基于领域驱动设计原则和模式的软件架构设计

目录

[1 快速入门 8](#_Toc142056411)

[2 入门教程 24](#_Toc142056412)

[2.1 Web应用程序 24](#_Toc142056413)

[2.1.1 设置你的开发环境 24](#_Toc142056414)

[2.1.2 创建新的解决方案 25](#_Toc142056415)

[2.1.3 运行解决方案 27](#_Toc142056416)

[2.2 控制台应用程序 31](#_Toc142056417)

[2.3 WPF应用程序 31](#_Toc142056418)

[2.4 空Web应用程序 32](#_Toc142056419)

[3 启动模版 35](#_Toc142056420)

[3.1 概述 35](#_Toc142056421)

[3.2 应用程序 35](#_Toc142056422)

[3.3 模块 45](#_Toc142056423)

[3.4 控制台 49](#_Toc142056424)

[3.5 WPF 50](#_Toc142056425)

[3.6 MAUI 50](#_Toc142056426)

[4 教程 51](#_Toc142056427)

[4.1 应用开发 51](#_Toc142056428)

[4.1.1 创建应用服务端 51](#_Toc142056429)

[4.1.2 图书列表页面 61](#_Toc142056430)

[4.1.3 创建,更新和删除书籍 75](#_Toc142056431)

[4.1.4 集成测试 106](#_Toc142056432)

[4.1.5 授权 111](#_Toc142056433)

[4.1.6 作者:领域层 123](#_Toc142056434)

[4.1.7 作者:数据库集成 129](#_Toc142056435)

[4.1.8 作者:应用服务层 134](#_Toc142056436)

[4.1.9 作者:用户界面 145](#_Toc142056437)

[4.1.10 图书-作者关系 169](#_Toc142056438)

[4.2 社区文章 193](#_Toc142056439)

[4.3 从ASP.NET Boilerplate迁移 193](#_Toc142056440)

[5 基础知识 212](#_Toc142056441)

[5.1 配置 212](#_Toc142056442)

[5.2 选项 212](#_Toc142056443)

[5.3 依赖注入 215](#_Toc142056444)

[5.3.1 Autofac 集成 223](#_Toc142056445)

[5.4 本地化 225](#_Toc142056446)

[5.5 异常处理 230](#_Toc142056447)

[5.6 验证 237](#_Toc142056448)

[5.6.1 FluentValidation集成 240](#_Toc142056449)

[5.7 授权 242](#_Toc142056450)

[5.8 缓存 251](#_Toc142056451)

[5.8.1 Redis 缓存 257](#_Toc142056452)

[5.9 日志 258](#_Toc142056453)

[5.10 设置管理 259](#_Toc142056454)

[5.11 链接字符串 264](#_Toc142056455)

[5.12 对象扩展 267](#_Toc142056456)

[6 基础设施 275](#_Toc142056457)

[6.1 后台服务 275](#_Toc142056458)

[6.1.1 后台作业 275](#_Toc142056459)

[6.1.2 Hangfire 集成 279](#_Toc142056460)

[6.1.3 RabbitMQ 集成 282](#_Toc142056461)

[6.1.4 Quartz 集成 285](#_Toc142056462)

[6.2 事件总线 295](#_Toc142056463)

[6.2.1 本地 Event Bus 296](#_Toc142056464)

[6.2.2 分布式 Event Bus 300](#_Toc142056465)

[6.3 种子数据 314](#_Toc142056466)

[6.4 分布式锁 318](#_Toc142056467)

[6.5 虚拟文件系统 321](#_Toc142056468)

[6.6 审计日志 325](#_Toc142056469)

[6.7 数据过滤 334](#_Toc142056470)

[6.8 当前用户 339](#_Toc142056471)

[6.9 对象到对象映射 343](#_Toc142056472)

[6.10 邮件发送 350](#_Toc142056473)

[6.10.1 邮件发送系统 350](#_Toc142056474)

[6.10.2 MailKit集成 356](#_Toc142056475)

[6.11 BLOB存储 358](#_Toc142056476)

[6.11.1 BLOB存储系统 358](#_Toc142056477)

[6.11.2 存储提供程序 365](#_Toc142056478)

[6.12 文本模版 380](#_Toc142056479)

[6.12.1 Razor 集成 381](#_Toc142056480)

[6.12.2 Scriban 集成 392](#_Toc142056481)

[6.13 GUID 生成 402](#_Toc142056482)

[6.14 时钟 405](#_Toc142056483)

[7 架构 408](#_Toc142056484)

[7.1 模块化 409](#_Toc142056485)

[7.1.1 基础 409](#_Toc142056486)

[7.1.2 模块插件 412](#_Toc142056487)

[7.1.3 自定义应用模块 417](#_Toc142056488)

[7.1.4 最佳实践 428](#_Toc142056489)

[7.2 领域驱动设计 451](#_Toc142056490)

[7.2.1 概述 451](#_Toc142056491)

[7.2.2 领域层 452](#_Toc142056492)

[7.2.3 应用服务层 479](#_Toc142056493)

[7.2.4 指南: 实现DDD 500](#_Toc142056494)

[7.3 多租户 542](#_Toc142056495)

[7.4 微服务架构 551](#_Toc142056496)

[8 API 553](#_Toc142056497)

[8.1 自动API控制器 553](#_Toc142056498)

[8.2 动态C# API客户端 556](#_Toc142056499)

[8.3 ABP端点 560](#_Toc142056500)

[8.3.1 应用程序配置 560](#_Toc142056501)

[8.4 API版本控制 560](#_Toc142056502)

[9 用户界面 564](#_Toc142056503)

[9.1 ASP.NET Core 564](#_Toc142056504)

[9.1.1 客户端包管理 564](#_Toc142056505)

[9.1.2 捆绑&压缩 567](#_Toc142056506)

[9.1.3 Tag Helpers 575](#_Toc142056507)

[9.1.4 仪表板和小部件(Widget)系统 588](#_Toc142056508)

[9.1.5 自定义/扩展UI 593](#_Toc142056509)

[9.1.6 测试 603](#_Toc142056510)

[9.1.7 主题化 607](#_Toc142056511)

[9.2 Angular 618](#_Toc142056512)

[9.2.1 v2.x 到 v3 迁移指南 618](#_Toc142056513)

[9.2.2 环境 627](#_Toc142056514)

[9.2.3 服务代理 631](#_Toc142056515)

[9.2.4 单元测试 632](#_Toc142056516)

[9.2.5 HTTP请求 640](#_Toc142056517)

[9.2.6 本地化 643](#_Toc142056518)

[9.2.7 权限管理 648](#_Toc142056519)

[9.2.8 多租户 649](#_Toc142056520)

[9.2.9 确认弹层 652](#_Toc142056521)

[9.2.10 Toast Overlay 656](#_Toc142056522)

[9.2.11 配置状态 659](#_Toc142056523)

[9.2.12 修改菜单 665](#_Toc142056524)

[9.2.13 替换组件 670](#_Toc142056525)

[9.2.14 懒加载Scripts与Styles 680](#_Toc142056526)

[9.2.15 DomInsertionService 684](#_Toc142056527)

[9.2.16 ContentProjectionService 687](#_Toc142056528)

[9.2.17 TrackByService 688](#_Toc142056529)

[9.2.18 SubscriptionService 691](#_Toc142056530)

[9.2.19 ListService 694](#_Toc142056531)

[9.3 React Native 698](#_Toc142056532)

[9.3.1 入门 698](#_Toc142056533)

[9.4 通用 699](#_Toc142056534)

[9.4.1 Utilities 699](#_Toc142056535)

[10 数据访问 727](#_Toc142056536)

[10.1 概述 727](#_Toc142056537)

[10.2 Entity Framework Core 727](#_Toc142056538)

[10.2.1 数据库迁移 746](#_Toc142056539)

[10.2.2 切换DMBS 765](#_Toc142056540)

[10.3 MongoDB 775](#_Toc142056541)

[10.4 Dapper 782](#_Toc142056542)

[11 实时 783](#_Toc142056543)

[11.1 SignalR集成 783](#_Toc142056544)

[12 测试 789](#_Toc142056545)

[13 部署 804](#_Toc142056546)

[13.1 部署到集群环境 805](#_Toc142056547)

[14 示例 808](#_Toc142056548)

[14.1 所有示例 808](#_Toc142056549)

[14.2 微服务示例 810](#_Toc142056550)

[15 应用模块 840](#_Toc142056551)

[15.1 概述 840](#_Toc142056552)

[15.2 账户 841](#_Toc142056553)

[15.3 审计日志 843](#_Toc142056554)

[15.4 后台作业 843](#_Toc142056555)

[15.5 CMS Kit 844](#_Toc142056556)

[15.6 文档 846](#_Toc142056557)

[15.7 功能管理 859](#_Toc142056558)

[15.8 Identity 862](#_Toc142056559)

[15.9 IdentityServer 869](#_Toc142056560)

[15.10 权限管理 872](#_Toc142056561)

[15.11 设置管理 875](#_Toc142056562)

[15.12 租户管理 881](#_Toc142056563)

[15.13 虚拟文件浏览器 883](#_Toc142056564)

[16 发布信息 885](#_Toc142056565)

[16.1 升级 885](#_Toc142056566)

[16.2 官方包 887](#_Toc142056567)

[16.3 预览版本 887](#_Toc142056568)

[16.4 每日构建 888](#_Toc142056569)

[16.5 路线图 888](#_Toc142056570)

[17 参考 890](#_Toc142056571)

[17.1 CLI 890](#_Toc142056572)

[17.1.1 新命令示例 900](#_Toc142056573)

[17.2 API文档 906](#_Toc142056574)

[18 贡献指南 906](#_Toc142056575)

# 快速入门

# 快速入门

````json

//[doc-params]

{

"UI": ["MVC", "Blazor", "BlazorServer", "NG"],

"DB": ["EF", "Mongo"]

}

````

这是一个由单个部分组成的快速入门教程, 旨在使用ABP框架构建一个简单的待办事项应用程序. 这是最终应用程序的屏幕截图:

![待办事项列表](todo-list.png)

你可以在[这里](https://github.com/abpframework/abp-samples/tree/master/TodoApp)找到已完成的项目源代码.

·

## 先决条件

\* 一个集成开发环境 (比如: [Visual Studio](https://visualstudio.microsoft.com/vs/)) 它需要支持 [.NET 5.0+](https://dotnet.microsoft.com/download/dotnet) 的开发.

{{if DB=="Mongo"}}

\* [MongoDB Server 4.0+](https://docs.mongodb.com/manual/administration/install-community/)

{{end}}

{{if UI=="NG"}}

\* [Node v16.x](https://nodejs.org/)

{{end}}

## 创建新的解决方案

我们将使用[ABP CLI](../../CLI.md) 创建带有ABP框架的新解决方案. 你可以在命令行终端中运行以下命令来安装它:

````bash

dotnet tool install -g Volo.Abp.Cli

````

然后创建一个空文件夹, 打开命令行终端并在终端中执行以下命令:

````bash

abp new TodoApp{{if UI=="Blazor"}} -u blazor{{else if UI=="BlazorServer"}} -u blazor-server{{else if UI=="NG"}} -u angular{{end}}{{if DB=="Mongo"}} -d mongodb{{end}}

````

{{if UI=="NG"}}

这将创建一个名为*\*TodoApp\**的新解决方案, 其中包含`angular`和`aspnet core`文件夹. 一旦解决方案就绪, 请在你最喜欢的 IDE 中打开 ASP.NET Core 解决方案.

{{else}}

这将创建一个名为*\*TodoApp\**的新解决方案. 一旦解决方案就绪, 请在你最喜欢的 IDE 中打开它.

{{end}}

### 创建数据库

如果你使用的是Visual Studio, 请右键单击`TodoApp.DbMigrator`项目, 选择*\*设置为启动项目\**, 然后按*\*Ctrl+F5\**运行它而不进行调试. 它将创建初始数据库并生成初始数据.

{{if DB=="EF"}}

> 一些IDE (例如Rider) 可能会在第一次运行时遇到问题, 因为 *\*DbMigrator\** 添加初始迁移并重新编译项目. 在这种情况下, 在 `.DbMigrator` 项目文件夹中打开命令行终端并执行 `dotnet run` 命令.

{{end}}

### 运行应用程序

{{if UI=="MVC" || UI=="BlazorServer"}}

最好在开始开发之前运行一下应用程序. 确保 {{if UI=="BlazorServer"}}`TodoApp.Blazor`{{else}}`TodoApp.Web`{{end}} 是启动项目, 然后运行应用程序(Visual Studio中是Ctrl+F5)来查看初始UI:

{{else if UI=="Blazor"}}

最好在开始开发之前运行一下应用程序. 解决方案中有两个主要应用程序

\* `TodoApp.HttpApi.Host` 承载服务器端的 HTTP API.

\* `TodoApp.Blazor` 是客户端的Blazor WebAssembly应用程序.

确保 `TodoApp.HttpApi.Host` 是启动项目, 然后运行应用程序(Visual Studio中的Ctrl+F5)打开[Swagger UI](https://swagger.io/tools/swagger-ui/)来查看服务器端的 HTTP API:

![todo-swagger-ui-initial](todo-swagger-ui-initial.png)

你可以使用此 UI 探索和测试你的 HTTP API. 现在, 我们可以将 `TodoApp.Blazor` 设置为启动项目并运行它来打开实际的 Blazor 应用程序UI:

{{else if UI=="NG"}}

最好在开始开发之前运行一下应用程序. 解决方案中有两个主要应用程序；

\* `TodoApp.HttpApi.Host` (在.NET解决方案中)承载服务器端HTTP API.

\* `Angular` 文件夹包含 Angular 应用程序.

确保 `TodoApp.HttpApi.Host` 是启动项目, 然后运行应用程序(Visual Studio中的Ctrl+F5)打开[Swagger UI](https://swagger.io/tools/swagger-ui/)来查看服务器端的 HTTP API:

![todo-swagger-ui-initial](todo-swagger-ui-initial.png)

你可以使用此 UI 探索和测试你的 HTTP API. 如果该功能正常, 我们可以运行Angular 客户端应用程序.

首先, 运行以下命令来还原NPM包:

````bash

npm install

````

安装所有软件包需要一些时间. 然后可以使用以下命令运行应用程序:

````bash

npm start

````

此命令需要时间, 但最终在你的默认浏览器中运行并打开应用程序:

{{end}}

![todo-ui-initial](todo-ui-initial.png)

你可以单击 *\*登录\** 按钮, 以`admin`作为用户名和`1q2w3E\*` 作为密码登录到应用程序.

一切就绪. 我们可以开始编程!

## 领域层

此应用程序只有一个 [实体](../../Entities.md), 接下来我们开始创建它. 在 *\*TodoApp.Domain\** 项目中创建一个新的 `TodoItem` 类:

````csharp

using System;

using Volo.Abp.Domain.Entities;

namespace TodoApp

{

public class TodoItem : BasicAggregateRoot<Guid>

{

public string Text { get; set; }

}

}

````

`BasicAggregateRoot` 是创建根实体的最简单的基础类. `Guid` 是这里实体的主键 (`Id`).

## 数据库集成

{{if DB=="EF"}}

下一步是设置 [Entity Framework Core](../../Entity-Framework-Core.md)配置

### 映射配置

打开在 *\*TodoApp.EntityFrameworkCore\** 项目中 `EntityFrameworkCore` 文件夹中的 `TodoAppDbContext` 类, 并向该类添加新的 `DbSet` 属性:

````csharp

public DbSet<TodoItem> TodoItems { get; set; }

````

然后在 `TodoAppDbContext` 类中定位到 `OnModelCreating` 方法, 并为 `TodoItem` 实体添加映射代码:

````csharp

protected override void OnModelCreating(ModelBuilder builder)

{

base.OnModelCreating(builder);

/\* Include modules to your migration db context \*/

builder.ConfigurePermissionManagement();

...

/\* Configure your own tables/entities inside here \*/

builder.Entity<TodoItem>(b =>

{

b.ToTable("TodoItems");

});

}

````

我们已经将 `TodoItem` 实体映射到数据库中的 `TodoItems` 表.

### Code First 迁移

解决方案快速模版已经配置为使用Entity Framework Core的 [Code First 迁移](https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations). 由于我们已经更改了数据库映射配置, 因此我们应该创建一个新的迁移并将更改应用于数据库.

在 *\*TodoApp.EntityFrameworkCore\** 项目目录中打开一个命令行终端并输入以下命令:

````bash

dotnet ef migrations add Added\_TodoItem

````

这将向项目添加一个新的迁移类:

![todo-efcore-migration](todo-efcore-migration.png)

你可以在同一命令行终端中使用以下命令将更改应用于数据库:

````bash

dotnet ef database update

````

> 如果你使用的是Visual Studio, 则可能希望在 *\*包管理器控制台 (PMC)\** 中使用 `Add-Migration Added\_TodoItem` 和 `Update-Database` 命令. 在这种情况下, 请确保 {{if UI=="MVC"}}`TodoApp.Web`{{else if UI=="BlazorServer"}}`TodoApp.Blazor`{{else if UI=="Blazor" || UI=="NG"}}`TodoApp.HttpApi.Host`{{end}} 是启动项目, 并且 `TodoApp.EntityFrameworkCore` 是PMC中的 *\*默认项目\**.

{{else if DB=="Mongo"}}

下一步是设置 [MongoDB](../../MongoDB.md) 配置. 打开 *\*TodoApp.MongoDB\** 项目中 `MongoDb` 文件夹的`TodoAppMongoDbContext` 类并做出以下更改；

1. 向类添加新属性:

````csharp

public IMongoCollection<TodoItem> TodoItems => Collection<TodoItem>();

````

2. 在 `CreateModel` 方法中添加以下代码:

````csharp

modelBuilder.Entity<TodoItem>(b =>

{

b.CollectionName = "TodoItems";

});

````

{{end}}

现在, 我们可以使用ABP仓库保存和检索待办事项, 如我们在下一节所做的那样.

## 应用层

[应用程序服务](../../Application-Services.md) 用于执行应用程序的用例. 我们需要执行以下用例:

\* 获取待办事项列表

\* 创建新的待办事项

\* 删除现有的待办事项

### 应用服务接口

我们可以先从为应用程序服务定义接口开始. 在 *\*TodoApp.Application.Contracts\** 项目中创建新的 `ITodoAppService` 接口, 如下所示:

````csharp

using System;

using System.Collections.Generic;

using System.Threading.Tasks;

using Volo.Abp.Application.Services;

namespace TodoApp

{

public interface ITodoAppService : IApplicationService

{

Task<List<TodoItemDto>> GetListAsync();

Task<TodoItemDto> CreateAsync(string text);

Task DeleteAsync(Guid id);

}

}

````

### 数据传输对象

`GetListAsync` 和`CreateAsync` 方法返回 `TodoItemDto`. `应用程序服务` 通常获取并返回 DTO([数据传输对象](../../Data-Transfer-Objects.md)) 而不是实体. 因此, 我们应该在这里定义DTO类. 在 *\*TodoApp.Application.Contracts\** 项目中创建新的 `TodoItemDto` 类:

````csharp

using System;

namespace TodoApp

{

public class TodoItemDto

{

public Guid Id { get; set; }

public string Text { get; set; }

}

}

````

这是一个非常简单的DTO类, 它与我们的 `TodoItem` 实体相对应. 接下来, 我们准备实现 `ITodoAppService`接口.

### 应用服务实现

在 *\*TodoApp.Application\** 项目中创建 `TodoAppService` 类, 如下所示:

````csharp

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Volo.Abp.Application.Services;

using Volo.Abp.Domain.Repositories;

namespace TodoApp

{

public class TodoAppService : ApplicationService, ITodoAppService

{

private readonly IRepository<TodoItem, Guid> \_todoItemRepository;

public TodoAppService(IRepository<TodoItem, Guid> todoItemRepository)

{

\_todoItemRepository = todoItemRepository;

}

// TODO: Implement the methods here...

}

}

````

该类继承自ABP框架的`ApplicationService`类, 并实现了之前定义的 `ITodoAppService`接口. ABP为实体提供默认的泛型 [仓储](../../Repositories.md). 我们可以使用它们来执行基本的数据库操作. 此类中 [注入](../../Dependency-Injection.md) 的 `IRepository<TodoItem, Guid>`, 它就是 `TodoItem` 实体的默认存储库. 我们将使用它来实现之前描述的用例.

#### 获取待办事项

让我们先实现 `GetListAsync` 方法:

````csharp

public async Task<List<TodoItemDto>> GetListAsync()

{

var items = await \_todoItemRepository.GetListAsync();

return items

.Select(item => new TodoItemDto

{

Id = item.Id,

Text = item.Text

}).ToList();

}

````

我们只是从数据库中获取完整的`TodoItem`列表, 将它们映射到`TodoItemDto`对象并作为结果返回.

#### 创建一个新的待办事项

下一个我们可以实现方法是 `CreateAsync` , 如下所示:

````csharp

public async Task<TodoItemDto> CreateAsync(string text)

{

var todoItem = await \_todoItemRepository.InsertAsync(

new TodoItem {Text = text}

);

return new TodoItemDto

{

Id = todoItem.Id,

Text = todoItem.Text

};

}

````

仓储的 `InsertAsync` 方法将给定的`TodoItem`插入数据库, 并返回相同的`TodoItem` 对象. 它还设置了`Id`, 因此我们可以在返回对象上使用它. 我们只是通过从新的 `TodoItem` 实体创建和返回 `TodoItemDto`.

#### 删除待办事项

最后, 我们来实现 `DeleteAsync` 方法, 代码如下:

````csharp

public async Task DeleteAsync(Guid id)

{

await \_todoItemRepository.DeleteAsync(id);

}

````

至此, 应用程序服务已准备好了让UI层来使用.

## 用户界面层

现在是在UI上显示待办事项的时候了! 在开始编写代码之前, 最好记住我们正在尝试构建的内容. 这里是示例程序的最终用户界面的截图:

![待办事项列表](todo-list.png)

> **\*\*我们将在本教程中保持最简洁的UI端, 以使本教程简单且重点突出. 请参阅 [**web应用程序开发教程**](../Part-1.md) 来了解构建实际应用各个方面.\*\***

{{if UI=="MVC"}}

### Index.cshtml.cs

打开在 *\*TodoApp.Web\** 项目的 `Pages` 文件夹中`Index.cshtml.cs`文件, 并用以下代码块替换它的默认内容:

````csharp

using System.Collections.Generic;

using System.Threading.Tasks;

namespace TodoApp.Web.Pages

{

public class IndexModel : TodoAppPageModel

{

public List<TodoItemDto> TodoItems { get; set; }

private readonly ITodoAppService \_todoAppService;

public IndexModel(ITodoAppService todoAppService)

{

\_todoAppService = todoAppService;

}

public async Task OnGetAsync()

{

TodoItems = await \_todoAppService.GetListAsync();

}

}

}

````

此类使用 `ITodoAppService` 获取待办事项列表并将它赋值给 `TodoItems` 属性. 我们将用它来渲染razor页面上的待办事项目列表.

### Index.cshtml

打开 *\*TodoApp.Web\** 项目中`Pages`文件夹下的`Index.cshtml`, 并替换为以下内容:

````xml

@page

@model TodoApp.Web.Pages.IndexModel

@section styles {

<abp-style src="/Pages/Index.css" />

}

@section scripts {

<abp-script src="/Pages/Index.js" />

}

<div class="container">

<abp-card>

<abp-card-header>

<abp-card-title>

TODO LIST

</abp-card-title>

</abp-card-header>

<abp-card-body>

<!-- FORM FOR NEW TODO ITEMS -->

<form id="NewItemForm" class="form-inline">

<input id="NewItemText"

type="text"

class="form-control mr-2"

placeholder="enter text...">

<button type="submit" class="btn btn-primary">Submit</button>

</form>

<!-- TODO ITEMS LIST -->

<ul id="TodoList">

@foreach (var todoItem in Model.TodoItems)

{

<li data-id="@todoItem.Id">

<i class="fa fa-trash-o"></i> @todoItem.Text

</li>

}

</ul>

</abp-card-body>

</abp-card>

</div>

````

我们使用 ABP 的 [卡片标签助手](../../UI/AspNetCore/Tag-Helpers/Cards.md) 来创建一个简单的卡片视图. 你可以直接使用标准引导HTML结构, 但ABP [标签助手](../../UI/AspNetCore/Tag-Helpers/Index.md) 使它更容易并且更安全.

此页面导入一个 CSS 和 JavaScript 文件, 所以我们需要创建它们.

### Index.js

打开 *\*TodoApp.Web\** 项目中`Pages`文件夹下的`Index.js`, 并替换为以下内容:

````js

$(function () {

// DELETING ITEMS /////////////////////////////////////////

$('#TodoList').on('click', 'li i', function(){

var $li = $(this).parent();

var id = $li.attr('data-id');

todoApp.todo.delete(id).then(function(){

$li.remove();

abp.notify.info('Deleted the todo item.');

});

});

// CREATING NEW ITEMS /////////////////////////////////////

$('#NewItemForm').submit(function(e){

e.preventDefault();

var todoText = $('#NewItemText').val();

todoApp.todo.create(todoText).then(function(result){

$('<li data-id="' + result.id + '">')

.html('<i class="fa fa-trash-o"></i> ' + result.text)

.appendTo($('#TodoList'));

$('#NewItemText').val('');

});

});

});

````

在第一部分中, 我们注册了在待办事项旁边的删除图标的点击事件, 来删除服务器上的相关项目并在UI上显示通知. 此外, 我们会从 DOM 中移除已删除的项目, 所以我们不需要刷新页面.

在第二部分中, 我们在服务器上创建一个新的待办事项. 如果成功, 我们将操纵DOM来将新的 `<li>` 元素插入到待办事项列表. 这样我们不需要在创建新的待办事项后刷新整个页面.

这里有趣的部分是我们如何与服务器通信. 请参阅 *\*动态JavaScript代理和自动API控制器\** 部分来了解其工作原理. 但是现在让我们继续并完成这个应用程序.

### Index.css

最后, 请打开 *\*TodoApp.Web\** 项目的 `Pages` 文件夹中的 `Index.css` 文件, 并替换为以下内容:

````css

#TodoList{

list-style: none;

margin: 0;

padding: 0;

}

#TodoList li {

padding: 5px;

margin: 5px 0px;

border: 1px solid #cccccc;

background-color: #f5f5f5;

}

#TodoList li i

{

opacity: 0.5;

}

#TodoList li i:hover

{

opacity: 1;

color: #ff0000;

cursor: pointer;

}

````

这是待办事项页面的简单样式. 我们相信你可以做得更好 :)

现在, 你可以再次运行应用程序并看到结果.

### 动态JavaScript代理和自动API控制器

在 `Index.js` 文件中, 我们使用了 `todoApp.todo.delete(...)` 和 `todoApp.todo.create(...)` 方法来与服务器通信. 这些函数是由ABP框架动态创建的, 这要归功于 [动态JavaScript客户端代理](../../UI/AspNetCore/Dynamic-JavaScript-Proxies.md) 系统. 它们执行 HTTP API 到服务器调用并返回一个Promise对象, 这样你就可以使用 `then` 函数注册一个回调, 像之前所做的那样.

但是, 你可能会注意到我们还没有创建任何API控制器, 所以服务器如何处理这些请求? 这个问题为我们引出了ABP 框架的 [自动API控制器](../../API/Auto-API-Controllers.md) 功能. 它通过约定自动将应用程序服务转换为 API 控制器.

如果你在应用程序URL中输入 `/swagger` 来打开 [Swagger UI](https://swagger.io/tools/swagger-ui/) , 就会看到Todo API:

![todo-api](todo-api.png)

{{else if UI=="Blazor" || UI=="BlazorServer"}}

### Index.razor.cs

打开在*\*TodoApp.Blazor\**项目的`Pages`文件夹中`Index.razor.cs`文件, 并替换为一下内容:

````csharp

using Microsoft.AspNetCore.Components;

using System.Collections.Generic;

using System.Threading.Tasks;

namespace TodoApp.Blazor.Pages

{

public partial class Index

{

[Inject]

private ITodoAppService TodoAppService { get; set; }

private List<TodoItemDto> TodoItems { get; set; } = new List<TodoItemDto>();

private string NewTodoText { get; set; }

protected async override Task OnInitializedAsync()

{

TodoItems = await TodoAppService.GetListAsync();

}

private async Task Create()

{

var result = await TodoAppService.CreateAsync(NewTodoText);

TodoItems.Add(result);

NewTodoText = null;

}

private async Task Delete(TodoItemDto todoItem)

{

await TodoAppService.DeleteAsync(todoItem.Id);

await Notify.Info("Deleted the todo item.");

TodoItems.Remove(todoItem);

}

}

}

````

此类使用 `ITodoAppService` 来执行待办事项的操作. 它在创建和删除操作后操纵 `TodoItems` 列表. 这样, 我们不需要从服务器刷新整个待办事项列表.

{{if UI=="Blazor"}}

请参阅下面的 *\*动态C#代理和自动API控制器\** 部分, 来了解我们如何在浏览器上运行的 Blazor 应用程序中使用注入和应用服务接口的! 但是现在让我们继续并完成这个应用程序.

{{end # Blazor}}

### Index.razor

打开在 *\*TodoApp.Blazor\** 项目的`Pages`文件夹中 `Index.razor`文件, 并替换为以下代码块内容:

````xml

@page "/"

@inherits TodoAppComponentBase

<div class="container">

<Card>

<CardHeader>

<CardTitle>

TODO LIST

</CardTitle>

</CardHeader>

<CardBody>

<!-- FORM FOR NEW TODO ITEMS -->

<form id="NewItemForm"

@onsubmit:preventDefault

@onsubmit="() => Create()"

class="form-inline">

<input type="text"

@bind-value="@NewTodoText"

class="form-control mr-2"

placeholder="enter text...">

<button type="submit" class="btn btn-primary">Submit</button>

</form>

<!-- TODO ITEMS LIST -->

<ul id="TodoList">

@foreach (var todoItem in TodoItems)

{

<li data-id="@todoItem.Id">

<i class="far fa-trash-alt"

@onclick="() => Delete(todoItem)"

></i> @todoItem.Text

</li>

}

</ul>

</CardBody>

</Card>

</div>

````

### Index.razor.css

最后, 请打开 *\*TodoApp.Blazor\** 项目的 `Pages` 文件夹中的 `Index.razor.css`文件, 并替换为以下内容:

````css

#TodoList{

list-style: none;

margin: 0;

padding: 0;

}

#TodoList li {

padding: 5px;

margin: 5px 0px;

border: 1px solid #cccccc;

background-color: #f5f5f5;

}

#TodoList li i

{

opacity: 0.5;

}

#TodoList li i:hover

{

opacity: 1;

color: #ff0000;

cursor: pointer;

}

````

这是待办事项页面的简单样式. 我们相信你可以做得更好 :)

现在, 你可以再次运行应用程序来查看结果.

{{if UI=="Blazor"}}

### 动态C#代理和自动API控制器

在 `Index.razor.cs` 文件中, 我们已经注入(使用 `[Inject]` 特性)并像使用本地服务一样使用 `ITodoAppService`. 请记住, Blazor应用程序是在浏览器上运行, 但此应用程序服务的实现却是在服务器上运行的.

这个神奇的过程是由ABP框架的 [动态C#客户端代理](../../API/Dynamic-CSharp-API-Clients.md) 系统完成. 它使用标准的`HttpClient` 并执行HTTP API到远程服务器请求. 它还为我们处理所有标准任务, 包括授权、JSON序列化和异常处理.

但是, 你可能会问我们还没有创建任何API控制器, 那么服务器如何处理这些请求呢? 这个问题为我们引出了ABP 框架的 [自动API控制器](../../API/Auto-API-Controllers.md) 功能. 它通过约定自动将应用程序服务转换为 API 控制器.

如果你运行 `TodoApp.HttpApi.Host` 应用程序, 你可以看到Todo API:

![todo-api](todo-api.png)

{{end # Blazor}}

{{else if UI=="NG"}}

### 服务代理生成

ABP提供了一个便捷的功能来自动创建客户端服务, 以方便地使用由服务器提供的HTTP API.

你首先需要运行 `TodoApp.HttpApi.Host` 项目, 因为代理生成器从服务器应用程序读取API定义.

> **\*\*请注意\*\***: IIS Express有一个问题: 它不允许从另一个进程连接到应用程序. 如果你使用的是Visual Studio, 请在 “运行” 按钮下拉列表中选择`TodoApp.HttpApi.Host` 来替代IIS Express, 如下图所示:

![运行时不使用 iisexpress](run-without-iisexpress.png)

当启动 `TodoApp.HttpApi.Host` 项目后, 在`angular`文件夹中打开一个命令行终端并输入以下命令:

````bash

abp generate-proxy -t ng

````

如果一切顺利, 它应该生成如下输出:

````bash

CREATE src/app/proxy/generate-proxy.json (170978 bytes)

CREATE src/app/proxy/README.md (1000 bytes)

CREATE src/app/proxy/todo.service.ts (794 bytes)

CREATE src/app/proxy/models.ts (66 bytes)

CREATE src/app/proxy/index.ts (58 bytes)

````

然后, 我们可以使用 `todoService` 来使用服务器端HTTP API, 就像我们将在下一节中做的那样.

### home.component.ts

打开 `/angular/src/app/home/home.component.ts` 文件, 并用以下代码块替换其内容:

````js

import { ToasterService } from '@abp/ng.theme.shared';

import { Component, OnInit } from '@angular/core';

import { TodoItemDto, TodoService } from '@proxy';

@Component({

selector: 'app-home',

templateUrl: './home.component.html',

styleUrls: ['./home.component.scss']

})

export class HomeComponent implements OnInit {

todoItems: TodoItemDto[];

newTodoText: string;

constructor(

private todoService: TodoService,

private toasterService: ToasterService)

{ }

ngOnInit(): void {

this.todoService.getList().subscribe(response => {

this.todoItems = response;

});

}

create(): void{

this.todoService.create(this.newTodoText).subscribe((result) => {

this.todoItems = this.todoItems.concat(result);

this.newTodoText = null;

});

}

delete(id: string): void {

this.todoService.delete(id).subscribe(() => {

this.todoItems = this.todoItems.filter(item => item.id !== id);

this.toasterService.info('Deleted the todo item.');

});

}

}

````

我们已经实现了使用 `todoService` 来获取待办事项目列表, 并将返回值赋值给 `todoItems` 数组. 我们还添加了 `create` 和 `delete` 方法. 这些方法将在视图端使用.

### home.component.html

打开 `/angular/src/app/home/home.component.html` 文件, 并用以下代码块替换其内容:

````html

<div class="container">

<div class="card">

<div class="card-header">

<div class="card-title">TODO LIST</div>

</div>

<div class="card-body">

<!-- FORM FOR NEW TODO ITEMS -->

<form class="form-inline" (ngSubmit)="create()">

<input

name="NewTodoText"

type="text"

[(ngModel)]="newTodoText"

class="form-control mr-2"

placeholder="enter text..."

/>

<button type="submit" class="btn btn-primary">Submit</button>

</form>

<!-- TODO ITEMS LIST -->

<ul id="TodoList">

<li \*ngFor="let todoItem of todoItems">

<i class="fa fa-trash-o" (click)="delete(todoItem.id)"></i> {%{{{ todoItem.text }}}%}

</li>

</ul>

</div>

</div>

</div>

````

### home.component.scss

最后, 打开`/angular/src/app/home/home.component.scss` 文件, 并用以下代码块替换其内容:

````css

#TodoList{

list-style: none;

margin: 0;

padding: 0;

}

#TodoList li {

padding: 5px;

margin: 5px 0px;

border: 1px solid #cccccc;

background-color: #f5f5f5;

}

#TodoList li i

{

opacity: 0.5;

}

#TodoList li i:hover

{

opacity: 1;

color: #ff0000;

cursor: pointer;

}

````

这是待办事项页面的简单样式. 我们相信你可以做得更好 :)

现在, 你可以再次运行应用程序来查看结果.

{{end}}

## 总结

在这个教程中, 我们已经建立了一个非常简单的应用程序来熟悉 ABP 框架. 如果你想要构建一个实际场景的应用程序, 请查看 [web 应用程序开发教程](../Part-1.md), 该教程涵盖了真实的 Web 应用程序开发的所有方面.

## 源代码

你可以在[这里](https://github.com/abpframework/abp-samples/tree/master/TodoApp)找到已完成的项目源代码.

## 另请参见

\* [Web应用程序开发教程](../Part-1.md)

# 入门教程

## Web应用程序

````json

//[doc-params]

{

"UI": ["MVC", "Blazor", "BlazorServer", "NG"],

"DB": ["EF", "Mongo"],

"Tiered": ["Yes", "No"]

}

````

> 本文档假设你更喜欢使用 **\*\*{{ UI\_Value }}\*\*** 作为 UI 框架, 使用 **\*\*{{ DB\_Value }}\*\*** 作为数据库提供程序. 对于其他选项, 请更改本文档顶部的首选项.

## 内容

本教程介绍如何使用 ABP 框架 **\*\*创建和运行\*\*** 新的 Web 应用程序. 请按照以下步骤操作;

1. [设置你的开发环境](Getting-Started-Setup-Environment.md)

2. [创建新的解决方案](Getting-Started-Create-Solution.md)

3. [运行解决方案](Getting-Started-Running-Solution.md)

设置你的开发环境

第一件事! 在创建项目之前, 让我们先设置你的开发环境.

### 先决条件

开发计算机上应安装以下工具:

\* 一个集成开发环境 (比如: [Visual Studio](https://visualstudio.microsoft.com/vs/)) 它需要支持 [.NET 7.0+](https://dotnet.microsoft.com/download/dotnet) 的开发.

{{ if UI != "Blazor" }}

\* [Node v16 或 v18](https://nodejs.org/)

\* [Yarn v1.20+ (不是v2)](https://classic.yarnpkg.com/en/docs/install) <sup id="a-yarn">[1](#f-yarn)</sup> 或 npm v6+ (已跟随Node一起安装)

{{ end }}

{{ if Tiered == "Yes" }}

\* [Redis](https://redis.io/) (启动解决方案使用 Redis 作为 [分布式缓存](Caching.md)).

{{ end }}

{{ if UI != "Blazor" }}

<sup id="f-yarn"><b>1</b></sup> *\_Yarn v2 工作方式不同, 不被支持.\_* <sup>[↩](#a-yarn)</sup>

{{ end }}

### 安装 ABP CLI

[ABP CLI](./CLI.md) 是一个命令行界面, 用于自动执行基于 ABP 的解决方案的一些常见任务. 首先, 你需要使用以下命令安装 ABP CLI：

````shell

dotnet tool install -g Volo.Abp.Cli

````

如果已安装, 则可以使用以下命令对其进行更新:

````shell

dotnet tool update -g Volo.Abp.Cli

````

## 下一步

\* [创建新的解决方案](Getting-Started-Create-Solution.md)

创建新的解决方案

我们将使用 ABP CLI 创建一个新的 ABP 项目.

> 或者, 你可以使用[ABP Framework 网站](https://abp.io/get-started)页面上的选项轻松的 **\*\*创建并下载\*\*** 项目.

使用 ABP CLI 的 `new` 命令创建一个新项目:

````shell

abp new Acme.BookStore{{if UI == "NG"}} -u angular{{else if UI == "Blazor"}} -u blazor{{else if UI == "BlazorServer"}} -u blazor-server{{end}}{{if DB == "Mongo"}} -d mongodb{{end}}{{if Tiered == "Yes"}}{{if UI == "MVC" || UI == "BlazorServer"}} --tiered{{else}} --separate-auth-server{{end}}{{end}}

````

*\*你可以使用不同级别的命名空间, 例如: BookStore、Acme.BookStore或 Acme.Retail.BookStore.\**

{{ if Tiered == "Yes" }}

{{ if UI == "MVC" || UI == "BlazorServer" }}

\* `--tified` 参数用于创建认证服务器、 UI 和 API 实际分隔的 N-层解决方案.

{{ else }}

\* `--separate-auth-server` 参数用于将Auth Server应用程序与API主机应用程序分隔开. 如果未指定, 则服务器上将只有一个端点.

{{ end }}

{{ end }}

> [ABP CLI 文档](./CLI.md) 涵盖了所有可用的命令和选项.

## 移动端开发

如果你想要在你的解决方案中包含 [React Native](https://reactnative.dev/) 项目, 将 `-m react-native` (or `--mobile react-native`) 参数添加到项目创建命令. 这是一个基础的 React Native 启动模板, 用于开发基于你的 ABP 后端的移动应用程序.

请参阅 [React Native 入门](Getting-Started-React-Native.md) 文档, 了解如何配置和运行 React Native 应用程序.

### 解决方案结构

该解决方案具有分层结构 (基于 [域驱动设计](Domain-Driven-Design.md)), 并包含单元 & 集成测试项目. 请参阅 [应用程序模板文档](Startup-Templates/Application.md) 以详细了解解决方案结构.

{{ if DB == "Mongo" }}

#### MongoDB 事务

[启动模板](Startup-templates/Index.md) 默认在`.MongoDB`项目中**\*\*禁用\*\***事务. 如果你的MongoDB服务器支持事务, 你可以在*\*YourProjectMongoDbModule\**类中的`ConfigureServices`方法开启它:

```csharp

Configure<AbpUnitOfWorkDefaultOptions>(options =>

{

options.TransactionBehavior = UnitOfWorkTransactionBehavior.Auto;

});

```

> 或者你可以删除该代码, 因为 `Auto` 已经是默认行为.

{{ end }}

## 下一步

\* [运行解决方案](Getting-Started-Running-Solution.md)

运行解决方案

## 创建数据库

### 连接字符串

检查在 {{if Tiered == "Yes"}}`.AuthServer` 和`.HttpApi.Host` 项目{{else}}{{if UI=="MVC"}}`.Web` 项目{{else if UI=="BlazorServer"}}`.Blazor` 项目{{else}}`.HttpApi.Host` 项目{{end}}{{end}} 中 `appsettings.json` 文件里的**\*\*连接字符串\*\***.

{{ if DB == "EF" }}

````json

"ConnectionStrings": {

"Default": "Server=(LocalDb)\MSSQLLocalDB;Database=BookStore;Trusted\_Connection=True"

}

````

> **\*\*关于连接字符串和数据库管理系统\*\***

>

> 解决方案配置为默认使用 **\*\*Entity Framework Core\*\*** 与 **\*\*MS SQL Server\*\***. 但是, 如果在执行ABP CLI 的`new`命令时使用了`-dbms`参数来选择其他DBMS (如`-dbms MySQL`), 那么连接字符串可能不同.

>

> EF Core 支持 [多种](https://docs.microsoft.com/en-us/ef/core/providers/) 据库提供程序, 因此你可以使用任何受支持的DBMS. 你可以需要时候参阅[Entity Framework 集成文档](Entity-Framework-Core.md) 来学习如何[切换到另一个DBMS](Entity-Framework-Core-Other-DBMS.md).

### 数据库迁移

该解决方案使用[Entity Framework Core Code First 迁移](https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/?tabs=dotnet-core-cli). 带有 `.DbMigrator` 的控制台程序用于 **\*\*应用迁移\*\*** 和 **\*\*初始化种子数据\*\***. 它在**\*\*开发\*\***和**\*\*生产\*\***环境中都很有用.

> `.dbMigator` 项目有自己的 `appsettings.json`. 因此, 如果你更改了之前的连接字符串, 那么也应该更改这个连接字符串.

### 初次迁移

`.dbMigator` 应用程序在首次运行时自动**\*\*创建初始迁移\*\***.

**\*\*如果你使用的是 Visual Studio, 你可以跳到 *\*运行 dbMigrator\** 部分.\*\*** 但是, 其他 IDE (例如 Rider) 在首次运行时可能会遇到问题, 因为它会添加初始迁移并编译项目. 在这种情况下, 请在 `.dbMigration` 项目的文件夹中打开命令行终端, 然后运行以下命令:

````bash

dotnet run

````

下次, 你可以像往常一样在 IDE 中运行它.

### 运行迁移

右键单击 `.dbMigration` 项目, 然后选择 **\*\*设置为启动项目\*\***

![set-as-startup-project](images/set-as-startup-project.png)

按F5(或Ctrl + F5) 运行应用程序. 它将具有如下所示的输出:

![db-migrator-output](images/db-migrator-output.png)

> 初始的[种子数据](Data-Seeding.md)在数据库中创建了 `admin` 用户(密码为`1q2w3E\*`) 用于登录应用程序. 所以, 对于新数据库至少使用 `.DbMigrator` 一次.

{{ else if DB == "Mongo" }}

````json

"ConnectionStrings": {

"Default": "mongodb://localhost:27017/BookStore"

}

````

该解决方案配置为在本地计算机中使用 **\*\*MongoDB\*\***, 因此你需要启动并运行 MongoDB 服务器实例, 或者将连接字符串更改为另一台 MongoDB 服务器.

### 种子初始数据

该解决方案带有 `.DbMigrator` 的控制台程序用于 **\*\*初始化种子数据\*\***. 它在**\*\*开发\*\***和**\*\*生产\*\***环境中都很有用.

> `.dbMigator` 项目有自己的 `appsettings.json`. 因此, 如果你更改了之前的连接字符串, 那么也应该更改这个连接字符串.

右键单击 `.dbMigration` 项目, 然后选择 **\*\*设置为启动项目\*\***

![set-as-startup-project](images/set-as-startup-project.png)

按F5(或Ctrl + F5) 运行应用程序. 它将具有如下所示的输出:

![db-migrator-output](images/db-migrator-output.png)

> 初始的[种子数据](Data-Seeding.md)在数据库中创建了 `admin` 用户(密码为`1q2w3E\*`) 用于登录应用程序. 所以, 对于新数据库至少使用 `.DbMigrator` 一次.

{{ end }}

## 运行应用程序

{{ if UI == "MVC" || UI == "BlazorServer" }}

{{ if Tiered == "Yes" }}

> 分层解决方案使用 **\*\*Redis\*\*** 作为分布式缓存. 确保它已安装并在本地计算机上运行. 如果你使用的是远程 Redis 服务器, 请修改项目的 ` appsettings.json` 文件中的配置.

1. 确保 `.AuthServer` 项目是启动项目. 运行此应用程序, 它将在浏览器中打开 **\*\*登录\*\*** 页面.

> 在 Visual Studio 中使用 Ctrl+F5(而不是F5) 在不进行调试的情况下运行应用程序. 如果你没有调试目的, 这会更快.

你可以登录, 但不能在这里进入主应用程序. 这 **\*\*只是身份验证服务器\*\***.

2. 确保 `.httpapi.Host` 项目是启动项目, 然后运行应用将在浏览器中打开 **\*\*Swagger UI\*\***.

![swagger-ui](images/swagger-ui.png)

这是 Web 应用程序使用的 HTTP API.

3. 最后, 确保 {{if UI=="MVC"}}`.Web`{{else}}`.Blazor`{{end}} 项目是启动项目,然后运行应用程序,它将在浏览器中打开 **\*\*欢迎\*\*** 页面

![mvc-tiered-app-home](images/bookstore-home.png)

单击 **\*\*login\*\*** 按钮, 它将重定向到 *\*身份验证服务器\** 以登录到应用程序:

![bookstore-login](images/bookstore-login.png)

{{ else # Tiered != "Yes" }}

确保 {{if UI=="MVC"}}`.Web`{{else}}`.Blazor`{{end}} 项目是启动项目. 运行应用程序将会在浏览器中打开 **\*\*login\*\*** 页面:

> 在 Visual Studio 中使用 Ctrl+F5(而不是F5) 在不进行调试的情况下运行应用程序. 如果你没有调试目的, 这会更快.

![bookstore-login](images/bookstore-login.png)

{{ end # Tiered }}

{{ else # UI != MVC || BlazorServer }}

### 运行 HTTP API 主机(服务器端)

{{ if Tiered == "Yes" }}

> 分层解决方案使用 Redis 作为分布式缓存. 确保它已安装并在本地计算机上运行. 如果你使用的是远程 Redis 服务器, 请修改项目的 `appsettings.json` 文件中的配置.

确保 `.AuthServer` 项目是启动项目. 运行此应用程序, 它将在浏览器中打开 **\*\*登录\*\*** 页面.

> 在 Visual Studio 中使用 Ctrl+F5(而不是F5) 在不进行调试的情况下运行应用程序. 如果你没有调试目的, 这会更快.

你可以登录, 但不能在这里进入主应用程序. 这 **\*\*只是身份验证服务器\*\***.

确保 `.HttpApi.Host` 项目是启动项目, 然后运行应用程序将打开 Swagger UI 的:

{{ else # Tiered == "No" }}

确保 `.HttpApi.Host` 项目是启动项目, 然后运行应用程序将打开 Swagger UI 的:

> 在 Visual Studio 中使用 Ctrl+F5(而不是F5) 在不进行调试的情况下运行应用程序. 如果你没有调试目的, 这会更快.

{{ end # Tiered }}

![swagger-ui](images/swagger-ui.png)

你可以在这里查看应用程序的API并测试它们. 获取Swagger UI的[更多信息](https://swagger.io/tools/swagger-ui/).

{{ end # UI }}

{{ if UI == "Blazor" }}

### 运行 Blazor 应用程序 (客户端)

确保 `.Blazor` 项目是启动项目并运行应用程序.

> 在 Visual Studio 中使用 Ctrl+F5(而不是F5) 在不进行调试的情况下运行应用程序. 如果你没有调试目的, 这会更快.

应用程序启动后, 单击页头上的 **\*\*Login\*\*** 链接, 你将重定向到身份验证服务器以输入用户名和密码:

![bookstore-login](images/bookstore-login.png)

{{ else if UI == "NG" }}

### 运行 Angular 应用程序 (客户端)

转到 `Angular` 文件夹, 打开命令行终端, 键入 `yarn` 命令(我们建议使用 [yarn](https://yarnpkg.com/) 软件包管理器, 而 `npm install` 也可以使用)

```bash

yarn

```

当所有node模块加载完毕后, 执行 `yarn start` (或 `npm start`) 命令：

```bash

yarn start

```

初次构建可能需要更长的时间. 完成后, 它会在默认浏览器中使用 [localhost:4200](http://localhost:4200/) 地址打开 Angular UI.

![bookstore-login](images/bookstore-login.png)

{{ end }}

输入用户名 **\*\*admin\*\*** 和 密码 **\*\*1q2w3E\*\***\* 登录应用程序. 应用程序已启动并正在运行. 你可以根据此启动模板开始开发应用程序.

## 另请参见

\* [Web 应用程序开发教程](Tutorials/Part-1.md)

\* [应用程序启动模板](Startup-Templates/Application.md)

## 控制台应用程序

# 控制台应用程序启动模板

此模板用于创建一个最小的依赖关系的ABP控制台应用程序项目.

## 如何开始?

首先,如果你没有安装[ABP CLI](../CLI.md),请先安装它:

````bash

dotnet tool install -g Volo.Abp.Cli

````

在一个空文件夹使用 `abp new` 命令创建新解决方案:

````bash

abp new Acme.MyConsoleApp -t console

````

`Acme.MyConsoleApp` 是解决方案的名称, 如*\*YourCompany.YourProduct\**. 你可以使用单级或多级名称.

## 解决方案结构

使用以上命令创建解决方案后,你会得到如下所示的解决方案:

![basic-console-application-solution](../images/basic-console-application-solution.png)

\* `HelloWorldService` 是一个实现了 `ITransientDependency` 接口的示例服务. 它会自动注册到[依赖注入](../Dependency-Injection.md)系统.

## WPF应用程序

# WPF应用程序启动模板

此模板用于创建一个最小的依赖关系的ABP WPF应用程序项目.

## 如何开始?

首先,如果你没有安装[ABP CLI](../CLI.md),请先安装它:

````bash

dotnet tool install -g Volo.Abp.Cli

````

在一个空文件夹使用 `abp new` 命令创建新解决方案:

````bash

abp new Acme.MyWpfApp -t wpf

````

`Acme.MyWpfApp` 是解决方案的名称, 如*\*YourCompany.YourProduct\**. 你可以使用单级或多级名称.

## 解决方案结构

使用以上命令创建解决方案后,你会得到如下所示的解决方案:

![basic-wpf-application-solution](../images/basic-wpf-application-solution.png)

\* `HelloWorldService` 是一个实现了 `ITransientDependency` 接口的示例服务. 它会自动注册到[依赖注入](../Dependency-Injection.md)系统.

## 空Web应用程序

# 在ASP.NET Core MVC Web 应用程序中使用ABP

本教程将介绍如何开始以最少的依赖关系开始使用ABP开发.

通常情况下你希望从 **\*\*[**启动模板**](Getting-Started-AspNetCore-MVC-Template.md)\*\*** 开始.

## 创建一个新项目

1. 使用Visual Studio 2022 (17.0.0+)创建一个新的ASP.NET Core Web应用程序:

![](images/create-new-aspnet-core-application-v2.png)

2. 配置新的项目:

![](images/select-empty-web-application-v2.png)

3. 完成创建:

![](images/create-aspnet-core-application.png)

## 安装 Volo.Abp.AspNetCore.Mvc 包

Volo.Abp.AspNetCore.Mvc是ABP集成ASP.NET Core MVC的包,请安装它到你项目中:

````

Install-Package Volo.Abp.AspNetCore.Mvc

````

## 创建ABP模块

ABP是一个模块化框架,它需要一个**\*\*启动(根)模块\*\***继承自 `AbpModule`:

````C#

using Microsoft.AspNetCore.Builder;

using Microsoft.Extensions.Hosting;

using Volo.Abp;

using Volo.Abp.AspNetCore.Mvc;

using Volo.Abp.Modularity;

namespace BasicAspNetCoreApplication

{

[DependsOn(typeof(AbpAspNetCoreMvcModule))]

public class AppModule : AbpModule

{

public override void OnApplicationInitialization(ApplicationInitializationContext context)

{

var app = context.GetApplicationBuilder();

var env = context.GetEnvironment();

// Configure the HTTP request pipeline.

if (env.IsDevelopment())

{

app.UseExceptionHandler("/Error");

// The default HSTS value is 30 days. You may want to change this for production scenarios, see https://aka.ms/aspnetcore-hsts.

app.UseHsts();

}

app.UseHttpsRedirection();

app.UseStaticFiles();

app.UseRouting();

app.UseConfiguredEndpoints();

}

}

}

````

``AppModule`` 是应用程序启动模块的好名称.

ABP的包定义了模块类,模块可以依赖其它模块.在上面的代码中 ``AppModule`` 依赖于 ``AbpAspNetCoreMvcModule`` (由[Volo.Abp.AspNetCore.Mvc](https://www.nuget.org/packages/Volo.Abp.AspNetCore.Mvc)包定义). 安装新的ABP的包后添加``DependsOn``特性是很常见的做法.

我们在此模块类中配置ASP.NET Core管道,而不是Startup类中.

### 启动类

接下来修改启动类集成到ABP模块系统:

````C#

using BasicAspNetCoreApplication;

var builder = WebApplication.CreateBuilder(args);

builder.Services.ReplaceConfiguration(builder.Configuration);

builder.Services.AddApplication<AppModule>();

var app = builder.Build();

app.InitializeApplication();

app.Run();

````

``services.AddApplication<AppModule>()``添加了从``AppModule``中启动的所有模块中定义的所有服务.

``app.InitializeApplication()`` 初始化并启动应用程序.

## 运行应用程序!

启动该应用,它将按预期运行.

## 使用 Autofac 依赖注入框架

虽然ASP.NET Core的依赖注入(DI)系统适用于基本要求,但[Autofac](https://autofac.org/)提供了属性注入和方法拦截等高级功能,这些功能是ABP执行高级应用程序框架功能所必需的.

用Autofac取代ASP.NET Core的DI系统并集成到ABP非常简单.

1. 安装 [Volo.Abp.Autofac](https://www.nuget.org/packages/Volo.Abp.Autofac) 包

````

Install-Package Volo.Abp.Autofac

````

2. 添加 ``AbpAutofacModule`` 依赖

````C#

[DependsOn(typeof(AbpAspNetCoreMvcModule))]

[DependsOn(typeof(AbpAutofacModule))] // 在模块上添加依赖AbpAutofacModule

public class AppModule : AbpModule

{

...

}

````

3. 修改``Program.cs``以使用Autofac:

````C#

using BasicAspNetCoreApplication;

var builder = WebApplication.CreateBuilder(args);

builder.Host.UseAutofac(); //Add this line

builder.Services.ReplaceConfiguration(builder.Configuration);

builder.Services.AddApplication<AppModule>();

var app = builder.Build();

app.InitializeApplication();

app.Run();

````

## 源码

从[此处](https://github.com/abpframework/abp-samples/tree/master/BasicAspNetCoreApplication)获取本教程中创建的示例项目的源代码.

# 启动模版

## 概述

# 启动模板

虽然你可以从一个空项目开始并手动添加所需的包,但启动模板可以非常轻松,舒适地使用ABP框架启动新的解决方案.

单击下面列表中的名称以查看相关启动模板的文档:

\* [\*\*app\*\*](Application.md): 应用程序模板.

\* [\*\*module\*\*](Module.md): 模块/服务模板.

\* [\*\*console\*\*](Console.md): 控制台模板.

\* [\*\*WPF\*\*](WPF.md): WPF模板.

\* [\*\*MAUI\*\*](MAUI.md): MAUI模板.

## 应用程序

# 应用程序启动模板

## 介绍

应用程序启动模板是基于[领域驱动设计](../Domain-Driven-Design.md)(DDD)分层的应用程序结构.

在这篇文档中详细介绍了**\*\*解决方案结构\*\***和项目,如果你想快速入门,请遵循以下指南:

\* 参阅[ASP.NET Core MVC 模板入门](../Getting-Started-With-Startup-Templates.md)创建此模板的新解决方案并运行它.

\* 参阅[ASP.NET Core MVC 教程](../Tutorials/AspNetCore-Mvc/Part-I.md)学习使用此模板开发应用程序.

\* [入门文档](../Getting-Started-AspNetCore-MVC-Template.md) 介绍了如何在几分钟内创建一个新的应用程序.

\* [应用程序开发教程](../Tutorials/AspNetCore-Mvc/Part-I.md) 学习使用此模板开发应用程序.

## 如何开始

你可以使用[ABP CLI](../CLI.md)创建基于此启动模板的新项目,或者你也可以在[入门](https://abp.io/get-started)页面创建并下载项目. 在这里我们使用CLI创建新项目.

如果未安装ABP CLI,第一步是安装ABP CLI

````bash

dotnet tool install -g Volo.Abp.Cli

````

然后使用 `abp new` 命令在空文件夹中创建新解决方案:

````bash

abp new Acme.BookStore -t app

````

\* `Acme.BookStore` 是解决方案的名称, 如*\*YourCompany.YourProduct\**. 你可以使用单级或多级名称.

\* 示例中指定了启动模板 (`-t` 或 `--template` 选项). 示指定模板时,默认模板是`app` .

### 指定UI框架

模板提供了多个UI框架

\* `mvc`: ASP.NET Core MVC Razor页面 (默认)

\* `angular`: Angular UI

使用 `-u` 或 `--ui` 选择指定UI框架:

````bash

abp new Acme.BookStore -u angular

````

### 指定数据库提供程序

`MVC`模板支持以下数据库提供程序:

- `ef`: Entity Framework Core (默认)

- `mongodb`: MongoDB

使用 `-d` (或 `--database-provider`) 选项指定数据库提供程序:

````bash

abp new Acme.BookStore -d mongodb

````

### 指定移动应用程序框架

该模板支持以下移动应用程序框架:

- `react-native`: React Native

使用 `-m` (or `--mobile`) 选项来指定移动应用程序框架:

````bash

abp new Acme.BookStore -m react-native

````

如果未指定, 则不会创建任何移动应用程序.

## 解决方案结构

根据命令的选项,会创建略有不同的解决方案结构.

### 默认结构

如果未指定附加选项,你会得到如下所示的解决方案:

![bookstore-visual-studio-solution-v3](../images/bookstore-visual-studio-solution-v3.png)

项目组织在`src`和`test`文件夹中. `src`文件夹包含实际应用程序,该应用程序基于前面提到的[DDD](../Domain-Driven-Design.md)原则进行分层. 下图展示了解决方案的层和项目的依赖关系:

![layered-project-dependencies](../images/layered-project-dependencies.png)

下面介绍解决方案中的项目及依赖关系.

#### .Domain.Shared 项目

项目包含常量,枚举和其他对象,这些对象实际上是领域层的一部分,但是解决方案中所有的层/项目中都会使用到.

例如 `BookType` 枚举和 `BookConsts` 类 (可能是 `Book` 实体用到的常数字段,像`MaxNameLength`)都适合放在这个项目中.

\* 该项目不依赖解决方案中的其他项目. 其他项目直接或间接依赖该项目

#### .Domain 项目

解决方案的领域层. 它主要包含 [实体, 集合根](../Entities.md), [领域服务](../Domain-Services.md), [值类型](../Value-Types.md), [仓储接口](../Repositories.md) 和解决方案的其他领域对象.

例如 `Book` 实体和 `IBookRepository` 接口都适合放在这个项目中.

\* 它依赖 `.Domain.Shared` 项目,因为项目中会用到它的一些常量,枚举和定义其他对象.

#### .Application.Contracts 项目

项目主要包含 [应用服务](../Application-Services.md) **\*\*interfaces\*\*** 和应用层的 [数据传输对象](../Data-Transfer-Objects.md) (DTO). 它用于分离应用层的接口和实现. 这种方式可以将接口项目做为约定包共享给客户端.

例如 `IBookAppService` 接口和 `BookCreationDto` 类都适合放在这个项目中.

\* 它依赖 `.Domain.Shared` 因为它可能会在应用接口和DTO中使用常量,枚举和其他的共享对象.

#### .Application 项目

项目包含 `.Application.Contracts` 项目的 [应用服务](../Application-Services.md) 接口**\*\*实现\*\***.

例如 `BookAppService` 类适合放在这个项目中.

\* 它依赖 `.Application.Contracts` 项目, 因为它需要实现接口与使用DTO.

\* 它依赖 `.Domain` 项目,因为它需要使用领域对象(实体,仓储接口等)执行应用程序逻辑.

#### .EntityFrameworkCore 项目

这是集成EF Core的项目. 它定义了 `DbContext` 并实现 `.Domain` 项目中定义的仓储接口.

\* 它依赖 `.Domain` 项目,因为它需要引用实体和仓储接口.

> 只有在你使用了EF Core做为数据库提供程序时,此项目才会可用. 如果选择的是其他数据库提供程序那么项目的名称会改变

#### .EntityFrameworkCore.DbMigrations 项目

包含解决方案的EF Core数据库迁移. 它有独立的 `DbContext` 来专门管理迁移.

ABP是一个模块化的框架,理想的设计是让每个模块都有自己的 `DbContext` 类. 这时用于迁移的 `DbContext` 就会发挥作用. 它将所有的 `DbContext` 配置统一到单个模型中以维护单个数据库的模式. 对于更高级的场景,可以程序可以拥有多个数据库(每个数据库有一个或多个模块表)和多个迁移`DbContext`(每个都维护不同的数据库模式)

需要注意,迁移 `DbContext` 仅用于数据库迁移,而不在*\*运行时\**使用.

\* 它依赖 `.EntityFrameworkCore` 项目,因为它重用了应用程序的 `DbContext` 配置 .

> 只有在你使用了EF Core做为数据库提供程序时,此项目才会可用.

> 参阅[Entity Framework Core迁移指南](../Entity-Framework-Core-Migrations.md)了解这个项目的详细信息.

#### .DbMigrator 项目

这是一个控制台应用程序,它简化了在开发和生产环境执行数据库迁移的操作.当你使用它时;

\* 必要时创建数据库(没有数据库时).

\* 应用未迁移的数据库迁移.

\* 初始化种子数据(当你需要时).

> 这个项目有自己的 `appsettings.json` 文件. 所以如果要更改数据库连接字符串,请记得也要更改此文件.

初始化种子数据很重要,ABP具有模块化的种子数据基础设施. 种子数据的更多信息,请参阅[文档](../Data-Seeding.md).

虽然创建数据库和应用迁移似乎只对关系数据库有用,但即使你选择NoSQL数据库提供程序(如MongoDB),也会生成此项目. 这时,它会为应用程序提供必要的初始数据.

\* 它依赖 `.EntityFrameworkCore.DbMigrations` 项目 (针对EF Core),因为它需要访问迁移文件.

\* 它依赖 `.Application.Contracts` 项目,因为它需要访问权限定义在初始化种子数据时为管理员用户赋予所有权限.

#### .HttpApi 项目

用于定义API控制器.

大多数情况下,你不需要手动定义API控制器,因为ABP的[动态API](../API/Auto-API-Controllers.md)功能会根据你的应用层自动创建API控制器. 但是,如果你需要编写API控制器,那么它是最合适的地方.

\* 它依赖 `.Application.Contracts` 项目,因为它需要注入应用服务接口.

#### .HttpApi.Client 项目

定义C#客户端代理使用解决方案的HTTP API项目. 可以将上编辑共享给第三方客户端,使其轻松的在DotNet应用程序中使用你的HTTP API(其他类型的应用程序可以手动或使用其平台的工具来使用你的API).

ABP有[动态 C# API 客户端](../API/Dynamic-CSharp-API-Clients.md)功能,所以大多数情况下你不需要手动的创建C#客户端代理.

`.HttpApi.Client.ConsoleTestApp` 项目是一个用于演示客户端代理用法的控制台应用程序.

\* 它依赖 `.Application.Contracts` 项目,因为它需要使用应用服务接口和DTO.

> 如果你不需要为API创建动态C#客户端代理,可以删除此项目和依赖项

#### .Web 项目

包含应用程序的用户界面(UI).如果使用ASP.NET Core MVC UI, 它包括Razor页面,javascript文件,样式文件,图片等...

包含应用程序主要的 `appsettings.json` 配置文件,用于配置数据库连接字符串和应用程序的其他配置

\* 依赖 `.HttpApi` 项目,因为UI层需要使用解决方案的API和应用服务接口.

> 如果查看 `.Web.csproj` 源码, 你会看到对 `.Application` 和 `.EntityFrameworkCore.DbMigrations` 项目的引用.

>

> 在编写UI层时实际上不需要这些引用. 因为UI层通常不依赖于EF Core或应用层的实现. 这个启动模板已经为分层部署做好了准备,API层托管在不同与UI层的服务器中.

>

> 但是如果你不选择 `--tiered` 选项, .Web项目会有这些引用,以便能够将Web,Api和应用层托管在单个应用程序站点.

>

> 你可以在表示层中使用领域实体和仓储,但是根据DDD的理论,这被认为是一种不好的做法.

#### Test 项目

解决方案有多个测试项目,每一层都会有一个:

\* `.Domain.Tests` 用于测试领域层.

\* `.Application.Tests` 用于测试应用层.

\* `.EntityFrameworkCore.Tests` 用于测试EF Core配置与自定义仓储.

\* `.Web.Tests` 用于测试UI(适用于ASP.NET Core MVC UI).

\* `.TestBase` 所有测试项目的基础(共享)项目.

此外, `.HttpApi.Client.ConsoleTestApp` 是一个控制台应用程序(不是自动化测试项目),它用于演示.Net应用程序中HTTP API的用法.

测试项目是用于做集成测试的:

\* 它完全集成到ABP框架和应用程序的所有服务.

\* 如果数据库提供程序是EF Core,测试项目会使用SQLite内存数据库,如果是MongoDB,它使用[Mongo2Go](https://github.com/Mongo2Go/Mongo2Go)库.

\* 授权被禁用,任何的应用服务都可以在测试中轻松调用.

你依然可以编写单元测试,只不过它很难写(因为你需要准备mock/fake对象),但它的运行速度更快(因为只测试单个类并跳过所有初始化过程).

#### 如何运行?

设置`.Web`为启动项目. 默认用户名 `admin`, 密码 `1q2w3E\*`.

更多信息请参阅[ASP.NET Core MVC 模板入门](../Getting-Started-AspNetCore-MVC-Template.md).

### 分层结构

如果你选择了ASP.NET Core UI并指定了 `--tiered` 选项,CLI会创建分层解决方案. 分层结构的目的是**\*\*将Web应用程序和HTTP API部署到不同的服务器\*\***:

![tiered-solution-servers](../images/tiered-solution-servers.png)

\* 浏览器渲染HTML,执行CSS和JavaScript来运行UI.

\* Web服务器托管静态文件(CSS,JavaScript,图片...等)和动态组件(如Razor页面),它通过HTTP请求到API服务器执行应用程序的业务逻辑.

\* API服务器托管HTTP API,使用应用程序的应用层和领域层执行业务逻辑.

\* 最后数据库服务器托管数据库.

与之前默认结构的三层部署比较,分层解决方案允许四层部署.

> 除非你真的需要四层部署,一般建议采用默认结构,它更易于开发,部署和维护.

解决方案结构如下所示:

![bookstore-visual-studio-solution-tiered](../images/bookstore-visual-studio-solution-tiered.png)

与默认结构不同,我们得到了两个新项目: `.AuthServer` 和 `.HttpApi.Host`.

#### .AuthServer 项目

用于其他项目的身份验证服务器. `.Web`项目使用OpenId AuthServer. 然后使用访问令牌调用HTTP API服务器. HTTP API服务器使用bearer token从访问令牌获取声明授权当前用户.

![tiered-solution-applications](../images/tiered-solution-applications.png)

ABP使用开源的[IdentityServer4](https://identityserver.io/) 和 [OpenIddict](https://github.com/openiddict/) 框架做应用程序间的身份验证. 有关IdentityServer4,OpenIddict和OpenId Connect协议的详细信息请参阅[IdentityServer4文档](http://docs.identityserver.io) [OpenIddict文档](https://documentation.openiddict.com/).

它有自己的`appsettings.json`文件(数据库连接字符串等其他配置).

#### .HttpApi.Host 项目

该项目是一个承载解决方案API的应用程序. 它有自己的`appsettings.json`文件(数据库连接字符串等其他配置).

#### .Web 项目

与默认结构一样,包含应用程序的用户界面(UI). 包括Razor页面,javascript文件,样式文件,图片等...

项目包含`appsetting.json`文件,但没有连接字符串配置, 它不需要连接到数据库. 文件中主要包含远程API服务器端点和身份验证服务器

#### 前置条件

\* [Redis](https://redis.io/): 应用程序使用Redis做分布式缓存,你需要安装并运行Redis.

#### 如何运行?

你应该按照以下顺序运行应用:

\* 首先运行`.AuthServer`,因为其他应用程序依赖它做身份验证.

\* 然后运行`.HttpApi.Host`,因为`.Web`应用程序需要访问HTTI API.

\* 最后运行`.Web`并登录到应用程序(用户名: `admin` 密码: `1q2w3E\*`).

### Angular UI

如果你选择Angular做为UI框架(使用 `-u angular` 选择), 解决方案会被分成两个文件夹:

\* `angular` 是客户端部分,文件夹中包含了Angular UI 解决方案.

\* `aspnet-core` 是服务端部分,文件夹中包含了ASP.NET Core解决方案.

服务端部分与上面的描述的解决方案非常相似. `.HttpApi.Host` 项目提供API接口, Angular应用程序使用它提供的接口.

`angular/src/environments` 文件夹下的文件含有应用程序的基础配置.

#### AppModule（应用程序模块）

`AppModule` 是应用程序的根模块. 一些ABP模块和一些基本模块导入到 `AppModule` 中.

ABP 配置模块也已经导入到 `AppModule` 中, 以满足可延迟加载 ABP 模块的初始需求.

#### AppRoutingModule（应用程序路由模块）

在 `AppRoutingModule` 中有可延迟加载的 ABP 模块作为路由.

> 不应更改ABP模块的路径.

你应该在 `data` 对象中添加 `routes` 属性, 以便在菜单中添加一个链接来重定向到自定义页面.

```js

{

path: 'dashboard',

loadChildren: () => import('./dashboard/dashboard.module').then(m => m.DashboardModule),

canActivate: [AuthGuard, PermissionGuard],

data: {

routes: {

name: 'ProjectName::Menu:Dashboard',

order: 2,

iconClass: 'fa fa-dashboard',

requiredPolicy: 'ProjectName.Dashboard.Host'

} as ABP.Route

}

}

```

在上面的例子中;

\* 如果用户没有登录, AuthGuard 会阻塞访问并重定向到登录页面.

\* PermissionGuard 使用 `rotues` 对象的 `requiredPolicy` 属性检查用户的权限. 如果用户未被授权访问该页, 则显示403页.

\* `routes` 的 `name` 属性是菜单链接标签. 可以定义本地化 key.

\* `routes` 对象的 `iconClass` 属性是菜单链接图标类.

\* `routes` 对象的 `requiredPolicy` 属性是访问页面所需的策略 key.

在上述 `routes` 定义之后, 如果用户被授权, 仪表盘链接将出现在菜单上.

#### Shared Module（共享模块）

所有模块可能需要的模块已导入到 `SharedModule`. 你应该将 `SharedModule` 导入所有模块.

参见 [Sharing Modules（共享模块）](https://angular.io/guide/sharing-ngmodules) 文档.

#### Environments（环境）

`src/environments` 文件夹下的文件包含应用程序的基本配置.

#### Home Module

Home模块是一个可延迟加载的模块, 它加载应用程序的根地址.

#### Styles（样式）

在 `angular.json` 中向 `styles` 数组添加所需的样式文件. `AppComponent` 在主包加载后通过 `LazyLoadService` 加载一些样式文件, 以缩短第一次绘制的时间.

#### Testing（测试）

你应该在与要测试的文件相同的文件夹中创建测试.

参见[测试文档](https://angular.io/guide/testing/).

#### Depended Packages（依赖包）

\* [NG Bootstrap](https://ng-bootstrap.github.io/) 被用作UI组件库.

\* [NGXS](https://www.ngxs.io/) 被用作状态管理库.

\* [angular-oauth2-oidc](https://github.com/manfredsteyer/angular-oauth2-oidc) 用于支持OAuth 2和OpenId Connect (OIDC).

\* [Chart.js](https://www.chartjs.org/) 用于创建小部件.

\* [ngx-validate](https://github.com/ng-turkey/ngx-validate) 用于对交互表单进行动态验证.

### React Native

如果使用 `-m react-native` 选项解决方案将[React Native](https://reactnative.dev/)应用程序作为默认值包含在 `react-native` 文件夹中.

服务器端类似于上面描述的解决方案. `\*.HttpApi.Host` 的项目提供 API, 所以 React 本机应用程序使用它.

React 本机应用程序是用 [Expo](https://expo.io/)生成的. Expo 是一套基于 React Native 构建的工具, 帮助你快速启动一个应用程序, 尽管它有很多功能.

React Native 应用文件夹结构, 如下图所示:

![react-native-folder-structure](../images/react-native-folder-structure.png)

\* `App.js` 是应用程序的引导组件.

\* `Environment.js` f文件有应用程序的基本配置. 在这个文件中定义了 `prod` and `dev` 配置.

\* [Contexts](https://reactjs.org/docs/context.html) 是在 `src/contexts` 文件夹中创建的.

\* [Higher order components](https://reactjs.org/docs/higher-order-components.html) 是在 `src/hocs` 文件夹中创建的.

\* [Custom hooks](https://reactjs.org/docs/hooks-custom.html#extracting-a-custom-hook) 是在 `src/hooks` 中创建的.

\* [Axios interceptors](https://github.com/axios/axios#interceptors) 是在 `src/interceptors` 文件夹中创建.

\* 工具函数从 `src/utils` 文件夹导出.

#### Components（组件）

可以在所有屏幕上使用的组件是在 `src/components` 文件夹中创建的. 所有组件都是作为一个能够使用 [hooks](https://reactjs.org/docs/hooks-intro.html) 的函数创建的.

#### Screens（屏幕）

![react-native-navigation-structure](../images/react-native-navigation-structure.png)

Screens 是通过在 `src/screens` 文件夹中创建将名称分开的文件夹来创建的. 某些 screens 的某些部分可以拆分为组件.

每个 screen 都在 `src/navigators` 文件夹中的导航器中使用.

#### Navigation（导航）

[React Navigation](https://reactnavigation.org/) 被用作导航库. 导航器是在 `src/navigators` 中创建的. 一个 [drawer](https://reactnavigation.org/docs/drawer-based-navigation/) 导航器和几个 [stack](https://reactnavigation.org/docs/hello-react-navigation/#installing-the-stack-navigator-library) 导航器在此文件夹中创建. 查看 [上图](#screens) 中的导航结构.

#### State Management（状态管理）

[Redux](https://redux.js.org/) 被用作状态管理库. [Redux Toolkit](https://redux-toolkit.js.org/) 库被用作高效Redux开发的工具集.

在 `src/store` 文件夹中创建 Actions, reducers, sagas, selectors. 存储文件夹如下:

![react-native-store-folder](../images/react-native-store-folder.png)

\* [\*\*Store\*\*](https://redux.js.org/basics/store) 在 `src/store/index.js` 文件中定义.

\* [\*\*Actions\*\*](https://redux.js.org/basics/actions/) 是将数据从应用程序发送到存储的有效信息负载.

\* [\*\*Reducers\*\*](https://redux.js.org/basics/reducers) 指定应用程序的状态如何更改以响应发送到存储的操作.

\* [\*\*Redux-Saga\*\*](https://redux-saga.js.org/) 是一个库, 旨在使应用程序的副作用(即异步的事情, 如数据获取和不纯的事情, 如访问浏览器缓存)更容易管理. Sagas 是在 `src/store/sagas` 文件夹中创建的.

\* [\*\*Reselect\*\*](https://github.com/reduxjs/reselect) 库用于创建缓存的选择器. 选择器是在 `src/store/selectors` 文件夹中创建的.

#### APIs

[Axios](https://github.com/axios/axios) 用作HTTP客户端库. Axios 实例从 `src/api/API.js` 导出 . 使用相同的配置进行HTTP调用. `src/api` 文件夹中还有为 API 调用创建的 API 文件.

#### Theming（主题）

[Native Base](https://nativebase.io/) 被用作UI组件库. 本地基本组件可以很容易地进行自定义.参见[Native Base customize](https://docs.nativebase.io/Customize.html#Customize) 文档.我们沿着同样的路走.

\* Native Base 主题变量在 `src/theme/variables` 文件夹中.

\* Native Base 组件样式在 `src/theme/components` 文件夹中.这些文件是用 Native Base's `ejectTheme` 脚本生成的.

\* 组件样式用 `src/theme/overrides` 文件夹下的文件覆盖.

#### Testing（单元测试）

将创建单元测试.

参见[测试概述](https://reactjs.org/docs/testing.html)文档.

#### Depended Libraries（依赖库）

\* [Native Base](https://nativebase.io/) 用作UI组件库.

\* [React Navigation](https://reactnavigation.org/) 用作导航库.

\* [Axios](https://github.com/axios/axios) 用作HTTP客户端库.

\* [Redux](https://redux.js.org/) 用作状态管理库.

\* [Redux Toolkit](https://redux-toolkit.js.org/) 库被用作高效Redux开发的工具集.

\* [Redux-Saga](https://redux-saga.js.org/) 用于管理异步进程.

\* [Redux Persist](https://github.com/rt2zz/redux-persist) 被用作状态持久化.

\* [Reselect](https://github.com/reduxjs/reselect) 用于创建缓存的选择器.

\* [i18n-js](https://github.com/fnando/i18n-js) 作为国际化库使用.

\* [expo-font](https://docs.expo.io/versions/latest/sdk/font/) 库可以轻松加载字体.

\* [Formik](https://github.com/jaredpalmer/formik) 用于构建表单.

\* [Yup](https://github.com/jquense/yup) 用于表单验证.

## 社交/外部登录

如果你想要为你的应用程序配置社交/外部登录,请参阅[社交/外部登录](../Authentication/Social-External-Logins.md)文档.

## 下一步是什么?

\* [入门文档](../Getting-Started.md)介绍了如何在几分钟内创建新应用程序.

\* [Web应用程序开发教程](../Tutorials/Part-1.md)逐步介绍了应用程序开发.

## 模块

# MVC模块启动模板

可用此模板开发基于[模块开发最佳实践和约定](../Best-Practices/Index.md)的可**\*\*复用 [**应用程序模块**](../Modules/Index.md)\*\*** . 它同样适用于开发**\*\*微服务\*\***.

## 如何开始?

你可以使用[ABP CLI](../CLI.md)创建基于此启动模板的新项目,或者你也可以在[入门](https://abp.io/get-started)页面创建并下载项目. 在这里我们使用CLI创建新项目.

如果未安装ABP CLI,第一步是安装ABP CLI

````bash

dotnet tool install -g Volo.Abp.Cli

````

然后使用 `abp new` 命令在空文件夹中创建新解决方案:

````bash

abp new Acme.IssueManagement -t module

````

\* `Acme.IssueManagement` 是解决方案的名称, 如*\*YourCompany.YourProduct\**. 你可以使用单级或多级名称.

### 无用户界面

默认情况模板附带一个MVC UI,你可以使用`--no-ui`选项指定不包含UI层.

````bash

abp new Acme.IssueManagement -t module --no-ui

````

## 解决方案结构

根据命令的选项,会创建略有不同的解决方案结构. 如果未指定选项,你会得到如下所示的解决方案:

![issuemanagement-module-solution](../images/issuemanagement-module-solution.png)

项目组织在`src`和`test`和`host`文件夹中:

\* `src` 文件夹包含基于[DDD](../Domain-Driven-Design.md)原则分层的实际模块.

\* `test` 文件夹包含单元和集成测试.

\* `host` 文件夹包含具有不同配置的应用程序,用于演示在应用程序中如何托管模块. 这些不是模块的一部分,但开发时会很有用.

下图展示了模块的层和项目的依赖关系:

![layered-project-dependencies-module](../images/layered-project-dependencies-module.png)

下面介绍解决方案中的项目及依赖关系:

### .Domain.Shared 项目

项目包含常量,枚举和其他对象,这些对象实际上是领域层的一部分,但是解决方案中所有的层/项目中都会使用到.

例如 `IssueType` 枚举和 `IssueConsts` 类 (可能是 `Issue` 实体用到的常数字段,像`MaxTitleLength`)都适合放在这个项目中.

\* 该项目不依赖解决方案中的其他项目. 其他项目直接或间接依赖该项目.

### .Domain 项目

解决方案的领域层. 它主要包含 [实体, 集合根](../Entities.md), [领域服务](../Domain-Services.md), 值类型, [仓储接口](../Repositories.md) 和解决方案的其他领域对象.

例如 `Issue` 实体, `IssueManager` 领域服务和 `IIssueRepository` 接口都适合放在这个项目中.

\* 它依赖 `.Domain.Shared` 项目,因为项目中会用到它的一些常量,枚举和定义其他对象.

### .Application.Contracts 项目

项目主要包含 [应用服务](../Application-Services.md) **\*\*interfaces\*\*** 和应用层的 [数据传输对象](../Data-Transfer-Objects.md) (DTO). 它用于分离应用层的接口和实现. 这种方式可以将接口项目做为约定包共享给客户端.

例如 `IIssueAppService` 接口和 `IssueCreationDto` 类都适合放在这个项目中.

\* 它依赖 `.Domain.Shared` 因为它可能会在应用接口和DTO中使用常量,枚举和其他的共享对象.

### .Application 项目

项目包含 `.Application.Contracts` 项目的 [应用服务](../Application-Services.md) 接口**\*\*实现\*\***.

例如 `IssueAppService` 类适合放在这个项目中.

\* 它依赖 `.Application.Contracts` 项目, 因为它需要实现接口与使用DTO.

\* 它依赖 `.Domain` 项目,因为它需要使用领域对象(实体,仓储接口等)执行应用程序逻辑.

### .EntityFrameworkCore 项目

这是集成EF Core的项目. 它定义了 `DbContext` 并实现 `.Domain` 项目中定义的仓储接口.

\* 它依赖 `.Domain` 项目,因为它需要引用实体和仓储接口.

> 如果不需要使用EF Core,可以删除此项目.

### .MongoDB 项目

这是集成MongoDB的项目.

\* 它依赖 `.Domain` 项目,因为它需要引用实体和仓储接口.

> 如果不需要使用MongoDB,可以删除此项目.

#### Test 项目

解决方案有多个测试项目,每一层都会有一个:

\* `.Domain.Tests` 用于测试领域层.

\* `.Application.Tests` 用于测试应用层.

\* `.EntityFrameworkCore.Tests` 用于测试EF Core配置与自定义仓储.

\* `.MongoDB.Tests` 用于测试MongoDB配置与自定义仓储.

\* `.TestBase` 所有测试项目的基础(共享)项目.

此外, `.HttpApi.Client.ConsoleTestApp` 是一个控制台应用程序(不是自动化测试项目),它用于演示DotNet应用程序中HTTP API的用法.

测试项目是用于做集成测试的:

\* 它完全集成到ABP框架和应用程序的所有服务.

\* 如果数据库提供程序是EF Core,测试项目会使用SQLite内存数据库,如果是MongoDB,它使用[Mongo2Go](https://github.com/Mongo2Go/Mongo2Go)库.

\* 授权被禁用,任何的应用服务都可以在测试中轻松调用.

你依然可以编写单元测试,只不过它很难写(因为你需要准备mock/fake对象),但它的运行速度更快(因为只测试单个类并跳过所有初始化过程).

> 领域层和应用层测试使用EF Core. 如果你删除了EF Core集成项目或者要使用MongoDB测试这些层,需要手动的修改项目引用和依赖.

### Host 项目

解决方案中有一些宿主应用程序可以在开发时运行模块. 在开发时会很有用. 除了正在开发的模块外,宿主应用程序包括其他模块:

宿主应用程序支持两种类型的场景.

#### 单体 (统一) 应用场景

如果模块含有UI, 那么 `.Web.Unified` 应用程序于在一个站点上托管UI和API. 它有自己的 `appsettings.json` 配置文件(包括数据库连接字符串)和EF Core数据库迁移.

`.Web.Unified` 应用程序有一个名为`YourProjectName\_Unified`的数据库(本例中名为 *\*IssueManagement\_Unified\**).

> 如果你选择了 `--no-ui` 选项, 这个项目不会出现在你的解决方案中.

##### 如何运行?

将`host/YourProjectName.Web.Unified`设置成启动项, 使用包管理控制台运行 `Update-Database` 命令迁移数据库,然后运行应用程序. 默认用户名: `admin` 密码: `1q2w3E\*`.

#### 分离部署&数据库 场景

在这个场景中,有三个应用程序;

\* `.AuthServer` 应用程序是其他应用程序使用的身份验证服务器,它有自己的 `appsettings.json` 包含数据库连接字符串和其他配置.

\* `.HttpApi.Host` 托管模块的HTTP API. 它有自己的 `appsettings.json` 包含数据库连接字符串和其他配置.

\* `.Web.Host` 托管模块的UI. 它包含 `appsettings.json` 文件, 但是其中没有数据库连接字符串, 因为它不需要连接到数据库, 它主要调用远程API服务器和身份认证服务器.

下图展示了应用程序的关系:

![tiered-solution-applications](../images/tiered-solution-applications.png)

`.Web.Host` 项目使用OpenId Connect身份认证从`.AuthServer`获取当前用户的身份和访问令牌. 然后使用访问令牌调用 `.HttpApi.Host`. HTTP API 服务器使用bearer token验证访问令牌获取当前用户声明并授权用户.

##### 前置条件

\* [Redis](https://redis.io/): 应用程序使用Redis做分布式缓存,你需要安装并运行Redis.

##### 如何运行?

你需要按照以下顺序运行应用程序:

- 首先, 运行 `.AuthServer`,因为其他应用程序依赖它做身份认证.

- 然后运行 `.HttpApi.Host`,因为`.Web.Host`应用程序使用API接口.

- 最后运行 `.Web.Host` 使用用户名: `admin` 密码: `1q2w3E\*` 登录到应用程序.

## 控制台

# 控制台应用程序启动模板

此模板用于创建一个最小的依赖关系的ABP控制台应用程序项目.

## 如何开始?

首先,如果你没有安装[ABP CLI](../CLI.md),请先安装它:

````bash

dotnet tool install -g Volo.Abp.Cli

````

在一个空文件夹使用 `abp new` 命令创建新解决方案:

````bash

abp new Acme.MyConsoleApp -t console

````

`Acme.MyConsoleApp` 是解决方案的名称, 如*\*YourCompany.YourProduct\**. 你可以使用单级或多级名称.

## 解决方案结构

使用以上命令创建解决方案后,你会得到如下所示的解决方案:

![basic-console-application-solution](../images/basic-console-application-solution.png)

\* `HelloWorldService` 是一个实现了 `ITransientDependency` 接口的示例服务. 它会自动注册到[依赖注入](../Dependency-Injection.md)系统.

## WPF

# WPF应用程序启动模板

此模板用于创建一个最小的依赖关系的ABP WPF应用程序项目.

## 如何开始?

首先,如果你没有安装[ABP CLI](../CLI.md),请先安装它:

````bash

dotnet tool install -g Volo.Abp.Cli

````

在一个空文件夹使用 `abp new` 命令创建新解决方案:

````bash

abp new Acme.MyWpfApp -t wpf

````

`Acme.MyWpfApp` 是解决方案的名称, 如*\*YourCompany.YourProduct\**. 你可以使用单级或多级名称.

## 解决方案结构

使用以上命令创建解决方案后,你会得到如下所示的解决方案:

![basic-wpf-application-solution](../images/basic-wpf-application-solution.png)

\* `HelloWorldService` 是一个实现了 `ITransientDependency` 接口的示例服务. 它会自动注册到[依赖注入](../Dependency-Injection.md)系统.

## MAUI

# MAUI应用程序启动模板

此模板用于创建一个最小的依赖关系的ABP MAUI应用程序项目.

## 如何开始?

首先,如果你没有安装[ABP CLI](../CLI.md),请先安装它:

````bash

dotnet tool install -g Volo.Abp.Cli

````

在一个空文件夹使用 `abp new` 命令创建新解决方案:

````bash

abp new Acme.MyMauiApp -t maui

````

`Acme.MyMauiApp` 是解决方案的名称, 如*\*YourCompany.YourProduct\**. 你可以使用单级或多级名称.

## 解决方案结构

使用以上命令创建解决方案后,你会得到如下所示的解决方案:

![basic-maui-application-solution](../images/basic-maui-application-solution.png)

\* `HelloWorldService` 是一个实现了 `ITransientDependency` 接口的示例服务. 它会自动注册到[依赖注入](../Dependency-Injection.md)系统.

# 教程

## 应用开发

创建应用服务端

# Web应用程序开发教程 - 第一章: 创建服务端

````json

//[doc-params]

{

"UI": ["MVC","Blazor","BlazorServer","NG"],

"DB": ["EF","Mongo"]

}

````

## 关于本教程

在本系列教程中, 你将构建一个名为 `Acme.BookStore` 的用于管理书籍及其作者列表的基于ABP的应用程序. 它是使用以下技术开发的:

\* **\*\*{{DB\_Value}}\*\*** 做为数据库提供程序.

\* **\*\*{{UI\_Value}}\*\*** 做为UI框架.

本教程分为以下部分:

- **\*\*Part 1: 创建服务端 (本章)\*\***

- [Part 2: 图书列表页面](Part-2.md)

- [Part 3: 创建,更新和删除图书](Part-3.md)

- [Part 4: 集成测试](Part-4.md)

- [Part 5: 授权](Part-5.md)

- [Part 6: 作者: 领域层](Part-6.md)

- [Part 7: 作者: 数据库集成](Part-7.md)

- [Part 8: 作者: 应用服务层](Part-8.md)

- [Part 9: 作者: 用户界面](Part-9.md)

- [Part 10: 图书到作者的关系](Part-10.md)

## 下载源码

本教程根据你的**\*\*UI\*\*** 和 **\*\*数据库\*\***偏好有多个版本,我们准备了几种可供下载的源码组合:

\* [MVC (Razor Pages) UI 与 EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Mvc-EfCore)

\* [Blazor UI 与 EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Blazor-EfCore)

\* [Angular UI 与 MongoDB](https://github.com/abpframework/abp-samples/tree/master/BookStore-Angular-MongoDb)

> 如果你在Windows中遇到 "文件名太长" or "解压错误", 很可能与Windows最大文件路径限制有关. Windows文件路径的最大长度为250字符. 为了解决这个问题,参阅 [在Windows 10中启用长路径](https://docs.microsoft.com/en-us/windows/win32/fileio/maximum-file-path-limitation?tabs=cmd#enable-long-paths-in-windows-10-version-1607-and-later).

> 如果你遇到与Git相关的长路径错误, 尝试使用下面的命令在Windows中启用长路径. 参阅 https://github.com/msysgit/msysgit/wiki/Git-cannot-create-a-file-or-directory-with-a-long-path

> `git config --system core.longpaths true`

{{if UI == "MVC" && DB == "EF"}}

### 视频教程

本章也被录制为视频教程 **\*\***<a href="https://www.youtube.com/watch?v=cJzyIFfAlp8&list=PLsNclT2aHJcPNaCf7Io3DbMN6yAk\_DgWJ&index=1" target="\_blank">**发布在YouTube**</a>**\*\***.

{{end}}

## 创建解决方案

在开始开发之前,请按照[入门教程](../Getting-Started.md)创建名为 `Acme.BookStore` 的新解决方案.

## 创建Book实体

启动模板中的**\*\*领域层\*\***分为两个项目:

- `Acme.BookStore.Domain`包含你的[实体](../Entities.md), [领域服务](../Domain-Services.md)和其他核心域对象.

- `Acme.BookStore.Domain.Shared`包含可与客户共享的常量,枚举或其他域相关对象.

在解决方案的**\*\*领域层\*\***(`Acme.BookStore.Domain`项目)中定义你的实体.

该应用程序的主要实体是`Book`. 在`Acme.BookStore.Domain`项目中创建一个 `Books` 文件夹(命名空间),并在其中添加名为 `Book` 的类,如下所示:

````csharp

using System;

using Volo.Abp.Domain.Entities.Auditing;

namespace Acme.BookStore.Books

{

public class Book : AuditedAggregateRoot<Guid>

{

public string Name { get; set; }

public BookType Type { get; set; }

public DateTime PublishDate { get; set; }

public float Price { get; set; }

}

}

````

\* ABP为实体提供了两个基本的基类: `AggregateRoot`和`Entity`. **\*\*Aggregate Root\*\***是[\*\*领域驱动设计\*\*](../Domain-Driven-Design.md) 概念之一. 可以视为直接查询和处理的根实体(请参阅[实体文档](../Entities.md)).

\* `Book`实体继承了`AuditedAggregateRoot`,`AuditedAggregateRoot`类在`AggregateRoot`类的基础上添加了一些基础[审计](../Audit-Logging.md)属性(例如`CreationTime`, `CreatorId`, `LastModificationTime` 等). ABP框架自动为你管理这些属性.

\* `Guid`是`Book`实体的**\*\*主键类型\*\***.

> 为了保持简单,本教程将实体属性保留为 **\*\*public get/set\*\*** . 如果你想了解关于DDD最佳实践,请参阅[实体文档](../Entities.md).

### BookType枚举

`Book`实体使用了`BookType`枚举. 在`Acme.BookStore.Domain.Shared`项目中创建`Books`文件夹(命名空间),并在其中添加`BookType`:

````csharp

namespace Acme.BookStore.Books

{

public enum BookType

{

Undefined,

Adventure,

Biography,

Dystopia,

Fantastic,

Horror,

Science,

ScienceFiction,

Poetry

}

}

````

最终的文件夹/文件结构应该如下所示:

![bookstore-book-and-booktype](images/bookstore-book-and-booktype.png)

### 将Book实体添加到DbContext中

{{if DB == "EF"}}

EF Core需要你将实体和 `DbContext` 建立关联.最简单的做法是在`Acme.BookStore.EntityFrameworkCore`项目的`BookStoreDbContext`类中添加`DbSet`属性.如下所示:

````csharp

public class BookStoreDbContext : AbpDbContext<BookStoreDbContext>

{

public DbSet<Book> Books { get; set; }

//...

}

````

{{end}}

{{if DB == "Mongo"}}

添加 `IMongoCollection<Book> Book` 属性到 `Acme.BookStore.MongoDB` 项目的 `BookStoreMongoDbContext` 中.

```csharp

public class BookStoreMongoDbContext : AbpMongoDbContext

{

public IMongoCollection<Book> Books => Collection<Book>();

//...

}

```

{{end}}

{{if DB == "EF"}}

### 将Book实体映射到数据库表

打开`BookStoreDbContext`类的`OnModelCreating`方法,为`Book`实体添加映射代码:

````csharp

using Acme.BookStore.Books;

...

namespace Acme.BookStore.EntityFrameworkCore

{

public class BookStoreDbContext :

AbpDbContext<BookStoreDbContext>,

IIdentityDbContext,

ITenantManagementDbContext

{

...

protected override void OnModelCreating(ModelBuilder builder)

{

base.OnModelCreating(builder);

/\* Include modules to your migration db context \*/

builder.ConfigurePermissionManagement();

...

/\* Configure your own tables/entities inside here \*/

builder.Entity<Book>(b =>

{

b.ToTable(BookStoreConsts.DbTablePrefix + "Books",

BookStoreConsts.DbSchema);

b.ConfigureByConvention(); //auto configure for the base class props

b.Property(x => x.Name).IsRequired().HasMaxLength(128);

});

}

}

}

````

\* `BookStoreConsts` 含有用于表的架构和表前缀的常量值. 使用它不是强制的,但建议在统一的地方控制表前缀.

\* `ConfigureByConvention()` 方法优雅的配置/映射继承的属性,应对所有的实体使用它.

### 添加数据迁移

本示例使用[EF Core Code First Migrations](https://docs.microsoft.com/zh-cn/ef/core/managing-schemas/migrations/).因为我们修改了数据库映射配置，我们必须创建一个新的迁移并且应用到数据库.

在 `Acme.BookStore.EntityFrameworkCore` 目录打开命令行终端输入以下命令:

```bash

dotnet ef migrations add Created\_Book\_Entity

```

它会添加新迁移类到项目中:

![bookstore-efcore-migration](./images/bookstore-efcore-migration.png)

> 如果你使用Visual Studio, 你也许想要在*\*包管理控制台(PMC)\*中使用 `Add-Migration Created\_Book\_Entity -c BookStoreDbContext` 和 `Update-Database -Context BookStoreDbContext` 命令. 确保 {{if UI=="MVC"}}`Acme.BookStore.Web`{{else if UI=="BlazorServer"}}`Acme.BookStore.Blazor`{{else if UI=="Blazor" || UI=="NG"}}`Acme.BookStore.HttpApi.Host`{{end}} 是启动项目并且 `Acme.BookStore.EntityFrameworkCore.DbMigrations` 是 PMC 的\*默认项目\**.

{{end}}

#### 添加种子数据

> >在运行应用程序之前最好将初始数据添加到数据库中. 本节介绍ABP框架的[数据种子系统](../Data-Seeding.md). 如果你不想创建种子数据可以跳过本节,但是建议你遵循它来学习这个有用的ABP Framework功能。

在 `\*.Domain` 项目下创建 `IDataSeedContributor` 的派生类,并且拷贝以下代码:

```csharp

using System;

using System.Threading.Tasks;

using Acme.BookStore.Books;

using Volo.Abp.Data;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Domain.Repositories;

namespace Acme.BookStore

{

public class BookStoreDataSeederContributor

: IDataSeedContributor, ITransientDependency

{

private readonly IRepository<Book, Guid> \_bookRepository;

public BookStoreDataSeederContributor(IRepository<Book, Guid> bookRepository)

{

\_bookRepository = bookRepository;

}

public async Task SeedAsync(DataSeedContext context)

{

if (await \_bookRepository.GetCountAsync() <= 0)

{

await \_bookRepository.InsertAsync(

new Book

{

Name = "1984",

Type = BookType.Dystopia,

PublishDate = new DateTime(1949, 6, 8),

Price = 19.84f

},

autoSave: true

);

await \_bookRepository.InsertAsync(

new Book

{

Name = "The Hitchhiker's Guide to the Galaxy",

Type = BookType.ScienceFiction,

PublishDate = new DateTime(1995, 9, 27),

Price = 42.0f

},

autoSave: true

);

}

}

}

}

```

\* 如果数据库中当前没有图书,则此代码使用 `IRepository<Book, Guid>`(默认[repository](../Repositories.md))将两本书插入数据库.

### 更新数据库

运行 `Acme.BookStore.DbMigrator` 应用程序来更新数据库:

![bookstore-dbmigrator-on-solution](images/bookstore-dbmigrator-on-solution.png)

`.DbMigrator` 是一个控制台使用程序,可以在**\*\*开发\*\***和**\*\*生产\*\***环境**\*\*迁移数据库架构\*\***和**\*\*初始化种子数据\*\***.

## 创建应用程序

应用程序层由两个分离的项目组成:

\* `Acme.BookStore.Application.Contracts` 包含你的[DTO](../Data-Transfer-Objects.md)和[应用服务](../Application-Services.md)接口.

\* `Acme.BookStore.Application` 包含你的应用服务实现.

在本部分中,你将创建一个应用程序服务,使用ABP Framework的 `CrudAppService` 基类来获取,创建,更新和删除书籍.

### BookDto

`CrudAppService` 基类需要定义实体的基本DTO. 在 `Acme.BookStore.Application.Contracts` 项目中创建 `Books` 文件夹(命名空间), 并在其中添加名为 `BookDto` 的DTO类:

````C#

using System;

using Volo.Abp.Application.Dtos;

namespace Acme.BookStore.Books

{

public class BookDto : AuditedEntityDto<Guid>

{

public string Name { get; set; }

public BookType Type { get; set; }

public DateTime PublishDate { get; set; }

public float Price { get; set; }

}

}

````

\* **\*\*DTO\*\***类被用来在 **\*\*表示层\*\*** 和 **\*\*应用层\*\*** **\*\*传递数据\*\***.参阅[DTO文档](https://docs.abp.io/zh-Hans/abp/latest/Data-Transfer-Objects).

\* 为了在用户界面上展示书籍信息,`BookDto`被用来将书籍数据传递到表示层.

\* `BookDto`继承自 `AuditedEntityDto<Guid>`.与上面定义的 `Book` 实体一样具有一些审计属性.

在将书籍返回到表示层时,需要将`Book`实体转换为`BookDto`对象. [AutoMapper](https://automapper.org)库可以在定义了正确的映射时自动执行此转换. 启动模板配置了AutoMapper,因此你只需在`Acme.BookStore.Application`项目的`BookStoreApplicationAutoMapperProfile`类中定义映射:

````csharp

using Acme.BookStore.Books;

using AutoMapper;

namespace Acme.BookStore

{

public class BookStoreApplicationAutoMapperProfile : Profile

{

public BookStoreApplicationAutoMapperProfile()

{

CreateMap<Book, BookDto>();

}

}

}

````

> 参阅 [对象到对象映射](../Object-To-Object-Mapping.md) 文档了解详情.

### CreateUpdateBookDto

在`Acme.BookStore.Application.Contracts`项目中创建 `Books` 文件夹(命名空间),并在其中添加名为 `CreateUpdateBookDto` 的DTO类:

````csharp

using System;

using System.ComponentModel.DataAnnotations;

namespace Acme.BookStore.Books

{

public class CreateUpdateBookDto

{

[Required]

[StringLength(128)]

public string Name { get; set; }

[Required]

public BookType Type { get; set; } = BookType.Undefined;

[Required]

[DataType(DataType.Date)]

public DateTime PublishDate { get; set; } = DateTime.Now;

[Required]

public float Price { get; set; }

}

}

````

\* 这个DTO类被用于在创建或更新书籍的时候从用户界面获取图书信息.

\* 它定义了数据注释特性(如`[Required]`)来定义属性的验证规则. DTO由ABP框架[自动验证](https://docs.abp.io/zh-Hans/abp/latest/Validation).

就像上面的`BookDto`一样,创建一个从`CreateUpdateBookDto`对象到`Book`实体的映射,最终映射配置类如下:

````csharp

using Acme.BookStore.Books;

using AutoMapper;

namespace Acme.BookStore

{

public class BookStoreApplicationAutoMapperProfile : Profile

{

public BookStoreApplicationAutoMapperProfile()

{

CreateMap<Book, BookDto>();

CreateMap<CreateUpdateBookDto, Book>();

}

}

}

````

### IBookAppService

下一步是为应用程序定义接口,在`Acme.BookStore.Application.Contracts`项目创建 `Books` 文件夹(命名空间),并在其中添加名为`IBookAppService`的接口:

````csharp

using System;

using Volo.Abp.Application.Dtos;

using Volo.Abp.Application.Services;

namespace Acme.BookStore.Books

{

public interface IBookAppService :

ICrudAppService< //Defines CRUD methods

BookDto, //Used to show books

Guid, //Primary key of the book entity

PagedAndSortedResultRequestDto, //Used for paging/sorting

CreateUpdateBookDto> //Used to create/update a book

{

}

}

````

\* 框架定义应用程序服务的接口**\*\*不是必需的\*\***. 但是,它被建议作为最佳实践.

\* `ICrudAppService`定义了常见的**\*\*CRUD\*\***方法:`GetAsync`,`GetListAsync`,`CreateAsync`,`UpdateAsync`和`DeleteAsync`. 从这个接口扩展不是必需的,你可以从空的`IApplicationService`接口继承并手动定义自己的方法(将在下一部分中完成).

\* `ICrudAppService`有一些变体, 你可以在每个方法中使用单独的DTO(例如使用不同的DTO进行创建和更新).

### BookAppService

是时候实现`IBookAppService`接口了.在`Acme.BookStore.Application`项目中创建 `Books` 文件夹(命名空间),并在其中添加名为 `BookAppService` 的类:

````csharp

using System;

using Volo.Abp.Application.Dtos;

using Volo.Abp.Application.Services;

using Volo.Abp.Domain.Repositories;

namespace Acme.BookStore.Books

{

public class BookAppService :

CrudAppService<

Book, //The Book entity

BookDto, //Used to show books

Guid, //Primary key of the book entity

PagedAndSortedResultRequestDto, //Used for paging/sorting

CreateUpdateBookDto>, //Used to create/update a book

IBookAppService //implement the IBookAppService

{

public BookAppService(IRepository<Book, Guid> repository)

: base(repository)

{

}

}

}

````

\* `BookAppService`继承了`CrudAppService<...>`.它实现了 `ICrudAppService` 定义的CRUD方法.

\* `BookAppService`注入`IRepository <Book,Guid>`,这是`Book`实体的默认仓储. ABP自动为每个聚合根(或实体)创建默认仓储. 请参阅[仓储文档](https://docs.abp.io/zh-Hans/abp/latest/Repositories)

\* `BookAppService`使用[`IObjectMapper`](../Object-To-Object-Mapping.md)将`Book`对象转换为`BookDto`对象, 将`CreateUpdateBookDto`对象转换为`Book`对象. 启动模板使用[AutoMapper](http://automapper.org/)库作为对象映射提供程序. 我们之前定义了映射, 因此它将按预期工作.

### 自动生成API Controllers

在典型的ASP.NET Core应用程序中,你创建**\*\*API Controller\*\***以将应用程序服务公开为**\*\*HTTP API\*\***端点. 这将允许浏览器或第三方客户端通过HTTP调用它们.

ABP可以[\*\*自动\*\*](../API/Auto-API-Controllers.md)按照约定将你的应用程序服务配置为MVC API控制器.

### Swagger UI

启动模板配置为使用[Swashbuckle.AspNetCore](https://github.com/domaindrivendev/Swashbuckle.AspNetCore)运行[swagger UI](https://swagger.io/tools/swagger-ui/). 运行应用程序并在浏览器中输入`https://localhost:XXXX/swagger/`(用你自己的端口替换XXXX)作为URL.

使用`CTRL+F5`运行应用程序 ({{if UI=="MVC"}}`Acme.BookStore.Web`{{else}}`Acme.BookStore.HttpApi.Host`{{end}})并使用浏览器访问`https://localhost:<port>/swagger/` on your browser. 使用你自己的端口号替换 `<port>`.

你会看到一些内置的服务端点和`Book`服务,它们都是REST风格的端点:

![bookstore-swagger](./images/bookstore-swagger.png)

Swagger有一个很好的UI来测试API.

你可以尝试执行`[GET] /api/app/book` API来获取书籍列表, 服务端会返回以下JSON结果:

````json

{

"totalCount": 2,

"items": [

{

"name": "The Hitchhiker's Guide to the Galaxy",

"type": 7,

"publishDate": "1995-09-27T00:00:00",

"price": 42,

"lastModificationTime": null,

"lastModifierId": null,

"creationTime": "2020-07-03T21:04:18.4607218",

"creatorId": null,

"id": "86100bb6-cbc1-25be-6643-39f62806969c"

},

{

"name": "1984",

"type": 3,

"publishDate": "1949-06-08T00:00:00",

"price": 19.84,

"lastModificationTime": null,

"lastModifierId": null,

"creationTime": "2020-07-03T21:04:18.3174016",

"creatorId": null,

"id": "41055277-cce8-37d7-bb37-39f62806960b"

}

]

}

````

这很酷,因为我们没有编写任何代码来创建API控制器,但是现在我们有了一个可以正常使用的REST API!

## 下一章

参阅教程的[下一章](part-2.md).

图书列表页面

# Web应用程序开发教程 - 第二章: 图书列表页面

````json

//[doc-params]

{

"UI": ["MVC","Blazor","BlazorServer","NG"],

"DB": ["EF","Mongo"]

}

````

## 关于本教程

在本系列教程中, 你将构建一个名为 `Acme.BookStore` 的用于管理书籍及其作者列表的基于ABP的应用程序. 它是使用以下技术开发的:

\* **\*\*{{DB\_Value}}\*\*** 做为ORM提供程序.

\* **\*\*{{UI\_Value}}\*\*** 做为UI框架.

本教程分为以下部分:

- [Part 1: 创建服务端](Part-1.md)

- **\*\*Part 2: 图书列表页面 (本章)\*\***

- [Part 3: 创建,更新和删除图书](Part-3.md)

- [Part 4: 集成测试](Part-4.md)

- [Part 5: 授权](Part-5.md)

- [Part 6: 作者: 领域层](Part-6.md)

- [Part 7: 作者: 数据库集成](Part-7.md)

- [Part 8: 作者: 应用服务层](Part-8.md)

- [Part 9: 作者: 用户页面](Part-9.md)

- [Part 10: 图书到作者的关系](Part-10.md)

## 下载源码

本教程根据你的**\*\*UI\*\*** 和 **\*\*数据库\*\***偏好有多个版本,我们准备了几种可供下载的源码组合:

\* [MVC (Razor Pages) UI 与 EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Mvc-EfCore)

\* [Blazor UI 与 EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Blazor-EfCore)

\* [Angular UI 与 MongoDB](https://github.com/abpframework/abp-samples/tree/master/BookStore-Angular-MongoDb)

> 如果你在Windows中遇到 "文件名太长" or "解压错误", 很可能与Windows最大文件路径限制有关. Windows文件路径的最大长度为250字符. 为了解决这个问题,参阅 [在Windows 10中启用长路径](https://docs.microsoft.com/en-us/windows/win32/fileio/maximum-file-path-limitation?tabs=cmd#enable-long-paths-in-windows-10-version-1607-and-later).

> 如果你遇到与Git相关的长路径错误, 尝试使用下面的命令在Windows中启用长路径. 参阅 https://github.com/msysgit/msysgit/wiki/Git-cannot-create-a-file-or-directory-with-a-long-path

> `git config --system core.longpaths true`

{{if UI == "MVC" && DB == "EF"}}

### 视频教程

本章也被录制为视频教程 **\*\***<a href="https://www.youtube.com/watch?v=UDNlLiPiBiw&list=PLsNclT2aHJcPNaCf7Io3DbMN6yAk\_DgWJ&index=2" target="\_blank">**发布在YouTube**</a>**\*\***.

{{end}}

{{if UI == "MVC"}}

## 动态JavaScript代理

在 **\*\*JavaScript\*\*** 端通过AJAX调用HTTP API端点是常见的做法. 你可以使用 `$.ajax` 或其他工具来调用端点. 但是ABP提供了更好的方法.

ABP**\*\*动态\*\***为所有API端点创建 **\*\*[**JavaScript代理**](../UI/AspNetCore/Dynamic-JavaScript-Proxies.md)\*\***. 所以你可以像调用**\*\*Javascript本地方法\*\***一样使用任何**\*\*端点\*\***.

### 在开发者控制台中进行测试

你可以在自己喜欢的浏览器的**\*\*开发者控制台\*\***轻松的测试JavaScript代理. 运行应用程序,打开浏览器的**\*\*开发者人员工具\*\***(*\*快捷键通常是F12\**),切换到**\*\*控制台\*\***选项卡,输入以下代码然后按回车:

````js

acme.bookStore.books.book.getList({}).done(function (result) { console.log(result); });

````

\* `acme.bookStore.books` 是 `BookAppService` 的命令空间转换成[小驼峰](https://en.wikipedia.org/wiki/Camel\_case)形式.

\* `book` 是 `BookAppService` 的约定名称(删除`AppService`后缀并且转换为小驼峰).

\* `getList` 是 `CrudAppService` 基类定义的 `GetListAsync` 方法的约定名称(删除`Async`后缀并且转换为小驼峰).

\* `{}` 参数将空对象发送到 `GetListAsync` 方法,该方法通常需要一个类型为 `PagedAndSortedResultRequestDto` 的对象,该对象用于将分页和排序选项发送到服务器(所有属性都是可选的,具有默认值. 因此你可以发送一个空对象).

\* `getList` 函数返回一个 `promise`. 你可以传递一个回调到 `then`(或`done`)函数来获取从服务器返回的结果.

运行该代码会产生以下输出:

![bookstore-javascript-proxy-console](images/bookstore-javascript-proxy-console.png)

你可以看到服务端返回的 **\*\*图书列表\*\***. 你也可以在开发者人员工具的 **\*\*网络\*\*** 选项卡查看客户端到服务端的通信:

![bookstore-getlist-result-network](images/bookstore-getlist-result-network.png)

让我们使用 `create` 函数**\*\*创建一本书\*\***:

````js

acme.bookStore.books.book.create({

name: 'Foundation',

type: 7,

publishDate: '1951-05-24',

price: 21.5

}).then(function (result) {

console.log('successfully created the book with id: ' + result.id);

});

````

> 如果你下载了本教程的源代码并按照示例中的步骤操作,你需要传递`authorId`参数给创建方法以**\*\*创建一本新书\*\***.

您应该在控制台中看到类似以下的消息:

````text

successfully created the book with id: 439b0ea8-923e-8e1e-5d97-39f2c7ac4246

````

检查数据库中的 `Books` 表你会看到新的一行. 你可以自己尝试使用 `get`, `update` 和 `delete` 函数.

在接下来的章节,我们将利用这些动态代理函数与服务器通信.

{{end}}

## 本地化

开始的UI开发之前,我们首先要准备本地化的文本(这是你通常在开发应用程序时需要做的).

本地化文本位于 `Acme.BookStore.Domain.Shared` 项目的 `Localization/BookStore` 文件夹下:

![bookstore-localization-files](images/bookstore-localization-files-v2.png)

打开 `en.json` (*\*英文翻译\**)文件并更改内容,如下所示:

````json

{

"Culture": "en",

"Texts": {

"Menu:Home": "Home",

"Welcome": "Welcome",

"LongWelcomeMessage": "Welcome to the application. This is a startup project based on the ABP framework. For more information, visit abp.io.",

"Menu:BookStore": "Book Store",

"Menu:Books": "Books",

"Actions": "Actions",

"Close": "Close",

"Delete": "Delete",

"Edit": "Edit",

"PublishDate": "Publish date",

"NewBook": "New book",

"Name": "Name",

"Type": "Type",

"Price": "Price",

"CreationTime": "Creation time",

"AreYouSure": "Are you sure?",

"AreYouSureToDelete": "Are you sure you want to delete this item?",

"Enum:BookType.Undefined": "Undefined",

"Enum:BookType.Adventure": "Adventure",

"Enum:BookType.Biography": "Biography",

"Enum:BookType.Dystopia": "Dystopia",

"Enum:BookType.Fantastic": "Fantastic",

"Enum:BookType.Horror": "Horror",

"Enum:BookType.Science": "Science",

"Enum:BookType.ScienceFiction": "Science fiction",

"Enum:BookType.Poetry": "Poetry"

}

}

````

> 简体中文翻译请打开`zh-Hans.json`文件 ,并将"Texts"对象中对应的值替换为中文.

\* 本地化关键字名称是任意的. 你可以设置任何名称. 对于特定的文本类型,我们更喜欢遵循一些约定:

\* 为按钮项添加 `Menu:` 前缀.

\* 使用 `Enum:<enum-type>:<enum-name>` 或 `<enum-type>.<enum-name>` 或 `<enum-name>` 命名约定来本地化枚举成员. 当您这样做时ABP可以在某些适当的情况下自动将枚举本地化.

如果未在本地化文件中定义文本,则文本将**\*\*回退\*\***到本地化键(ASP.NET Core的标准行为).

> ABP本地化系统建立在[ASP.NET Core标准本地化](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/localization)系统之上,并以多种方式进行了扩展. 有关详细信息请参见[本地化文档](../Localization.md).

{{if UI == "MVC"}}

## 创建图书页面

是时候创建可见的和可用的东西了! 我们将使用微软推荐的[Razor Pages UI](https://docs.microsoft.com/zh-cn/aspnet/core/tutorials/razor-pages/razor-pages-start),而不是经典的MVC.

在 `Acme.BookStore.Web` 项目的 `Pages` 文件夹下创建一个名为新的 `Books` 的文件夹. 然后在文件夹右键选择 **\*\*添加 > Razor Page\*\*** 菜单. 输入名称 `Index`:

![bookstore-add-index-page](./images/bookstore-add-index-page-v2.png)

打开 `Index.cshtml` 并把内容修改成下面这样:

````html

@page

@using Acme.BookStore.Web.Pages.Books

@model IndexModel

<h2>Books</h2>

````

`Index.cshtml.cs` 内容应该是:

```csharp

using Microsoft.AspNetCore.Mvc.RazorPages;

namespace Acme.BookStore.Web.Pages.Books

{

public class IndexModel : PageModel

{

public void OnGet()

{

}

}

}

```

### 将图书页面添加到主菜单

打开 `Menus` 文件夹中的 `BookStoreMenuContributor` 类,在 `ConfigureMainMenuAsync` 方法的底部添加如下代码:

````csharp

context.Menu.AddItem(

new ApplicationMenuItem(

"BooksStore",

l["Menu:BookStore"],

icon: "fa fa-book"

).AddItem(

new ApplicationMenuItem(

"BooksStore.Books",

l["Menu:Books"],

url: "/Books"

)

)

);

````

运行项目,使用用户名 `admin` 和密码 `1q2w3E\*` 登录到应用程序. 看到新菜单项已添加到顶部栏:

![bookstore-menu-items](./images/bookstore-new-menu-item.png)

点击BookStore下的Books子菜单项就会跳转到空的图书页面.

### 图书列表

我们将使用[Datatables.net](https://datatables.net/)JQuery插件来显示图书列表. [Datatables](https://datatables.net/)可以完全通过AJAX工作,速度快,并提供良好的用户体验.

> Datatables插件在启动模板中配置,因此你可以直接在任何页面中使用它,无需在页面中引用样式和脚本文件.

#### Index.cshtml

将 `Pages/Book/Index.cshtml` 改成下面的样子:

````html

@page

@using Acme.BookStore.Localization

@using Acme.BookStore.Web.Pages.Books

@using Microsoft.Extensions.Localization

@model IndexModel

@inject IStringLocalizer<BookStoreResource> L

@section scripts

{

<abp-script src="/Pages/Books/Index.js" />

}

<abp-card>

<abp-card-header>

<h2>@L["Books"]</h2>

</abp-card-header>

<abp-card-body>

<abp-table striped-rows="true" id="BooksTable"></abp-table>

</abp-card-body>

</abp-card>

````

\* `abp-script` [tag helper](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/tag-helpers/intro)用于将外部的 **\*\*脚本\*\*** 添加到页面中.它比标准的`script`标签多了很多额外的功能.它可以处理 **\*\*最小化\*\***和 **\*\*版本\*\***.查看[捆绑 & 压缩文档](../UI/AspNetCore/Bundling-Minification.md)获取更多信息.

\* `abp-card` 和 `abp-table` 是为Twitter Bootstrap的[card component](http://getbootstrap.com/docs/4.5/components/card/)封装的 **\*\*tag helpers\*\***.ABP中有很多tag helpers,可以很方便的使用大多数[bootstrap](https://getbootstrap.com/)组件.你也可以使用原生的HTML标签代替tag helpers.使用tag helper可以通过智能提示和编译时类型检查减少HTML代码并防止错误.查看[tag helpers 文档](../UI/AspNetCore/Tag-Helpers/Index.md).

#### Index.js

在 `Pages/Books/` 文件夹中创建 `index.js`文件

![bookstore-index-js-file](images/bookstore-index-js-file-v3.png)

`index.js` 的内容如下:

````js

$(function () {

var l = abp.localization.getResource('BookStore');

var dataTable = $('#BooksTable').DataTable(

abp.libs.datatables.normalizeConfiguration({

serverSide: true,

paging: true,

order: [[1, "asc"]],

searching: false,

scrollX: true,

ajax: abp.libs.datatables.createAjax(acme.bookStore.books.book.getList),

columnDefs: [

{

title: l('Name'),

data: "name"

},

{

title: l('Type'),

data: "type",

render: function (data) {

return l('Enum:BookType:' + data);

}

},

{

title: l('PublishDate'),

data: "publishDate",

render: function (data) {

return luxon

.DateTime

.fromISO(data, {

locale: abp.localization.currentCulture.name

}).toLocaleString();

}

},

{

title: l('Price'),

data: "price"

},

{

title: l('CreationTime'), data: "creationTime",

render: function (data) {

return luxon

.DateTime

.fromISO(data, {

locale: abp.localization.currentCulture.name

}).toLocaleString(luxon.DateTime.DATETIME\_SHORT);

}

}

]

})

);

});

````

\* `abp.localization.getResource` 获取一个函数,该函数用于使用服务器端定义的相同JSON文件对文本进行本地化. 通过这种方式你可以与客户端共享本地化值.

\* `abp.libs.datatables.normalizeConfiguration`是一个辅助方法.不是必须的, 但是它通过为缺省的选项提供约定的值来简化[Datatables](https://datatables.net/)配置.

\* `abp.libs.datatables.createAjax`是另一个辅助方法,用来适配ABP的动态JavaScript API代理和[Datatable](https://datatables.net/)期望的参数格式.

\* `acme.bookStore.books.book.getList` 是动态JavaScript代理函数(上面已经介绍过了)

\* [luxon](https://moment.github.io/luxon/) 库也是该解决方案中预先配置的标准库,你可以轻松地执行日期/时间操作.

> 查看 [Datatable文档](https://datatables.net/manual/) 了解更多配置项.

## 运行最终应用程序

你可以运行应用程序!该部分的最终用户界面如下所示:

![Book list](images/bookstore-book-list-3.png)

这是一个可以正常工作的,服务端分页,排序和本地化的图书列表.

{{else if UI == "NG"}}

## 安装NPM包

>注意: 本教程基于ABP Framework v3.1.0+. 如果你的项目版本较旧,请升级您的解决方案. 如果要升级现有的v2.x项目,请参阅[迁移指南](../UI/Angular/Migration-Guide-v3.md).

在 `angular` 目录下打开命令行窗口,选择 `yarn` 命令安装NPM包:

```bash

yarn

```

## 创建图书页面

是时候创建可见和可用的东西了!开发ABP Angular前端应用程序时,需要使用一些工具:

- [Ng Bootstrap](https://ng-bootstrap.github.io/#/home) 用做UI组件库.

- [ngx-datatable](https://swimlane.gitbook.io/ngx-datatable/) 用做 datatable 类库.

运行以下命令在angular应用程序根目录创建一个名为 `BookModule` 的新模块:

```bash

yarn ng generate module book --module app --routing --route books

```

该命令应该产生以下的输出:

````bash

> yarn ng generate module book --module app --routing --route books

yarn run v1.19.1

$ ng generate module book --module app --routing --route books

CREATE src/app/book/book-routing.module.ts (336 bytes)

CREATE src/app/book/book.module.ts (335 bytes)

CREATE src/app/book/book.component.html (19 bytes)

CREATE src/app/book/book.component.spec.ts (614 bytes)

CREATE src/app/book/book.component.ts (268 bytes)

CREATE src/app/book/book.component.scss (0 bytes)

UPDATE src/app/app-routing.module.ts (1289 bytes)

Done in 3.88s.

````

### BookModule

打开 `/src/app/book/book.module.ts` 并使用以下内容替换:

````js

import { NgModule } from '@angular/core';

import { SharedModule } from '../shared/shared.module';

import { BookRoutingModule } from './book-routing.module';

import { BookComponent } from './book.component';

@NgModule({

declarations: [BookComponent],

imports: [

BookRoutingModule,

SharedModule

]

})

export class BookModule { }

````

\* 添加了 `SharedModule`. `SharedModule` 导出了一些创建用户界面所需的通用模块.

\* `SharedModule` 已经导出了 `CommonModule`,所以我们删除了 `CommonModule`.

### 路由

生成的代码将新的路由定义放在 `src/app/app-routing.module.ts` 文件中,如下所示:

````js

const routes: Routes = [

// other route definitions...

{ path: 'books', loadChildren: () => import('./book/book.module').then(m => m.BookModule) },

];

````

现在打开 `src/app/route.provider.ts` 替换 `configureRoutes` 函数为以下代码:

```js

function configureRoutes(routes: RoutesService) {

return () => {

routes.add([

{

path: '/',

name: '::Menu:Home',

iconClass: 'fas fa-home',

order: 1,

layout: eLayoutType.application,

},

{

path: '/book-store',

name: '::Menu:BookStore',

iconClass: 'fas fa-book',

order: 2,

layout: eLayoutType.application,

},

{

path: '/books',

name: '::Menu:Books',

parentName: '::Menu:BookStore',

layout: eLayoutType.application,

},

]);

};

}

```

`RoutesService` 是ABP框架提供的用于配置主菜单和路由的服务.

\* `path` 路由的URL.

\* `name` 菜单项的名称(参阅[本地化文档](../UI/Angular/Localization.md)了解更多).

\* `iconClass` 菜单项的图标(你可以使用默认的[Font Awesome](https://fontawesome.com/)图标).

\* `order` 菜单项的排序.

\* `layout` BooksModule路由的布局. (有三个预定义的布局类型: `eLayoutType.application`, `eLayoutType.account` 或 `eLayoutType.empty`).

更多信息请参阅[RoutesService 文档](../UI/Angular/Modifying-the-Menu.md#via-routesservice).

### 生成服务代理

[ABP CLI](../CLI.md) 提供 `generate-proxy` 命令为HTTP APIs生成客户端代理.有了这些代理,在客户端使用HTTP APIs变得更加方便. 运行 `generate-proxy` 命令前, 你的 host 必须正在运行.

> **\*\*警告\*\***: 使用IIS Express时有一个问题; 它不允许从另一个进程连接应用程序. 如果你使用Visual Studio, 在运行按钮的下拉框中选择`Acme.BookStore.HttpApi.Host`,不要选择IIS Express, 如下图:

![vs-run-without-iisexpress](images/vs-run-without-iisexpress.png)

启动host应用程序后,在 `angular` 文件夹下运行以下命令:

```bash

abp generate-proxy -t ng

```

这个命令将在`/src/app/proxy/books`文件夹下产生以下文件:

![Generated files](images/generated-proxies-3.png)

### BookComponent

打开 `/src/app/book/book.component.ts` 用以下内容替换它:

```js

import { ListService, PagedResultDto } from '@abp/ng.core';

import { Component, OnInit } from '@angular/core';

import { BookService, BookDto } from '@proxy/books';

@Component({

selector: 'app-book',

templateUrl: './book.component.html',

styleUrls: ['./book.component.scss'],

providers: [ListService],

})

export class BookComponent implements OnInit {

book = { items: [], totalCount: 0 } as PagedResultDto<BookDto>;

constructor(public readonly list: ListService, private bookService: BookService) {}

ngOnInit() {

const bookStreamCreator = (query) => this.bookService.getList(query);

this.list.hookToQuery(bookStreamCreator).subscribe((response) => {

this.book = response;

});

}

}

```

\* 我们引入并注入了生成的 `BookService`.

\* 我们使用 [ListService](../UI/Angular/List-Service.md),它是一个工具服务,提供了易用的分页,排序和搜索.

打开 `/src/app/book/book.component.html` 用以下内容替换它:

```html

<div class="card">

<div class="card-header">

<div class="row">

<div class="col col-md-6">

<h5 class="card-title">

{%{{{ '::Menu:Books' | abpLocalization }}}%}

</h5>

</div>

<div class="text-end col col-md-6"></div>

</div>

</div>

<div class="card-body">

<ngx-datatable [rows]="book.items" [count]="book.totalCount" [list]="list" default>

<ngx-datatable-column [name]="'::Name' | abpLocalization" prop="name"></ngx-datatable-column>

<ngx-datatable-column [name]="'::Type' | abpLocalization" prop="type">

<ng-template let-row="row" ngx-datatable-cell-template>

{%{{{ '::Enum:BookType:' + row.type | abpLocalization }}}%}

</ng-template>

</ngx-datatable-column>

<ngx-datatable-column [name]="'::PublishDate' | abpLocalization" prop="publishDate">

<ng-template let-row="row" ngx-datatable-cell-template>

{%{{{ row.publishDate | date }}}%}

</ng-template>

</ngx-datatable-column>

<ngx-datatable-column [name]="'::Price' | abpLocalization" prop="price">

<ng-template let-row="row" ngx-datatable-cell-template>

{%{{{ row.price | currency }}}%}

</ng-template>

</ngx-datatable-column>

</ngx-datatable>

</div>

</div>

```

现在你可以在浏览器看到最终结果:

![图书列表最终结果](images/bookstore-book-list.png)

{{else if UI == "Blazor" || UI == "BlazorServer"}}

## 创建图书页面

是时候创建可见和可用的东西了! 右击`Acme.BookStore.Blazor`项目下的`Pages`文件夹,新建一个名为`Books.razor`的**\*\*razor组件\*\***.

![blazor-add-books-component](images/blazor-add-books-component.png)

用以下内容替换这个组件的内容:

````html

@page "/books"

<h2>Books</h2>

@code {

}

````

### 将图书页面添加到主菜单

打开`Blazor`项目中的`BookStoreMenuContributor`类,在 `ConfigureMainMenuAsync` 方法的底部添加如下代码:

````csharp

context.Menu.AddItem(

new ApplicationMenuItem(

"BooksStore",

l["Menu:BookStore"],

icon: "fa fa-book"

).AddItem(

new ApplicationMenuItem(

"BooksStore.Books",

l["Menu:Books"],

url: "/books"

)

)

);

````

运行项目,使用用户名 `admin` 和密码 `1q2w3E\*` 登录到应用程序. 看到新菜单项已添加到顶部栏:

![blazor-menu-bookstore](images/blazor-menu-bookstore.png)

点击BookStore下的Books子菜单项就会跳转到空的图书页面.

### 图书列表

我们将使用[Blazorise library](https://blazorise.com/)作为UI组件.它是一个强大的库,支持主要的HTML/CSS框架,包括Bootstrap.

ABP提供了一个通用的基类,`AbpCrudPageBase<...>`,用来创建CRUD风格的页面.这个基类兼容用来构建`IBookAppService`的`ICrudAppService`.所以我们从`AbpCrudPageBase`继承,获得标准CRUD的默认实现.

打开`Books.razor` 并把内容修改成下面这样:

````xml

@page "/books"

@using Volo.Abp.Application.Dtos

@using Acme.BookStore.Books

@using Acme.BookStore.Localization

@using Microsoft.Extensions.Localization

@inject IStringLocalizer<BookStoreResource> L

@inherits AbpCrudPageBase<IBookAppService, BookDto, Guid, PagedAndSortedResultRequestDto, CreateUpdateBookDto>

<Card>

<CardHeader>

<h2>@L["Books"]</h2>

</CardHeader>

<CardBody>

<DataGrid TItem="BookDto"

Data="Entities"

ReadData="OnDataGridReadAsync"

TotalItems="TotalCount"

ShowPager="true"

PageSize="PageSize">

<DataGridColumns>

<DataGridColumn TItem="BookDto"

Field="@nameof(BookDto.Name)"

Caption="@L["Name"]"></DataGridColumn>

<DataGridColumn TItem="BookDto"

Field="@nameof(BookDto.Type)"

Caption="@L["Type"]">

<DisplayTemplate>

@L[$"Enum:BookType.{Enum.GetName(context.Type)}"]

</DisplayTemplate>

</DataGridColumn>

<DataGridColumn TItem="BookDto"

Field="@nameof(BookDto.PublishDate)"

Caption="@L["PublishDate"]">

<DisplayTemplate>

@context.PublishDate.ToShortDateString()

</DisplayTemplate>

</DataGridColumn>

<DataGridColumn TItem="BookDto"

Field="@nameof(BookDto.Price)"

Caption="@L["Price"]">

</DataGridColumn>

<DataGridColumn TItem="BookDto"

Field="@nameof(BookDto.CreationTime)"

Caption="@L["CreationTime"]">

<DisplayTemplate>

@context.CreationTime.ToLongDateString()

</DisplayTemplate>

</DataGridColumn>

</DataGridColumns>

</DataGrid>

</CardBody>

</Card>

````

> 如果你可以编译并运行成功,但看到一些语法错误.你可以忽略这些错误,因为Visual Studio处理Blazor还有一些bug.

\* `AbpCrudPageBase<IBookAppService, BookDto, Guid, PagedAndSortedResultRequestDto, CreateUpdateBookDto>`实现了所有的CRUD细节,我们从它继承.

\* `Entities`, `TotalCount`, `PageSize`, `OnDataGridReadAsync`定义在基类中.

\* 注入`IStringLocalizer<BookStoreResource>` (作为`L`对象),用于本地化.

虽然上面的代码非常容易理解,你仍然可以查看Blazorise [Card](https://blazorise.com/docs/components/card/)和[DataGrid](https://blazorise.com/docs/extensions/datagrid/)文档以更好地理解它们.

#### 关于AbpCrudPageBase

对于图书页面,我们将持续从`AbpCrudPageBase`获得益处. 你可以只注入`IBookAppService`并自己执行所有的服务端调用(感谢ABP的[动态C# HTTP API客户端代理](../API/Dynamic-CSharp-API-Clients.md)系统).

## 运行最终应用程序

你可以运行应用程序!该部分的最终用户界面如下所示:

![blazor-bookstore-book-list](images/blazor-bookstore-book-list.png)

这是一个可以正常工作的,服务端分页,排序和本地化的图书列表.

{{end # UI }}

## 下一章

查看本教程的[下一章](Part-3.md).

创建,更新和删除书籍

# Web应用程序开发教程 - 第三章: 创建,更新和删除图书

````json

//[doc-params]

{

"UI": ["MVC","Blazor","BlazorServer","NG"],

"DB": ["EF","Mongo"]

}

````

## 关于本教程

在本系列教程中, 你将构建一个名为 `Acme.BookStore` 的用于管理书籍及其作者列表的基于ABP的应用程序. 它是使用以下技术开发的:

\* **\*\*{{DB\_Value}}\*\*** 做为ORM提供程序.

\* **\*\*{{UI\_Value}}\*\*** 做为UI框架.

本教程分为以下部分:

- [Part 1: 创建服务端](Part-1.md)

- [Part 2: 图书列表页面](Part-2.md)

- **\*\*Part 3: 创建,更新和删除图书 (本章)\*\***

- [Part 4: 集成测试](Part-4.md)

- [Part 5: 授权](Part-5.md)

- [Part 6: 作者: 领域层](Part-6.md)

- [Part 7: 作者: 数据库集成](Part-7.md)

- [Part 8: 作者: 应用服务层](Part-8.md)

- [Part 9: 作者: 用户页面](Part-9.md)

- [Part 10: 图书到作者的关系](Part-10.md)

## 下载源码

本教程根据你的**\*\*UI\*\*** 和 **\*\*数据库\*\***偏好有多个版本,我们准备了几种可供下载的源码组合:

\* [MVC (Razor Pages) UI 与 EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Mvc-EfCore)

\* [Blazor UI 与 EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Blazor-EfCore)

\* [Angular UI 与 MongoDB](https://github.com/abpframework/abp-samples/tree/master/BookStore-Angular-MongoDb)

> 如果你在Windows中遇到 "文件名太长" or "解压错误", 很可能与Windows最大文件路径限制有关. Windows文件路径的最大长度为250字符. 为了解决这个问题,参阅 [在Windows 10中启用长路径](https://docs.microsoft.com/en-us/windows/win32/fileio/maximum-file-path-limitation?tabs=cmd#enable-long-paths-in-windows-10-version-1607-and-later).

> 如果你遇到与Git相关的长路径错误, 尝试使用下面的命令在Windows中启用长路径. 参阅 https://github.com/msysgit/msysgit/wiki/Git-cannot-create-a-file-or-directory-with-a-long-path

> `git config --system core.longpaths true`

{{if UI == "MVC" && DB == "EF"}}

### 视频教程

本章也被录制为视频教程 **\*\***<a href="https://www.youtube.com/watch?v=TLShZO8u2VE&list=PLsNclT2aHJcPNaCf7Io3DbMN6yAk\_DgWJ&index=3" target="\_blank">**发布在YouTube**</a>**\*\***.

{{end}}

{{if UI == "MVC"}}

## 创建新书籍

通过本节, 你将会了解如何创建一个 modal form 实现新增书籍的功能. model dialog将如下图所示:

![bookstore-create-dialog](./images/bookstore-create-dialog-2.png)

### 创建 modal form

在 `Acme.BookStore.Web` 项目的 `Pages/Books` 目录下新建一个 `CreateModal.cshtml` Razor页面:

![bookstore-add-create-dialog](./images/bookstore-add-create-dialog-v2.png)

#### CreateModal.cshtml.cs

打开 `CreateModal.cshtml.cs` 代码文件(`CreateModalModel` 类),替换成以下代码:

````C#

using System.Threading.Tasks;

using Acme.BookStore.Books;

using Microsoft.AspNetCore.Mvc;

namespace Acme.BookStore.Web.Pages.Books

{

public class CreateModalModel : BookStorePageModel

{

[BindProperty]

public CreateUpdateBookDto Book { get; set; }

private readonly IBookAppService \_bookAppService;

public CreateModalModel(IBookAppService bookAppService)

{

\_bookAppService = bookAppService;

}

public void OnGet()

{

Book = new CreateUpdateBookDto();

}

public async Task<IActionResult> OnPostAsync()

{

await \_bookAppService.CreateAsync(Book);

return NoContent();

}

}

}

````

\* 该类派生于 `BookStorePageModel` 而非默认的 `PageModel`. `BookStorePageModel` 间接继承了 `PageModel` 并且添加了一些可以被你的page model类使用的通用属性和方法.

\* `Book` 属性上的 `[BindProperty]` 特性将post请求提交上来的数据绑定到该属性上.

\* 该类通过构造函数注入了 `IBookAppService` 应用服务,并且在 `OnPostAsync` 处理程序中调用了服务的 `CreateAsync` 方法.

\* 它在 `OnGet` 方法中创建一个新的 `CreateUpdateBookDto` 对象。 ASP.NET Core不需要像这样创建一个新实例就可以正常工作. 但是它不会为你创建实例,并且如果你的类在类构造函数中赋值一些默认值或执行一些代码,它们将无法工作. 对于这种情况,我们为某些 `CreateUpdateBookDto` 属性设置了默认值.

#### CreateModal.cshtml

打开 `CreateModal.cshtml` 文件并粘贴如下代码:

````html

@page

@using Acme.BookStore.Localization

@using Acme.BookStore.Web.Pages.Books

@using Microsoft.Extensions.Localization

@using Volo.Abp.AspNetCore.Mvc.UI.Bootstrap.TagHelpers.Modal

@model CreateModalModel

@inject IStringLocalizer<BookStoreResource> L

@{

Layout = null;

}

<abp-dynamic-form abp-model="Book" asp-page="/Books/CreateModal">

<abp-modal>

<abp-modal-header title="@L["NewBook"].Value"></abp-modal-header>

<abp-modal-body>

<abp-form-content />

</abp-modal-body>

<abp-modal-footer buttons="@(AbpModalButtons.Cancel|AbpModalButtons.Save)"></abp-modal-footer>

</abp-modal>

</abp-dynamic-form>

````

\* 这个 modal 使用 `abp-dynamic-form` [tag Helper](../UI/AspNetCore/Tag-Helpers/Dynamic-Forms.md) 根据 `CreateUpdateBookDto` 类自动构建了表单.

\* `abp-model` 指定了 `Book` 属性为模型对象.

\* `abp-form-content` tag helper 作为表单控件渲染位置的占位符 (这是可选的,只有你在 `abp-dynamic-form` 中像本示例这样添加了其他内容才需要).

> 提示: 就像在本示例中一样,`Layout` 应该为 `null`,因为当通过AJAX加载模态窗口时,我们不希望包括所有布局.

### 添加 "New book" 按钮

打开 `Pages/Books/Index.cshtml` 并按如下代码修改 `abp-card-header` :

````html

<abp-card-header>

<abp-row>

<abp-column size-md="\_6">

<abp-card-title>@L["Books"]</abp-card-title>

</abp-column>

<abp-column size-md="\_6" class="text-end">

<abp-button id="NewBookButton"

text="@L["NewBook"].Value"

icon="plus"

button-type="Primary"/>

</abp-column>

</abp-row>

</abp-card-header>

````

`Index.cshtml` 的内容最终如下所示:

````html

@page

@using Acme.BookStore.Localization

@using Acme.BookStore.Web.Pages.Books

@using Microsoft.Extensions.Localization

@model IndexModel

@inject IStringLocalizer<BookStoreResource> L

@section scripts

{

<abp-script src="/Pages/Books/Index.js"/>

}

<abp-card>

<abp-card-header>

<abp-row>

<abp-column size-md="\_6">

<abp-card-title>@L["Books"]</abp-card-title>

</abp-column>

<abp-column size-md="\_6" class="text-end">

<abp-button id="NewBookButton"

text="@L["NewBook"].Value"

icon="plus"

button-type="Primary"/>

</abp-column>

</abp-row>

</abp-card-header>

<abp-card-body>

<abp-table striped-rows="true" id="BooksTable"></abp-table>

</abp-card-body>

</abp-card>

````

如下图所示,只是在表格 **\*\*右上方\*\*** 添加了 **\*\*New book\*\*** 按钮:

![bookstore-new-book-button](images/bookstore-new-book-button-2.png)

打开 `Pages/Book/Index.js` 在 `datatable` 配置代码后面添加如下代码:

````js

var createModal = new abp.ModalManager(abp.appPath + 'Books/CreateModal');

createModal.onResult(function () {

dataTable.ajax.reload();

});

$('#NewBookButton').click(function (e) {

e.preventDefault();

createModal.open();

});

````

\* `abp.ModalManager` 是一个在客户端管理modal的辅助类.它内部使用了Twitter Bootstrap的标准modal组件,但通过简化的API抽象了许多细节.

\* `createModal.onResult(...)` 用于在创建书籍后刷新数据表格.

\* `createModal.open();` 用于打开modal创建新书籍.

`Index.js` 的内容最终如下所示:

````js

$(function () {

var l = abp.localization.getResource('BookStore');

var dataTable = $('#BooksTable').DataTable(

abp.libs.datatables.normalizeConfiguration({

serverSide: true,

paging: true,

order: [[1, "asc"]],

searching: false,

scrollX: true,

ajax: abp.libs.datatables.createAjax(acme.bookStore.books.book.getList),

columnDefs: [

{

title: l('Name'),

data: "name"

},

{

title: l('Type'),

data: "type",

render: function (data) {

return l('Enum:BookType:' + data);

}

},

{

title: l('PublishDate'),

data: "publishDate",

render: function (data) {

return luxon

.DateTime

.fromISO(data, {

locale: abp.localization.currentCulture.name

}).toLocaleString();

}

},

{

title: l('Price'),

data: "price"

},

{

title: l('CreationTime'), data: "creationTime",

render: function (data) {

return luxon

.DateTime

.fromISO(data, {

locale: abp.localization.currentCulture.name

}).toLocaleString(luxon.DateTime.DATETIME\_SHORT);

}

}

]

})

);

var createModal = new abp.ModalManager(abp.appPath + 'Books/CreateModal');

createModal.onResult(function () {

dataTable.ajax.reload();

});

$('#NewBookButton').click(function (e) {

e.preventDefault();

createModal.open();

});

});

````

现在,你可以 **\*\*运行程序\*\*** 通过新的 modal form 来创建书籍了.

## 更新书籍

在 `Acme.BookStore.Web` 项目的 `Pages/Books` 目录下新建一个名叫 `EditModal.cshtml` 的Razor页面:

![bookstore-add-edit-dialog](./images/bookstore-add-edit-dialog.png)

### EditModal.cshtml.cs

打开 `EditModal.cshtml.cs` 文件(`EditModalModel`类) 并替换成以下代码:

````csharp

using System;

using System.Threading.Tasks;

using Acme.BookStore.Books;

using Microsoft.AspNetCore.Mvc;

namespace Acme.BookStore.Web.Pages.Books

{

public class EditModalModel : BookStorePageModel

{

[HiddenInput]

[BindProperty(SupportsGet = true)]

public Guid Id { get; set; }

[BindProperty]

public CreateUpdateBookDto Book { get; set; }

private readonly IBookAppService \_bookAppService;

public EditModalModel(IBookAppService bookAppService)

{

\_bookAppService = bookAppService;

}

public async Task OnGetAsync()

{

var bookDto = await \_bookAppService.GetAsync(Id);

Book = ObjectMapper.Map<BookDto, CreateUpdateBookDto>(bookDto);

}

public async Task<IActionResult> OnPostAsync()

{

await \_bookAppService.UpdateAsync(Id, Book);

return NoContent();

}

}

}

````

\* `[HiddenInput]` 和 `[BindProperty]` 是标准的 ASP.NET Core MVC 特性.这里启用 `SupportsGet` 从Http请求的查询字符串参数中获取Id的值.

\* 在 `OnGetAsync` 方法中, 我们从 `BookAppService` 获得 `BookDto` ,并将它映射成DTO对象 `CreateUpdateBookDto`.

\* `OnPostAsync` 方法直接使用 `BookAppService.UpdateAsync` 来更新实体.

### BookDto 到 CreateUpdateBookDto 对象映射

为了执行 `BookDto` 到 `CreateUpdateBookDto` 对象映射,请打开 `Acme.BookStore.Web` 项目中的 `BookStoreWebAutoMapperProfile.cs` 并更改它,如下所示:

````csharp

using AutoMapper;

namespace Acme.BookStore.Web

{

public class BookStoreWebAutoMapperProfile : Profile

{

public BookStoreWebAutoMapperProfile()

{

CreateMap<BookDto, CreateUpdateBookDto>();

}

}

}

````

\* 我们添加了 `CreateMap<BookDto, CreateUpdateBookDto>();` 作为映射定义.

> 请注意,我们在Web层中进行映射定义是一种最佳实践,因为仅在该层中需要它.

### EditModal.cshtml

将 `EditModal.cshtml` 页面内容替换成如下代码:

````html

@page

@using Acme.BookStore.Localization

@using Acme.BookStore.Web.Pages.Books

@using Microsoft.Extensions.Localization

@using Volo.Abp.AspNetCore.Mvc.UI.Bootstrap.TagHelpers.Modal

@model EditModalModel

@inject IStringLocalizer<BookStoreResource> L

@{

Layout = null;

}

<abp-dynamic-form abp-model="Book" asp-page="/Books/EditModal">

<abp-modal>

<abp-modal-header title="@L["Update"].Value"></abp-modal-header>

<abp-modal-body>

<abp-input asp-for="Id" />

<abp-form-content />

</abp-modal-body>

<abp-modal-footer buttons="@(AbpModalButtons.Cancel|AbpModalButtons.Save)"></abp-modal-footer>

</abp-modal>

</abp-dynamic-form>

````

这个页面内容和 `CreateModal.cshtml` 非常相似,除了以下几点:

\* 它包含`id`属性的`abp-input`, 用于存储被编辑书籍的 `id` (它是隐藏的Input)

\* 此页面指定的post地址是`Books/EditModal`.

### 为表格添加 "操作(Actions)" 下拉菜单

我们将为表格每行添加下拉按钮 ("Actions"):

打开 `Pages/Books/Index.js` 页面,并按下方所示修改表格部分的代码:

````js

$(function () {

var l = abp.localization.getResource('BookStore');

var createModal = new abp.ModalManager(abp.appPath + 'Books/CreateModal');

var editModal = new abp.ModalManager(abp.appPath + 'Books/EditModal');

var dataTable = $('#BooksTable').DataTable(

abp.libs.datatables.normalizeConfiguration({

serverSide: true,

paging: true,

order: [[1, "asc"]],

searching: false,

scrollX: true,

ajax: abp.libs.datatables.createAjax(acme.bookStore.books.book.getList),

columnDefs: [

{

title: l('Actions'),

rowAction: {

items:

[

{

text: l('Edit'),

action: function (data) {

editModal.open({ id: data.record.id });

}

}

]

}

},

{

title: l('Name'),

data: "name"

},

{

title: l('Type'),

data: "type",

render: function (data) {

return l('Enum:BookType:' + data);

}

},

{

title: l('PublishDate'),

data: "publishDate",

render: function (data) {

return luxon

.DateTime

.fromISO(data, {

locale: abp.localization.currentCulture.name

}).toLocaleString();

}

},

{

title: l('Price'),

data: "price"

},

{

title: l('CreationTime'), data: "creationTime",

render: function (data) {

return luxon

.DateTime

.fromISO(data, {

locale: abp.localization.currentCulture.name

}).toLocaleString(luxon.DateTime.DATETIME\_SHORT);

}

}

]

})

);

createModal.onResult(function () {

dataTable.ajax.reload();

});

editModal.onResult(function () {

dataTable.ajax.reload();

});

$('#NewBookButton').click(function (e) {

e.preventDefault();

createModal.open();

});

});

````

\* 增加了一个新的 `ModalManager` 名为 `editModal` 打开编辑模态框.

\* 在 `columnDefs` 部分的开头添加了一个新列,用于"*\*Actions\**"下拉按钮.

\* "*\*Edit\**" 动作简单地调用 `editModal.open()` 打开编辑模态框.

\* `editModal.onResult(...)` 当你关闭编程模态框时进行回调刷新数据表格.

你可以运行应用程序,并通过选择一本书的编辑操作编辑任何一本书.

最终的UI看起来如下:

![bookstore-books-table-actions](./images/bookstore-edit-button-2.png)

## 删除书籍

打开 `Pages/book/index.js` 文件,在 `rowAction` `items` 下新增一项:

````js

{

text: l('Delete'),

confirmMessage: function (data) {

return l('BookDeletionConfirmationMessage', data.record.name);

},

action: function (data) {

acme.bookStore.books.book

.delete(data.record.id)

.then(function() {

abp.notify.info(l('SuccessfullyDeleted'));

dataTable.ajax.reload();

});

}

}

````

\* `confirmMessage` 执行 `action` 前向用户进行确认.

\* `acme.bookStore.books.book.delete(...)` 执行一个AJAX请求删除一个book.

\* `abp.notify.info` 执行删除操作后显示一个通知信息.

由于我们使用了两个新的本地化文本(`BookDeletionConfirmationMessage`和`SuccesslyDeleted`),因此你需要将它们添加到本地化文件(`Acme.BookStore.Domain.Shared`项目的`Localization/BookStore`文件夹下的`en.json`):

````json

"BookDeletionConfirmationMessage": "Are you sure to delete the book '{0}'?",

"SuccessfullyDeleted": "Successfully deleted!"

````

> 简体中文翻译请打开`zh-Hans.json`文件 ,并将"Texts"对象中对应的值替换为中文.

`Index.js` 的内容最终如下所示:

````js

$(function () {

var l = abp.localization.getResource('BookStore');

var createModal = new abp.ModalManager(abp.appPath + 'Books/CreateModal');

var editModal = new abp.ModalManager(abp.appPath + 'Books/EditModal');

var dataTable = $('#BooksTable').DataTable(

abp.libs.datatables.normalizeConfiguration({

serverSide: true,

paging: true,

order: [[1, "asc"]],

searching: false,

scrollX: true,

ajax: abp.libs.datatables.createAjax(acme.bookStore.books.book.getList),

columnDefs: [

{

title: l('Actions'),

rowAction: {

items:

[

{

text: l('Edit'),

action: function (data) {

editModal.open({ id: data.record.id });

}

},

{

text: l('Delete'),

confirmMessage: function (data) {

return l(

'BookDeletionConfirmationMessage',

data.record.name

);

},

action: function (data) {

acme.bookStore.books.book

.delete(data.record.id)

.then(function() {

abp.notify.info(

l('SuccessfullyDeleted')

);

dataTable.ajax.reload();

});

}

}

]

}

},

{

title: l('Name'),

data: "name"

},

{

title: l('Type'),

data: "type",

render: function (data) {

return l('Enum:BookType:' + data);

}

},

{

title: l('PublishDate'),

data: "publishDate",

render: function (data) {

return luxon

.DateTime

.fromISO(data, {

locale: abp.localization.currentCulture.name

}).toLocaleString();

}

},

{

title: l('Price'),

data: "price"

},

{

title: l('CreationTime'), data: "creationTime",

render: function (data) {

return luxon

.DateTime

.fromISO(data, {

locale: abp.localization.currentCulture.name

}).toLocaleString(luxon.DateTime.DATETIME\_SHORT);

}

}

]

})

);

createModal.onResult(function () {

dataTable.ajax.reload();

});

editModal.onResult(function () {

dataTable.ajax.reload();

});

$('#NewBookButton').click(function (e) {

e.preventDefault();

createModal.open();

});

});

````

你可以运行程序并尝试删除一本书.

{{end}}

{{if UI == "NG"}}

## 创建新书籍

下面的章节中,你将学习到如何创建一个新的模态窗口新增书籍.

### BookComponent

打开 `/src/app/book/book.component.ts` 使用以下内容替换:

```js

import { ListService, PagedResultDto } from '@abp/ng.core';

import { Component, OnInit } from '@angular/core';

import { BookService, BookDto } from '@proxy/books';

@Component({

selector: 'app-book',

templateUrl: './book.component.html',

styleUrls: ['./book.component.scss'],

providers: [ListService],

})

export class BookComponent implements OnInit {

book = { items: [], totalCount: 0 } as PagedResultDto<BookDto>;

isModalOpen = false; // add this line

constructor(public readonly list: ListService, private bookService: BookService) {}

ngOnInit() {

const bookStreamCreator = (query) => this.bookService.getList(query);

this.list.hookToQuery(bookStreamCreator).subscribe((response) => {

this.book = response;

});

}

// add new method

createBook() {

this.isModalOpen = true;

}

}

```

\* 我们定义了一个名为 `isModalOpen` 的属性和 `createBook` 方法.

打开 `/src/app/book/book.component.html` 做以下更改:

```html

<div class="card">

<div class="card-header">

<div class="row">

<div class="col col-md-6">

<h5 class="card-title">{%{{{ '::Menu:Books' | abpLocalization }}}%}</h5>

</div>

<div class="text-end col col-md-6">

<!-- Add the "new book" button here -->

<div class="text-lg-end pt-2">

<button id="create" class="btn btn-primary" type="button" (click)="createBook()">

<i class="fa fa-plus mr-1"></i>

<span>{%{{{ "::NewBook" | abpLocalization }}}%}</span>

</button>

</div>

</div>

</div>

</div>

<div class="card-body">

<!-- ngx-datatable should be here! -->

</div>

</div>

<!-- Add the modal here -->

<abp-modal [(visible)]="isModalOpen">

<ng-template #abpHeader>

<h3>{%{{{ '::NewBook' | abpLocalization }}}%}</h3>

</ng-template>

<ng-template #abpBody> </ng-template>

<ng-template #abpFooter>

<button type="button" class="btn btn-secondary" abpClose>

{%{{{ '::Close' | abpLocalization }}}%}

</button>

</ng-template>

</abp-modal>

```

\* 添加了 `New book` 按钮到卡片头部.

\* 添加了 `abp-modal` 渲染模态框,允许用户创建新书. `abp-modal` 是显示模态框的预构建组件. 你也可以使用其它方法显示模态框,但 `abp-modal` 提供了一些额外的好处.

你可以打开浏览器,点击**\*\*New book\*\***按钮看到模态框.

![Empty modal for new book](images/bookstore-empty-new-book-modal.png)

### 添加响应式表单

[响应式表单](https://angular.io/guide/reactive-forms) 提供一种模型驱动的方法来处理其值随时间变化的表单输入.

打开 `/src/app/book/book.component.ts` 使用以下内容替换:

```js

import { ListService, PagedResultDto } from '@abp/ng.core';

import { Component, OnInit } from '@angular/core';

import { BookService, BookDto, bookTypeOptions } from '@proxy/books'; // add bookTypeOptions

import { FormGroup, FormBuilder, Validators } from '@angular/forms'; // add this

@Component({

selector: 'app-book',

templateUrl: './book.component.html',

styleUrls: ['./book.component.scss'],

providers: [ListService],

})

export class BookComponent implements OnInit {

book = { items: [], totalCount: 0 } as PagedResultDto<BookDto>;

form: FormGroup; // add this line

// add bookTypes as a list of BookType enum members

bookTypes = bookTypeOptions;

isModalOpen = false;

constructor(

public readonly list: ListService,

private bookService: BookService,

private fb: FormBuilder // inject FormBuilder

) {}

ngOnInit() {

const bookStreamCreator = (query) => this.bookService.getList(query);

this.list.hookToQuery(bookStreamCreator).subscribe((response) => {

this.book = response;

});

}

createBook() {

this.buildForm(); // add this line

this.isModalOpen = true;

}

// add buildForm method

buildForm() {

this.form = this.fb.group({

name: ['', Validators.required],

type: [null, Validators.required],

publishDate: [null, Validators.required],

price: [null, Validators.required],

});

}

// add save method

save() {

if (this.form.invalid) {

return;

}

this.bookService.create(this.form.value).subscribe(() => {

this.isModalOpen = false;

this.form.reset();

this.list.get();

});

}

}

```

\* 从` @angular/forms `导入了 `FormGroup, FormBuilder and Validators`.

\* 添加了 `form: FormGroup` 变量.

\* 添加了 `bookTypes` 属性作为 `BookType` 枚举成员列表. 将在表单选项中使用.

\* 我们注入了 `FormBuilder` 到构造函数. [FormBuilder](https://angular.io/api/forms/FormBuilder) 提供了简便的方法生成表单控件. 它减少了构建复杂表单所需的样板文件的数量.

\* 我们添加了 `buildForm` 方法到文件末尾, 在 `createBook` 方法调用 `buildForm()` 方法.

\* 添加了`save` 方法.

打开 `/src/app/book/book.component.html`,使用以下内容替换 `<ng-template #abpBody> </ng-template>`:

```html

<ng-template #abpBody>

<form [formGroup]="form" (ngSubmit)="save()">

<div class="form-group">

<label for="book-name">Name</label><span> \* </span>

<input type="text" id="book-name" class="form-control" formControlName="name" autofocus />

</div>

<div class="form-group">

<label for="book-price">Price</label><span> \* </span>

<input type="number" id="book-price" class="form-control" formControlName="price" />

</div>

<div class="form-group">

<label for="book-type">Type</label><span> \* </span>

<select class="form-control" id="book-type" formControlName="type">

<option [ngValue]="null">Select a book type</option>

<option [ngValue]="type.value" \*ngFor="let type of bookTypes"> {%{{{ type.key }}}%}</option>

</select>

</div>

<div class="form-group">

<label>Publish date</label><span> \* </span>

<input

#datepicker="ngbDatepicker"

class="form-control"

name="datepicker"

formControlName="publishDate"

ngbDatepicker

(click)="datepicker.toggle()"

/>

</div>

</form>

</ng-template>

```

同时使用下面的代码部分替换 `<ng-template #abpFooter> </ng-template>`:

````html

<ng-template #abpFooter>

<button type="button" class="btn btn-secondary" abpClose>

{%{{{ '::Close' | abpLocalization }}}%}

</button>

<!--added save button-->

<button class="btn btn-primary" (click)="save()" [disabled]="form.invalid">

<i class="fa fa-check mr-1"></i>

{%{{{ '::Save' | abpLocalization }}}%}

</button>

</ng-template>

````

### Datepicker

我们在这个组件中使用了[NgBootstrap datepicker](https://ng-bootstrap.github.io/#/components/datepicker/overview). 因此需要添加与此组件相关的依赖项.

打开 `/src/app/book/book.module.ts` 使用以下内容替换:

```js

import { NgModule } from '@angular/core';

import { SharedModule } from '../shared/shared.module';

import { BookRoutingModule } from './book-routing.module';

import { BookComponent } from './book.component';

import { NgbDatepickerModule } from '@ng-bootstrap/ng-bootstrap'; // add this line

@NgModule({

declarations: [BookComponent],

imports: [

BookRoutingModule,

SharedModule,

NgbDatepickerModule, // add this line

]

})

export class BookModule { }

```

\* 我们导入了 `NgbDatepickerModule` 来使用日期选择器.

打开 `/src/app/book/book.component.ts` 使用以下内容替换:

```js

import { ListService, PagedResultDto } from '@abp/ng.core';

import { Component, OnInit } from '@angular/core';

import { BookService, BookDto, bookTypeOptions } from '@proxy/books';

import { FormGroup, FormBuilder, Validators } from '@angular/forms';

// added this line

import { NgbDateNativeAdapter, NgbDateAdapter } from '@ng-bootstrap/ng-bootstrap';

@Component({

selector: 'app-book',

templateUrl: './book.component.html',

styleUrls: ['./book.component.scss'],

providers: [

ListService,

{ provide: NgbDateAdapter, useClass: NgbDateNativeAdapter } // add this line

],

})

export class BookComponent implements OnInit {

book = { items: [], totalCount: 0 } as PagedResultDto<BookDto>;

form: FormGroup;

bookTypes = bookTypeOptions;

isModalOpen = false;

constructor(

public readonly list: ListService,

private bookService: BookService,

private fb: FormBuilder

) {}

ngOnInit() {

const bookStreamCreator = (query) => this.bookService.getList(query);

this.list.hookToQuery(bookStreamCreator).subscribe((response) => {

this.book = response;

});

}

createBook() {

this.buildForm();

this.isModalOpen = true;

}

buildForm() {

this.form = this.fb.group({

name: ['', Validators.required],

type: [null, Validators.required],

publishDate: [null, Validators.required],

price: [null, Validators.required],

});

}

save() {

if (this.form.invalid) {

return;

}

this.bookService.create(this.form.value).subscribe(() => {

this.isModalOpen = false;

this.form.reset();

this.list.get();

});

}

}

```

\* 导入了 `NgbDateNativeAdapter` 和 `NgbDateAdapter`.

\* 我们添加了一个新的 `NgbDateAdapter` 提供程序,它将Datepicker值转换为 `Date` 类型. 更多详细信息,请参见[datepicker adapters](https://ng-bootstrap.github.io/#/components/datepicker/overview).

现在你可以打开浏览器看到以下变化:

![Save button to the modal](images/bookstore-new-book-form-v2.png)

## 更新书籍

打开 `/src/app/book/book.component.ts` 使用以下内容替换:

```js

import { ListService, PagedResultDto } from '@abp/ng.core';

import { Component, OnInit } from '@angular/core';

import { BookService, BookDto, bookTypeOptions } from '@proxy/books';

import { FormGroup, FormBuilder, Validators } from '@angular/forms';

import { NgbDateNativeAdapter, NgbDateAdapter } from '@ng-bootstrap/ng-bootstrap';

@Component({

selector: 'app-book',

templateUrl: './book.component.html',

styleUrls: ['./book.component.scss'],

providers: [ListService, { provide: NgbDateAdapter, useClass: NgbDateNativeAdapter }],

})

export class BookComponent implements OnInit {

book = { items: [], totalCount: 0 } as PagedResultDto<BookDto>;

selectedBook = {} as BookDto; // declare selectedBook

form: FormGroup;

bookTypes = bookTypeOptions;

isModalOpen = false;

constructor(

public readonly list: ListService,

private bookService: BookService,

private fb: FormBuilder

) {}

ngOnInit() {

const bookStreamCreator = (query) => this.bookService.getList(query);

this.list.hookToQuery(bookStreamCreator).subscribe((response) => {

this.book = response;

});

}

createBook() {

this.selectedBook = {} as BookDto; // reset the selected book

this.buildForm();

this.isModalOpen = true;

}

// Add editBook method

editBook(id: string) {

this.bookService.get(id).subscribe((book) => {

this.selectedBook = book;

this.buildForm();

this.isModalOpen = true;

});

}

buildForm() {

this.form = this.fb.group({

name: [this.selectedBook.name || '', Validators.required],

type: [this.selectedBook.type || null, Validators.required],

publishDate: [

this.selectedBook.publishDate ? new Date(this.selectedBook.publishDate) : null,

Validators.required,

],

price: [this.selectedBook.price || null, Validators.required],

});

}

// change the save method

save() {

if (this.form.invalid) {

return;

}

const request = this.selectedBook.id

? this.bookService.update(this.selectedBook.id, this.form.value)

: this.bookService.create(this.form.value);

request.subscribe(() => {

this.isModalOpen = false;

this.form.reset();

this.list.get();

});

}

}

```

\* 我们声明了类型为 `BookDto` 的 `selectedBook` 变量.

\* 我们添加了 `editBook` 方法, 根据给定书籍 `Id` 设置 `selectedBook` 对象.

\* 我们替换了 `buildForm` 方法使用 `selectedBook` 数据创建表单.

\* 我们替换了 `createBook` 方法,设置 `selectedBook` 为空对象.

\* 我们修改了 `save` 方法,同时处理新建和更新操作.

### 添加 "Actions" 下拉框到表格

打开 `/src/app/book/book.component.html` 在 `ngx-datatable` 第一列添加 `ngx-datatable-column` 定义:

```html

<ngx-datatable-column

[name]="'::Actions' | abpLocalization"

[maxWidth]="150"

[sortable]="false"

>

<ng-template let-row="row" ngx-datatable-cell-template>

<div ngbDropdown container="body" class="d-inline-block">

<button

class="btn btn-primary btn-sm dropdown-toggle"

data-toggle="dropdown"

aria-haspopup="true"

ngbDropdownToggle

>

<i class="fa fa-cog mr-1"></i>{%{{{ '::Actions' | abpLocalization }}}%}

</button>

<div ngbDropdownMenu>

<button ngbDropdownItem (click)="editBook(row.id)">

{%{{{ '::Edit' | abpLocalization }}}%}

</button>

</div>

</div>

</ng-template>

</ngx-datatable-column>

```

在表格的第一列添加了一个 "Actions" 下拉菜单,如下图所示:

![Action buttons](images/bookstore-actions-buttons.png)

同时如下所示更改 `ng-template #abpHeader` 部分:

```html

<ng-template #abpHeader>

<h3>{%{{{ (selectedBook.id ? '::Edit' : '::NewBook' ) | abpLocalization }}}%}</h3>

</ng-template>

```

模板将在标题中显示 **\*\*Edit\*\*** 文本用于编辑记录操作, **\*\*New Book\*\*** 用于添加新记录操作.

## 删除书籍

打开 `/src/app/book/book.component.ts` 注入 `ConfirmationService`.

所示替换构造函数:

```js

// ...

// add new imports

import { ConfirmationService, Confirmation } from '@abp/ng.theme.shared';

//change the constructor

constructor(

public readonly list: ListService,

private bookService: BookService,

private fb: FormBuilder,

private confirmation: ConfirmationService // inject the ConfirmationService

) {}

// Add a delete method

delete(id: string) {

this.confirmation.warn('::AreYouSureToDelete', '::AreYouSure').subscribe((status) => {

if (status === Confirmation.Status.confirm) {

this.bookService.delete(id).subscribe(() => this.list.get());

}

});

}

```

\* 我们引入了 `ConfirmationService`.

\* 我们注入了 `ConfirmationService` 到构造函数.

\* 添加了 `delete` 方法.

> 参阅[确认弹层文档](../UI/Angular/Confirmation-Service)了解该服务的更多信息.

### 添加删除按钮:

打开 `/src/app/book/book.component.html` 修改 `ngbDropdownMenu` 添加删除按钮:

```html

<div ngbDropdownMenu>

<!-- add the Delete button -->

<button ngbDropdownItem (click)="delete(row.id)">

{%{{{ '::Delete' | abpLocalization }}}%}

</button>

</div>

```

最终操作下拉框UI看起来如下:

![bookstore-final-actions-dropdown](images/bookstore-final-actions-dropdown.png)

点击 `delete` 操作调用 `delete` 方法,然后显示一个确认弹层如下图所示.

![bookstore-confirmation-popup](images/bookstore-confirmation-popup.png)

{{end}}

{{if UI == "Blazor" || UI == "BlazorServer"}}

## 创建新书籍

通过本节, 你将会了解如何创建一个模态窗口实现新增书籍的功能. 因为我们已经从 `AbpCrudPageBase` 继承, 所以只需要开发视图部分.

### 添加 "New Button" 按钮

打开 `Books.razor` 替换 `<CardHeader>` 部分为以下代码:

````xml

<CardHeader>

<Row Class="justify-content-between">

<Column ColumnSize="ColumnSize.IsAuto">

<h2>@L["Books"]</h2>

</Column>

<Column ColumnSize="ColumnSize.IsAuto">

<Button Color="Color.Primary"

Clicked="OpenCreateModalAsync">@L["NewBook"]</Button>

</Column>

</Row>

</CardHeader>

````

如下图所示,卡片头 **\*\*右侧\*\*** 添加了 **\*\*New book\*\*** 按钮:

![blazor-add-book-button](images/blazor-add-book-button.png)

现在, 我们可以添加点击按钮后打开的模态窗口了.

### 书籍创建模态窗口

打开 `Books.razor`, 添加以下代码到页面底部:

````xml

<Modal @ref="@CreateModal">

<ModalBackdrop />

<ModalContent IsCentered="true">

<Form>

<ModalHeader>

<ModalTitle>@L["NewBook"]</ModalTitle>

<CloseButton Clicked="CloseCreateModalAsync"/>

</ModalHeader>

<ModalBody>

<Validations @ref="@CreateValidationsRef" Model="@NewEntity" ValidateOnLoad="false">

<Validation MessageLocalizer="@LH.Localize">

<Field>

<FieldLabel>@L["Name"]</FieldLabel>

<TextEdit @bind-Text="@NewEntity.Name">

<Feedback>

<ValidationError/>

</Feedback>

</TextEdit>

</Field>

</Validation>

<Field>

<FieldLabel>@L["Type"]</FieldLabel>

<Select TValue="BookType" @bind-SelectedValue="@NewEntity.Type">

@foreach (int bookTypeValue in Enum.GetValues(typeof(BookType)))

{

<SelectItem TValue="BookType" Value="@((BookType) bookTypeValue)">

@L[$"Enum:BookType.{Enum.GetName((BookType)bookTypeValue)}"]

</SelectItem>

}

</Select>

</Field>

<Field>

<FieldLabel>@L["PublishDate"]</FieldLabel>

<DateEdit TValue="DateTime" @bind-Date="NewEntity.PublishDate"/>

</Field>

<Field>

<FieldLabel>@L["Price"]</FieldLabel>

<NumericEdit TValue="float" @bind-Value="NewEntity.Price"/>

</Field>

</Validations>

</ModalBody>

<ModalFooter>

<Button Color="Color.Secondary"

Clicked="CloseCreateModalAsync">@L["Cancel"]</Button>

<Button Color="Color.Primary"

Type="@ButtonType.Submit"

PreventDefaultOnSubmit="true"

Clicked="CreateEntityAsync">@L["Save"]</Button>

</ModalFooter>

</Form>

</ModalContent>

</Modal>

````

这段代码需要一个服务; 在文件顶部, `@inherits...` 行前, 注入 `AbpBlazorMessageLocalizerHelper<T>`:

````csharp

@inject AbpBlazorMessageLocalizerHelper<BookStoreResource> LH

````

\* 表单实现了验证功能, `AbpBlazorMessageLocalizerHelper` 用于本地化验证消息.

\* `CreateModal` 对象, `CloseCreateModalAsync` 和 `CreateEntityAsync` 方法定义在基类中. 参阅 [Blazorise文档](https://blazorise.com/docs/) 以深入理解 `Modal` 和其它组件.

这就是全部了. 运行应用程序, 尝试添加一本新书.

![blazor-new-book-modal](images/blazor-new-book-modal.png)

## 更新书籍

编辑书籍与新建书籍很类似.

### 操作下拉菜单

打开 `Books.razor` , 在 `DataGridColumns` 中添加以下 `DataGridEntityActionsColumn` 作为第一项:

````xml

<DataGridEntityActionsColumn TItem="BookDto" @ref="@EntityActionsColumn">

<DisplayTemplate>

<EntityActions TItem="BookDto" EntityActionsColumn="@EntityActionsColumn">

<EntityAction TItem="BookDto"

Text="@L["Edit"]"

Clicked="() => OpenEditModalAsync(context)" />

</EntityActions>

</DisplayTemplate>

</DataGridEntityActionsColumn>

````

\* `OpenEditModalAsync` 定义在基类中, 它接收实体(书籍)参数, 编辑这个实体.

`DataGridEntityActionsColumn` 组件用于显示 `DataGrid` 每一行中的"操作" 下拉菜单. 如果其中只有唯一的操作, `DataGridEntityActionsColumn` 显示 **\*\*唯一按钮\*\***, 而不是下拉菜单.

![blazor-edit-book-action](images/blazor-edit-book-action-2.png)

### 编辑模态窗口

我们现在可以定义一个模态窗口编辑书籍. 加入下面的代码到 `Books.razor` 页面的底部:

````xml

<Modal @ref="@EditModal">

<ModalBackdrop />

<ModalContent IsCentered="true">

<Form>

<ModalHeader>

<ModalTitle>@EditingEntity.Name</ModalTitle>

<CloseButton Clicked="CloseEditModalAsync"/>

</ModalHeader>

<ModalBody>

<Validations @ref="@EditValidationsRef" Model="@NewEntity" ValidateOnLoad="false">

<Validation MessageLocalizer="@LH.Localize">

<Field>

<FieldLabel>@L["Name"]</FieldLabel>

<TextEdit @bind-Text="@EditingEntity.Name">

<Feedback>

<ValidationError/>

</Feedback>

</TextEdit>

</Field>

</Validation>

<Field>

<FieldLabel>@L["Type"]</FieldLabel>

<Select TValue="BookType" @bind-SelectedValue="@EditingEntity.Type">

@foreach (int bookTypeValue in Enum.GetValues(typeof(BookType)))

{

<SelectItem TValue="BookType" Value="@((BookType) bookTypeValue)">

@L[$"Enum:BookType.{Enum.GetName((BookType)bookTypeValue)}"]

</SelectItem>

}

</Select>

</Field>

<Field>

<FieldLabel>@L["PublishDate"]</FieldLabel>

<DateEdit TValue="DateTime" @bind-Date="EditingEntity.PublishDate"/>

</Field>

<Field>

<FieldLabel>@L["Price"]</FieldLabel>

<NumericEdit TValue="float" @bind-Value="EditingEntity.Price"/>

</Field>

</Validations>

</ModalBody>

<ModalFooter>

<Button Color="Color.Secondary"

Clicked="CloseEditModalAsync">@L["Cancel"]</Button>

<Button Color="Color.Primary"

Type="@ButtonType.Submit"

PreventDefaultOnSubmit="true"

Clicked="UpdateEntityAsync">@L["Save"]</Button>

</ModalFooter>

</Form>

</ModalContent>

</Modal>

````

### AutoMapper 配置

基类 `AbpCrudPageBase` 使用 [对象到对象映射](../Object-To-Object-Mapping.md) 系统将 `BookDto` 对象转化为`CreateUpdateBookDto` 对象. 因此, 我们需要定义映射.

打开 `Acme.BookStore.Blazor` 项目中的 `BookStoreBlazorAutoMapperProfile `, 替换成以下内容:

````csharp

using Acme.BookStore.Books;

using AutoMapper;

namespace Acme.BookStore.Blazor

{

public class BookStoreBlazorAutoMapperProfile : Profile

{

public BookStoreBlazorAutoMapperProfile()

{

CreateMap<BookDto, CreateUpdateBookDto>();

}

}

}

````

\* `CreateMap<BookDto, CreateUpdateBookDto>();` 行用于定义映射.

### 测试编辑模态窗口

你可以运行程序并尝试编辑一本书.

![blazor-edit-book-modal](images/blazor-edit-book-modal.png)

> 提示: 尝试保留 *\*Name\** 字段为空并提交表单, 将显示验证错误消息.

## 删除书籍

打开 `Books.razor` 页面, 在 `EntityActions` 中的"编辑" 操作下面加入以下的 `EntityAction`:

````xml

<EntityAction TItem="BookDto"

Text="@L["Delete"]"

Clicked="() => DeleteEntityAsync(context)"

ConfirmationMessage="() => GetDeleteConfirmationMessage(context)" />

````

\* `DeleteEntityAsync` 定义在基类中. 通过向服务器发起请求删除实体.

\* `ConfirmationMessage` 执行操作前显示确认消息的回调函数.

\* `GetDeleteConfirmationMessage` 定义在基类中. 你可以覆写这个方法 (或传递其它值给 `ConfirmationMessage` 参数) 以定制本地化消息.

因为"操作" 按钮现在有了两个操作, 变成了下拉菜单:

![blazor-edit-book-action](images/blazor-delete-book-action.png)

运行程序并尝试删除一本书.

## 完整的 CRUD UI 代码

下面是完整的创建图书管理CRUD页面的代码, 这些代码在上面是分成两部分开发的:

````xml

@page "/books"

@using Volo.Abp.Application.Dtos

@using Acme.BookStore.Books

@using Acme.BookStore.Localization

@using Microsoft.Extensions.Localization

@using Volo.Abp.AspNetCore.Components.Web

@inject IStringLocalizer<BookStoreResource> L

@inject AbpBlazorMessageLocalizerHelper<BookStoreResource> LH

@inherits AbpCrudPageBase<IBookAppService, BookDto, Guid, PagedAndSortedResultRequestDto, CreateUpdateBookDto>

<Card>

<CardHeader>

<Row Class="justify-content-between">

<Column ColumnSize="ColumnSize.IsAuto">

<h2>@L["Books"]</h2>

</Column>

<Column ColumnSize="ColumnSize.IsAuto">

<Button Color="Color.Primary"

Clicked="OpenCreateModalAsync">@L["NewBook"]</Button>

</Column>

</Row>

</CardHeader>

<CardBody>

<DataGrid TItem="BookDto"

Data="Entities"

ReadData="OnDataGridReadAsync"

CurrentPage="CurrentPage"

TotalItems="TotalCount"

ShowPager="true"

PageSize="PageSize">

<DataGridColumns>

<DataGridEntityActionsColumn TItem="BookDto" @ref="@EntityActionsColumn">

<DisplayTemplate>

<EntityActions TItem="BookDto" EntityActionsColumn="@EntityActionsColumn">

<EntityAction TItem="BookDto"

Text="@L["Edit"]"

Clicked="() => OpenEditModalAsync(context)" />

<EntityAction TItem="BookDto"

Text="@L["Delete"]"

Clicked="() => DeleteEntityAsync(context)"

ConfirmationMessage="()=>GetDeleteConfirmationMessage(context)" />

</EntityActions>

</DisplayTemplate>

</DataGridEntityActionsColumn>

<DataGridColumn TItem="BookDto"

Field="@nameof(BookDto.Name)"

Caption="@L["Name"]"></DataGridColumn>

<DataGridColumn TItem="BookDto"

Field="@nameof(BookDto.Type)"

Caption="@L["Type"]">

<DisplayTemplate>

@L[$"Enum:BookType.{Enum.GetName(context.Type)}"]

</DisplayTemplate>

</DataGridColumn>

<DataGridColumn TItem="BookDto"

Field="@nameof(BookDto.PublishDate)"

Caption="@L["PublishDate"]">

<DisplayTemplate>

@context.PublishDate.ToShortDateString()

</DisplayTemplate>

</DataGridColumn>

<DataGridColumn TItem="BookDto"

Field="@nameof(BookDto.Price)"

Caption="@L["Price"]">

</DataGridColumn>

<DataGridColumn TItem="BookDto"

Field="@nameof(BookDto.CreationTime)"

Caption="@L["CreationTime"]">

<DisplayTemplate>

@context.CreationTime.ToLongDateString()

</DisplayTemplate>

</DataGridColumn>

</DataGridColumns>

</DataGrid>

</CardBody>

</Card>

<Modal @ref="@CreateModal">

<ModalBackdrop />

<ModalContent IsCentered="true">

<Form>

<ModalHeader>

<ModalTitle>@L["NewBook"]</ModalTitle>

<CloseButton Clicked="CloseCreateModalAsync"/>

</ModalHeader>

<ModalBody>

<Validations @ref="@CreateValidationsRef" Model="@NewEntity" ValidateOnLoad="false">

<Validation MessageLocalizer="@LH.Localize">

<Field>

<FieldLabel>@L["Name"]</FieldLabel>

<TextEdit @bind-Text="@NewEntity.Name">

<Feedback>

<ValidationError/>

</Feedback>

</TextEdit>

</Field>

</Validation>

<Field>

<FieldLabel>@L["Type"]</FieldLabel>

<Select TValue="BookType" @bind-SelectedValue="@NewEntity.Type">

@foreach (int bookTypeValue in Enum.GetValues(typeof(BookType)))

{

<SelectItem TValue="BookType" Value="@((BookType) bookTypeValue)">

@L[$"Enum:BookType.{Enum.GetName((BookType)bookTypeValue)}"]

</SelectItem>

}

</Select>

</Field>

<Field>

<FieldLabel>@L["PublishDate"]</FieldLabel>

<DateEdit TValue="DateTime" @bind-Date="NewEntity.PublishDate"/>

</Field>

<Field>

<FieldLabel>@L["Price"]</FieldLabel>

<NumericEdit TValue="float" @bind-Value="NewEntity.Price"/>

</Field>

</Validations>

</ModalBody>

<ModalFooter>

<Button Color="Color.Secondary"

Clicked="CloseCreateModalAsync">@L["Cancel"]</Button>

<Button Color="Color.Primary"

Type="@ButtonType.Submit"

PreventDefaultOnSubmit="true"

Clicked="CreateEntityAsync">@L["Save"]</Button>

</ModalFooter>

</Form>

</ModalContent>

</Modal>

<Modal @ref="@EditModal">

<ModalBackdrop />

<ModalContent IsCentered="true">

<Form>

<ModalHeader>

<ModalTitle>@EditingEntity.Name</ModalTitle>

<CloseButton Clicked="CloseEditModalAsync"/>

</ModalHeader>

<ModalBody>

<Validations @ref="@EditValidationsRef" Model="@NewEntity" ValidateOnLoad="false">

<Validation MessageLocalizer="@LH.Localize">

<Field>

<FieldLabel>@L["Name"]</FieldLabel>

<TextEdit @bind-Text="@EditingEntity.Name">

<Feedback>

<ValidationError/>

</Feedback>

</TextEdit>

</Field>

</Validation>

<Field>

<FieldLabel>@L["Type"]</FieldLabel>

<Select TValue="BookType" @bind-SelectedValue="@EditingEntity.Type">

@foreach (int bookTypeValue in Enum.GetValues(typeof(BookType)))

{

<SelectItem TValue="BookType" Value="@((BookType) bookTypeValue)">

@L[$"Enum:BookType.{Enum.GetName((BookType)bookTypeValue)}"]

</SelectItem>

}

</Select>

</Field>

<Field>

<FieldLabel>@L["PublishDate"]</FieldLabel>

<DateEdit TValue="DateTime" @bind-Date="EditingEntity.PublishDate"/>

</Field>

<Field>

<FieldLabel>@L["Price"]</FieldLabel>

<NumericEdit TValue="float" @bind-Value="EditingEntity.Price"/>

</Field>

</Validations>

</ModalBody>

<ModalFooter>

<Button Color="Color.Secondary"

Clicked="CloseEditModalAsync">@L["Cancel"]</Button>

<Button Color="Color.Primary"

Type="@ButtonType.Submit"

PreventDefaultOnSubmit="true"

Clicked="UpdateEntityAsync">@L["Save"]</Button>

</ModalFooter>

</Form>

</ModalContent>

</Modal>

````

{{end}}

## 下一章

查看本教程的[下一章](Part-4.md).

集成测试

# Web应用程序开发教程 - 第四章: 集成测试

````json

//[doc-params]

{

"UI": ["MVC","Blazor","BlazorServer","NG"],

"DB": ["EF","Mongo"]

}

````

## 关于本教程

在本系列教程中, 你将构建一个名为 `Acme.BookStore` 的用于管理书籍及其作者列表的基于ABP的应用程序. 它是使用以下技术开发的:

\* **\*\*{{DB\_Value}}\*\*** 做为ORM提供程序.

\* **\*\*{{UI\_Value}}\*\*** 做为UI框架.

本教程分为以下部分:

- [Part 1: 创建服务端](Part-1.md)

- [Part 2: 图书列表页面](Part-2.md)

- [Part 3: 创建,更新和删除图书](Part-3.md)

- **\*\*Part 4: 集成测试 (本章)\*\***

- [Part 5: 授权](Part-5.md)

- [Part 6: 作者: 领域层](Part-6.md)

- [Part 7: 作者: 数据库集成](Part-7.md)

- [Part 8: 作者: 应用服务层](Part-8.md)

- [Part 9: 作者: 用户页面](Part-9.md)

- [Part 10: 图书到作者的关系](Part-10.md)

## 下载源码

本教程根据你的**\*\*UI\*\*** 和 **\*\*数据库\*\***偏好有多个版本,我们准备了几种可供下载的源码组合:

\* [MVC (Razor Pages) UI 与 EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Mvc-EfCore)

\* [Blazor UI 与 EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Blazor-EfCore)

\* [Angular UI 与 MongoDB](https://github.com/abpframework/abp-samples/tree/master/BookStore-Angular-MongoDb)

> 如果你在Windows中遇到 "文件名太长" or "解压错误", 很可能与Windows最大文件路径限制有关. Windows文件路径的最大长度为250字符. 为了解决这个问题,参阅 [在Windows 10中启用长路径](https://docs.microsoft.com/en-us/windows/win32/fileio/maximum-file-path-limitation?tabs=cmd#enable-long-paths-in-windows-10-version-1607-and-later).

> 如果你遇到与Git相关的长路径错误, 尝试使用下面的命令在Windows中启用长路径. 参阅 https://github.com/msysgit/msysgit/wiki/Git-cannot-create-a-file-or-directory-with-a-long-path

> `git config --system core.longpaths true`

{{if UI == "MVC" && DB == "EF"}}

### 视频教程

本章也被录制为视频教程 **\*\***<a href="https://www.youtube.com/watch?v=aidRB4YFDLM&list=PLsNclT2aHJcPNaCf7Io3DbMN6yAk\_DgWJ&index=4" target="\_blank">**发布在YouTube**</a>**\*\***.

{{end}}

## 解决方案中的测试项目

这一部分涵盖了 **\*\*服务器端\*\*** 测试. 解决方案中有多个测试项目:

![bookstore-test-projects-v2](./images/bookstore-test-projects-mvc.png)

> 根据你选择的UI和数据库, 测试项目略微有所不同. 例如, 如果选择MongoDB, 那么 `Acme.BookStore.EntityFrameworkCore.Tests` 会变为 `Acme.BookStore.MongoDB.Tests`.

每个项目用于测试相关的应用程序项目.测试项目使用以下库进行测试:

\* [xunit](https://xunit.github.io/) 作为主测试框架.

\* [Shoudly](http://shouldly.readthedocs.io/en/latest/) 作为断言库.

\* [NSubstitute](http://nsubstitute.github.io/) 作为模拟库.

{{if DB=="EF"}}

> 测试项目配置为使用 **\*\*SQLite内存\*\*** 作为数据库. 创建一个单独的数据库实例并使用[数据种子系统](../Data-Seeding.md)初始化种子数据,为每个测试准备一个新的数据库.

{{else if DB=="Mongo"}}

> \*\*[Mongo2Go](https://github.com/Mongo2Go/Mongo2Go)\*\*库用于模拟MongoDB数据库. 创建一个单独的数据库实例并使用[数据种子系统](../Data-Seeding.md)初始化种子数据,为每个测试准备一个新的数据库.

{{end}}

## 添加测试数据

如果你已经按照[第一部分](Part-1.md)中的描述创建了数据种子贡献者,则相同的数据也在测试中可用. 因此你可以跳过此部分. 如果你尚未创建种子贡献者,可以使用 `BookStoreTestDataSeedContributor` 来为要在以下测试中使用的相同数据提供种子.

## 测试 BookAppService

在 `Acme.BookStore.Application.Tests` 项目的 `Books` 命名空间(文件夹)中创建一个名叫 `BookAppService\_Tests` 的测试类:

````csharp

using System;

using System.Linq;

using System.Threading.Tasks;

using Shouldly;

using Volo.Abp.Application.Dtos;

using Volo.Abp.Validation;

using Xunit;

namespace Acme.BookStore.Books

{ {{if DB=="Mongo"}}

[Collection(BookStoreTestConsts.CollectionDefinitionName)]{{end}}

public class BookAppService\_Tests : BookStoreApplicationTestBase

{

private readonly IBookAppService \_bookAppService;

public BookAppService\_Tests()

{

\_bookAppService = GetRequiredService<IBookAppService>();

}

[Fact]

public async Task Should\_Get\_List\_Of\_Books()

{

//Act

var result = await \_bookAppService.GetListAsync(

new PagedAndSortedResultRequestDto()

);

//Assert

result.TotalCount.ShouldBeGreaterThan(0);

result.Items.ShouldContain(b => b.Name == "1984");

}

}

}

````

\* 测试方法 `Should\_Get\_List\_Of\_Books` 直接使用 `BookAppService.GetListAsync` 方法来获取用户列表,并执行检查.

\* 我们可以安全地检查 "1984" 这本书的名称,因为我们知道这本书可以在数据库中找到,我们已将其添加到种子数据中.

新增测试方法,用以测试创建一个**\*\*合法\*\***book实体的场景:

````csharp

[Fact]

public async Task Should\_Create\_A\_Valid\_Book()

{

//Act

var result = await \_bookAppService.CreateAsync(

new CreateUpdateBookDto

{

Name = "New test book 42",

Price = 10,

PublishDate = DateTime.Now,

Type = BookType.ScienceFiction

}

);

//Assert

result.Id.ShouldNotBe(Guid.Empty);

result.Name.ShouldBe("New test book 42");

}

````

新增测试方法,用以测试创建一个非法book实体失败的场景:

````csharp

[Fact]

public async Task Should\_Not\_Create\_A\_Book\_Without\_Name()

{

var exception = await Assert.ThrowsAsync<AbpValidationException>(async () =>

{

await \_bookAppService.CreateAsync(

new CreateUpdateBookDto

{

Name = "",

Price = 10,

PublishDate = DateTime.Now,

Type = BookType.ScienceFiction

}

);

});

exception.ValidationErrors

.ShouldContain(err => err.MemberNames.Any(mem => mem == "Name"));

}

````

\* 由于 `Name` 是空值, ABP 抛出一个 `AbpValidationException` 异常.

最终的测试类如下所示:

````csharp

using System;

using System.Linq;

using System.Threading.Tasks;

using Shouldly;

using Volo.Abp.Application.Dtos;

using Volo.Abp.Validation;

using Xunit;

namespace Acme.BookStore.Books

{ {{if DB=="Mongo"}}

[Collection(BookStoreTestConsts.CollectionDefinitionName)]{{end}}

public class BookAppService\_Tests : BookStoreApplicationTestBase

{

private readonly IBookAppService \_bookAppService;

public BookAppService\_Tests()

{

\_bookAppService = GetRequiredService<IBookAppService>();

}

[Fact]

public async Task Should\_Get\_List\_Of\_Books()

{

//Act

var result = await \_bookAppService.GetListAsync(

new PagedAndSortedResultRequestDto()

);

//Assert

result.TotalCount.ShouldBeGreaterThan(0);

result.Items.ShouldContain(b => b.Name == "1984");

}

[Fact]

public async Task Should\_Create\_A\_Valid\_Book()

{

//Act

var result = await \_bookAppService.CreateAsync(

new CreateUpdateBookDto

{

Name = "New test book 42",

Price = 10,

PublishDate = DateTime.Now,

Type = BookType.ScienceFiction

}

);

//Assert

result.Id.ShouldNotBe(Guid.Empty);

result.Name.ShouldBe("New test book 42");

}

[Fact]

public async Task Should\_Not\_Create\_A\_Book\_Without\_Name()

{

var exception = await Assert.ThrowsAsync<AbpValidationException>(async () =>

{

await \_bookAppService.CreateAsync(

new CreateUpdateBookDto

{

Name = "",

Price = 10,

PublishDate = DateTime.Now,

Type = BookType.ScienceFiction

}

);

});

exception.ValidationErrors

.ShouldContain(err => err.MemberNames.Any(mem => mem == "Name"));

}

}

}

````

打开**\*\*测试资源管理器\*\***(测试 -> Windows -> 测试资源管理器)并**\*\*执行所有\*\***测试:

![bookstore-appservice-tests](./images/bookstore-appservice-tests.png)

恭喜你, **\*\*绿色图标\*\***表示测试已成功通过!

## 下一章

查看本教程的[下一章](Part-5.md).

授权

# Web应用程序开发教程 - 第五章: 授权

````json

//[doc-params]

{

"UI": ["MVC","Blazor","BlazorServer","NG"],

"DB": ["EF","Mongo"]

}

````

## 关于本教程

在本系列教程中, 你将构建一个名为 `Acme.BookStore` 的用于管理书籍及其作者列表的基于ABP的应用程序. 它是使用以下技术开发的:

\* **\*\*{{DB\_Value}}\*\*** 做为ORM提供程序.

\* **\*\*{{UI\_Value}}\*\*** 做为UI框架.

本教程分为以下部分:

- [Part 1: 创建服务端](Part-1.md)

- [Part 2: 图书列表页面](Part-2.md)

- [Part 3: 创建,更新和删除图书](Part-2.md)

- [Part 4: 集成测试](Part-4.md)

- **\*\*Part 5: 授权 (本章)\*\***

- [Part 6: 作者: 领域层](Part-6.md)

- [Part 7: 作者: 数据库集成](Part-7.md)

- [Part 8: 作者: 应用服务层](Part-8.md)

- [Part 9: 作者: 用户页面](Part-9.md)

- [Part 10: 图书到作者的关系](Part-10.md)

## 下载源码

本教程根据你的**\*\*UI\*\*** 和 **\*\*数据库\*\***偏好有多个版本,我们准备了几种可供下载的源码组合:

\* [MVC (Razor Pages) UI 与 EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Mvc-EfCore)

\* [Blazor UI 与 EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Blazor-EfCore)

\* [Angular UI 与 MongoDB](https://github.com/abpframework/abp-samples/tree/master/BookStore-Angular-MongoDb)

> 如果你在Windows中遇到 "文件名太长" 或 "解压错误", 很可能与Windows最大文件路径限制有关. Windows文件路径的最大长度为250字符. 为了解决这个问题,参阅 [在Windows 10中启用长路径](https://docs.microsoft.com/zh-cn/windows/win32/fileio/maximum-file-path-limitation?tabs=cmd#enable-long-paths-in-windows-10-version-1607-and-later).

> 如果你遇到与Git相关的长路径错误, 尝试使用下面的命令在Windows中启用长路径. 参阅 https://github.com/msysgit/msysgit/wiki/Git-cannot-create-a-file-or-directory-with-a-long-path

> `git config --system core.longpaths true`

{{if UI == "MVC" && DB == "EF"}}

### 视频教程

本章也被录制为视频教程 **\*\***<a href="https://www.youtube.com/watch?v=1WsfMITN\_Jk&list=PLsNclT2aHJcPNaCf7Io3DbMN6yAk\_DgWJ&index=5" target="\_blank">**发布在YouTube**</a>**\*\***.

{{end}}

## 权限

ABP框架提供了一个基于ASP.NET Core[授权基础架构](https://docs.microsoft.com/zh-cn/aspnet/core/security/authorization/introduction)的[授权系统](../Authorization.md). 基于标准授权基础架构的一个主要功能是添加了 **\*\*权限系统\*\***, 这个系统允许定义权限并且根据角色, 用户或客户端启用/禁用权限.

### 权限名称

权限必须有唯一的名称 (一个 `字符串`). 最好的方法是把它定义为一个 `常量`, 这样我们就可以重用这个权限名称了.

打开 `Acme.BookStore.Application.Contracts` 项目中的 `BookStorePermissions` 类 (位于 `Permissions` 文件夹) 并替换为以下代码:

````csharp

namespace Acme.BookStore.Permissions

{

public static class BookStorePermissions

{

public const string GroupName = "BookStore";

public static class Books

{

public const string Default = GroupName + ".Books";

public const string Create = Default + ".Create";

public const string Edit = Default + ".Edit";

public const string Delete = Default + ".Delete";

}

}

}

````

权限名称具有层次结构. 例如, "创建图书" 权限被定义为 `BookStore.Books.Create`. ABP不强制必须如此, 但这是一种有益的做法.

### 权限定义

在使用权限前必须定义它们.

打开 `Acme.BookStore.Application.Contracts` 项目中的 `BookStorePermissionDefinitionProvider` 类 (位于 `Permissions` 文件夹) 并替换为以下代码:

````csharp

using Acme.BookStore.Localization;

using Volo.Abp.Authorization.Permissions;

using Volo.Abp.Localization;

namespace Acme.BookStore.Permissions

{

public class BookStorePermissionDefinitionProvider : PermissionDefinitionProvider

{

public override void Define(IPermissionDefinitionContext context)

{

var bookStoreGroup = context.AddGroup(BookStorePermissions.GroupName, L("Permission:BookStore"));

var booksPermission = bookStoreGroup.AddPermission(BookStorePermissions.Books.Default, L("Permission:Books"));

booksPermission.AddChild(BookStorePermissions.Books.Create, L("Permission:Books.Create"));

booksPermission.AddChild(BookStorePermissions.Books.Edit, L("Permission:Books.Edit"));

booksPermission.AddChild(BookStorePermissions.Books.Delete, L("Permission:Books.Delete"));

}

private static LocalizableString L(string name)

{

return LocalizableString.Create<BookStoreResource>(name);

}

}

}

````

这个类定义了一个 **\*\*权限组\*\*** (在UI上分组权限, 下文会看到) 和 权限组中的**\*\*4个权限\*\***. 而且, **\*\*创建\*\***, **\*\*编辑\*\*** 和 **\*\*删除\*\*** 是 `BookStorePermissions.Books.Default` 权限的子权限. **\*\*仅当父权限被选择\*\***时, 子权限才能被选择.

最后, 编辑本地化文件 (`Acme.BookStore.Domain.Shared` 项目的 `Localization/BookStore` 文件夹中的 `en.json`) 定义上面使用的本地化键:

````json

"Permission:BookStore": "Book Store",

"Permission:Books": "Book Management",

"Permission:Books.Create": "Creating new books",

"Permission:Books.Edit": "Editing the books",

"Permission:Books.Delete": "Deleting the books"

````

> 本地化键名可以是任意的, 并没有强制的规则. 但我们推荐上面使用的约定. 简体中文翻译请打开`zh-Hans.json`文件 ,并将"Texts"对象中对应的值替换为中文.

### 权限管理界面

完成权限定义后, 可以在**\*\*权限管理模态窗口\*\***看到它们.

在*\*管理 -> Identity -> 角色\** 页面, 选择admin角色的 *\*权限\** 操作, 打开权限管理模态窗口:

![bookstore-permissions-ui](images/bookstore-permissions-ui.png)

授予你希望的权限并保存.

> **\*\*提示\*\***: 如果运行 `Acme.BookStore.DbMigrator` 应用程序, 新权限会被自动授予admin.

## 授权

现在, 你可以使用权限授权图书管理.

### 应用层 和 HTTP API

打开 the `BookAppService` 类, 设置策略名称为上面定义的权限名称.

````csharp

using System;

using Acme.BookStore.Permissions;

using Volo.Abp.Application.Dtos;

using Volo.Abp.Application.Services;

using Volo.Abp.Domain.Repositories;

namespace Acme.BookStore.Books

{

public class BookAppService :

CrudAppService<

Book, //The Book entity

BookDto, //Used to show books

Guid, //Primary key of the book entity

PagedAndSortedResultRequestDto, //Used for paging/sorting

CreateUpdateBookDto>, //Used to create/update a book

IBookAppService //implement the IBookAppService

{

public BookAppService(IRepository<Book, Guid> repository)

: base(repository)

{

GetPolicyName = BookStorePermissions.Books.Default;

GetListPolicyName = BookStorePermissions.Books.Default;

CreatePolicyName = BookStorePermissions.Books.Create;

UpdatePolicyName = BookStorePermissions.Books.Edit;

DeletePolicyName = BookStorePermissions.Books.Delete;

}

}

}

````

加入代码到构造器. 基类中的 `CrudAppService` 自动在CRUD操作中使用这些权限. 这不仅实现了 **\*\*应用服务\*\*** 的安全性, 也实现了 **\*\*HTTP API\*\*** 安全性, 因为如前解释的, HTTP API 自动使用这些服务. (参阅 [自动 API controllers](../API/Auto-API-Controllers.md)).

> 在稍后开发作者管理功能时, 你将会看到声明式授权, 使用 `[Authorize(...)]` 特性.

{{if UI == "MVC"}}

### Razor 页面

虽然安全的 HTTP API和应用服务阻止未授权用户使用服务, 但他们依然可以导航到图书管理页面. 虽然当页面发起第一个访问服务器的AJAX请求时会收到授权异常, 但为了更好的用户体验和安全性, 我们应该对页面进行授权.

打开 `BookStoreWebModule` 在 `ConfigureServices` 方法中加入以下代码:

````csharp

Configure<RazorPagesOptions>(options =>

{

options.Conventions.AuthorizePage("/Books/Index", BookStorePermissions.Books.Default);

options.Conventions.AuthorizePage("/Books/CreateModal", BookStorePermissions.Books.Create);

options.Conventions.AuthorizePage("/Books/EditModal", BookStorePermissions.Books.Edit);

});

````

现在未授权用户会被重定向至**\*\*登录页面\*\***.

#### 隐藏新建图书按钮

图书管理页面有一个 *\*新建图书\** 按钮, 当用户没有 *\*图书新建\** 权限时就不可见的.

![bookstore-new-book-button-small](images/bookstore-new-book-button-small.png)

打开 `Pages/Books/Index.cshtml` 文件, 替换内容为以下代码:

````html

@page

@using Acme.BookStore.Localization

@using Acme.BookStore.Permissions

@using Acme.BookStore.Web.Pages.Books

@using Microsoft.AspNetCore.Authorization

@using Microsoft.Extensions.Localization

@model IndexModel

@inject IStringLocalizer<BookStoreResource> L

@inject IAuthorizationService AuthorizationService

@section scripts

{

<abp-script src="/Pages/Books/Index.js"/>

}

<abp-card>

<abp-card-header>

<abp-row>

<abp-column size-md="\_6">

<abp-card-title>@L["Books"]</abp-card-title>

</abp-column>

<abp-column size-md="\_6" class="text-right">

@if (await AuthorizationService.IsGrantedAsync(BookStorePermissions.Books.Create))

{

<abp-button id="NewBookButton"

text="@L["NewBook"].Value"

icon="plus"

button-type="Primary"/>

}

</abp-column>

</abp-row>

</abp-card-header>

<abp-card-body>

<abp-table striped-rows="true" id="BooksTable"></abp-table>

</abp-card-body>

</abp-card>

````

\* 加入 `@inject IAuthorizationService AuthorizationService` 以访问授权服务.

\* 使用 `@if (await AuthorizationService.IsGrantedAsync(BookStorePermissions.Books.Create))` 检查图书创建权限, 条件显示 *\*新建图书\** 按钮.

### JavaScript端

图书管理页面中的图书表格每行都有操作按钮. 操作按钮包括 *\*编辑\** 和 *\*删除\** 操作:

![bookstore-edit-delete-actions](images/bookstore-edit-delete-actions.png)

如果用户没有权限, 应该隐藏相关的操作. 表格行中的操作有一个 `visible` 属性, 可以设置为 `false` 隐藏操作项.

打开 `Acme.BookStore.Web` 项目中的 `Pages/Books/Index.js`, 为 `编辑` 操作加入 `visible` 属性:

````js

{

text: l('Edit'),

visible: abp.auth.isGranted('BookStore.Books.Edit'), //CHECK for the PERMISSION

action: function (data) {

editModal.open({ id: data.record.id });

}

}

````

对 `Delete` 操作进行同样的操作:

````js

visible: abp.auth.isGranted('BookStore.Books.Delete')

````

\* `abp.auth.isGranted(...)` 检查前面定义的权限.

\* `visible` 也可以是一个返回 `bool` 值的函数. 这个函数可以稍后根据某些条件计算.

### 菜单项

即使我们在图书管理页面的所有层都控制了权限, 应用程序的主菜单依然会显示. 我们应该隐藏用户没有权限的菜单项.

打开 `BookStoreMenuContributor` 类, 找到下面的代码:

````csharp

context.Menu.AddItem(

new ApplicationMenuItem(

"BooksStore",

l["Menu:BookStore"],

icon: "fa fa-book"

).AddItem(

new ApplicationMenuItem(

"BooksStore.Books",

l["Menu:Books"],

url: "/Books"

)

)

);

````

替换为以下代码:

````csharp

var bookStoreMenu = new ApplicationMenuItem(

"BooksStore",

l["Menu:BookStore"],

icon: "fa fa-book"

);

context.Menu.AddItem(bookStoreMenu);

//CHECK the PERMISSION

if (await context.IsGrantedAsync(BookStorePermissions.Books.Default))

{

bookStoreMenu.AddItem(new ApplicationMenuItem(

"BooksStore.Books",

l["Menu:Books"],

url: "/Books"

));

}

````

你需要为 `ConfigureMenuAsync` 方法加入 `async` 关键字, 并重新组织返回值. 最终的 `BookStoreMenuContributor` 类应该如下:

````csharp

using System.Threading.Tasks;

using Microsoft.Extensions.DependencyInjection;

using Microsoft.Extensions.Localization;

using Acme.BookStore.Localization;

using Acme.BookStore.MultiTenancy;

using Acme.BookStore.Permissions;

using Volo.Abp.TenantManagement.Web.Navigation;

using Volo.Abp.UI.Navigation;

namespace Acme.BookStore.Web.Menus

{

public class BookStoreMenuContributor : IMenuContributor

{

public async Task ConfigureMenuAsync(MenuConfigurationContext context)

{

if (context.Menu.Name == StandardMenus.Main)

{

await ConfigureMainMenuAsync(context);

}

}

private async Task ConfigureMainMenuAsync(MenuConfigurationContext context)

{

if (!MultiTenancyConsts.IsEnabled)

{

var administration = context.Menu.GetAdministration();

administration.TryRemoveMenuItem(TenantManagementMenuNames.GroupName);

}

var l = context.GetLocalizer<BookStoreResource>();

context.Menu.Items.Insert(0, new ApplicationMenuItem("BookStore.Home", l["Menu:Home"], "~/"));

var bookStoreMenu = new ApplicationMenuItem(

"BooksStore",

l["Menu:BookStore"],

icon: "fa fa-book"

);

context.Menu.AddItem(bookStoreMenu);

//CHECK the PERMISSION

if (await context.IsGrantedAsync(BookStorePermissions.Books.Default))

{

bookStoreMenu.AddItem(new ApplicationMenuItem(

"BooksStore.Books",

l["Menu:Books"],

url: "/Books"

));

}

}

}

}

````

{{else if UI == "NG"}}

### Angular Guard 配置

UI的第一步是防止未认证用户看见"图书"菜单项并进入图书管理页面.

打开 `/src/app/book/book-routing.module.ts` 替换为以下代码:

````js

import { NgModule } from '@angular/core';

import { Routes, RouterModule } from '@angular/router';

import { AuthGuard, PermissionGuard } from '@abp/ng.core';

import { BookComponent } from './book.component';

const routes: Routes = [

{ path: '', component: BookComponent, canActivate: [AuthGuard, PermissionGuard] },

];

@NgModule({

imports: [RouterModule.forChild(routes)],

exports: [RouterModule],

})

export class BookRoutingModule {}

````

\* 从 `@abp/ng.core` 引入 `AuthGuard` 和 `PermissionGuard`.

\* 在路由定义中添加 `canActivate: [AuthGuard, PermissionGuard]`.

打开 `/src/app/route.provider.ts`, 在 `/books` 路由中添加 `requiredPolicy: 'BookStore.Books'`. `/books` 路由应该如以下配置:

````js

{

path: '/books',

name: '::Menu:Books',

parentName: '::Menu:BookStore',

layout: eLayoutType.application,

requiredPolicy: 'BookStore.Books',

}

````

### 隐藏新建图书按钮

当用户没有 *\*图书新建\** 权限时, 图书管理页面上的 *\*新建图书\** 按钮应该不可见.

![bookstore-new-book-button-small](images/bookstore-new-book-button-small.png)

打开 `/src/app/book/book.component.html` 文件, 替换创建按钮的HTML内容如下:

````html

<!-- Add the abpPermission directive -->

<button \*abpPermission="'BookStore.Books.Create'" id="create" class="btn btn-primary" type="button" (click)="createBook()">

<i class="fa fa-plus mr-1"></i>

<span>{%{{{ '::NewBook' | abpLocalization }}}%}</span>

</button>

````

\* 加入 `\*abpPermission="'BookStore.Books.Create'"`, 当用户没有权限时隐藏按钮.

### 隐藏编辑和删除操作

图书管理页面中的图书表格每行都有操作按钮. 操作按钮包括 *\*编辑\** 和 *\*删除\** 操作:

![bookstore-edit-delete-actions](images/bookstore-edit-delete-actions.png)

如果用户没有权限, 应该隐藏相关的操作.

打开 `/src/app/book/book.component.html` 文件, 替换编辑和删除按钮的内容如下:

````html

<!-- Add the abpPermission directive -->

<button \*abpPermission="'BookStore.Books.Edit'" ngbDropdownItem (click)="editBook(row.id)">

{%{{{ '::Edit' | abpLocalization }}}%}

</button>

<!-- Add the abpPermission directive -->

<button \*abpPermission="'BookStore.Books.Delete'" ngbDropdownItem (click)="delete(row.id)">

{%{{{ '::Delete' | abpLocalization }}}%}

</button>

````

\* 加入 `\*abpPermission="'BookStore.Books.Edit'"`, 当用户没有编辑权限时隐藏按钮.

\* 加入 `\*abpPermission="'BookStore.Books.Delete'"`, 当用户没有删除权限时隐藏按钮.

{{else if UI == "Blazor"}}

### Razor验证组件

打开 `Acme.BookStore.Blazor` 项目中的 `/Pages/Books.razor` 文件, 在` @page` 指令和命名空间引入(`@using` 行)后添加 `Authorize` 特性, 如下所示:

````html

@page "/books"

@attribute [Authorize(BookStorePermissions.Books.Default)]

@using Acme.BookStore.Permissions

@using Microsoft.AspNetCore.Authorization

...

````

添加这个特性阻止未登录用户或未授权用户访问这个页面. 用户重试后, 会被重定向到登录页面.

### 显示/隐藏操作

图书管理页面上的每一种图书都有 *\*新建\** 按钮和 *\*编辑\**, *\*删除\** 操作. 如果用户没有相关权限, 这些按钮/操作应该被隐藏.

基类 `AbpCrudPageBase` 已经具有这些操作需要的功能.

#### 设置策略 (权限) 名称

加入以下代码到 `Books.razor` 文件结尾:

````csharp

@code

{

public Books() // Constructor

{

CreatePolicyName = BookStorePermissions.Books.Create;

UpdatePolicyName = BookStorePermissions.Books.Edit;

DeletePolicyName = BookStorePermissions.Books.Delete;

}

}

````

基类 `AbpCrudPageBase` 自动检查相关操作的权限. 如果需要手动检查, 它也定义了相应的属性.

\* `HasCreatePermission`: True, 如果用户具有新建实体的权限.

\* `HasUpdatePermission`: True, 如果用户具有编辑/更新实体的权限.

\* `HasDeletePermission`: True, 如果用户具有删除实体的权限.

> **\*\*Blazor 提示\*\***: 当添加少量代码到 `@code` 是没有问题的. 当添加的代码变长时, 建议使用代码后置方法以便于维护. 我们将在作者部分使用这个方法.

#### 隐藏新建图书按钮

检查 *\*新建图书\** 按钮权限:

````xml

@if (HasCreatePermission)

{

<Button Color="Color.Primary"

Clicked="OpenCreateModalAsync">@L["NewBook"]</Button>

}

````

#### 隐藏编辑/删除操作

`EntityAction` 组件定义了 `Visible` 属性 (参数) 以条件显示操作.

更新 `EntityActions` 部分:

````xml

<EntityActions TItem="BookDto" EntityActionsColumn="@EntityActionsColumn">

<EntityAction TItem="BookDto"

Text="@L["Edit"]"

Visible=HasUpdatePermission

Clicked="() => OpenEditModalAsync(context)" />

<EntityAction TItem="BookDto"

Text="@L["Delete"]"

Visible=HasDeletePermission

Clicked="() => DeleteEntityAsync(context)"

ConfirmationMessage="()=>GetDeleteConfirmationMessage(context)" />

</EntityActions>

````

#### 关于权限缓存

你可以运行和测试权限. 从admin角色中移除一个图书相关权限, 观察到相关按钮/操作从UI上消失.

在客户端, **\*\*ABP框架缓存当前用户的权限\*\*** . 所以, 当你修改了你的权限, 你需要手工 **\*\*刷新页面\*\***. 如果不刷新并试图使用被禁的操作, 你会从服务器收到一个HTTP 403 (forbidden) 响应.

> 修改角色或用户的权限在服务端立即生效. 所以, 缓存系统不会导致安全问题.

### 菜单项

即使我们在图书管理页面的所有层都控制了权限, 应用程序的主菜单依然会显示. 我们应该隐藏用户没有权限的菜单项.

打开 `Acme.BookStore.Blazor` 项目中的 `BookStoreMenuContributor` 类, 找到以下代码:

````csharp

context.Menu.AddItem(

new ApplicationMenuItem(

"BooksStore",

l["Menu:BookStore"],

icon: "fa fa-book"

).AddItem(

new ApplicationMenuItem(

"BooksStore.Books",

l["Menu:Books"],

url: "/books"

)

)

);

````

替换为以下代码:

````csharp

var bookStoreMenu = new ApplicationMenuItem(

"BooksStore",

l["Menu:BookStore"],

icon: "fa fa-book"

);

context.Menu.AddItem(bookStoreMenu);

//CHECK the PERMISSION

if (await context.IsGrantedAsync(BookStorePermissions.Books.Default))

{

bookStoreMenu.AddItem(new ApplicationMenuItem(

"BooksStore.Books",

l["Menu:Books"],

url: "/books"

));

}

````

你需要为 `ConfigureMenuAsync` 方法加入 `async` 关键字并重新整理返回值. 最终的 `ConfigureMainMenuAsync` 方法如下:

````csharp

private async Task ConfigureMainMenuAsync(MenuConfigurationContext context)

{

var l = context.GetLocalizer<BookStoreResource>();

context.Menu.Items.Insert(

0,

new ApplicationMenuItem(

"BookStore.Home",

l["Menu:Home"],

"/",

icon: "fas fa-home"

)

);

var bookStoreMenu = new ApplicationMenuItem(

"BooksStore",

l["Menu:BookStore"],

icon: "fa fa-book"

);

context.Menu.AddItem(bookStoreMenu);

//CHECK the PERMISSION

if (await context.IsGrantedAsync(BookStorePermissions.Books.Default))

{

bookStoreMenu.AddItem(new ApplicationMenuItem(

"BooksStore.Books",

l["Menu:Books"],

url: "/books"

));

}

}

````

{{end}}

## 下一章

查看本教程的[下一章](Part-6.md).

作者:领域层

# Web应用程序开发教程 - 第六章: 作者: 领域层

````json

//[doc-params]

{

"UI": ["MVC","Blazor","BlazorServer","NG"],

"DB": ["EF","Mongo"]

}

````

## 关于本教程

在本系列教程中, 你将构建一个名为 `Acme.BookStore` 的用于管理书籍及其作者列表的基于ABP的应用程序. 它是使用以下技术开发的:

\* **\*\*{{DB\_Value}}\*\*** 做为ORM提供程序.

\* **\*\*{{UI\_Value}}\*\*** 做为UI框架.

本教程分为以下部分:

- [Part 1: 创建服务端](Part-1.md)

- [Part 2: 图书列表页面](Part-2.md)

- [Part 3: 创建,更新和删除图书](Part-2.md)

- [Part 4: 集成测试](Part-4.md)

- [Part 5: 授权](Part-5.md)

- **\*\*Part 6: 作者: 领域层 (本章)\*\***

- [Part 7: 作者: 数据库集成](Part-7.md)

- [Part 8: 作者: 应用服务层](Part-8.md)

- [Part 9: 作者: 用户页面](Part-9.md)

- [Part 10: 图书到作者的关系](Part-10.md)

## 下载源码

本教程根据你的**\*\*UI\*\*** 和 **\*\*数据库\*\***偏好有多个版本,我们准备了几种可供下载的源码组合:

\* [MVC (Razor Pages) UI 与 EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Mvc-EfCore)

\* [Blazor UI 与 EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Blazor-EfCore)

\* [Angular UI 与 MongoDB](https://github.com/abpframework/abp-samples/tree/master/BookStore-Angular-MongoDb)

> 如果你在Windows中遇到 "文件名太长" or "解压错误", 很可能与Windows最大文件路径限制有关. Windows文件路径的最大长度为250字符. 为了解决这个问题,参阅 [在Windows 10中启用长路径](https://docs.microsoft.com/en-us/windows/win32/fileio/maximum-file-path-limitation?tabs=cmd#enable-long-paths-in-windows-10-version-1607-and-later).

> 如果你遇到与Git相关的长路径错误, 尝试使用下面的命令在Windows中启用长路径. 参阅 https://github.com/msysgit/msysgit/wiki/Git-cannot-create-a-file-or-directory-with-a-long-path

> `git config --system core.longpaths true`

## 简介

在前面的章节中, 我们使用 ABP 框架轻松地构建了一些服务;

\* 使用 [CrudAppService](../Application-Services.md) 基类, 而不是为标准的增删改查操作手工开发应用服务.

\* 使用 [generic repositories](../Repositories.md) 自动完成数据层功能.

对于 "作者" 部分;

\* 我们将要展示在需要的情况下, 如何 **\*\*手工做一些事情\*\***.

\* 我们将要实现一些 **\*\*领域驱动设计 (DDD) 最佳实践\*\***.

> **\*\*开发将会逐层完成, 一次聚焦一层. 在真实项目中, 你会逐个功能(垂直)开发, 如同前面的教程. 通过这种方式, 你可以体验这两种方式\*\***

## 作者实体

在 `Acme.BookStore.Domain` 项目中创建 `Authors` 文件夹 (命名空间), 在其中加入 `Author` 类:

````csharp

using System;

using JetBrains.Annotations;

using Volo.Abp;

using Volo.Abp.Domain.Entities.Auditing;

namespace Acme.BookStore.Authors

{

public class Author : FullAuditedAggregateRoot<Guid>

{

public string Name { get; private set; }

public DateTime BirthDate { get; set; }

public string ShortBio { get; set; }

private Author()

{

/\* This constructor is for deserialization / ORM purpose \*/

}

internal Author(

Guid id,

[NotNull] string name,

DateTime birthDate,

[CanBeNull] string shortBio = null)

: base(id)

{

SetName(name);

BirthDate = birthDate;

ShortBio = shortBio;

}

internal Author ChangeName([NotNull] string name)

{

SetName(name);

return this;

}

private void SetName([NotNull] string name)

{

Name = Check.NotNullOrWhiteSpace(

name,

nameof(name),

maxLength: AuthorConsts.MaxNameLength

);

}

}

}

````

\* 由 `FullAuditedAggregateRoot<Guid>` 继承使得实体支持[软删除](../Data-Filtering.md) (指实体被删除时, 它并没有从数据库中被删除, 而只是被标记删除), 实体也具有了 [审计](../Entities.md) 属性.

\* `Name` 属性的 `private set` 限制从类的外部设置这个属性. 有两种方法设置名字 (两种都进行了验证):

\* 当新建一个作者时, 通过构造器.

\* 使用 `ChangeName` 方法更新名字.

\* `构造器` 和 `ChangeName` 方法的访问级别是 `internal`, 强制这些方法只能在领域层由 `AuthorManager` 使用. 稍后将对此进行解释.

\* `Check` 类是一个ABP框架工具类, 用于检查方法参数 (如果参数非法会抛出 `ArgumentException`).

`AuthorConsts` 是一个简单的类, 它位于 `Acme.BookStore.Domain.Shared` 项目的 `Authors` 命名空间 (文件夹)中:

````csharp

namespace Acme.BookStore.Authors

{

public static class AuthorConsts

{

public const int MaxNameLength = 64;

}

}

````

在 `Acme.BookStore.Domain.Shared` 项目中创建这个类, 因为[数据传输类](../Data-Transfer-Objects.md) (DTOs) 稍后会再一次用到它.

## AuthorManager: 领域服务

`Author` 构造器和 `ChangeName` 方法的访问级别是 `internal`, 所以它们只能在领域层使用. 在 `Acme.BookStore.Domain` 项目中的 `Authors` 文件夹 (命名空间)创建 `AuthorManager` 类:

````csharp

using System;

using System.Threading.Tasks;

using JetBrains.Annotations;

using Volo.Abp;

using Volo.Abp.Domain.Services;

namespace Acme.BookStore.Authors

{

public class AuthorManager : DomainService

{

private readonly IAuthorRepository \_authorRepository;

public AuthorManager(IAuthorRepository authorRepository)

{

\_authorRepository = authorRepository;

}

public async Task<Author> CreateAsync(

[NotNull] string name,

DateTime birthDate,

[CanBeNull] string shortBio = null)

{

Check.NotNullOrWhiteSpace(name, nameof(name));

var existingAuthor = await \_authorRepository.FindByNameAsync(name);

if (existingAuthor != null)

{

throw new AuthorAlreadyExistsException(name);

}

return new Author(

GuidGenerator.Create(),

name,

birthDate,

shortBio

);

}

public async Task ChangeNameAsync(

[NotNull] Author author,

[NotNull] string newName)

{

Check.NotNull(author, nameof(author));

Check.NotNullOrWhiteSpace(newName, nameof(newName));

var existingAuthor = await \_authorRepository.FindByNameAsync(newName);

if (existingAuthor != null && existingAuthor.Id != author.Id)

{

throw new AuthorAlreadyExistsException(newName);

}

author.ChangeName(newName);

}

}

}

````

\* `AuthorManager` 强制使用一种可控的方式创建作者和修改作者的名字. 应用层 (后面会介绍) 将会使用这些方法.

> **\*\*DDD 提示\*\***: 如非必须并且用于执行核心业务规则, 不要引入领域服务方法. 对于这个场景, 我们使用这个服务保证名字的唯一性.

两个方法都检查是否存在同名用户, 如果存在, 抛出业务异常 `AuthorAlreadyExistsException`, 这个异常定义在 `Acme.BookStore.Domain` 项目 (`Authors` 文件夹中):

````csharp

using Volo.Abp;

namespace Acme.BookStore.Authors

{

public class AuthorAlreadyExistsException : BusinessException

{

public AuthorAlreadyExistsException(string name)

: base(BookStoreDomainErrorCodes.AuthorAlreadyExists)

{

WithData("name", name);

}

}

}

````

`BusinessException` 是一个特殊的异常类型. 在需要时抛出领域相关异常是一个好的实践. ABP框架会自动处理它, 并且它也容易本地化. `WithData(...)` 方法提供额外的数据给异常对象, 这些数据将会在本地化中或出于其它一些目的被使用.

打开 `Acme.BookStore.Domain.Shared` 项目中的 `BookStoreDomainErrorCodes` 并修改为:

````csharp

namespace Acme.BookStore

{

public static class BookStoreDomainErrorCodes

{

public const string AuthorAlreadyExists = "BookStore:00001";

}

}

````

这里定义了一个字符串, 表示应用程序抛出的错误码, 这个错误码可以被客户端应用程序处理. 为了用户, 你可能希望本地化它. 打开 `Acme.BookStore.Domain.Shared` 项目中的 `Localization/BookStore/en.json` , 加入以下项:

````json

"BookStore:00001": "There is already an author with the same name: {name}"

````

> 简体中文翻译请打开`zh-Hans.json`文件 ,并将"Texts"对象中对应的值替换为中文.

当 `AuthorAlreadyExistsException` 被抛出, 终端用户将会在UI上看到组织好的错误消息.

## IAuthorRepository

`AuthorManager` 注入了 `IAuthorRepository`, 所以我们需要定义它. 在 `Acme.BookStore.Domain` 项目的 `Authors` 文件夹 (命名空间) 中创建这个新接口:

````csharp

using System;

using System.Collections.Generic;

using System.Threading.Tasks;

using Volo.Abp.Domain.Repositories;

namespace Acme.BookStore.Authors

{

public interface IAuthorRepository : IRepository<Author, Guid>

{

Task<Author> FindByNameAsync(string name);

Task<List<Author>> GetListAsync(

int skipCount,

int maxResultCount,

string sorting,

string filter = null

);

}

}

````

\* `IAuthorRepository` 扩展了标准 `IRepository<Author, Guid>` 接口, 所以所有的标准 [repository](../Repositories.md) 方法对于 `IAuthorRepository` 都是可用的.

\* `FindByNameAsync` 在 `AuthorManager` 中用来根据姓名查询用户.

\* `GetListAsync` 用于应用层以获得一个排序的, 经过过滤的作者列表, 显示在UI上.

我们会在下一章实现这个repository.

> 这两个方法似乎 **\*\*看上去没有必要\*\***, 因为标准repositories已经是 `IQueryable`, 你可以直接使用它们, 而不是自定义方法. 在实际应用程序中, 这么做是没问题的. 但在这个 **\*\*学习指南\*\***中, 解释如何在需要时创建自定义repository方法是有价值的.

## 结论

这一章覆盖了图书管理程序作者相关功能的领域层. 在这一章中创建/更新的文件在下图中被高亮:

![bookstore-author-domain-layer](images/bookstore-author-domain-layer.png)

## 下一章

查看本教程的[下一章](Part-7.md).

作者:数据库集成

# Web应用程序开发教程 - 第七章: 数据库集成

````json

//[doc-params]

{

"UI": ["MVC","Blazor","BlazorServer","NG"],

"DB": ["EF","Mongo"]

}

````

## 关于本教程

在本系列教程中, 你将构建一个名为 `Acme.BookStore` 的用于管理书籍及其作者列表的基于ABP的应用程序. 它是使用以下技术开发的:

\* **\*\*{{DB\_Value}}\*\*** 做为ORM提供程序.

\* **\*\*{{UI\_Value}}\*\*** 做为UI框架.

本教程分为以下部分:

- [Part 1: 创建服务端](Part-1.md)

- [Part 2: 图书列表页面](Part-2.md)

- [Part 3: 创建,更新和删除图书](Part-2.md)

- [Part 4: 集成测试](Part-4.md)

- [Part 5: 授权](Part-5.md)

- [Part 6: 作者: 领域层](Part-6.md)

- **\*\*Part 7: 数据库集成\*\***(本章)

- [Part 8: 作者: 应用服务层](Part-8.md)

- [Part 9: 作者: 用户页面](Part-9.md)

- [Part 10: 图书到作者的关系](Part-10.md)

## 下载源码

本教程根据你的**\*\*UI\*\*** 和 **\*\*数据库\*\***偏好有多个版本,我们准备了几种可供下载的源码组合:

\* [MVC (Razor Pages) UI 与 EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Mvc-EfCore)

\* [Blazor UI 与 EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Blazor-EfCore)

\* [Angular UI 与 MongoDB](https://github.com/abpframework/abp-samples/tree/master/BookStore-Angular-MongoDb)

> 如果你在Windows中遇到 "文件名太长" or "解压错误", 很可能与Windows最大文件路径限制有关. Windows文件路径的最大长度为250字符. 为了解决这个问题,参阅 [在Windows 10中启用长路径](https://docs.microsoft.com/en-us/windows/win32/fileio/maximum-file-path-limitation?tabs=cmd#enable-long-paths-in-windows-10-version-1607-and-later).

> 如果你遇到与Git相关的长路径错误, 尝试使用下面的命令在Windows中启用长路径. 参阅 https://github.com/msysgit/msysgit/wiki/Git-cannot-create-a-file-or-directory-with-a-long-path

> `git config --system core.longpaths true`

## 简介

这章阐述如何为前一章介绍的 `作者` 实体配置数据库集成.

{{if DB=="EF"}}

## DB Context

打开 `Acme.BookStore.EntityFrameworkCore` 项目中的 `BookStoreDbContext` 加入 `DbSet` 属性:

````csharp

public DbSet<Author> Authors { get; set; }

````

定位到相同项目中的 `BookStoreDbContext` 类中的 `OnModelCreating` 方法, 加入以下代码到方法的结尾:

````csharp

builder.Entity<Author>(b =>

{

b.ToTable(BookStoreConsts.DbTablePrefix + "Authors",

BookStoreConsts.DbSchema);

b.ConfigureByConvention();

b.Property(x => x.Name)

.IsRequired()

.HasMaxLength(AuthorConsts.MaxNameLength);

b.HasIndex(x => x.Name);

});

````

这和前面的 `Book` 实体做的一样, 所以不再赘述.

## 创建数据库迁移

配置启动解决方案为使用 [Entity Framework Core Code First Migrations](https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/). 因为我们还没有修改数据库映射配置，所以需要创建一个新的迁移并对数据库应用变更.

打开命令行终端, 切换当前目录为 `Acme.BookStore.EntityFrameworkCore` 项目目录, 输入以下命令:

````bash

dotnet ef migrations add Added\_Authors

````

这会在项目中添加一个迁移类:

![bookstore-efcore-migration-authors](./images/bookstore-efcore-migration-authors.png)

你可以在同一个命令行终端中使用以下命令对数据库应用更改:

````bash

dotnet ef database update

````

> 如果你使用 Visual Studio, 可能希望在 *\*Package Manager Console (PMC)\** 使用 `Add-Migration Added\_Authors -c BookStoreDbContext` 和 `Update-Database -Context BookStoreDbContext` 命令. 如果这样, 保证 {{if UI=="MVC"}}`Acme.BookStore.Web`{{else if UI=="BlazorServer"}}`Acme.BookStore.Blazor`{{else if UI=="Blazor" || UI=="NG"}}`Acme.BookStore.HttpApi.Host`{{end}} 是启动项目并且在PMC中 `Acme.BookStore.EntityFrameworkCore` 是 *\*默认项目\** .

{{else if DB=="Mongo"}}

## DB Context

打开 `Acme.BookStore.MongoDB` 项目 `MongoDb 文件夹`中的 `BookStoreMongoDbContext`, 在类中加入以下属性:

````csharp

public IMongoCollection<Author> Authors => Collection<Author>();

````

{{end}}

## 实现 IAuthorRepository

{{if DB=="EF"}}

在 `Acme.BookStore.EntityFrameworkCore` 项目 (`Authors` 文件夹)中创建一个新类 `EfCoreAuthorRepository`, 粘贴以下代码:

````csharp

using System;

using System.Collections.Generic;

using System.Linq;

using System.Linq.Dynamic.Core;

using System.Threading.Tasks;

using Acme.BookStore.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore;

using Volo.Abp.Domain.Repositories.EntityFrameworkCore;

using Volo.Abp.EntityFrameworkCore;

namespace Acme.BookStore.Authors

{

public class EfCoreAuthorRepository

: EfCoreRepository<BookStoreDbContext, Author, Guid>,

IAuthorRepository

{

public EfCoreAuthorRepository(

IDbContextProvider<BookStoreDbContext> dbContextProvider)

: base(dbContextProvider)

{

}

public async Task<Author> FindByNameAsync(string name)

{

var dbSet = await GetDbSetAsync();

return await dbSet.FirstOrDefaultAsync(author => author.Name == name);

}

public async Task<List<Author>> GetListAsync(

int skipCount,

int maxResultCount,

string sorting,

string filter = null)

{

var dbSet = await GetDbSetAsync();

return await dbSet

.WhereIf(

!filter.IsNullOrWhiteSpace(),

author => author.Name.Contains(filter)

)

.OrderBy(sorting)

.Skip(skipCount)

.Take(maxResultCount)

.ToListAsync();

}

}

}

````

\* 继承自 `EfCoreRepository`, 所以继承了标准repository的方法实现.

\* `WhereIf` 是ABP 框架的快捷扩展方法. 它仅当第一个条件满足时, 执行 `Where` 查询. (根据名字查询, 仅当 filter 不为空). 你可以不使用这个方法, 但这些快捷方法可以提高效率.

\* `sorting` 可以是一个字符串, 如 `Name`, `Name ASC` 或 `Name DESC`. 通过使用 [System.Linq.Dynamic.Core](https://www.nuget.org/packages/System.Linq.Dynamic.Core) NuGet 包是可能的.

> 参阅 [EF Core 集成文档](../Entity-Framework-Core.md) 获得基于EF Core的repositories的更多信息.

{{else if DB=="Mongo"}}

在 `Acme.BookStore.MongoDB` 项目 (`Authors` 文件夹)中创建一个新类 `MongoDbAuthorRepository`, 粘贴以下代码:

```csharp

using System;

using System.Linq;

using System.Linq.Dynamic.Core;

using System.Collections.Generic;

using System.Threading.Tasks;

using Acme.BookStore.MongoDB;

using MongoDB.Driver;

using MongoDB.Driver.Linq;

using Volo.Abp.Domain.Repositories.MongoDB;

using Volo.Abp.MongoDB;

namespace Acme.BookStore.Authors

{

public class MongoDbAuthorRepository

: MongoDbRepository<BookStoreMongoDbContext, Author, Guid>,

IAuthorRepository

{

public MongoDbAuthorRepository(

IMongoDbContextProvider<BookStoreMongoDbContext> dbContextProvider

) : base(dbContextProvider)

{

}

public async Task<Author> FindByNameAsync(string name)

{

var queryable = await GetMongoQueryableAsync();

return await queryable.FirstOrDefaultAsync(author => author.Name == name);

}

public async Task<List<Author>> GetListAsync(

int skipCount,

int maxResultCount,

string sorting,

string filter = null)

{

var queryable = await GetMongoQueryableAsync();

return await queryable

.WhereIf<Author, IMongoQueryable<Author>>(

!filter.IsNullOrWhiteSpace(),

author => author.Name.Contains(filter)

)

.OrderBy(sorting)

.As<IMongoQueryable<Author>>()

.Skip(skipCount)

.Take(maxResultCount)

.ToListAsync();

}

}

}

```

\* 继承自 `MongoDbRepository`, 所以继承了标准repository的方法实现.

\* `WhereIf` 是ABP 框架的快捷扩展方法. 它仅当第一个条件满足时, 执行 `Where` 查询. (根据名字查询, 仅当 filter 不为空). 你可以不使用这个方法, 但这些快捷方法可以提高效率.

\* `sorting` 可以是一个字符串, 如 `Name`, `Name ASC` 或 `Name DESC`. 通过使用 [System.Linq.Dynamic.Core](https://www.nuget.org/packages/System.Linq.Dynamic.Core) NuGet 包是可能的.

> 参阅 [MongoDB 集成文档](../MongoDB.md) 获得基于MongoDB的repositories的更多信息.

{{end}}

## 下一章

查看本教程的[下一章](Part-8.md).

作者:应用服务层

# Web应用程序开发教程 - 第八章: 作者: 应用服务层

````json

//[doc-params]

{

"UI": ["MVC","Blazor","BlazorServer","NG"],

"DB": ["EF","Mongo"]

}

````

## 关于本教程

在本系列教程中, 你将构建一个名为 `Acme.BookStore` 的用于管理书籍及其作者列表的基于ABP的应用程序. 它是使用以下技术开发的:

\* **\*\*{{DB\_Value}}\*\*** 做为ORM提供程序.

\* **\*\*{{UI\_Value}}\*\*** 做为UI框架.

本教程分为以下部分:

- [Part 1: 创建服务端](Part-1.md)

- [Part 2: 图书列表页面](Part-2.md)

- [Part 3: 创建,更新和删除图书](Part-2.md)

- [Part 4: 集成测试](Part-4.md)

- [Part 5: 授权](Part-5.md)

- [Part 6: 作者: 领域层](Part-6.md)

- [Part 7: 作者: 数据库集成](Part-7.md)

- **\*\*Part 8: 作者: 应用服务层 (本章)\*\***

- [Part 9: 作者: 用户页面](Part-9.md)

- [Part 10: 图书到作者的关系](Part-10.md)

## 下载源码

本教程根据你的**\*\*UI\*\*** 和 **\*\*数据库\*\***偏好有多个版本,我们准备了几种可供下载的源码组合:

\* [MVC (Razor Pages) UI 与 EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Mvc-EfCore)

\* [Blazor UI 与 EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Blazor-EfCore)

\* [Angular UI 与 MongoDB](https://github.com/abpframework/abp-samples/tree/master/BookStore-Angular-MongoDb)

> 如果你在Windows中遇到 "文件名太长" or "解压错误", 很可能与Windows最大文件路径限制有关. Windows文件路径的最大长度为250字符. 为了解决这个问题,参阅 [在Windows 10中启用长路径](https://docs.microsoft.com/en-us/windows/win32/fileio/maximum-file-path-limitation?tabs=cmd#enable-long-paths-in-windows-10-version-1607-and-later).

> 如果你遇到与Git相关的长路径错误, 尝试使用下面的命令在Windows中启用长路径. 参阅 https://github.com/msysgit/msysgit/wiki/Git-cannot-create-a-file-or-directory-with-a-long-path

> `git config --system core.longpaths true`

## 简介

这章阐述如何为前一章介绍的 `作者` 实体创建应用服务层.

## IAuthorAppService

我们首先创建 [应用服务](../Application-Services.md) 接口和相关的 [DTO](../Data-Transfer-Objects.md)s. 在 `Acme.BookStore.Application.Contracts` 项目的 `Authors` 命名空间 (文件夹) 创建一个新接口 `IAuthorAppService`:

````csharp

using System;

using System.Threading.Tasks;

using Volo.Abp.Application.Dtos;

using Volo.Abp.Application.Services;

namespace Acme.BookStore.Authors

{

public interface IAuthorAppService : IApplicationService

{

Task<AuthorDto> GetAsync(Guid id);

Task<PagedResultDto<AuthorDto>> GetListAsync(GetAuthorListDto input);

Task<AuthorDto> CreateAsync(CreateAuthorDto input);

Task UpdateAsync(Guid id, UpdateAuthorDto input);

Task DeleteAsync(Guid id);

}

}

````

\* `IApplicationService` 是一个常规接口, 所有应用服务都继承自它, 所以 ABP 框架可以识别它们.

\* 在 `Author` 实体中定义标准方法用于CRUD操作.

\* `PagedResultDto` 是一个ABP框架中预定义的 DTO 类. 它拥有一个 `Items` 集合 和一个 `TotalCount` 属性, 用于返回分页结果.

\* 优先从 `CreateAsync` 方法返回 `AuthorDto` (新创建的作者), 虽然在这个程序中没有这么做 - 这里只是展示一种不同用法.

这个类使用下面定义的DTOs (为你的项目创建它们).

### AuthorDto

````csharp

using System;

using Volo.Abp.Application.Dtos;

namespace Acme.BookStore.Authors

{

public class AuthorDto : EntityDto<Guid>

{

public string Name { get; set; }

public DateTime BirthDate { get; set; }

public string ShortBio { get; set; }

}

}

````

\* `EntityDto<T>` 只有一个类型为指定泛型参数的 `Id` 属性. 你可以自己创建 `Id` 属性, 而不是继承自 `EntityDto<T>`.

### GetAuthorListDto

````csharp

using Volo.Abp.Application.Dtos;

namespace Acme.BookStore.Authors

{

public class GetAuthorListDto : PagedAndSortedResultRequestDto

{

public string? Filter { get; set; }

}

}

````

\* `Filter` 用于搜索作者. 它可以是 `null` (或空字符串) 以获得所有用户.

\* `PagedAndSortedResultRequestDto` 具有标准分页和排序属性: `int MaxResultCount`, `int SkipCount` 和 `string Sorting`.

> ABP 框架拥有这些基本的DTO类以简化并标准化你的DTOs. 参阅 [DTO 文档](../Data-Transfer-Objects.md) 获得所有DTO类的详细信息.

### CreateAuthorDto

````csharp

using System;

using System.ComponentModel.DataAnnotations;

namespace Acme.BookStore.Authors

{

public class CreateAuthorDto

{

[Required]

[StringLength(AuthorConsts.MaxNameLength)]

public string Name { get; set; }

[Required]

public DateTime BirthDate { get; set; }

public string ShortBio { get; set; }

}

}

````

数据标记特性可以用来验证DTO. 参阅 [验证文档](../Validation.md) 获得详细信息.

### UpdateAuthorDto

````csharp

using System;

using System.ComponentModel.DataAnnotations;

namespace Acme.BookStore.Authors

{

public class UpdateAuthorDto

{

[Required]

[StringLength(AuthorConsts.MaxNameLength)]

public string Name { get; set; }

[Required]

public DateTime BirthDate { get; set; }

public string ShortBio { get; set; }

}

}

````

> 我们可以在创建和更新操作间分享 (重用) 相同的DTO. 虽然可以这么做, 但我们推荐为这些操作创建不同的DTOs, 因为我们发现随着时间的推移, 它们通常会变得有差异. 所以, 与紧耦合相比, 代码重复也是合理的.

## AuthorAppService

是时候实现 `IAuthorAppService` 接口了. 在 `Acme.BookStore.Application` 项目的 `Authors` 命名空间 (文件夹) 中创建一个新类 `AuthorAppService` :

````csharp

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Acme.BookStore.Permissions;

using Microsoft.AspNetCore.Authorization;

using Volo.Abp.Application.Dtos;

using Volo.Abp.Domain.Repositories;

namespace Acme.BookStore.Authors

{

[Authorize(BookStorePermissions.Authors.Default)]

public class AuthorAppService : BookStoreAppService, IAuthorAppService

{

private readonly IAuthorRepository \_authorRepository;

private readonly AuthorManager \_authorManager;

public AuthorAppService(

IAuthorRepository authorRepository,

AuthorManager authorManager)

{

\_authorRepository = authorRepository;

\_authorManager = authorManager;

}

//...SERVICE METHODS WILL COME HERE...

}

}

````

\* `[Authorize(BookStorePermissions.Authors.Default)]` 是一个检查权限(策略)的声明式方法, 用来给当前用户授权. 参阅 [授权文档](../Authorization.md) 获得详细信息. `BookStorePermissions` 类在后文会被更新, 现在不需要担心编译错误.

\* 由 `BookStoreAppService` 派生, 这个类是一个简单基类, 可以做为模板. 它继承自标准的 `ApplicationService` 类.

\* 实现上面定义的 `IAuthorAppService` .

\* 注入 `IAuthorRepository` 和 `AuthorManager` 以使用服务方法.

现在, 我们逐个介绍服务方法. 复制这些方法到 `AuthorAppService` 类.

### GetAsync

````csharp

public async Task<AuthorDto> GetAsync(Guid id)

{

var author = await \_authorRepository.GetAsync(id);

return ObjectMapper.Map<Author, AuthorDto>(author);

}

````

这个方法根据 `Id` 获得 `Author` 实体, 使用 [对象到对象映射](../Object-To-Object-Mapping.md) 转换为 `AuthorDto`. 这需要配置AutoMapper, 后面会介绍.

### GetListAsync

````csharp

public async Task<PagedResultDto<AuthorDto>> GetListAsync(GetAuthorListDto input)

{

if (input.Sorting.IsNullOrWhiteSpace())

{

input.Sorting = nameof(Author.Name);

}

var authors = await \_authorRepository.GetListAsync(

input.SkipCount,

input.MaxResultCount,

input.Sorting,

input.Filter

);

var totalCount = input.Filter == null

? await \_authorRepository.CountAsync()

: await \_authorRepository.CountAsync(

author => author.Name.Contains(input.Filter));

return new PagedResultDto<AuthorDto>(

totalCount,

ObjectMapper.Map<List<Author>, List<AuthorDto>>(authors)

);

}

````

\* 为处理客户端没有设置的情况, 在方法的开头设置默认排序是 "根据作者名".

\* 使用 `IAuthorRepository.GetListAsync` 从数据库中获得分页的, 排序的和过滤的作者列表. 我们已经在教程的前一章中实现了它. 再一次强调, 实际上不需要创建这个方法, 因为我们可以从数据库中直接查询, 这里只是演示如何创建自定义repository方法.

\* 直接查询 `AuthorRepository` , 得到作者的数量. 如果客户端发送了过滤条件, 会得到过滤后的作者数量.

\* 最后, 通过映射 `Author` 列表到 `AuthorDto` 列表, 返回分页后的结果.

### CreateAsync

````csharp

[Authorize(BookStorePermissions.Authors.Create)]

public async Task<AuthorDto> CreateAsync(CreateAuthorDto input)

{

var author = await \_authorManager.CreateAsync(

input.Name,

input.BirthDate,

input.ShortBio

);

await \_authorRepository.InsertAsync(author);

return ObjectMapper.Map<Author, AuthorDto>(author);

}

````

\* `CreateAsync` 需要 `BookStorePermissions.Authors.Create` 权限 (另外包括 `AuthorAppService` 类声明的 `BookStorePermissions.Authors.Default` 权限).

\* 使用 `AuthorManager` (领域服务) 创建新作者.

\* 使用 `IAuthorRepository.InsertAsync` 插入新作者到数据库.

\* 使用 `ObjectMapper` 返回 `AuthorDto` , 代表新创建的作者.

> **\*\*DDD提示\*\***: 一些开发者可能会发现可以在 `\_authorManager.CreateAsync` 插入新实体. 我们认为把它留给应用层是更好的设计, 因为应用层更了解应该何时插入实体到数据库(在插入实体前可能需要额外的工作. 如果在领域层插入, 可能需要额外的更新操作). 但是, 你拥有最终的决定权.

### UpdateAsync

````csharp

[Authorize(BookStorePermissions.Authors.Edit)]

public async Task UpdateAsync(Guid id, UpdateAuthorDto input)

{

var author = await \_authorRepository.GetAsync(id);

if (author.Name != input.Name)

{

await \_authorManager.ChangeNameAsync(author, input.Name);

}

author.BirthDate = input.BirthDate;

author.ShortBio = input.ShortBio;

await \_authorRepository.UpdateAsync(author);

}

````

\* `UpdateAsync` 需要额外的 `BookStorePermissions.Authors.Edit` 权限.

\* 使用 `IAuthorRepository.GetAsync` 从数据库中获得作者实体. 如果给定的id没有找到作者, `GetAsync` 抛出 `EntityNotFoundException`, 这在web应用程序中导致一个 `404` HTTP 状态码. 在更新操作中先获取实体再更新它, 是一个好的实践.

\* 如果客户端请求, 使用 `AuthorManager.ChangeNameAsync` (领域服务方法) 修改作者姓名.

\* 因为没有任何业务逻辑, 直接更新 `BirthDate` 和 `ShortBio`, 它们可以接受任何值.

\* 最后, 调用 `IAuthorRepository.UpdateAsync` 更新实体到数据库.

{{if DB == "EF"}}

> **\*\*EF Core 提示\*\***: Entity Framework Core 拥有 **\*\*change tracking\*\*** 系统并在unit of work 结束时 **\*\*自动保存\*\*** 任何修改到实体 (你可以简单地认为APB框架在方法结束时自动调用 `SaveChanges`). 所以, 即使你在方法结束时没有调用 `\_authorRepository.UpdateAsync(...)` , 它依然可以工作. 如果你不考虑以后修改EF Core, 你可以移除这一行.

{{end}}

### DeleteAsync

````csharp

[Authorize(BookStorePermissions.Authors.Delete)]

public async Task DeleteAsync(Guid id)

{

await \_authorRepository.DeleteAsync(id);

}

````

\* `DeleteAsync` 需要额外的 `BookStorePermissions.Authors.Delete` 权限.

\* 直接使用repository的 `DeleteAsync` 方法.

## 权限定义

你还不能编译代码, 因为它需要 `BookStorePermissions` 类定义中一些常数.

打开 `Acme.BookStore.Application.Contracts` 项目中的 `BookStorePermissions` 类 (在 `Permissions` 文件夹中), 修改为如下代码:

````csharp

namespace Acme.BookStore.Permissions

{

public static class BookStorePermissions

{

public const string GroupName = "BookStore";

public static class Books

{

public const string Default = GroupName + ".Books";

public const string Create = Default + ".Create";

public const string Edit = Default + ".Edit";

public const string Delete = Default + ".Delete";

}

// \*\*\* ADDED a NEW NESTED CLASS \*\*\*

public static class Authors

{

public const string Default = GroupName + ".Authors";

public const string Create = Default + ".Create";

public const string Edit = Default + ".Edit";

public const string Delete = Default + ".Delete";

}

}

}

````

然后打开同一项目中的 `BookStorePermissionDefinitionProvider`, 在 `Define` 方法的结尾加入以下行:

````csharp

var authorsPermission = bookStoreGroup.AddPermission(

BookStorePermissions.Authors.Default, L("Permission:Authors"));

authorsPermission.AddChild(

BookStorePermissions.Authors.Create, L("Permission:Authors.Create"));

authorsPermission.AddChild(

BookStorePermissions.Authors.Edit, L("Permission:Authors.Edit"));

authorsPermission.AddChild(

BookStorePermissions.Authors.Delete, L("Permission:Authors.Delete"));

````

最后, 在 `Acme.BookStore.Domain.Shared` 项目中的 `Localization/BookStore/en.json` 加入以下项, 用以本地化权限名称:

````csharp

"Permission:Authors": "Author Management",

"Permission:Authors.Create": "Creating new authors",

"Permission:Authors.Edit": "Editing the authors",

"Permission:Authors.Delete": "Deleting the authors"

````

> 简体中文翻译请打开`zh-Hans.json`文件 ,并将"Texts"对象中对应的值替换为中文.

## 对象到对象映射

`AuthorAppService` 使用 `ObjectMapper` 将 `Author` 对象 转换为 `AuthorDto` 对象. 所以, 我们需要在 AutoMapper 配置中定义映射.

打开 `Acme.BookStore.Application` 项目中的 `BookStoreApplicationAutoMapperProfile` 类, 加入以下行到构造函数:

````csharp

CreateMap<Author, AuthorDto>();

````

## 数据种子

如同图书管理部分所做的, 在数据库中生成一些初始作者实体. 不仅当第一次运行应用程序时是有用的, 对自动化测试也是很有用的.

打开 `Acme.BookStore.Domain` 项目中的 `BookStoreDataSeederContributor`, 修改文件内容如下:

````csharp

using System;

using System.Threading.Tasks;

using Acme.BookStore.Authors;

using Acme.BookStore.Books;

using Volo.Abp.Data;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Domain.Repositories;

namespace Acme.BookStore

{

public class BookStoreDataSeederContributor

: IDataSeedContributor, ITransientDependency

{

private readonly IRepository<Book, Guid> \_bookRepository;

private readonly IAuthorRepository \_authorRepository;

private readonly AuthorManager \_authorManager;

public BookStoreDataSeederContributor(

IRepository<Book, Guid> bookRepository,

IAuthorRepository authorRepository,

AuthorManager authorManager)

{

\_bookRepository = bookRepository;

\_authorRepository = authorRepository;

\_authorManager = authorManager;

}

public async Task SeedAsync(DataSeedContext context)

{

if (await \_bookRepository.GetCountAsync() <= 0)

{

await \_bookRepository.InsertAsync(

new Book

{

Name = "1984",

Type = BookType.Dystopia,

PublishDate = new DateTime(1949, 6, 8),

Price = 19.84f

},

autoSave: true

);

await \_bookRepository.InsertAsync(

new Book

{

Name = "The Hitchhiker's Guide to the Galaxy",

Type = BookType.ScienceFiction,

PublishDate = new DateTime(1995, 9, 27),

Price = 42.0f

},

autoSave: true

);

}

// ADDED SEED DATA FOR AUTHORS

if (await \_authorRepository.GetCountAsync() <= 0)

{

await \_authorRepository.InsertAsync(

await \_authorManager.CreateAsync(

"George Orwell",

new DateTime(1903, 06, 25),

"Orwell produced literary criticism and poetry, fiction and polemical journalism; and is best known for the allegorical novella Animal Farm (1945) and the dystopian novel Nineteen Eighty-Four (1949)."

)

);

await \_authorRepository.InsertAsync(

await \_authorManager.CreateAsync(

"Douglas Adams",

new DateTime(1952, 03, 11),

"Douglas Adams was an English author, screenwriter, essayist, humorist, satirist and dramatist. Adams was an advocate for environmentalism and conservation, a lover of fast cars, technological innovation and the Apple Macintosh, and a self-proclaimed 'radical atheist'."

)

);

}

}

}

}

````

{{if DB=="EF"}}

你现在可以运行 `.DbMigrator` 控制台应用程序, **\*\*迁移\*\*** **\*\*数据库 schema\*\*** 并生成 **\*\*种子\*\*** 初始数据.

{{else if DB=="Mongo"}}

你现在可以运行 `.DbMigrator` 控制台应用程序, **\*\*迁移\*\*** **\*\*数据库 schema\*\*** 并生成 **\*\*种子\*\*** 初始数据.

{{end}}

## 测试作者应用服务

最后, 你可以为 `IAuthorAppService` 写一些测试. 在 `Acme.BookStore.Application.Tests` 项目的 `Authors` 命名空间(文件夹)中加入一个名为 `AuthorAppService\_Tests` 新类:

````csharp

using System;

using System.Threading.Tasks;

using Shouldly;

using Xunit;

namespace Acme.BookStore.Authors

{ {{if DB=="Mongo"}}

[Collection(BookStoreTestConsts.CollectionDefinitionName)]{{end}}

public class AuthorAppService\_Tests : BookStoreApplicationTestBase

{

private readonly IAuthorAppService \_authorAppService;

public AuthorAppService\_Tests()

{

\_authorAppService = GetRequiredService<IAuthorAppService>();

}

[Fact]

public async Task Should\_Get\_All\_Authors\_Without\_Any\_Filter()

{

var result = await \_authorAppService.GetListAsync(new GetAuthorListDto());

result.TotalCount.ShouldBeGreaterThanOrEqualTo(2);

result.Items.ShouldContain(author => author.Name == "George Orwell");

result.Items.ShouldContain(author => author.Name == "Douglas Adams");

}

[Fact]

public async Task Should\_Get\_Filtered\_Authors()

{

var result = await \_authorAppService.GetListAsync(

new GetAuthorListDto {Filter = "George"});

result.TotalCount.ShouldBeGreaterThanOrEqualTo(1);

result.Items.ShouldContain(author => author.Name == "George Orwell");

result.Items.ShouldNotContain(author => author.Name == "Douglas Adams");

}

[Fact]

public async Task Should\_Create\_A\_New\_Author()

{

var authorDto = await \_authorAppService.CreateAsync(

new CreateAuthorDto

{

Name = "Edward Bellamy",

BirthDate = new DateTime(1850, 05, 22),

ShortBio = "Edward Bellamy was an American author..."

}

);

authorDto.Id.ShouldNotBe(Guid.Empty);

authorDto.Name.ShouldBe("Edward Bellamy");

}

[Fact]

public async Task Should\_Not\_Allow\_To\_Create\_Duplicate\_Author()

{

await Assert.ThrowsAsync<AuthorAlreadyExistsException>(async () =>

{

await \_authorAppService.CreateAsync(

new CreateAuthorDto

{

Name = "Douglas Adams",

BirthDate = DateTime.Now,

ShortBio = "..."

}

);

});

}

//TODO: Test other methods...

}

}

````

完成应用服务方法的测试, 它们应该很容易理解.

## 下一章

查看本教程的[下一章](Part-9.md).

作者:用户界面

# Web应用程序开发教程 - 第九章: 作者: 用户页面

````json

//[doc-params]

{

"UI": ["MVC","Blazor","BlazorServer","NG"],

"DB": ["EF","Mongo"]

}

````

## 关于本教程

在本系列教程中, 你将构建一个名为 `Acme.BookStore` 的用于管理书籍及其作者列表的基于ABP的应用程序. 它是使用以下技术开发的:

\* **\*\*{{DB\_Value}}\*\*** 做为ORM提供程序.

\* **\*\*{{UI\_Value}}\*\*** 做为UI框架.

本教程分为以下部分:

- [Part 1: 创建服务端](Part-1.md)

- [Part 2: 图书列表页面](Part-2.md)

- [Part 3: 创建,更新和删除图书](Part-2.md)

- [Part 4: 集成测试](Part-4.md)

- [Part 5: 授权](Part-5.md)

- [Part 6: 作者: 领域层](Part-6.md)

- [Part 7: 作者: 数据库集成](Part-7.md)

- [Part 8: 作者: 应用服务层](Part-8.md)

- **\*\*Part 9: 作者: 用户页面 (本章)\*\***

- [Part 10: 图书到作者的关系](Part-10.md)

## 下载源码

本教程根据你的**\*\*UI\*\*** 和 **\*\*数据库\*\***偏好有多个版本,我们准备了几种可供下载的源码组合:

\* [MVC (Razor Pages) UI 与 EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Mvc-EfCore)

\* [Blazor UI 与 EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Blazor-EfCore)

\* [Angular UI 与 MongoDB](https://github.com/abpframework/abp-samples/tree/master/BookStore-Angular-MongoDb)

> 如果你在Windows中遇到 "文件名太长" or "解压错误", 很可能与Windows最大文件路径限制有关. Windows文件路径的最大长度为250字符. 为了解决这个问题,参阅 [在Windows 10中启用长路径](https://docs.microsoft.com/en-us/windows/win32/fileio/maximum-file-path-limitation?tabs=cmd#enable-long-paths-in-windows-10-version-1607-and-later).

> 如果你遇到与Git相关的长路径错误, 尝试使用下面的命令在Windows中启用长路径. 参阅 https://github.com/msysgit/msysgit/wiki/Git-cannot-create-a-file-or-directory-with-a-long-path

> `git config --system core.longpaths true`

## 简介

这章阐述如何为前一章介绍的 `作者` 实体创建CRUD页面.

{{if UI == "MVC"}}

## 作者列表页面

在 `Acme.BookStore.Web` 项目的 `Pages/Authors` 文件夹下创建一个新的razor页面, `Index.cshtml`, 修改文件内容如下.

### Index.cshtml

````html

@page

@using Acme.BookStore.Localization

@using Acme.BookStore.Permissions

@using Acme.BookStore.Web.Pages.Authors

@using Microsoft.AspNetCore.Authorization

@using Microsoft.Extensions.Localization

@inject IStringLocalizer<BookStoreResource> L

@inject IAuthorizationService AuthorizationService

@model IndexModel

@section scripts

{

<abp-script src="/Pages/Authors/Index.js"/>

}

<abp-card>

<abp-card-header>

<abp-row>

<abp-column size-md="\_6">

<abp-card-title>@L["Authors"]</abp-card-title>

</abp-column>

<abp-column size-md="\_6" class="text-right">

@if (await AuthorizationService

.IsGrantedAsync(BookStorePermissions.Authors.Create))

{

<abp-button id="NewAuthorButton"

text="@L["NewAuthor"].Value"

icon="plus"

button-type="Primary"/>

}

</abp-column>

</abp-row>

</abp-card-header>

<abp-card-body>

<abp-table striped-rows="true" id="AuthorsTable"></abp-table>

</abp-card-body>

</abp-card>

````

这是一个简单的页面, 和我们以前创建的图书页面一样. 它导入了一个JavaScript文件, 我们后面会进行介绍这个文件.

### IndexModel.cshtml.cs

````csharp

using Microsoft.AspNetCore.Mvc.RazorPages;

namespace Acme.BookStore.Web.Pages.Authors

{

public class IndexModel : PageModel

{

public void OnGet()

{

}

}

}

````

### Index.js

````js

$(function () {

var l = abp.localization.getResource('BookStore');

var createModal = new abp.ModalManager(abp.appPath + 'Authors/CreateModal');

var editModal = new abp.ModalManager(abp.appPath + 'Authors/EditModal');

var dataTable = $('#AuthorsTable').DataTable(

abp.libs.datatables.normalizeConfiguration({

serverSide: true,

paging: true,

order: [[1, "asc"]],

searching: false,

scrollX: true,

ajax: abp.libs.datatables.createAjax(acme.bookStore.authors.author.getList),

columnDefs: [

{

title: l('Actions'),

rowAction: {

items:

[

{

text: l('Edit'),

visible:

abp.auth.isGranted('BookStore.Authors.Edit'),

action: function (data) {

editModal.open({ id: data.record.id });

}

},

{

text: l('Delete'),

visible:

abp.auth.isGranted('BookStore.Authors.Delete'),

confirmMessage: function (data) {

return l(

'AuthorDeletionConfirmationMessage',

data.record.name

);

},

action: function (data) {

acme.bookStore.authors.author

.delete(data.record.id)

.then(function() {

abp.notify.info(

l('SuccessfullyDeleted')

);

dataTable.ajax.reload();

});

}

}

]

}

},

{

title: l('Name'),

data: "name"

},

{

title: l('BirthDate'),

data: "birthDate",

render: function (data) {

return luxon

.DateTime

.fromISO(data, {

locale: abp.localization.currentCulture.name

}).toLocaleString();

}

}

]

})

);

createModal.onResult(function () {

dataTable.ajax.reload();

});

editModal.onResult(function () {

dataTable.ajax.reload();

});

$('#NewAuthorButton').click(function (e) {

e.preventDefault();

createModal.open();

});

});

````

简单来说, 这个JavaScript页面:

\* 创建了一个具有 `操作`, `姓名` 和 `生日` 列的数据表格.

\* `Actions` 列用来添加 *\*编辑\** 和 *\*删除\** 操作.

\* `生日` 提供了一个 `render` 函数, 使用 [luxon](https://moment.github.io/luxon/) 库格式化 `DateTime` 值.

\* 使用 `abp.ModalManager` 打开 *\*新建\** 和 *\*编辑\** 模态表单.

这块代码与以前创建的图书页面非常相似, 所以我们不再赘述.

### 本地化

这个页面使用了一些需要声明的本地化键. 打开 `Acme.BookStore.Domain.Shared` 项目中 `Localization/BookStore` 文件夹下的 `en.json` 文件, 加入以下条目:

````json

"Menu:Authors": "Authors",

"Authors": "Authors",

"AuthorDeletionConfirmationMessage": "Are you sure to delete the author '{0}'?",

"BirthDate": "Birth date",

"NewAuthor": "New author"

````

> 简体中文翻译请打开`zh-Hans.json`文件 ,并将"Texts"对象中对应的值替换为中文.

注意我们加入了额外的键. 它们会在下面的小节中被使用.

### 加入主菜单

打开 `Acme.BookStore.Web` 项目的 `Menus` 文件夹中的 `BookStoreMenuContributor.cs` , 在 `ConfigureMainMenuAsync` 方法的结尾加入以下代码:

````csharp

if (await context.IsGrantedAsync(BookStorePermissions.Authors.Default))

{

bookStoreMenu.AddItem(new ApplicationMenuItem(

"BooksStore.Authors",

l["Menu:Authors"],

url: "/Authors"

));

}

````

### 运行应用程序

运行并登录应用程序. **\*\*因为你还没有权限, 所以不能看见菜单项\*\***. 转到 `Identity/Roles` 页面, 点击 *\*操作\** 按钮并选择**\*\*管理员角色\*\***的*\*权限\**操作:

![bookstore-author-permissions](images/bookstore-author-permissions.png)

如你所见, 管理员角色还没有*\*作者管理\**权限. 单击复选框并保存, 赋予权限. **\*\*刷新页面\*\***后, 你会在主菜单中的*\*图书商店\**下看到*\*作者\**菜单项:

![bookstore-authors-page](images/bookstore-authors-page.png)

页面是完全可以工作的, 除了 *\*新建作者\** 和 *\*操作/编辑\**, 因为它们还没有实现 .

> **\*\*提示\*\***: 如果你在定义一个新权限后运行 `.DbMigrator` 控制台程序, 它会自动将这些权限赋予管理员角色, 你不需要手工赋予权限.

## 新建模态窗口

在 `Acme.BookStore.Web` 项目的 `Pages/Authors` 文件夹下创建一个 razor 页面 `CreateModal.cshtml`, 修改它的内容如下:

### CreateModal.cshtml

```html

@page

@using Acme.BookStore.Localization

@using Acme.BookStore.Web.Pages.Authors

@using Microsoft.Extensions.Localization

@using Volo.Abp.AspNetCore.Mvc.UI.Bootstrap.TagHelpers.Modal

@model CreateModalModel

@inject IStringLocalizer<BookStoreResource> L

@{

Layout = null;

}

<form asp-page="/Authors/CreateModal">

<abp-modal>

<abp-modal-header title="@L["NewAuthor"].Value"></abp-modal-header>

<abp-modal-body>

<abp-input asp-for="Author.Name" />

<abp-input asp-for="Author.BirthDate" />

<abp-input asp-for="Author.ShortBio" />

</abp-modal-body>

<abp-modal-footer buttons="@(AbpModalButtons.Cancel|AbpModalButtons.Save)"></abp-modal-footer>

</abp-modal>

</form>

```

之前我们已经使用ABP框架的 [动态表单](../UI/AspNetCore/Tag-Helpers/Dynamic-Forms.md)开发了图书页面. 这里可以使用相同的方法, 但我们希望展示如何手工完成它. 实际上, 没有那么手工化, 因为在这个例子中我们使用了 `abp-input` 标签简化了表单元素的创建.

你当然可以使用标准Bootstrap HTML结构, 但是这需要写很多代码. `abp-input` 自动添加验证, 本地化和根据数据类型生成标准元素.

### CreateModal.cshtml.cs

```csharp

using System;

using System.ComponentModel.DataAnnotations;

using System.Threading.Tasks;

using Acme.BookStore.Authors;

using Microsoft.AspNetCore.Mvc;

using Volo.Abp.AspNetCore.Mvc.UI.Bootstrap.TagHelpers.Form;

namespace Acme.BookStore.Web.Pages.Authors

{

public class CreateModalModel : BookStorePageModel

{

[BindProperty]

public CreateAuthorViewModel Author { get; set; }

private readonly IAuthorAppService \_authorAppService;

public CreateModalModel(IAuthorAppService authorAppService)

{

\_authorAppService = authorAppService;

}

public void OnGet()

{

Author = new CreateAuthorViewModel();

}

public async Task<IActionResult> OnPostAsync()

{

var dto = ObjectMapper.Map<CreateAuthorViewModel, CreateAuthorDto>(Author);

await \_authorAppService.CreateAsync(dto);

return NoContent();

}

public class CreateAuthorViewModel

{

[Required]

[StringLength(AuthorConsts.MaxNameLength)]

public string Name { get; set; }

[Required]

[DataType(DataType.Date)]

public DateTime BirthDate { get; set; }

[TextArea]

public string ShortBio { get; set; }

}

}

}

```

这个页面模型类注入和使用 `IAuthorAppService` 创建新作者. 它和图书创建模型类之间主要的区别是这个模型类为视图模型声明了一个新类 `CreateAuthorViewModel`, 而不是重用 `CreateAuthorDto`.

这么做的主要原因是展示在页面中如何使用不同的模型. 但还有一个好处: 我们为类成员添加了两个不存在于 `CreateAuthorDto` 中的特性:

\* 为 `BirthDate` 添加 `[DataType(DataType.Date)]` 特性, 这会在UI为这个属性显示一个日期选择控件.

\* 为 `ShortBio` 添加 `[TextArea]` 特性, 这会显示一个多行文本框, 而不是标准文本框.

通过这种方式, 可以根据UI需求定制视图模型类, 而无需修改DTO. 这么做的一个结果是: 使用 `ObjectMapper` 将 `CreateAuthorViewModel` 映射到 `CreateAuthorDto`. 为了完成映射, 需要在 `BookStoreWebAutoMapperProfile` 构造函数中加入新的映射代码:

````csharp

using Acme.BookStore.Authors; // ADDED NAMESPACE IMPORT

using Acme.BookStore.Books;

using AutoMapper;

namespace Acme.BookStore.Web

{

public class BookStoreWebAutoMapperProfile : Profile

{

public BookStoreWebAutoMapperProfile()

{

CreateMap<BookDto, CreateUpdateBookDto>();

// ADD a NEW MAPPING

CreateMap<Pages.Authors.CreateModalModel.CreateAuthorViewModel,

CreateAuthorDto>();

}

}

}

````

当你重新运行应用程序后, 点击"新建作者" 按钮会打开一个新的模态窗口.

![bookstore-new-author-modal](images/bookstore-new-author-modal.png)

## 编辑模态窗口

在 `Acme.BookStore.Web` 项目的 `Pages/Authors` 文件夹下创建一个 razor 页面 `EditModal.cshtml`, 修改它的内容如下:

### EditModal.cshtml

````html

@page

@using Acme.BookStore.Localization

@using Acme.BookStore.Web.Pages.Authors

@using Microsoft.Extensions.Localization

@using Volo.Abp.AspNetCore.Mvc.UI.Bootstrap.TagHelpers.Modal

@model EditModalModel

@inject IStringLocalizer<BookStoreResource> L

@{

Layout = null;

}

<form asp-page="/Authors/EditModal">

<abp-modal>

<abp-modal-header title="@L["Update"].Value"></abp-modal-header>

<abp-modal-body>

<abp-input asp-for="Author.Id" />

<abp-input asp-for="Author.Name" />

<abp-input asp-for="Author.BirthDate" />

<abp-input asp-for="Author.ShortBio" />

</abp-modal-body>

<abp-modal-footer buttons="@(AbpModalButtons.Cancel|AbpModalButtons.Save)"></abp-modal-footer>

</abp-modal>

</form>

````

### EditModal.cshtml.cs

```csharp

using System;

using System.ComponentModel.DataAnnotations;

using System.Threading.Tasks;

using Acme.BookStore.Authors;

using Microsoft.AspNetCore.Mvc;

using Volo.Abp.AspNetCore.Mvc.UI.Bootstrap.TagHelpers.Form;

namespace Acme.BookStore.Web.Pages.Authors

{

public class EditModalModel : BookStorePageModel

{

[BindProperty]

public EditAuthorViewModel Author { get; set; }

private readonly IAuthorAppService \_authorAppService;

public EditModalModel(IAuthorAppService authorAppService)

{

\_authorAppService = authorAppService;

}

public async Task OnGetAsync(Guid id)

{

var authorDto = await \_authorAppService.GetAsync(id);

Author = ObjectMapper.Map<AuthorDto, EditAuthorViewModel>(authorDto);

}

public async Task<IActionResult> OnPostAsync()

{

await \_authorAppService.UpdateAsync(

Author.Id,

ObjectMapper.Map<EditAuthorViewModel, UpdateAuthorDto>(Author)

);

return NoContent();

}

public class EditAuthorViewModel

{

[HiddenInput]

public Guid Id { get; set; }

[Required]

[StringLength(AuthorConsts.MaxNameLength)]

public string Name { get; set; }

[Required]

[DataType(DataType.Date)]

public DateTime BirthDate { get; set; }

[TextArea]

public string ShortBio { get; set; }

}

}

}

```

这个类与 `CreateModal.cshtml.cs` 类似, 主要不同是:

\* 使用 `IAuthorAppService.GetAsync(...)` 方法从应用层获取正在编辑的作者.

\* `EditAuthorViewModel` 拥有一个额外的 `Id` 属性, 它被 `[HiddenInput]` 特性标记, 会为这个属性在页面上创建一个隐藏输入框.

这个类要求在 `BookStoreWebAutoMapperProfile` 类中添加两个对象映射声明:

```csharp

using Acme.BookStore.Authors;

using Acme.BookStore.Books;

using AutoMapper;

namespace Acme.BookStore.Web

{

public class BookStoreWebAutoMapperProfile : Profile

{

public BookStoreWebAutoMapperProfile()

{

CreateMap<BookDto, CreateUpdateBookDto>();

CreateMap<Pages.Authors.CreateModalModel.CreateAuthorViewModel,

CreateAuthorDto>();

// ADD THESE NEW MAPPINGS

CreateMap<AuthorDto, Pages.Authors.EditModalModel.EditAuthorViewModel>();

CreateMap<Pages.Authors.EditModalModel.EditAuthorViewModel,

UpdateAuthorDto>();

}

}

}

```

这就是全部了! 你可以运行应用程序并尝试编辑一个作者.

{{else if UI == "NG"}}

## 作者管理页面

运行下面的命令行在angular应用程序的根目录创建一个名为 `AuthorModule` 新模块:

```bash

yarn ng generate module author --module app --routing --route authors

```

这上命令应该产生以下输出:

```bash

> yarn ng generate module author --module app --routing --route authors

yarn run v1.19.1

$ ng generate module author --module app --routing --route authors

CREATE src/app/author/author-routing.module.ts (344 bytes)

CREATE src/app/author/author.module.ts (349 bytes)

CREATE src/app/author/author.component.html (21 bytes)

CREATE src/app/author/author.component.spec.ts (628 bytes)

CREATE src/app/author/author.component.ts (276 bytes)

CREATE src/app/author/author.component.scss (0 bytes)

UPDATE src/app/app-routing.module.ts (1396 bytes)

Done in 2.22s.

```

### AuthorModule

打开 `/src/app/author/author.module.ts,` 替换成以下内容:

```js

import { NgModule } from '@angular/core';

import { SharedModule } from '../shared/shared.module';

import { AuthorRoutingModule } from './author-routing.module';

import { AuthorComponent } from './author.component';

import { NgbDatepickerModule } from '@ng-bootstrap/ng-bootstrap';

@NgModule({

declarations: [AuthorComponent],

imports: [SharedModule, AuthorRoutingModule, NgbDatepickerModule],

})

export class AuthorModule {}

```

- 添加 `SharedModule`. `SharedModule` 导出一些创建用户页面需要的通用模块.

- `SharedModule` 已经导出 `CommonModule`, 所以移除 `CommonModule`.

- 添加 `NgbDatepickerModule`, 后面用于作者创建和编辑表单.

### 菜单定义

打开 `src/app/route.provider.ts` 文件, 加入以下的菜单定义:

````js

{

path: '/authors',

name: '::Menu:Authors',

parentName: '::Menu:BookStore',

layout: eLayoutType.application,

requiredPolicy: 'BookStore.Authors',

}

````

最终的 `configureRoutes` 函数声明应该如下:

```js

function configureRoutes(routes: RoutesService) {

return () => {

routes.add([

{

path: '/',

name: '::Menu:Home',

iconClass: 'fas fa-home',

order: 1,

layout: eLayoutType.application,

},

{

path: '/book-store',

name: '::Menu:BookStore',

iconClass: 'fas fa-book',

order: 2,

layout: eLayoutType.application,

},

{

path: '/books',

name: '::Menu:Books',

parentName: '::Menu:BookStore',

layout: eLayoutType.application,

requiredPolicy: 'BookStore.Books',

},

{

path: '/authors',

name: '::Menu:Authors',

parentName: '::Menu:BookStore',

layout: eLayoutType.application,

requiredPolicy: 'BookStore.Authors',

},

]);

};

}

```

### 生成服务代理

[ABP CLI](https://docs.abp.io/en/abp/latest/CLI) 提供 `generate-proxy` 命令为HTTP APIs生成客户端代理, 使得使用HTTP APIs更容易. `generate-proxy` 命令运行前, 应用程序必须启动运行.

在 `angular` 文件夹运行以下命令:

```bash

abp generate-proxy -t ng

```

这个命令为作者服务和相关模型(DTO)类生成服务代理:

![bookstore-angular-service-proxy-author](images/bookstore-angular-service-proxy-author-2.png)

### AuthorComponent

打开 `/src/app/author/author.component.ts` 文件, 替换成以下内容:

```js

import { Component, OnInit } from '@angular/core';

import { ListService, PagedResultDto } from '@abp/ng.core';

import { AuthorService, AuthorDto } from '@proxy/authors';

import { FormGroup, FormBuilder, Validators } from '@angular/forms';

import { NgbDateNativeAdapter, NgbDateAdapter } from '@ng-bootstrap/ng-bootstrap';

import { ConfirmationService, Confirmation } from '@abp/ng.theme.shared';

@Component({

selector: 'app-author',

templateUrl: './author.component.html',

styleUrls: ['./author.component.scss'],

providers: [ListService, { provide: NgbDateAdapter, useClass: NgbDateNativeAdapter }],

})

export class AuthorComponent implements OnInit {

author = { items: [], totalCount: 0 } as PagedResultDto<AuthorDto>;

isModalOpen = false;

form: FormGroup;

selectedAuthor = {} as AuthorDto;

constructor(

public readonly list: ListService,

private authorService: AuthorService,

private fb: FormBuilder,

private confirmation: ConfirmationService

) {}

ngOnInit(): void {

const authorStreamCreator = (query) => this.authorService.getList(query);

this.list.hookToQuery(authorStreamCreator).subscribe((response) => {

this.author = response;

});

}

createAuthor() {

this.selectedAuthor = {} as AuthorDto;

this.buildForm();

this.isModalOpen = true;

}

editAuthor(id: string) {

this.authorService.get(id).subscribe((author) => {

this.selectedAuthor = author;

this.buildForm();

this.isModalOpen = true;

});

}

buildForm() {

this.form = this.fb.group({

name: [this.selectedAuthor.name || '', Validators.required],

birthDate: [

this.selectedAuthor.birthDate ? new Date(this.selectedAuthor.birthDate) : null,

Validators.required,

],

});

}

save() {

if (this.form.invalid) {

return;

}

if (this.selectedAuthor.id) {

this.authorService

.update(this.selectedAuthor.id, this.form.value)

.subscribe(() => {

this.isModalOpen = false;

this.form.reset();

this.list.get();

});

} else {

this.authorService.create(this.form.value).subscribe(() => {

this.isModalOpen = false;

this.form.reset();

this.list.get();

});

}

}

delete(id: string) {

this.confirmation.warn('::AreYouSureToDelete', '::AreYouSure')

.subscribe((status) => {

if (status === Confirmation.Status.confirm) {

this.authorService.delete(id).subscribe(() => this.list.get());

}

});

}

}

```

打开 `/src/app/author/author.component.html` 文件, 替换成以下内容:

````html

<div class="card">

<div class="card-header">

<div class="row">

<div class="col col-md-6">

<h5 class="card-title">

{%{{{ '::Menu:Authors' | abpLocalization }}}%}

</h5>

</div>

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

<div class="text-lg-right pt-2">

<button \*abpPermission="'BookStore.Authors.Create'" id="create" class="btn btn-primary" type="button" (click)="createAuthor()">

<i class="fa fa-plus mr-1"></i>

<span>{%{{{ '::NewAuthor' | abpLocalization }}}%}</span>

</button>

</div>

</div>

</div>

</div>

<div class="card-body">

<ngx-datatable [rows]="author.items" [count]="author.totalCount" [list]="list" default>

<ngx-datatable-column

[name]="'::Actions' | abpLocalization"

[maxWidth]="150"

[sortable]="false"

>

<ng-template let-row="row" ngx-datatable-cell-template>

<div ngbDropdown container="body" class="d-inline-block">

<button

class="btn btn-primary btn-sm dropdown-toggle"

data-toggle="dropdown"

aria-haspopup="true"

ngbDropdownToggle

>

<i class="fa fa-cog mr-1"></i>{%{{{ '::Actions' | abpLocalization }}}%}

</button>

<div ngbDropdownMenu>

<button \*abpPermission="'BookStore.Authors.Edit'" ngbDropdownItem (click)="editAuthor(row.id)">

{%{{{ '::Edit' | abpLocalization }}}%}

</button>

<button \*abpPermission="'BookStore.Authors.Delete'" ngbDropdownItem (click)="delete(row.id)">

{%{{{ '::Delete' | abpLocalization }}}%}

</button>

</div>

</div>

</ng-template>

</ngx-datatable-column>

<ngx-datatable-column [name]="'::Name' | abpLocalization" prop="name"></ngx-datatable-column>

<ngx-datatable-column [name]="'::BirthDate' | abpLocalization">

<ng-template let-row="row" ngx-datatable-cell-template>

{%{{{ row.birthDate | date }}}%}

</ng-template>

</ngx-datatable-column>

</ngx-datatable>

</div>

</div>

<abp-modal [(visible)]="isModalOpen">

<ng-template #abpHeader>

<h3>{%{{{ (selectedAuthor.id ? '::Edit' : '::NewAuthor') | abpLocalization }}}%}</h3>

</ng-template>

<ng-template #abpBody>

<form [formGroup]="form" (ngSubmit)="save()">

<div class="form-group">

<label for="author-name">Name</label><span> \* </span>

<input type="text" id="author-name" class="form-control" formControlName="name" autofocus />

</div>

<div class="form-group">

<label>Birth date</label><span> \* </span>

<input

#datepicker="ngbDatepicker"

class="form-control"

name="datepicker"

formControlName="birthDate"

ngbDatepicker

(click)="datepicker.toggle()"

/>

</div>

</form>

</ng-template>

<ng-template #abpFooter>

<button type="button" class="btn btn-secondary" abpClose>

{%{{{ '::Close' | abpLocalization }}}%}

</button>

<button class="btn btn-primary" (click)="save()" [disabled]="form.invalid">

<i class="fa fa-check mr-1"></i>

{%{{{ '::Save' | abpLocalization }}}%}

</button>

</ng-template>

</abp-modal>

````

### 本地化

这个页面使用了一些需要声明的本地化键. 打开 `Acme.BookStore.Domain.Shared` 项目中 `Localization/BookStore` 文件夹下的 `en.json` 文件, 加入以下条目:

````json

"Menu:Authors": "Authors",

"Authors": "Authors",

"AuthorDeletionConfirmationMessage": "Are you sure to delete the author '{0}'?",

"BirthDate": "Birth date",

"NewAuthor": "New author"

````

> 简体中文翻译请打开`zh-Hans.json`文件 ,并将"Texts"对象中对应的值替换为中文.

### 运行应用程序

运行并登录应用程序. **\*\*因为你还没有权限, 所以不能看见菜单项\*\***. 转到 `Identity/Roles` 页面, 点击 *\*操作\** 按钮并选择**\*\*管理员角色\*\***的*\*权限\**操作:

![bookstore-author-permissions](images/bookstore-author-permissions.png)

如你所见, 管理员角色还没有*\*作者管理\**权限. 单击复选框并保存, 赋予权限. **\*\*刷新页面\*\***后, 你会在主菜单中的*\*图书商店\**下看到*\*作者\**菜单项:

![bookstore-authors-page](images/bookstore-angular-authors-page.png)

这就是全部了! 这是一个完整的, 可以工作的页面. 你可以新建, 编辑和删除作者.

> **\*\*提示\*\***: 如果你在定义一个新权限后运行 `.DbMigrator` 控制台程序, 它会自动将这些权限赋予管理员角色, 你不需要手工赋予权限.

{{end}}

{{if UI == "Blazor" || UI == "BlazorServer"}}

## 作者管理页面

### 作者Razor组件

在 `Acme.BookStore.Blazor` 项目中新建一个 Razor组件页面 `/Pages/Authors.razor`, 替换成以下内容:

````xml

@page "/authors"

@using Acme.BookStore.Authors

@using Acme.BookStore.Localization

@using Volo.Abp.AspNetCore.Components.Web

@inherits BookStoreComponentBase

@inject IAuthorAppService AuthorAppService

@inject AbpBlazorMessageLocalizerHelper<BookStoreResource> LH

<Card>

<CardHeader>

<Row>

<Column ColumnSize="ColumnSize.Is6">

<h2>@L["Authors"]</h2>

</Column>

<Column ColumnSize="ColumnSize.Is6">

<Paragraph Alignment="TextAlignment.Right">

@if (CanCreateAuthor)

{

<Button Color="Color.Primary"

Clicked="OpenCreateAuthorModal">

@L["NewAuthor"]

</Button>

}

</Paragraph>

</Column>

</Row>

</CardHeader>

<CardBody>

<DataGrid TItem="AuthorDto"

Data="AuthorList"

ReadData="OnDataGridReadAsync"

TotalItems="TotalCount"

ShowPager="true"

PageSize="PageSize">

<DataGridColumns>

<DataGridColumn Width="150px"

TItem="AuthorDto"

Field="@nameof(AuthorDto.Id)"

Sortable="false"

Caption="@L["Actions"]">

<DisplayTemplate>

<Dropdown>

<DropdownToggle Color="Color.Primary">

@L["Actions"]

</DropdownToggle>

<DropdownMenu>

@if (CanEditAuthor)

{

<DropdownItem Clicked="() => OpenEditAuthorModal(context)">

@L["Edit"]

</DropdownItem>

}

@if (CanDeleteAuthor)

{

<DropdownItem Clicked="() => DeleteAuthorAsync(context)">

@L["Delete"]

</DropdownItem>

}

</DropdownMenu>

</Dropdown>

</DisplayTemplate>

</DataGridColumn>

<DataGridColumn TItem="AuthorDto"

Field="@nameof(AuthorDto.Name)"

Caption="@L["Name"]"></DataGridColumn>

<DataGridColumn TItem="AuthorDto"

Field="@nameof(AuthorDto.BirthDate)"

Caption="@L["BirthDate"]">

<DisplayTemplate>

@context.BirthDate.ToShortDateString()

</DisplayTemplate>

</DataGridColumn>

</DataGridColumns>

</DataGrid>

</CardBody>

</Card>

<Modal @ref="CreateAuthorModal">

<ModalBackdrop />

<ModalContent IsCentered="true">

<Form>

<ModalHeader>

<ModalTitle>@L["NewAuthor"]</ModalTitle>

<CloseButton Clicked="CloseCreateAuthorModal" />

</ModalHeader>

<ModalBody>

<Validations @ref="@CreateValidationsRef" Model="@NewAuthor" ValidateOnLoad="false">

<Validation MessageLocalizer="@LH.Localize">

<Field>

<FieldLabel>@L["Name"]</FieldLabel>

<TextEdit @bind-Text="@NewAuthor.Name">

<Feedback>

<ValidationError/>

</Feedback>

</TextEdit>

</Field>

</Validation>

<Field>

<FieldLabel>@L["BirthDate"]</FieldLabel>

<DateEdit TValue="DateTime" @bind-Date="@NewAuthor.BirthDate"/>

</Field>

<Validation MessageLocalizer="@LH.Localize">

<Field>

<FieldLabel>@L["ShortBio"]</FieldLabel>

<MemoEdit Rows="5" @bind-Text="@NewAuthor.ShortBio">

<Feedback>

<ValidationError/>

</Feedback>

</MemoEdit>

</Field>

</Validation>

</Validations>

</ModalBody>

<ModalFooter>

<Button Color="Color.Secondary"

Clicked="CloseCreateAuthorModal">

@L["Cancel"]

</Button>

<Button Color="Color.Primary"

Type="@ButtonType.Submit"

PreventDefaultOnSubmit="true"

Clicked="CreateAuthorAsync">

@L["Save"]

</Button>

</ModalFooter>

</Form>

</ModalContent>

</Modal>

<Modal @ref="EditAuthorModal">

<ModalBackdrop />

<ModalContent IsCentered="true">

<Form>

<ModalHeader>

<ModalTitle>@EditingAuthor.Name</ModalTitle>

<CloseButton Clicked="CloseEditAuthorModal" />

</ModalHeader>

<ModalBody>

<Validations @ref="@EditValidationsRef" Model="@EditingAuthor" ValidateOnLoad="false">

<Validation MessageLocalizer="@LH.Localize">

<Field>

<FieldLabel>@L["Name"]</FieldLabel>

<TextEdit @bind-Text="@EditingAuthor.Name">

<Feedback>

<ValidationError/>

</Feedback>

</TextEdit>

</Field>

</Validation>

<Field>

<FieldLabel>@L["BirthDate"]</FieldLabel>

<DateEdit TValue="DateTime" @bind-Date="@EditingAuthor.BirthDate"/>

</Field>

<Validation>

<Field>

<FieldLabel>@L["ShortBio"]</FieldLabel>

<MemoEdit Rows="5" @bind-Text="@EditingAuthor.ShortBio">

<Feedback>

<ValidationError/>

</Feedback>

</MemoEdit>

</Field>

</Validation>

</Validations>

</ModalBody>

<ModalFooter>

<Button Color="Color.Secondary"

Clicked="CloseEditAuthorModal">

@L["Cancel"]

</Button>

<Button Color="Color.Primary"

Type="@ButtonType.Submit"

PreventDefaultOnSubmit="true"

Clicked="UpdateAuthorAsync">

@L["Save"]

</Button>

</ModalFooter>

</Form>

</ModalContent>

</Modal>

````

\* 这些代码类似 `Books.razor`, 除了不继承自 `AbpCrudPageBase`, 它使用自己的实现.

\* 注入 `IAuthorAppService` , 从UI使用服务器端的HTTP APIs . 我们可以直接注入应用服务接口并在 [动态 C# HTTP API 客户端代理系统](../API/Dynamic-CSharp-API-Clients.md)的帮助下像使用普通的方法一样使用它们, 动态 C# HTTP API 客户端代理系统会为我们调用REST API. 参考下面的 `Authors` 类获得使用方法.

\* 注入 `IAuthorizationService` 检查 [权限](../Authorization.md).

\* 注入 `IObjectMapper` 进行 [对象到对象映射](../Object-To-Object-Mapping.md).

在 `Pages` 文件夹下新建一个代码后置文件 `Authors.razor.cs`, 使用以下代码:

````csharp

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Acme.BookStore.Authors;

using Acme.BookStore.Permissions;

using Blazorise;

using Blazorise.DataGrid;

using Microsoft.AspNetCore.Authorization;

using Volo.Abp.Application.Dtos;

namespace Acme.BookStore.Blazor.Pages

{

public partial class Authors

{

private IReadOnlyList<AuthorDto> AuthorList { get; set; }

private int PageSize { get; } = LimitedResultRequestDto.DefaultMaxResultCount;

private int CurrentPage { get; set; }

private string CurrentSorting { get; set; }

private int TotalCount { get; set; }

private bool CanCreateAuthor { get; set; }

private bool CanEditAuthor { get; set; }

private bool CanDeleteAuthor { get; set; }

private CreateAuthorDto NewAuthor { get; set; }

private Guid EditingAuthorId { get; set; }

private UpdateAuthorDto EditingAuthor { get; set; }

private Modal CreateAuthorModal { get; set; }

private Modal EditAuthorModal { get; set; }

private Validations CreateValidationsRef;

private Validations EditValidationsRef;

public Authors()

{

NewAuthor = new CreateAuthorDto();

EditingAuthor = new UpdateAuthorDto();

}

protected override async Task OnInitializedAsync()

{

await SetPermissionsAsync();

await GetAuthorsAsync();

}

private async Task SetPermissionsAsync()

{

CanCreateAuthor = await AuthorizationService

.IsGrantedAsync(BookStorePermissions.Authors.Create);

CanEditAuthor = await AuthorizationService

.IsGrantedAsync(BookStorePermissions.Authors.Edit);

CanDeleteAuthor = await AuthorizationService

.IsGrantedAsync(BookStorePermissions.Authors.Delete);

}

private async Task GetAuthorsAsync()

{

var result = await AuthorAppService.GetListAsync(

new GetAuthorListDto

{

MaxResultCount = PageSize,

SkipCount = CurrentPage \* PageSize,

Sorting = CurrentSorting

}

);

AuthorList = result.Items;

TotalCount = (int)result.TotalCount;

}

private async Task OnDataGridReadAsync(DataGridReadDataEventArgs<AuthorDto> e)

{

CurrentSorting = e.Columns

.Where(c => c.Direction != SortDirection.None)

.Select(c => c.Field + (c.Direction == SortDirection.Descending ? " DESC" : ""))

.JoinAsString(",");

CurrentPage = e.Page - 1;

await GetAuthorsAsync();

await InvokeAsync(StateHasChanged);

}

private void OpenCreateAuthorModal()

{

CreateValidationsRef.ClearAll();

NewAuthor = new CreateAuthorDto();

CreateAuthorModal.Show();

}

private void CloseCreateAuthorModal()

{

CreateAuthorModal.Hide();

}

private void OpenEditAuthorModal(AuthorDto author)

{

EditValidationsRef.ClearAll();

EditingAuthorId = author.Id;

EditingAuthor = ObjectMapper.Map<AuthorDto, UpdateAuthorDto>(author);

EditAuthorModal.Show();

}

private async Task DeleteAuthorAsync(AuthorDto author)

{

var confirmMessage = L["AuthorDeletionConfirmationMessage", author.Name];

if (!await Message.Confirm(confirmMessage))

{

return;

}

await AuthorAppService.DeleteAsync(author.Id);

await GetAuthorsAsync();

}

private void CloseEditAuthorModal()

{

EditAuthorModal.Hide();

}

private async Task CreateAuthorAsync()

{

if (CreateValidationsRef.ValidateAll())

{

await AuthorAppService.CreateAsync(NewAuthor);

await GetAuthorsAsync();

CreateAuthorModal.Hide();

}

}

private async Task UpdateAuthorAsync()

{

if (EditValidationsRef.ValidateAll())

{

await AuthorAppService.UpdateAsync(EditingAuthorId, EditingAuthor);

await GetAuthorsAsync();

EditAuthorModal.Hide();

}

}

}

}

````

这个类定义了 `Authors.razor` 页面使用的属性和方法.

### 对象映射

`Authors` 类使用 `OpenEditAuthorModal` 方法中的 `IObjectMapper`. 所以需要定义这个映射.

打开 `Acme.BookStore.Blazor` 项目中的 `BookStoreBlazorAutoMapperProfile.cs`, 在构造函数中加入一个映射:

````csharp

CreateMap<AuthorDto, UpdateAuthorDto>();

````

你在文件开头需要 `using Acme.BookStore.Authors;` 声明语句.

### 加入主菜单

打开 `Acme.BookStore.Blazor` 项目的 `Menus` 文件夹中的 `BookStoreMenuContributor.cs` , 在 `ConfigureMainMenuAsync` 方法的结尾加入以下代码:

````csharp

if (await context.IsGrantedAsync(BookStorePermissions.Authors.Default))

{

bookStoreMenu.AddItem(new ApplicationMenuItem(

"BooksStore.Authors",

l["Menu:Authors"],

url: "/authors"

));

}

````

### 本地化

我们需要本地化上面的代码. 打开 `Acme.BookStore.Domain.Shared` 项目中 `Localization/BookStore` 文件夹下的 `en.json` 文件, 加入以下条目:

````json

"Menu:Authors": "Authors",

"Authors": "Authors",

"AuthorDeletionConfirmationMessage": "Are you sure to delete the author '{0}'?",

"BirthDate": "Birth date",

"NewAuthor": "New author"

````

> 简体中文翻译请打开`zh-Hans.json`文件 ,并将"Texts"对象中对应的值替换为中文.

### 运行应用程序

运行并登录应用程序. **\*\*因为你还没有权限, 所以不能看见菜单项\*\***. 转到 `Identity/Roles` 页面, 点击 *\*操作\** 按钮并选择**\*\*管理员角色\*\***的*\*权限\**操作:

![bookstore-author-permissions](images/bookstore-author-permissions.png)

如你所见, 管理员角色还没有*\*作者管理\**权限. 单击复选框并保存, 赋予权限. **\*\*刷新页面\*\***后, 你会在主菜单中的*\*图书商店\**下看到*\*作者\**菜单项:

![bookstore-authors-page](images/bookstore-authors-blazor-ui.png)

这就是全部了! 这是一个完整的, 可以工作的页面. 你可以新建, 编辑和删除作者.

> **\*\*提示\*\***: 如果你在定义一个新权限后运行 `.DbMigrator` 控制台程序, 它会自动将这些权限赋予管理员角色, 你不需要手工赋予权限.

{{end}}

## 下一章

查看本教程的[下一章](Part-10.md).

图书-作者关系

# Web应用程序开发教程 - 第十章: 图书到作者的关系

````json

//[doc-params]

{

"UI": ["MVC","Blazor","BlazorServer","NG"],

"DB": ["EF","Mongo"]

}

````

## 关于本教程

在本系列教程中, 你将构建一个名为 `Acme.BookStore` 的用于管理书籍及其作者列表的基于ABP的应用程序. 它是使用以下技术开发的:

\* **\*\*{{DB\_Text}}\*\*** 做为ORM提供程序.

\* **\*\*{{UI\_Value}}\*\*** 做为UI框架.

本教程分为以下部分:

- [Part 1: 创建服务端](Part-1.md)

- [Part 2: 图书列表页面](Part-2.md)

- [Part 3: 创建,更新和删除图书](Part-2.md)

- [Part 4: 集成测试](Part-4.md)

- [Part 5: 授权](Part-5.md)

- [Part 6: 作者: 领域层](Part-6.md)

- [Part 7: 作者: 数据库集成](Part-7.md)

- [Part 8: 作者: 应用服务层](Part-8.md)

- [Part 9: 作者: 用户页面](Part-9.md)

- **\*\*Part 10: 图书到作者的关系 (本章)\*\***

## 下载源码

本教程根据你的**\*\*UI\*\*** 和 **\*\*数据库\*\***偏好有多个版本,我们准备了几种可供下载的源码组合:

\* [MVC (Razor Pages) UI 与 EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Mvc-EfCore)

\* [Blazor UI 与 EF Core](https://github.com/abpframework/abp-samples/tree/master/BookStore-Blazor-EfCore)

\* [Angular UI 与 MongoDB](https://github.com/abpframework/abp-samples/tree/master/BookStore-Angular-MongoDb)

> 如果你在Windows中遇到 "文件名太长" or "解压错误", 很可能与Windows最大文件路径限制有关. Windows文件路径的最大长度为250字符. 为了解决这个问题,参阅 [在Windows 10中启用长路径](https://docs.microsoft.com/en-us/windows/win32/fileio/maximum-file-path-limitation?tabs=cmd#enable-long-paths-in-windows-10-version-1607-and-later).

> 如果你遇到与Git相关的长路径错误, 尝试使用下面的命令在Windows中启用长路径. 参阅 https://github.com/msysgit/msysgit/wiki/Git-cannot-create-a-file-or-directory-with-a-long-path

> `git config --system core.longpaths true`

## 简介

我们已经为图书管理应用程序创建了 `图书` 和 `作者` 功能. 然而, 这些实体间还没有关联.

在本章, 我们会在 `作者` 和 `图书` 实体间建立 **\*\*1 对 N\*\*** 的关系.

## 在图书实体中加入关系

打开 `Acme.BookStore.Domain` 项目中的 `Books/Book.cs`, 在 `Book` 实体中加入下列属性:

````csharp

public Guid AuthorId { get; set; }

````

{{if DB=="EF"}}

> 在本章中, 我们选择不在 `Book` 类中加入 `Author` 实体的 **\*\*导航属性\*\*** (例如 `public Author Author { get; set; }`). 这是为了遵循 DDD 最佳实践 (规则: 仅通过id引用其它聚合对象). 但是, 你自己可以添加这样的导航属性, 并为EF Core配置它. 这样, 你在获取图书和它们的作者时就不需要写join查询了(如同下面我们做的一样), 这会使代码更简洁一些.

{{end}}

## 数据库 & 数据迁移

为 `Book` 实体新增一个不为空的 `AuthorId` 属性. 但是, 数据库中**\*\*已存在的图书怎么办\*\***? 它们没有 `AuthorId`s, 当我们尝试运行应用程序时会出问题.

这是一个 **\*\*典型的迁移问题\*\***, 解决方案依赖于你的具体情况;

\* 如果你还没有发布应用程序到生产环境, 你可以直接删除数据库中的图书数据, 甚至你可以删除开发环境中的整个数据库.

\* 你可以在数据迁移或生成种子阶段使用代码更新已有数据.

\* 你可以手工处理这些数据.

我们倾向于 **\*\*删除数据库\*\*** {{if DB=="EF"}}(你可以在 *\*Package Manager 控制台\**中运行 `Drop-Database`){{end}}, 因为这只是个示例项目, 数据丢失并不要紧. 因为这个主题不是关于ABP框架的, 我们不会深入所有的场景.

{{if DB=="EF"}}

### 更新 EF Core 映射

定位到 `Acme.BookStore.EntityFrameworkCore` 项目的 `EntityFrameworkCore` 文件夹下的 `BookStoreDbContext` 类的 `OnModelCreating` 方法, 修改 `builder.Entity<Book>` 部分如下:

````csharp

builder.Entity<Book>(b =>

{

b.ToTable(BookStoreConsts.DbTablePrefix + "Books", BookStoreConsts.DbSchema);

b.ConfigureByConvention(); //auto configure for the base class props

b.Property(x => x.Name).IsRequired().HasMaxLength(128);

// ADD THE MAPPING FOR THE RELATION

b.HasOne<Author>().WithMany().HasForeignKey(x => x.AuthorId).IsRequired();

});

````

### 新增 EF Core 迁移

启动解决方案被配置为使用 [Entity Framework Core Code First Migrations](https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/). 因为我们修改了数据库映射配置, 我们需要新建一个迁移并应用于数据库.

在 `Acme.BookStore.EntityFrameworkCore` 项目的文件目录打开命令行终端, 输入命令:

````bash

dotnet ef migrations add Added\_AuthorId\_To\_Book

````

这会创建一个新的迁移类, 在它的 `Up` 方法中使用下列方法:

````csharp

migrationBuilder.AddColumn<Guid>(

name: "AuthorId",

table: "AppBooks",

nullable: false,

defaultValue: new Guid("00000000-0000-0000-0000-000000000000"));

migrationBuilder.CreateIndex(

name: "IX\_AppBooks\_AuthorId",

table: "AppBooks",

column: "AuthorId");

migrationBuilder.AddForeignKey(

name: "FK\_AppBooks\_AppAuthors\_AuthorId",

table: "AppBooks",

column: "AuthorId",

principalTable: "AppAuthors",

principalColumn: "Id",

onDelete: ReferentialAction.Cascade);

````

\* `AppBooks` 表增加一个 `AuthorId` 字段 .

\* 根据 `AuthorId` 字段新建一个索引.

\* 声明到 `AppAuthors` 表的外键.

> 如果你使用 Visual Studio, 可能希望在 *\*Package Manager Console (PMC)\** 使用 `Add-Migration Added\_AuthorId\_To\_Book -c BookStoreDbContext` 和 `Update-Database -Context BookStoreDbContext` 命令. 如果这样, 保证 {{if UI=="MVC"}}`Acme.BookStore.Web`{{else if UI=="BlazorServer"}}`Acme.BookStore.Blazor`{{else if UI=="Blazor" || UI=="NG"}}`Acme.BookStore.HttpApi.Host`{{end}} 是启动项目并且在PMC中 `Acme.BookStore.EntityFrameworkCore` 是 *\*默认项目\** .

{{end}}

## 修改数据种子

因为 `AuthorId` 是 `Book` 实体的不可为空属性, 当前的数据种子代码不能工作. 打开 `Acme.BookStore.Domain` 项目中的 `BookStoreDataSeederContributor`, 修改成以下代码:

````csharp

using System;

using System.Threading.Tasks;

using Acme.BookStore.Authors;

using Acme.BookStore.Books;

using Volo.Abp.Data;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Domain.Repositories;

namespace Acme.BookStore

{

public class BookStoreDataSeederContributor

: IDataSeedContributor, ITransientDependency

{

private readonly IRepository<Book, Guid> \_bookRepository;

private readonly IAuthorRepository \_authorRepository;

private readonly AuthorManager \_authorManager;

public BookStoreDataSeederContributor(

IRepository<Book, Guid> bookRepository,

IAuthorRepository authorRepository,

AuthorManager authorManager)

{

\_bookRepository = bookRepository;

\_authorRepository = authorRepository;

\_authorManager = authorManager;

}

public async Task SeedAsync(DataSeedContext context)

{

if (await \_bookRepository.GetCountAsync() > 0)

{

return;

}

var orwell = await \_authorRepository.InsertAsync(

await \_authorManager.CreateAsync(

"George Orwell",

new DateTime(1903, 06, 25),

"Orwell produced literary criticism and poetry, fiction and polemical journalism; and is best known for the allegorical novella Animal Farm (1945) and the dystopian novel Nineteen Eighty-Four (1949)."

)

);

var douglas = await \_authorRepository.InsertAsync(

await \_authorManager.CreateAsync(

"Douglas Adams",

new DateTime(1952, 03, 11),

"Douglas Adams was an English author, screenwriter, essayist, humorist, satirist and dramatist. Adams was an advocate for environmentalism and conservation, a lover of fast cars, technological innovation and the Apple Macintosh, and a self-proclaimed 'radical atheist'."

)

);

await \_bookRepository.InsertAsync(

new Book

{

AuthorId = orwell.Id, // SET THE AUTHOR

Name = "1984",

Type = BookType.Dystopia,

PublishDate = new DateTime(1949, 6, 8),

Price = 19.84f

},

autoSave: true

);

await \_bookRepository.InsertAsync(

new Book

{

AuthorId = douglas.Id, // SET THE AUTHOR

Name = "The Hitchhiker's Guide to the Galaxy",

Type = BookType.ScienceFiction,

PublishDate = new DateTime(1995, 9, 27),

Price = 42.0f

},

autoSave: true

);

}

}

}

````

唯一的区别是设置 `Book` 实体的 `AuthorId` 属性.

> 执行 `DbMigrator` 前删除已有图书或数据库. 参阅上面的 *\*数据库 & 数据迁移\** 小节获取详细信息.

{{if DB=="EF"}}

你现在可以运行 `.DbMigrator` 控制台应用程序, **\*\*迁移\*\*** **\*\*数据库 schema\*\*** 并生成 **\*\*种子\*\*** 初始数据.

{{else if DB=="Mongo"}}

你现在可以运行 `.DbMigrator` 控制台应用程序, **\*\*迁移\*\*** **\*\*数据库 schema\*\*** 并生成 **\*\*种子\*\*** 初始数据.

{{end}}

## 应用层

我们将修改 `BookAppService`, 支持作者关系.

### 数据传输对象

让我们从DTOs开始.

#### BookDto

打开 `Acme.BookStore.Application.Contracts` 项目的 `Books` 文件夹下的 `BookDto` 类, 添加如下属性:

```csharp

public Guid AuthorId { get; set; }

public string AuthorName { get; set; }

```

最终的 `BookDto` 类应该如下:

```csharp

using System;

using Volo.Abp.Application.Dtos;

namespace Acme.BookStore.Books

{

public class BookDto : AuditedEntityDto<Guid>

{

public Guid AuthorId { get; set; }

public string AuthorName { get; set; }

public string Name { get; set; }

public BookType Type { get; set; }

public DateTime PublishDate { get; set; }

public float Price { get; set; }

}

}

```

#### CreateUpdateBookDto

打开 `Acme.BookStore.Application.Contracts` 项目的 `Books` 文件夹下的 `CreateUpdateBookDto` 类, 添加 `AuthorId` 属性:

````csharp

public Guid AuthorId { get; set; }

````

#### AuthorLookupDto

在 `Acme.BookStore.Application.Contracts` 项目的 `Books` 文件夹下新建一个类 `AuthorLookupDto`:

````csharp

using System;

using Volo.Abp.Application.Dtos;

namespace Acme.BookStore.Books

{

public class AuthorLookupDto : EntityDto<Guid>

{

public string Name { get; set; }

}

}

````

它会被一个将要添加到 `IBookAppService` 的新方法使用.

### IBookAppService

打开 `Acme.BookStore.Application.Contracts` 项目的 `Books` 文件夹下的 `IBookAppService` 接口, 添加一个名为 `GetAuthorLookupAsync` 的新方法:

````csharp

using System;

using System.Threading.Tasks;

using Volo.Abp.Application.Dtos;

using Volo.Abp.Application.Services;

namespace Acme.BookStore.Books

{

public interface IBookAppService :

ICrudAppService< //Defines CRUD methods

BookDto, //Used to show books

Guid, //Primary key of the book entity

PagedAndSortedResultRequestDto, //Used for paging/sorting

CreateUpdateBookDto> //Used to create/update a book

{

// ADD the NEW METHOD

Task<ListResultDto<AuthorLookupDto>> GetAuthorLookupAsync();

}

}

````

这个新方法将被UI用来获取作者列表, 填充一个下拉框. 使用这个下拉框选择图书作者.

### BookAppService

打开 `Acme.BookStore.Application` 项目的 `Books` 文件夹下的 `BookAppService` 类, 更新为以下代码:

{{if DB=="EF"}}

```csharp

using System;

using System.Collections.Generic;

using System.Linq;

using System.Linq.Dynamic.Core;

using System.Threading.Tasks;

using Acme.BookStore.Authors;

using Acme.BookStore.Permissions;

using Microsoft.AspNetCore.Authorization;

using Volo.Abp.Application.Dtos;

using Volo.Abp.Application.Services;

using Volo.Abp.Domain.Entities;

using Volo.Abp.Domain.Repositories;

namespace Acme.BookStore.Books

{

[Authorize(BookStorePermissions.Books.Default)]

public class BookAppService :

CrudAppService<

Book, //The Book entity

BookDto, //Used to show books

Guid, //Primary key of the book entity

PagedAndSortedResultRequestDto, //Used for paging/sorting

CreateUpdateBookDto>, //Used to create/update a book

IBookAppService //implement the IBookAppService

{

private readonly IAuthorRepository \_authorRepository;

public BookAppService(

IRepository<Book, Guid> repository,

IAuthorRepository authorRepository)

: base(repository)

{

\_authorRepository = authorRepository;

GetPolicyName = BookStorePermissions.Books.Default;

GetListPolicyName = BookStorePermissions.Books.Default;

CreatePolicyName = BookStorePermissions.Books.Create;

UpdatePolicyName = BookStorePermissions.Books.Edit;

DeletePolicyName = BookStorePermissions.Books.Delete;

}

public override async Task<BookDto> GetAsync(Guid id)

{

//Get the IQueryable<Book> from the repository

var queryable = await Repository.GetQueryableAsync();

//Prepare a query to join books and authors

var query = from book in queryable

join author in await \_authorRepository.GetQueryableAsync() on book.AuthorId equals author.Id

where book.Id == id

select new { book, author };

//Execute the query and get the book with author

var queryResult = await AsyncExecuter.FirstOrDefaultAsync(query);

if (queryResult == null)

{

throw new EntityNotFoundException(typeof(Book), id);

}

var bookDto = ObjectMapper.Map<Book, BookDto>(queryResult.book);

bookDto.AuthorName = queryResult.author.Name;

return bookDto;

}

public override async Task<PagedResultDto<BookDto>> GetListAsync(PagedAndSortedResultRequestDto input)

{

//Get the IQueryable<Book> from the repository

var queryable = await Repository.GetQueryableAsync();

//Prepare a query to join books and authors

var query = from book in queryable

join author in await \_authorRepository.GetQueryableAsync() on book.AuthorId equals author.Id

select new {book, author};

//Paging

query = query

.OrderBy(NormalizeSorting(input.Sorting))

.Skip(input.SkipCount)

.Take(input.MaxResultCount);

//Execute the query and get a list

var queryResult = await AsyncExecuter.ToListAsync(query);

//Convert the query result to a list of BookDto objects

var bookDtos = queryResult.Select(x =>

{

var bookDto = ObjectMapper.Map<Book, BookDto>(x.book);

bookDto.AuthorName = x.author.Name;

return bookDto;

}).ToList();

//Get the total count with another query

var totalCount = await Repository.GetCountAsync();

return new PagedResultDto<BookDto>(

totalCount,

bookDtos

);

}

public async Task<ListResultDto<AuthorLookupDto>> GetAuthorLookupAsync()

{

var authors = await \_authorRepository.GetListAsync();

return new ListResultDto<AuthorLookupDto>(

ObjectMapper.Map<List<Author>, List<AuthorLookupDto>>(authors)

);

}

private static string NormalizeSorting(string sorting)

{

if (sorting.IsNullOrEmpty())

{

return $"book.{nameof(Book.Name)}";

}

if (sorting.Contains("authorName", StringComparison.OrdinalIgnoreCase))

{

return sorting.Replace(

"authorName",

"author.Name",

StringComparison.OrdinalIgnoreCase

);

}

return $"book.{sorting}";

}

}

}

```

我们做了以下修改:

\* 给所有新建/覆写的方法增加 `[Authorize(BookStorePermissions.Books.Default)]` 进行授权(当授权特性应用于类时, 它对这个类的所有方法有效).

\* 注入 `IAuthorRepository`, 从作者中查询.

\* 覆写基类 `CrudAppService` 的 `GetAsync` 方法. 这个方法根据给定的 `id` 返回单一 `BookDto` 对象.

\* 使用一个简单的LINQ表达式关联图书和作者, 根据给定的图书id查询, 查询结果同时包含图书和作者.

\* 使用 `AsyncExecuter.FirstOrDefaultAsync(...)` 执行查询并得到一个结果. 这是一种无需依赖database provider API, 使用异步LINQ扩展的方法. 参阅 [repository文档](../Repositories.md)以理解我们为什么使用它.

\* 如果请求的图书在数据库中不存在, 抛出一个 `EntityNotFoundException`, 这会导致一个 `HTTP 404` (not found) 状态码.

\* 最后, 使用 `ObjectMapper`创建一个 `BookDto` 对象, 然后手工给 `AuthorName` 赋值.

\* 覆写 `CrudAppService` 基类的 `GetListAsync` 方法, 返回图书列表. 逻辑与前一个方法类似, 所以很容易理解.

\* 新建一个方法: `GetAuthorLookupAsync`. 这个方法只是简单地获取所有作者. UI使用这个方法填充一个下拉框, 当编辑图书时用来选择作者.

{{else if DB=="Mongo"}}

```csharp

using System;

using System.Collections.Generic;

using System.Linq.Dynamic.Core;

using System.Linq;

using System.Threading.Tasks;

using Acme.BookStore.Authors;

using Acme.BookStore.Permissions;

using Microsoft.AspNetCore.Authorization;

using Volo.Abp.Application.Dtos;

using Volo.Abp.Application.Services;

using Volo.Abp.Domain.Repositories;

namespace Acme.BookStore.Books

{

[Authorize(BookStorePermissions.Books.Default)]

public class BookAppService :

CrudAppService<

Book, //The Book entity

BookDto, //Used to show books

Guid, //Primary key of the book entity

PagedAndSortedResultRequestDto, //Used for paging/sorting

CreateUpdateBookDto>, //Used to create/update a book

IBookAppService //implement the IBookAppService

{

private readonly IAuthorRepository \_authorRepository;

public BookAppService(

IRepository<Book, Guid> repository,

IAuthorRepository authorRepository)

: base(repository)

{

\_authorRepository = authorRepository;

GetPolicyName = BookStorePermissions.Books.Default;

GetListPolicyName = BookStorePermissions.Books.Default;

CreatePolicyName = BookStorePermissions.Books.Create;

UpdatePolicyName = BookStorePermissions.Books.Edit;

DeletePolicyName = BookStorePermissions.Books.Create;

}

public async override Task<BookDto> GetAsync(Guid id)

{

var book = await Repository.GetAsync(id);

var bookDto = ObjectMapper.Map<Book, BookDto>(book);

var author = await \_authorRepository.GetAsync(book.AuthorId);

bookDto.AuthorName = author.Name;

return bookDto;

}

public async override Task<PagedResultDto<BookDto>>

GetListAsync(PagedAndSortedResultRequestDto input)

{

//Set a default sorting, if not provided

if (input.Sorting.IsNullOrWhiteSpace())

{

input.Sorting = nameof(Book.Name);

}

//Get the IQueryable<Book> from the repository

var queryable = await Repository.GetQueryableAsync();

//Get the books

var books = await AsyncExecuter.ToListAsync(

queryable

.OrderBy(input.Sorting)

.Skip(input.SkipCount)

.Take(input.MaxResultCount)

);

//Convert to DTOs

var bookDtos = ObjectMapper.Map<List<Book>, List<BookDto>>(books);

//Get a lookup dictionary for the related authors

var authorDictionary = await GetAuthorDictionaryAsync(books);

//Set AuthorName for the DTOs

bookDtos.ForEach(bookDto => bookDto.AuthorName =

authorDictionary[bookDto.AuthorId].Name);

//Get the total count with another query (required for the paging)

var totalCount = await Repository.GetCountAsync();

return new PagedResultDto<BookDto>(

totalCount,

bookDtos

);

}

public async Task<ListResultDto<AuthorLookupDto>> GetAuthorLookupAsync()

{

var authors = await \_authorRepository.GetListAsync();

return new ListResultDto<AuthorLookupDto>(

ObjectMapper.Map<List<Author>, List<AuthorLookupDto>>(authors)

);

}

private async Task<Dictionary<Guid, Author>>

GetAuthorDictionaryAsync(List<Book> books)

{

var authorIds = books

.Select(b => b.AuthorId)

.Distinct()

.ToArray();

var queryable = await \_authorRepository.GetQueryableAsync();

var authors = await AsyncExecuter.ToListAsync(

queryable.Where(a => authorIds.Contains(a.Id))

);

return authors.ToDictionary(x => x.Id, x => x);

}

}

}

```

我们做了以下修改:

\* 给所有新建/覆写的方法增加 `[Authorize(BookStorePermissions.Books.Default)]` 进行授权(当授权特性应用于类时, 它对这个类的所有方法有效).

\* 注入 `IAuthorRepository`, 从作者中查询.

\* 覆写基类 `CrudAppService` 的 `GetAsync` 方法. 这个方法根据给定的 `id` 返回单一 `BookDto` 对象.

\* 覆写 `CrudAppService` 基类的 `GetListAsync` 方法, 返回图书列表. 这里将从数据库中查询作者和在应用层设置作者名进行了分离. 你也可以创建一个自定义repository方法, 执行一个join查询, 或者利用MongoDB AP在一个查询中获取图书的作者, 这种做法性能会更好.

\* 新建一个方法: `GetAuthorLookupAsync`. 这个方法只是简单地获取所有作者. UI使用这个方法填充一个下拉框, 当编辑图书时用来选择作者.

{{end}}

### 对象到对象映射映射

引入 `AuthorLookupDto` 类, 在 `GetAuthorLookupAsync` 方法中使用对象映射. 所以, 我们需要在 `Acme.BookStore.Application` 项目的 `BookStoreApplicationAutoMapperProfile.cs` 文件中加入一个新的映射定义.

````csharp

CreateMap<Author, AuthorLookupDto>();

````

## 单元测试

因为修改了 `AuthorAppService`, 一些单元测试失败了. 打开 `Acme.BookStore.Application.Tests` 项目的 `Books` 目录中的 `BookAppService\_Tests`, 修改成以下代码:

```csharp

using System;

using System.Linq;

using System.Threading.Tasks;

using Acme.BookStore.Authors;

using Shouldly;

using Volo.Abp.Application.Dtos;

using Volo.Abp.Validation;

using Xunit;

namespace Acme.BookStore.Books

{ {{if DB=="Mongo"}}

[Collection(BookStoreTestConsts.CollectionDefinitionName)]{{end}}

public class BookAppService\_Tests : BookStoreApplicationTestBase

{

private readonly IBookAppService \_bookAppService;

private readonly IAuthorAppService \_authorAppService;

public BookAppService\_Tests()

{

\_bookAppService = GetRequiredService<IBookAppService>();

\_authorAppService = GetRequiredService<IAuthorAppService>();

}

[Fact]

public async Task Should\_Get\_List\_Of\_Books()

{

//Act

var result = await \_bookAppService.GetListAsync(

new PagedAndSortedResultRequestDto()

);

//Assert

result.TotalCount.ShouldBeGreaterThan(0);

result.Items.ShouldContain(b => b.Name == "1984" &&

b.AuthorName == "George Orwell");

}

[Fact]

public async Task Should\_Create\_A\_Valid\_Book()

{

var authors = await \_authorAppService.GetListAsync(new GetAuthorListDto());

var firstAuthor = authors.Items.First();

//Act

var result = await \_bookAppService.CreateAsync(

new CreateUpdateBookDto

{

AuthorId = firstAuthor.Id,

Name = "New test book 42",

Price = 10,

PublishDate = System.DateTime.Now,

Type = BookType.ScienceFiction

}

);

//Assert

result.Id.ShouldNotBe(Guid.Empty);

result.Name.ShouldBe("New test book 42");

}

[Fact]

public async Task Should\_Not\_Create\_A\_Book\_Without\_Name()

{

var exception = await Assert.ThrowsAsync<AbpValidationException>(async () =>

{

await \_bookAppService.CreateAsync(

new CreateUpdateBookDto

{

Name = "",

Price = 10,

PublishDate = DateTime.Now,

Type = BookType.ScienceFiction

}

);

});

exception.ValidationErrors

.ShouldContain(err => err.MemberNames.Any(m => m == "Name"));

}

}

}

```

\* 修改 `Should\_Get\_List\_Of\_Books` 中的断言条件, 从 `b => b.Name == "1984"` 修改为 `b => b.Name == "1984" && b.AuthorName == "George Orwell"`, 检查用户名是否被填充.

\* 修改 `Should\_Create\_A\_Valid\_Book` 方法, 当新建图书时, 设置 `AuthorId`, 因为它现在是不可为空的了.

## 用户页面

{{if UI=="MVC"}}

### 图书列表

图书列表页面的修改很小. 打开 `Acme.BookStore.Web` 项目上的 `Pages/Books/Index.js`, 在 `name` and `type` 列之间加入如下列定义:

````js

...

{

title: l('Name'),

data: "name"

},

// ADDED the NEW AUTHOR NAME COLUMN

{

title: l('Author'),

data: "authorName"

},

{

title: l('Type'),

data: "type",

render: function (data) {

return l('Enum:BookType:' + data);

}

},

...

````

运行应用程序, 你会在表格中看到 *\*Author\** 列:

![bookstore-added-author-to-book-list](images/bookstore-added-author-to-book-list.png)

### 新建模态窗口

打开 `Acme.BookStore.Web` 项目中的 `Pages/Books/CreateModal.cshtml.cs`, 修改文件内容为:

```csharp

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.ComponentModel.DataAnnotations;

using System.Linq;

using System.Threading.Tasks;

using Acme.BookStore.Books;

using Microsoft.AspNetCore.Mvc;

using Microsoft.AspNetCore.Mvc.Rendering;

using Volo.Abp.AspNetCore.Mvc.UI.Bootstrap.TagHelpers.Form;

namespace Acme.BookStore.Web.Pages.Books

{

public class CreateModalModel : BookStorePageModel

{

[BindProperty]

public CreateBookViewModel Book { get; set; }

public List<SelectListItem> Authors { get; set; }

private readonly IBookAppService \_bookAppService;

public CreateModalModel(

IBookAppService bookAppService)

{

\_bookAppService = bookAppService;

}

public async Task OnGetAsync()

{

Book = new CreateBookViewModel();

var authorLookup = await \_bookAppService.GetAuthorLookupAsync();

Authors = authorLookup.Items

.Select(x => new SelectListItem(x.Name, x.Id.ToString()))

.ToList();

}

public async Task<IActionResult> OnPostAsync()

{

await \_bookAppService.CreateAsync(

ObjectMapper.Map<CreateBookViewModel, CreateUpdateBookDto>(Book)

);

return NoContent();

}

public class CreateBookViewModel

{

[SelectItems(nameof(Authors))]

[DisplayName("Author")]

public Guid AuthorId { get; set; }

[Required]

[StringLength(128)]

public string Name { get; set; }

[Required]

public BookType Type { get; set; } = BookType.Undefined;

[Required]

[DataType(DataType.Date)]

public DateTime PublishDate { get; set; } = DateTime.Now;

[Required]

public float Price { get; set; }

}

}

}

```

\* 将 `Book` 属性的类型从 `CreateUpdateBookDto` 修改为这个文件中新定义的 `CreateBookViewModel` 类. 这个修改的主要动机是根据UI需求自定义模型类. 我们不希望在 `CreateUpdateBookDto` 类中使用UI相关的 `[SelectItems(nameof(Authors))]` 和 `[DisplayName("Author")]` 特性.

\* 新增 `Authors` 属性, 在 `OnGetAsync` 方法中使用前面定义的 `IBookAppService.GetAuthorLookupAsync` 方法填充它.

\* 修改 `OnPostAsync` 方法, 映射 `CreateBookViewModel` 对象到 `CreateUpdateBookDto` 对象, 因为 `IBookAppService.CreateAsync` 需要一个这种类型的参数.

### 编辑模态窗口

打开 `Acme.BookStore.Web` 项目中的 `Pages/Books/EditModal.cshtml.cs`, 修改文件内容为:

```csharp

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.ComponentModel.DataAnnotations;

using System.Linq;

using System.Threading.Tasks;

using Acme.BookStore.Books;

using Microsoft.AspNetCore.Mvc;

using Microsoft.AspNetCore.Mvc.Rendering;

using Volo.Abp.AspNetCore.Mvc.UI.Bootstrap.TagHelpers.Form;

namespace Acme.BookStore.Web.Pages.Books

{

public class EditModalModel : BookStorePageModel

{

[BindProperty]

public EditBookViewModel Book { get; set; }

public List<SelectListItem> Authors { get; set; }

private readonly IBookAppService \_bookAppService;

public EditModalModel(IBookAppService bookAppService)

{

\_bookAppService = bookAppService;

}

public async Task OnGetAsync(Guid id)

{

var bookDto = await \_bookAppService.GetAsync(id);

Book = ObjectMapper.Map<BookDto, EditBookViewModel>(bookDto);

var authorLookup = await \_bookAppService.GetAuthorLookupAsync();

Authors = authorLookup.Items

.Select(x => new SelectListItem(x.Name, x.Id.ToString()))

.ToList();

}

public async Task<IActionResult> OnPostAsync()

{

await \_bookAppService.UpdateAsync(

Book.Id,

ObjectMapper.Map<EditBookViewModel, CreateUpdateBookDto>(Book)

);

return NoContent();

}

public class EditBookViewModel

{

[HiddenInput]

public Guid Id { get; set; }

[SelectItems(nameof(Authors))]

[DisplayName("Author")]

public Guid AuthorId { get; set; }

[Required]

[StringLength(128)]

public string Name { get; set; }

[Required]

public BookType Type { get; set; } = BookType.Undefined;

[Required]

[DataType(DataType.Date)]

public DateTime PublishDate { get; set; } = DateTime.Now;

[Required]

public float Price { get; set; }

}

}

}

```

\* 将 `Book` 属性的类型从 `CreateUpdateBookDto` 修改为这个文件中新定义的 `EditBookViewModel` 类, 和我们前面所做的创建模型的修改一样.

\* 移动新类 `EditBookViewModel` 的 `Id` 属性.

\* 新增 `Authors` 属性, 在 `OnGetAsync` 方法中使用前面定义的 `IBookAppService.GetAuthorLookupAsync` 方法填充它.

\* 修改 `OnPostAsync` 方法, 映射 `EditBookViewModel` 对象到 `CreateUpdateBookDto` 对象, 因为 `IBookAppService.UpdateAsync` 需要一个这种类型的参数.

这些修改需要对 `EditModal.cshtml` 进行一些小修改. 移除 `<abp-input asp-for="Id" />` 标签, 因为我们不再需要它了 (因为它被移动到 `EditBookViewModel` 中了). `EditModal.cshtml` 的最终内容应为:

````html

@page

@using Acme.BookStore.Localization

@using Acme.BookStore.Web.Pages.Books

@using Microsoft.Extensions.Localization

@using Volo.Abp.AspNetCore.Mvc.UI.Bootstrap.TagHelpers.Modal

@model EditModalModel

@inject IStringLocalizer<BookStoreResource> L

@{

Layout = null;

}

<abp-dynamic-form abp-model="Book" asp-page="/Books/EditModal">

<abp-modal>

<abp-modal-header title="@L["Update"].Value"></abp-modal-header>

<abp-modal-body>

<abp-form-content />

</abp-modal-body>

<abp-modal-footer buttons="@(AbpModalButtons.Cancel|AbpModalButtons.Save)"></abp-modal-footer>

</abp-modal>

</abp-dynamic-form>

````

### 对象到对象映射配置

以下修改需要定义一些对象到对象映射. 打开 `Acme.BookStore.Web` 项目中的 `BookStoreWebAutoMapperProfile.cs`, 在构造函数中添加下列映射定义:

```csharp

CreateMap<Pages.Books.CreateModalModel.CreateBookViewModel, CreateUpdateBookDto>();

CreateMap<BookDto, Pages.Books.EditModalModel.EditBookViewModel>();

CreateMap<Pages.Books.EditModalModel.EditBookViewModel, CreateUpdateBookDto>();

```

你可以运行应用程序, 尝试新建或更新一本书. 你将在新建/更新表单上看到一个下拉框, 使用它指定图书的作者:

![bookstore-added-authors-to-modals](images/bookstore-added-authors-to-modals.png)

{{else if UI=="NG"}}

### 生成服务代理

因为修改了 HTTP APIs, 你需要更新 Angular 客户端 [服务代理](../UI/Angular/Service-Proxies.md). 运行 `generate-proxy` 命令前, 应用程序必须启动运行.

在 `angular` 文件夹中运行下面的命令 (你可能需要停止angular应用程序):

```bash

abp generate-proxy -t ng

```

这个命令将会更新 `/src/app/proxy/` 文件夹中的服务代理文件.

### 图书列表

图书列表页面的修改很小. 打开 `/src/app/book/book.component.html`, 在 `Name` 和 `Type` 列之间加入以下列定义:

````html

<ngx-datatable-column

[name]="'::Author' | abpLocalization"

prop="authorName"

[sortable]="false"

></ngx-datatable-column>

````

运行应用程序, 你会在表格中看到 *\*作者\** 列:

![bookstore-books-with-authorname-angular](images/bookstore-books-with-authorname-angular.png)

### 新建/编辑 表单

下一步是添加作者选择下拉框到新建/编辑表单. 最终的页面如下图:

![bookstore-angular-author-selection](images/bookstore-angular-author-selection.png)

添加作者下拉框, 作为表单中的第一个元素.

打开 `/src/app/book/book.component.ts`, 修改文件内容为:

````js

import { ListService, PagedResultDto } from '@abp/ng.core';

import { Component, OnInit } from '@angular/core';

import { BookService, BookDto, bookTypeOptions, AuthorLookupDto } from '@proxy/books';

import { FormGroup, FormBuilder, Validators } from '@angular/forms';

import { NgbDateNativeAdapter, NgbDateAdapter } from '@ng-bootstrap/ng-bootstrap';

import { ConfirmationService, Confirmation } from '@abp/ng.theme.shared';

import { Observable } from 'rxjs';

import { map } from 'rxjs/operators';

@Component({

selector: 'app-book',

templateUrl: './book.component.html',

styleUrls: ['./book.component.scss'],

providers: [ListService, { provide: NgbDateAdapter, useClass: NgbDateNativeAdapter }],

})

export class BookComponent implements OnInit {

book = { items: [], totalCount: 0 } as PagedResultDto<BookDto>;

form: FormGroup;

selectedBook = {} as BookDto;

authors$: Observable<AuthorLookupDto[]>;

bookTypes = bookTypeOptions;

isModalOpen = false;

constructor(

public readonly list: ListService,

private bookService: BookService,

private fb: FormBuilder,

private confirmation: ConfirmationService

) {

this.authors$ = bookService.getAuthorLookup().pipe(map((r) => r.items));

}

ngOnInit() {

const bookStreamCreator = (query) => this.bookService.getList(query);

this.list.hookToQuery(bookStreamCreator).subscribe((response) => {

this.book = response;

});

}

createBook() {

this.selectedBook = {} as BookDto;

this.buildForm();

this.isModalOpen = true;

}

editBook(id: string) {

this.bookService.get(id).subscribe((book) => {

this.selectedBook = book;

this.buildForm();

this.isModalOpen = true;

});

}

buildForm() {

this.form = this.fb.group({

authorId: [this.selectedBook.authorId || null, Validators.required],

name: [this.selectedBook.name || null, Validators.required],

type: [this.selectedBook.type || null, Validators.required],

publishDate: [

this.selectedBook.publishDate ? new Date(this.selectedBook.publishDate) : null,

Validators.required,

],

price: [this.selectedBook.price || null, Validators.required],

});

}

save() {

if (this.form.invalid) {

return;

}

const request = this.selectedBook.id

? this.bookService.update(this.selectedBook.id, this.form.value)

: this.bookService.create(this.form.value);

request.subscribe(() => {

this.isModalOpen = false;

this.form.reset();

this.list.get();

});

}

delete(id: string) {

this.confirmation.warn('::AreYouSureToDelete', 'AbpAccount::AreYouSure').subscribe((status) => {

if (status === Confirmation.Status.confirm) {

this.bookService.delete(id).subscribe(() => this.list.get());

}

});

}

}

````

\* 为 `AuthorLookupDto`, `Observable` 和 `map` 添加引用.

\* 在 `selectedBook` 后添加 `authors$: Observable<AuthorLookupDto[]>;` 成员.

\* 在构造函数中添加 `this.authors$ = bookService.getAuthorLookup().pipe(map((r) => r.items));`.

\* 在 `buildForm()` 函数中添加 `authorId: [this.selectedBook.authorId || null, Validators.required]`.

打开 `/src/app/book/book.component.html`, 在图书名称表单组前加入下列表单组:

````html

<div class="form-group">

<label for="author-id">Author</label><span> \* </span>

<select class="form-control" id="author-id" formControlName="authorId">

<option [ngValue]="null">Select author</option>

<option [ngValue]="author.id" \*ngFor="let author of authors$ | async">

{%{{{ author.name }}}%}

</option>

</select>

</div>

````

这就是全部了. 运行应用程序, 尝试新建或编辑一个作者.

{{end}}

{{if UI == "Blazor" || UI == "BlazorServer"}}

### The Book List

在图书列表上显示 *\*作者姓名\** 非常容易. 打开 `Acme.BookStore.Blazor` 项目中的 the `/Pages/Books.razor` 文件, 在 `名称` (图书名) 列后加入下列 `DataGridColumn` 定义:

````xml

<DataGridColumn TItem="BookDto"

Field="@nameof(BookDto.AuthorName)"

Caption="@L["Author"]"></DataGridColumn>

````

运行应用程序, 你将在表格上看到 *\*作者\** 列:

![blazor-bookstore-book-list-with-authors](images/blazor-bookstore-book-list-with-authors.png)

### 新建图书模态窗口

在 `Books.razor` 文件的 `@code` 小节加入以下成员变量:

````csharp

IReadOnlyList<AuthorLookupDto> authorList = Array.Empty<AuthorLookupDto>();

````

覆写 `OnInitializedAsync` 方法, 加入以下代码:

````csharp

protected override async Task OnInitializedAsync()

{

await base.OnInitializedAsync();

authorList = (await AppService.GetAuthorLookupAsync()).Items;

}

````

\* 调用 `base.OnInitializedAsync()` 是有必要的, 因为 `AbpCrudPageBase` 有一些初始化代码要执行.

最终的 `@code` 代码如下:

````csharp

@code

{

//ADDED A NEW FIELD

IReadOnlyList<AuthorLookupDto> authorList = Array.Empty<AuthorLookupDto>();

public Books() // Constructor

{

CreatePolicyName = BookStorePermissions.Books.Create;

UpdatePolicyName = BookStorePermissions.Books.Edit;

DeletePolicyName = BookStorePermissions.Books.Delete;

}

//GET AUTHORS ON INITIALIZATION

protected override async Task OnInitializedAsync()

{

await base.OnInitializedAsync();

authorList = (await AppService.GetAuthorLookupAsync()).Items;

}

}

````

最后, 在 *\*新建\** 模态窗口的 `ModalBody` 加入以下 `Field` 定义, 在 `Name` field之前, 做为第一项:

````xml

<Field>

<FieldLabel>@L["Author"]</FieldLabel>

<Select TValue="Guid" @bind-SelectedValue="@NewEntity.AuthorId">

<SelectItem TValue="Guid" Value="Guid.Empty">@L["PickAnAuthor"]</SelectItem>

@foreach (var author in authorList)

{

<SelectItem TValue="Guid" Value="@author.Id">

@author.Name

</SelectItem>

}

</Select>

</Field>

````

这需要在 `en.json` 文件中加入一个新的本地化键:

````js

"PickAnAuthor": "Pick an author"

````

你可以运行应用程序, 当新建图书时, 可以看见 *\*作者选择\**:

![book-create-modal-with-author](images/book-create-modal-with-author.png)

### 编辑图书模态窗口

在 *\*编辑\** 模态窗口的 `ModalBody` 加入以下 `Field` 定义, 在 `Name` field之前, 做为第一项:

````xml

<Field>

<FieldLabel>@L["Author"]</FieldLabel>

<Select TValue="Guid" @bind-SelectedValue="@EditingEntity.AuthorId">

@foreach (var author in authorList)

{

<SelectItem TValue="Guid" Value="@author.Id">

@author.Name

</SelectItem>

}

</Select>

</Field>

````

这就是全部了. 我们重用了定义在 *\*新建\** 模态窗口中的 `authorList`.

{{end}}

## 社区文章

<https://community.abp.io/articles>

## 从ASP.NET Boilerplate迁移

# Migrating from ASP.NET Boilerplate to the ABP Framework

ABP Framework is **\*\*the successor\*\*** of the open source [ASP.NET Boilerplate](https://aspnetboilerplate.com/) framework. This guide aims to help you to **\*\*migrate your existing solutions\*\*** (you developed with the ASP.NET Boilerplate framework) to the ABP Framework.

## Introduction

**\*\*ASP.NET Boilerplate\*\*** is being **\*\*actively developed\*\*** [since 2013](https://github.com/aspnetboilerplate/aspnetboilerplate/graphs/contributors). It is loved, used and contributed by the community. It started as a side project of [a developer](http://halilibrahimkalkan.com/), but now it is officially maintained and improved by the company [Volosoft](https://volosoft.com/) in addition to the great community support.

ABP Framework has the same goal of the ASP.NET Boilerplate framework: **\*\*Don't Repeat Yourself\*\***! It provides infrastructure, tools and startup templates to make a developer's life easier while developing enterprise software solutions.

See [the introduction blog post](https://blog.abp.io/abp/Abp-vNext-Announcement) if you wonder why we needed to re-write the ASP.NET Boilerplate framework.

### Should I Migrate?

No, you don't have to!

\* ASP.NET Boilerplate is still in active development and maintenance.

\* It also works on the latest ASP.NET Core and related libraries and tools. It is up to date.

However, if you want to take the advantage of the new ABP Framework [features](https://abp.io/features) and the new architecture opportunities (like support for NoSQL databases, microservice compatibility, advanced modularity), you can use this document as a guide.

### What About the ASP.NET Zero?

[ASP.NET Zero](https://aspnetzero.com/) is a commercial product developed by the core ASP.NET Boilerplate team, on top of the ASP.NET Boilerplate framework. It provides pre-built application [features](https://aspnetzero.com/Features), code generation tooling and a nice looking modern UI. It is trusted and used by thousands of companies from all around the World.

We have created the [ABP Commercial](https://commercial.abp.io/) as an alternative to the ASP.NET Zero. ABP Commercial is more modular and upgradeable compared to the ASP.NET Zero. It currently has less features compared to ASP.NET Zero, but the gap will be closed by the time (it also has some features don't exist in the ASP.NET Zero).

We think ASP.NET Zero is still a good choice while starting a new application. It is production ready and mature solution delivered as a full source code. It is being actively developed and we are constantly adding new features.

We don't suggest to migrate your ASP.NET Zero based solution to the ABP Commercial if;

\* Your ASP.NET Zero solution is mature and it is in maintenance rather than a rapid development.

\* You don't have enough development time to perform the migration.

\* A monolithic solution fits in your business.

\* You've customized existing ASP.NET Zero features too much based on your requirements.

We also suggest you to compare the features of two products based on your needs.

If you have an ASP.NET Zero based solution and want to migrate to the ABP Commercial, this guide will also help you.

### ASP.NET MVC 5.x Projects

The ABP Framework doesn't support ASP.NET MVC 5.x, it only works with ASP.NET Core. So, if you migrate your ASP.NET MVC 5.x based projects, you will also deal with the .NET Core migration.

## The Migration Progress

We've designed the ABP Framework by **\*\*getting the best parts\*\*** of the ASP.NET Boilerplate framework, so it will be familiar to you if you've developed ASP.NET Boilerplate based applications.

In the ASP.NET Boilerplate, we have not worked much on the UI side, but used some free themes (we've used [metronic theme](https://keenthemes.com/metronic/) for ASP.NET Zero on the other side). In the ABP Framework, we worked a lot on the UI side (especially for the MVC / Razor Pages UI, because Angular already has a good modular system of its own). So, the **\*\*most challenging part\*\*** of the migration will be the **\*\*User Interface\*\*** of your solution.

ABP Framework is (and ASP.NET Boilerplate was) designed based on the [Domain Driven Design](https://docs.abp.io/en/abp/latest/Domain-Driven-Design) patterns & principles and the startup templates are layered based on the DDD layers. So, this guide respects to that layering model and explains the migration layer by layer.

## Creating the Solution

First step of the migration is to create a new solution. We suggest you to create a fresh new project using [the startup templates](https://abp.io/get-started) (see [this document](https://docs.abp.io/en/commercial/latest/getting-started) for the ABP Commercial).

After creating the project and running the application, you can copy your code from your existing solution to the new solution step by step, layer by layer.

### About Pre-Built Modules

The startup projects for the ABP Framework use the [pre-built modules](https://docs.abp.io/en/abp/latest/Modules/Index) (not all of them, but the essentials) and themes as NuGet/NPM packages. So, you don't see the source code of the modules/themes in your solution. This has an advantage that you can easily update these packages when a new version is released. However, you can not easily customize them as their source code in your hands.

We suggest to continue to use these modules as package references, in this way you can get new features easily (see [abp update command](https://docs.abp.io/en/abp/latest/CLI#update)). In this case, you have a few options to customize or extend the functionality of the used modules;

\* You can create your own entity and share the same database table with an entity in a used module. An example of this is the `AppUser` entity comes in the startup template.

\* You can [replace](https://docs.abp.io/en/abp/latest/Dependency-Injection#replace-a-service) a domain service, application service, controller, page model or other types of services with your own implementation. We suggest you to inherit from the existing implementation and override the method you need.

\* You can replace a `.cshtml` view, page, view component, partial view... with your own one using the [Virtual File System](https://docs.abp.io/en/abp/latest/Virtual-File-System).

\* You can override javascript, css, image or any other type of static files using the [Virtual File System](https://docs.abp.io/en/abp/latest/Virtual-File-System).

More extend/customization options will be developed and documented by the time. However, if you need to fully change the module implementation, it is best to add the [source code](https://github.com/abpframework/abp/tree/dev/modules) of the related module into your own solution and remove the package dependencies.

The source code of the modules and the themes are [MIT](https://opensource.org/licenses/MIT) licensed, you can fully own and customize it without any limitation (for the ABP Commercial, you can download the source code of a [module](https://commercial.abp.io/modules)/[theme](https://commercial.abp.io/themes) if you have a [license](https://commercial.abp.io/pricing) type that includes the source code).

## The Domain Layer

Most of your domain layer code will remain same, while you need to perform some minor changes in your domain objects.

### Aggregate Roots & Entities

The ABP Framework and the ASP.NET Boilerplate both have the `IEntity` and `IEntity<T>` interfaces and `Entity` and `Entity<T>` base classes to define entities but they have some differences.

If you have an entity in the ASP.NET Boilerplate application like that:

````csharp

public class Person : Entity //Default PK is int for the ASP.NET Boilerplate

{

...

}

````

Then your primary key (the `Id` property in the base class) is `int` which is the **\*\*default primary key\*\*** (PK) type for the ASP.NET Boilerplate. If you want to set another type of PK, you need to explicitly declare it:

````csharp

public class Person : Entity<Guid> //Set explicit PK in the ASP.NET Boilerplate

{

...

}

````

ABP Framework behaves differently and expects to **\*\*always explicitly set\*\*** the PK type:

````csharp

public class Person : Entity<Guid> //Set explicit PK in the ASP.NET Boilerplate

{

...

}

````

`Id` property (and the corresponding PK in the database) will be `Guid` in this case.

#### Composite Primary Keys

ABP Framework also has a non-generic `Entity` base class, but this time it has no `Id` property. Its purpose is to allow you to create entities with composite PKs. See [the documentation](https://docs.abp.io/en/abp/latest/Entities#entities-with-composite-keys) to learn more about the composite PKs.

#### Aggregate Root

It is best practice now to use the `AggregateRoot` base class instead of `Entity` for aggregate root entities. See [the documentation](https://docs.abp.io/en/abp/latest/Entities#aggregateroot-class) to learn more about the aggregate roots.

In opposite to the ASP.NET Boilerplate, the ABP Framework creates default repositories (`IRepository<T>`) **\*\*only for the aggregate roots\*\***. It doesn't create for other types derived from the `Entity`.

If you still want to create default repositories for all entity types, find the *\*YourProjectName\**EntityFrameworkCoreModule class in your solution and change `options.AddDefaultRepositories()` to `options.AddDefaultRepositories(includeAllEntities: true)` (it may be already like that for the application startup template).

#### Migrating the Existing Entities

We suggest & use the GUID as the PK type for all the ABP Framework modules. However, you can continue to use your existing PK types to migrate your database tables easier.

The challenging part will be the primary keys of the ASP.NET Boilerplate related entities, like Users, Roles, Tenants, Settings... etc. Our suggestion is to copy data from existing database to the new database tables using a tool or in a manual way (be careful about the foreign key values).

#### Documentation

See the documentation for details on the entities:

\* [ASP.NET Boilerplate - Entity documentation](https://aspnetboilerplate.com/Pages/Documents/Entities)

\* [ABP Framework - Entity documentation](https://docs.abp.io/en/abp/latest/Entities)

### Repositories

> ABP Framework creates default repositories (`IRepository<T>`) **\*\*only for the aggregate roots\*\***. It doesn't create for other types derived from the `Entity`. See the "Aggregate Root" section above for more information.

The ABP Framework and the ASP.NET Boilerplate both have the default generic repository system, but has some differences.

#### Injecting the Repositories

In the ASP.NET Boilerplate, there are two default repository interfaces you can directly inject and use:

\* `IRepository<TEntity>` (e.g. `IRepository<Person>`) is used for entities with `int` primary key (PK) which is the default PK type.

\* `IRepository<TEntity, TKey>` (e.g. `IRepository<Person, Guid>`) is used for entities with other types of PKs.

ABP Framework doesn't have a default PK type, so you need to **\*\*explicitly declare the PK type\*\*** of your entity, like `IRepository<Person, int>` or `IRepository<Person, Guid>`.

ABP Framework also has the `IRepository<TEntity>` (without PK), but it is mostly used when your entity has a composite PK (because this repository has no methods work with the `Id` property). See [the documentation](https://docs.abp.io/en/abp/latest/Entities#entities-with-composite-keys) to learn more about the **\*\*composite PKs\*\***.

#### Restricted Repositories

ABP Framework additionally provides a few repository interfaces:

\* `IBasicRepository<TEntity, TKey>` has the same methods with the `IRepository` except it doesn't have `IQueryable` support. It can be useful if you don't want to expose complex querying code to the application layer. In this case, you typically want to create custom repositories to encapsulate the querying logic. It is also useful for database providers those don't support `IQueryable`.

\* `IReadOnlyRepository<TEntity,TKey>` has the methods get data from the database, but doesn't contain any method change the database.

\* `IReadOnlyBasicRepository<TEntity, TKey>` is similar to the read only repository but also doesn't support `IQueryable`.

All the interfaces also have versions without `TKey` (like ``IReadOnlyRepository<TEntity>`) those can be used for composite PKs just like explained above.

#### GetAll() vs IQueryable

ASP.NET Boilerplate's repository has a `GetAll()` method that is used to obtain an `IQueryable` object to execute LINQ on it. An example application service calls the `GetAll()` method:

````csharp

public class PersonAppService : ApplicationService, IPersonAppService

{

private readonly IRepository<Person, Guid> \_personRepository;

public PersonAppService(IRepository<Person, Guid> personRepository)

{

\_personRepository = personRepository;

}

public async Task DoIt()

{

var people = await \_personRepository

.GetAll() //GetAll() returns IQueryable

.Where(p => p.BirthYear > 2000) //Use LINQ extension methods

.ToListAsync();

}

}

````

ABP Framework's repository have `GetQueryableAsync` instead:

````csharp

public class PersonAppService : ApplicationService, IPersonAppService

{

private readonly IRepository<Person, Guid> \_personRepository;

public PersonAppService(IRepository<Person, Guid> personRepository)

{

\_personRepository = personRepository;

}

public async Task DoIt()

{

var queryable = await \_personRepository.GetQueryableAsync();

var people = await queryable

.Where(p => p.BirthYear > 2000) //Use LINQ extension methods

.ToListAsync();

}

}

````

> Note that in order to use the async LINQ extension methods (like `ToListAsync` here), you may need to depend on the database provider (like EF Core) since these methods are defined in the database provider package, they are not standard LINQ methods. See the [repository document](Repositories.md) for alternative approaches for async query execution.

#### FirstOrDefault(predicate), Single()... Methods

ABP Framework repository has not such methods get predicate (expression) since the repository itself is `IQueryable` and all these methods are already standard LINQ extension methods those can be directly used.

However, it provides the following methods those can be used to query a single entity by its Id:

\* `FindAsync(id)` returns the entity or null if not found.

\* `GetAsync(id)` method returns the entity or throws an `EntityNotFoundException` (which causes HTTP 404 status code) if not found.

#### Sync vs Async

ABP Framework repository has no sync methods (like `Insert`). All the methods are async (like `InsertAsync`). So, if your application has sync repository method usages, convert them to async versions.

In general, ABP Framework forces you to completely use async everywhere, because mixing async & sync methods is not a recommended approach.

#### Documentation

See the documentation for details on the repositories:

\* [ASP.NET Boilerplate - Repository documentation](https://aspnetboilerplate.com/Pages/Documents/Repositories)

\* [ABP Framework - Repository documentation](https://docs.abp.io/en/abp/latest/Repositories)

### Domain Services

Your domain service logic mostly remains same on the migration. ABP Framework also defines the base `DomainService` class and the `IDomainService` interface just works like the ASP.NET Boilerplate.

## The Application Layer

Your application service logic remains similar on the migration. ABP Framework also defines the base `ApplicationService` class and the `IApplicationService` interface just works like the ASP.NET Boilerplate, but there are some differences in details.

### Declarative Authorization

ASP.NET Boilerplate has `AbpAuthorize` and `AbpMvcAuthorize` attributes for declarative authorization. Example usage:

````csharp

[AbpAuthorize("MyUserDeletionPermissionName")]

public async Task DeleteUserAsync(...)

{

...

}

````

ABP Framework doesn't has such a custom attribute. It uses the standard `Authorize` attribute in all layers.

````csharp

[Authorize("MyUserDeletionPermissionName")]

public async Task DeleteUserAsync(...)

{

...

}

````

This is possible with the better integration to the Microsoft Authorization Extensions libraries. See the Authorization section below for more information about the authorization system.

### CrudAppService and AsyncCrudAppService Classes

ASP.NET Boilerplate has `CrudAppService` (with sync service methods) and `AsyncCrudAppService` (with async service methods) classes.

ABP Framework only has the `CrudAppService` which actually has only the async methods (instead of sync methods).

ABP Framework's `CrudAppService` method signatures are slightly different than the old one. For example, old update method signature was ` Task<TEntityDto> UpdateAsync(TUpdateInput input) ` while the new one is ` Task<TGetOutputDto> UpdateAsync(TKey id, TUpdateInput input) `. The main difference is that it gets the Id of the updating entity as a separate parameter instead of including in the input DTO.

### Data Transfer Objects (DTOs)

There are similar base DTO classes (like `EntityDto`) in the ABP Framework too. So, you can find the corresponding DTO base class if you need.

#### Validation

You can continue to use the data annotation attributes to validate your DTOs just like in the ASP.NET Boilerplate.

ABP Framework doesn't include the ` ICustomValidate ` that does exists in the ASP.NET Boilerplate. Instead, you should implement the standard `IValidatableObject` interface for your custom validation logic.

## The Infrastructure Layer

### Namespaces

ASP.NET Boilerplate uses the `Abp.\*` namespaces while the ABP Framework uses the `Volo.Abp.\*` namespaces for the framework and pre-built fundamental modules.

In addition, there are also some pre-built application modules (like docs and blog modules) those are using the `Volo.\*` namespaces (like `Volo.Blogging.\*` and `Volo.Docs.\*`). We consider these modules as standalone open source products developed by Volosoft rather than add-ons or generic modules completing the ABP Framework and used in the applications. We've developed them as a module to make them re-usable as a part of a bigger solution.

### Module System

Both of the ASP.NET Boilerplate and the ABP Framework have the `AbpModule` while they are a bit different.

ASP.NET Boilerplate's `AbpModule` class has `PreInitialize`, `Initialize` and `PostInitialize` methods you can override and configure the framework and the depended modules. You can also register and resolve dependencies in these methods.

ABP Framework's `AbpModule` class has the `ConfigureServices` and `OnApplicationInitialization` methods (and their Pre and Post versions). It is similar to ASP.NET Core's Startup class. You configure other services and register dependencies in the `ConfigureServices`. However, you can now resolve dependencies in that point. You can resolve dependencies and configure the ASP.NET Core pipeline in the `OnApplicationInitialization` method while you can not register dependencies here. So, the new module classes separate dependency registration phase from dependency resolution phase since it follows the ASP.NET Core's approach.

### Dependency Injection

#### The DI Framework

ASP.NET Boilerplate is using the [Castle Windsor](http://www.castleproject.org/projects/windsor/) as the dependency injection framework. This is a fundamental dependency of the ASP.NET Boilerplate framework. We've got a lot of feedback to make the ASP.NET Boilerplate DI framework agnostic, but it was not so easy because of the design.

ABP Framework is dependency injection framework independent since it uses Microsoft's [Dependency Injection Extensions](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection) library as an abstraction. None of the ABP Framework or module packages depends on any specific library.

However, ABP Framework doesn't use the Microsoft's base DI library because it has some missing features ABP Framework needs to: Property Injection and Interception. All the startup templates and the samples are using the [Autofac](https://autofac.org/) as the DI library and it is the only [officially integrated](Autofac-Integration.md) library to the ABP Framework. We suggest you to use the Autofac with the ABP Framework if you have not a good reason. If you have a good reason, please create an [issue](https://github.com/abpframework/abp/issues/new) on GitHub to request it or just implement it and send a pull request :)

#### Registering the Dependencies

Registering the dependencies are similar and mostly handled by the framework conventionally (like repositories, application services, controllers... etc). Implement the same `ITransientDependency`, `ISingletonDependency` and `IScopedDependency` interfaces for the services not registered by conventions.

When you need to manually register dependencies, use the `context.Services` in the `ConfigureServices` method of your module. Example:

````csharp

public class BlogModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

//Register an instance as singleton

context.Services.AddSingleton<TaxCalculator>(new TaxCalculator(taxRatio: 0.18));

//Register a factory method that resolves from IServiceProvider

context.Services.AddScoped<ITaxCalculator>(

sp => sp.GetRequiredService<TaxCalculator>()

);

}

}

````

See the ABP Framework [dependency injection document](https://docs.abp.io/en/abp/latest/Dependency-Injection) for details.

### Configuration vs Options System

ASP.NET Boilerplate has its own configuration system to configure the framework and the modules. For example, you could disable the audit logging in the `Initialize` method of your [module](https://aspnetboilerplate.com/Pages/Documents/Module-System):

````csharp

public override void Initialize()

{

Configuration.Auditing.IsEnabled = false;

}

````

ABP Framework uses [the options pattern](Options.md) to configure the framework and the modules. You typically configure the options in the `ConfigureServices` method of your [module](Module-Development-Basics.md):

````csharp

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpAuditingOptions>(options =>

{

options.IsEnabled = false;

});

}

````

Instead of a central configuration object, there are separated option classes for every module and feature those are defined in the related documents.

### IAbpSession vs ICurrentUser and ICurrentTenant

ASP.NET Boilerplate's `IAbpSession` service is used to obtain the current user and tenant information, like ` UserId ` and `TenantId`.

ABP Framework doesn't have the same service. Instead, use `ICurrentUser` and `ICurrentTenant` services. These services are defined as base properties in some common classes (like `ApplicationService` and `AbpController`), so you generally don't need to manually inject them. They also have much properties compared to the `IAbpSession`.

### Authorization

ABP Framework extends the [ASP.NET Core Authorization](https://docs.microsoft.com/en-us/aspnet/core/security/authorization/introduction) by adding **\*\*permissions\*\*** as auto [policies](https://docs.microsoft.com/en-us/aspnet/core/security/authorization/policies) and allowing the authorization system to be usable in the [application services](Application-Services.md) too.

#### AbpAuthorize vs Authorize

Use the standard `[Authorize]` and `[AllowAnonymous]` attributes instead of ASP.NET Boilerplate's custom `[AbpAuthorize]` and `[AbpAllowAnonymous]` attributes.

#### IPermissionChecker vs IAuthorizationService

Use the standard `IAuthorizationService` to check permissions instead of the ASP.NET Boilerplate's `IPermissionChecker` service. While `IPermissionChecker` also exists in the ABP Framework, it is used to explicitly use the permissions. Using `IAuthorizationService` is the recommended way since it covers other type of policy checks too.

#### AuthorizationProvider vs PermissionDefinitionProvider

You inherit from the `AuthorizationProvider` in the ASP.NET Boilerplate to define your permissions. ABP Framework replaces it by the `PermissionDefinitionProvider` base class. So, define your permissions by inheriting from the `PermissionDefinitionProvider` class.

### Unit of Work

Unit of work system has been designed to work seamlessly. For most of the cases, you don't need to change anything.

`UnitOfWork` attribute of the ABP Framework doesn't have the `ScopeOption` (type of `TransactionScopeOption`) property. Instead, use `IUnitOfWorkManager.Begin()` method with `requiresNew = true` to create an independent inner transaction in a transaction scope.

#### Data Filters

ASP.NET Boilerplate implements the data filtering system as a part of the unit of work. ABP Framework has a separate `IDataFilter` service.

See the [data filtering document](Data-Filtering.md) to learn how to enable/disable a filter.

See [the UOW documentation](Unit-Of-Work.md) for more about the UOW system.

### Multi-Tenancy

#### IMustHaveTenant & IMayHaveTenant vs IMultiTenant

ASP.NET Boilerplate defines `IMustHaveTenant` and `IMayHaveTenant` interfaces to implement them for your entities. In this way, your entities are automatically filtered according to the current tenant. Because of the design, there was a problem: You had to create a "Default" tenant in the database with "1" as the Id if you want to create a non multi-tenant application (this "Default" tenant was used as the single tenant).

ABP Framework has a single interface for multi-tenant entities: `IMultiTenant` which defines a nullable `TenantId` property of type `Guid`. If your application is not multi-tenant, then your entities will have null TenantId (instead of a default one).

On the migration, you need to change the TenantId field type and replace these interfaces with the `IMultiTenant`

#### Switch Between Tenants

In some cases you might need to switch to a tenant for a code scope and work with the tenant's data in this scope.

In ASP.NET Boilerplate, it is done using the `IUnitOfWorkManager` service:

````csharp

public async Task<List<Product>> GetProducts(int tenantId)

{

using (\_unitOfWorkManager.Current.SetTenantId(tenantId))

{

return await \_productRepository.GetAllListAsync();

}

}

````

In the ABP Framework it is done with the `ICurrentTenant` service:

````csharp

public async Task<List<Product>> GetProducts(Guid tenantId)

{

using (\_currentTenant.Change(tenantId))

{

return await \_productRepository.GetListAsync();

}

}

````

Pass `null` to the `Change` method to switch to the host side.

### Caching

ASP.NET Boilerplate has its [own distributed caching abstraction](https://aspnetboilerplate.com/Pages/Documents/Caching) which has in-memory and Redis implementations. You typically inject the `ICacheManager` service and use its `GetCache(...)` method to obtain a cache, then get and set objects in the cache.

ABP Framework uses and extends ASP.NET Core's [distributed caching abstraction](Caching.md). It defines the `IDistributedCache<T>` services to inject a cache and get/set objects.

### Logging

ASP.NET Boilerplate uses Castle Windsor's [logging facility](https://github.com/castleproject/Windsor/blob/master/docs/logging-facility.md) as an abstraction and supports multiple logging providers including Log4Net (the default one comes with the startup projects) and Serilog. You typically property-inject the logger:

````csharp

using Castle.Core.Logging; //1: Import Logging namespace

public class TaskAppService : ITaskAppService

{

//2: Getting a logger using property injection

public ILogger Logger { get; set; }

public TaskAppService()

{

//3: Do not write logs if no Logger supplied.

Logger = NullLogger.Instance;

}

public void CreateTask(CreateTaskInput input)

{

//4: Write logs

Logger.Info("Creating a new task with description: " + input.Description);

//...

}

}

````

ABP Framework depends on Microsoft's [logging extensions](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging) library which is also an abstraction and there are many providers implement it. Startup templates are using the Serilog as the pre-configured logging libary while it is easy to change in your project. The usage pattern is similar:

````csharp

//1: Import the Logging namespaces

using Microsoft.Extensions.Logging;

using Microsoft.Extensions.Logging.Abstractions;

public class TaskAppService : ITaskAppService

{

//2: Getting a logger using property injection

public ILogger<TaskAppService> Logger { get; set; }

public TaskAppService()

{

//3: Do not write logs if no Logger supplied.

Logger = NullLogger<TaskAppService>.Instance;

}

public void CreateTask(CreateTaskInput input)

{

//4: Write logs

Logger.Info("Creating a new task with description: " + input.Description);

//...

}

}

````

You inject the `ILogger<T>` instead of the `ILogger`.

### Object to Object Mapping

#### IObjectMapper Service

ASP.NET Boilerplate defines an `IObjectMapper` service ([see](https://aspnetboilerplate.com/Pages/Documents/Object-To-Object-Mapping)) and has an integration to the [AutoMapper](https://automapper.org/) library.

Example usage: Create a `User` object with the given `CreateUserInput` object:

````csharp

public void CreateUser(CreateUserInput input)

{

var user = ObjectMapper.Map<User>(input);

...

}

````

Example: Update an existing `User` properties with the given `UpdateUserInput` object:

````csharp

public async Task UpdateUserAsync(Guid id, UpdateUserInput input)

{

var user = await \_userRepository.GetAsync(id);

ObjectMapper.Map(input, user);

}

````

ABP Framework has the same `IObjectMapper` service ([see](Object-To-Object-Mapping.md)) and the AutoMapper integration with a slightly different mapping methods.

Example usage: Create a `User` object with the given `CreateUserInput` object:

````csharp

public void CreateUser(CreateUserInput input)

{

var user = ObjectMapper.Map<CreateUserInput, User>(input);

}

````

This time you need to explicitly declare the source type and target type (while ASP.NET Boilerplate was requiring only the target type).

Example: Update an existing `User` properties with the given `UpdateUserInput` object:

````csharp

public async Task UpdateUserAsync(Guid id, UpdateUserInput input)

{

var user = await \_userRepository.GetAsync(id);

ObjectMapper.Map<UpdateUserInput, User>(input, user);

}

````

Again, ABP Framework expects to explicitly set the source and target types.

#### AutoMapper Integration

##### Auto Mapping Attributes

ASP.NET Boilerplate has `AutoMapTo`, `AutoMapFrom` and `AutoMap` attributes to automatically create mappings for the declared types. Example:

````csharp

[AutoMapTo(typeof(User))]

public class CreateUserInput

{

public string Name { get; set; }

public string Surname { get; set; }

...

}

````

ABP Framework has no such attributes, because AutoMapper as a [similar attribute](https://automapper.readthedocs.io/en/latest/Attribute-mapping.html) now. You need to switch to AutoMapper's attribute.

##### Mapping Definitions

ABP Framework follows AutoMapper principles closely. You can define classes derived from the `Profile` class to define your mappings.

##### Configuration Validation

Configuration validation is a best practice for the AutoMapper to maintain your mapping configuration in a safe way.

See [the documentation](Object-To-Object-Mapping.md) for more information related to the object mapping.

### Setting Management

#### Defining the Settings

In an ASP.NET Boilerplate based application, you create a class deriving from the `SettingProvider` class, implement the `GetSettingDefinitions` method and add your class to the `Configuration.Settings.Providers` list.

In the ABP Framework, you need to derive your class from the `SettingDefinitionProvider` and implement the `Define` method. You don't need to register your class since the ABP Framework automatically discovers it.

#### Getting the Setting Values

ASP.NET Boilerplate provides the `ISettingManager` to read the setting values in the server side and `abp.setting.get(...)` method in the JavaScript side.

ABP Framework has the `ISettingProvider` service to read the setting values in the server side and `abp.setting.get(...)` method in the JavaScript side.

#### Setting the Setting Values

For ASP.NET Boilerplate, you use the same `ISettingManager` service to change the setting values.

ABP Framework separates it and provides the setting management module (pre-added to the startup projects) which has the ` ISettingManager ` to change the setting values. This separation was introduced to support tiered deployment scenarios (where `ISettingProvider` can also work in the client application while `ISettingManager ` can also work in the server (API) side).

### Clock

ASP.NET Boilerplate has a static `Clock` service ([see](https://aspnetboilerplate.com/Pages/Documents/Timing)) which is used to abstract the `DateTime` kind, so you can easily switch between Local and UTC times. You don't inject it, but just use the `Clock.Now` static method to obtain the current time.

ABP Framework has the `IClock` service ([see](Timing.md)) which has a similar goal, but now you need to inject it whenever you need it.

### Event Bus

ASP.NET Boilerplate has an in-process event bus system. You typically inject the `IEventBus` (or use the static instance `EventBus.Default`) to trigger an event. It automatically triggers events for entity changes (like `EntityCreatingEventData` and `EntityUpdatedEventData`). You create a class by implementing the `IEventHandler<T>` interface.

ABP Framework separates the event bus into two services: `ILocalEventBus` and `IDistributedEventBus`.

The local event bus is similar to the event bus of the ASP.NET Boilerplate while the distributed event bus is new feature introduced in the ABP Framework.

So, to migrate your code;

\* Use the `ILocalEventBus` instead of the `IEventBus`.

\* Implement the `ILocalEventHandler` instead of the `IEventHandler`.

> Note that ABP Framework has also an `IEventBus` interface, but it does exists to be a common interface for the local and distributed event bus. It is not injected and directly used.

### Feature Management

Feature system is used in multi-tenant applications to define features of your application check if given feature is available for the current tenant.

#### Defining Features

In the ASP.NET Boilerplate ([see](https://aspnetboilerplate.com/Pages/Documents/Feature-Management)), you create a class inheriting from the `FeatureProvider`, override the `SetFeatures` method and add your class to the `Configuration.Features.Providers` list.

In the ABP Framework ([see](Features.md)), you derive your class from the `FeatureDefinitionProvider` and override the `Define` method. No need to add your class to the configuration, it is automatically discovered by the framework.

#### Checking Features

You can continue to use the `RequiresFeature` attribute and `IFeatureChecker` service to check if a feature is enabled for the current tenant.

#### Changing the Feature Values

In the ABP Framework you use the `IFeatureManager` to change a feature value for a tenant.

### Audit Logging

The ASP.NET Boilerplate ([see](https://aspnetboilerplate.com/Pages/Documents/Audit-Logging)) and the ABP Framework ([see](Audit-Logging.md)) has similar audit logging systems. ABP Framework requires to add `UseAuditing()` middleware to the ASP.NET Core pipeline, which is already added in the startup templates. So, most of the times it will be work out of the box.

### Localization

ASP.NET Boilerplate supports XML and JSON files to define the localization key-values for the UI ([see](https://aspnetboilerplate.com/Pages/Documents/Localization)). ABP Framework only supports the JSON formatter localization files ([see](Localization.md)). So, you need to convert your XML file to JSON.

The ASP.NET Boilerplate has its own the `ILocalizationManager` service to be injected and used for the localization in the server side.

The ABP Framework uses [Microsoft localization extension](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/localization) library, so it is completely integrated to ASP.NET Core. You use the `IStringLocalizer<T>` service to get a localized text. Example:

````csharp

public class MyService

{

private readonly IStringLocalizer<TestResource> \_localizer;

public MyService(IStringLocalizer<TestResource> localizer)

{

\_localizer = localizer;

}

public void Foo()

{

var str = \_localizer["HelloWorld"]; //Get a localized text

}

}

````

So, you need to replace `ILocalizationManager` usage by the `IStringLocalizer`.

It also provides API used in the client side:

````js

var testResource = abp.localization.getResource('Test');

var str = testResource('HelloWorld');

````

It was like `abp.localization.localize(...)` in the ASP.NET Boilerplate.

### Navigation vs Menu

In ASP.NET Boilerplate you create a class deriving from the `NavigationProvider` to define your menu elements. Menu items has `requiredPermissionName` attributes to restrict access to a menu element. Menu items were static and your class is executed only one time.

In the ABP Framework you need to create a class implements the `IMenuContributor` interface. Your class is executed whenever the menu needs to be rendered. So, you can conditionally add menu items.

As an example, this is the menu contributor of the tenant management module:

````csharp

public class AbpTenantManagementWebMainMenuContributor : IMenuContributor

{

public async Task ConfigureMenuAsync(MenuConfigurationContext context)

{

//Add items only to the main menu

if (context.Menu.Name != StandardMenus.Main)

{

return;

}

//Get the standard administration menu item

var administrationMenu = context.Menu.GetAdministration();

//Resolve some needed services from the DI container

var l = context.GetLocalizer<AbpTenantManagementResource>();

var tenantManagementMenuItem = new ApplicationMenuItem(

TenantManagementMenuNames.GroupName,

l["Menu:TenantManagement"],

icon: "fa fa-users");

administrationMenu.AddItem(tenantManagementMenuItem);

//Conditionally add the "Tenants" menu item based on the permission

if (await context.IsGrantedAsync(TenantManagementPermissions.Tenants.Default))

{

tenantManagementMenuItem.AddItem(

new ApplicationMenuItem(

TenantManagementMenuNames.Tenants,

l["Tenants"],

url: "/TenantManagement/Tenants"));

}

}

}

````

So, you need to check permission using the `IAuthorizationService` if you want to show a menu item only when the user has the related permission.

> Navigation/Menu system is only for ASP.NET Core MVC / Razor Pages applications. Angular applications has a different system implemented in the startup templates.

## Missing Features

The following features are not present for the ABP Framework. Here, a list of some major missing features (and the related issue for that feature waiting on the ABP Framework GitHub repository):

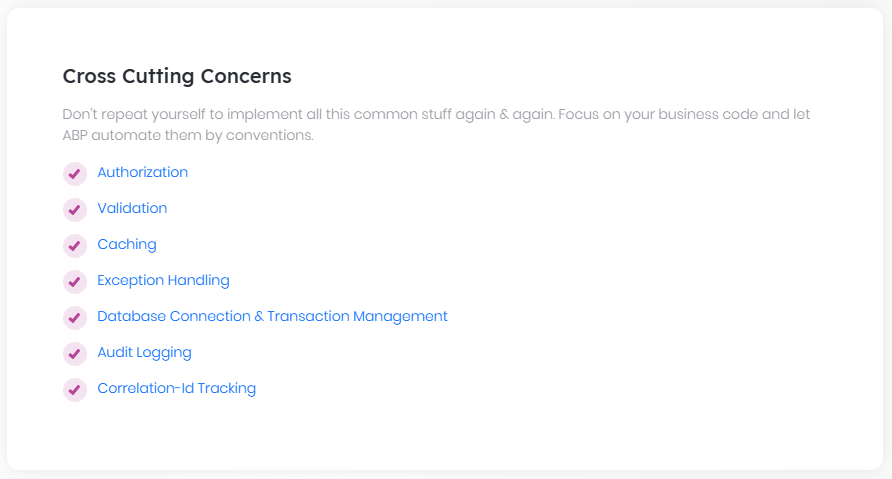
\* [Multi-Lingual Entities](https://aspnetboilerplate.com/Pages/Documents/Multi-Lingual-Entities) ([#1754](https://github.com/abpframework/abp/issues/1754))

\* [Real time notification system](https://aspnetboilerplate.com/Pages/Documents/Notification-System) ([#633](https://github.com/abpframework/abp/issues/633))

\* [NHibernate Integration](https://aspnetboilerplate.com/Pages/Documents/NHibernate-Integration) ([#339](https://github.com/abpframework/abp/issues/339)) - We don't intent to work on this, but any community contribution welcome.

Some of these features will eventually be implemented. However, you can implement them yourself if they are important for you. If you want, you can [contribute](Contribution/Index.md) to the framework, it is appreciated.

# 基础知识



## 配置

# 配置

ASP.NET Core 提供了一个灵活可扩展,基于键值的配置系统. 但是配置系统独立于ASP.NET Core是`Microsoft.Extensions` 类库的部分. 它可以用于任何类型的应用程序 . 参见 [微软文档](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/) 学习配置的更多内容. ABP 框架与配置系统百分百兼容.

## 选项

# 选项

微软引入[选项模式](https://docs.microsoft.com/zh-cn/aspnet/core/fundamentals/configuration/options),它是用于配置框架服务使用的设置. 选项模式由[Microsoft.Extensions.Options](https://www.nuget.org/packages/Microsoft.Extensions.Options)NuGet包实现,除了ASP.NET Core应用,它还适用于任何类型的应用程序.

ABP框架遵循选项模式,并定义了用于配置框架和模块的选项类(在相关功能文档中有详细的说明).

由于[微软的文档](https://docs.microsoft.com/zh-cn/aspnet/core/fundamentals/configuration/options)详细解释了选项模式,本文中只会介绍ABP增加的一些功能.

## 配置选项

通常配置选项在 `Startup` 类的 `ConfigureServices` 方法中. 但由于ABP框架提供了模块化基础设施,因此你可以在[模块](Module-Development-Basics.md)的`ConfigureServices` 方法配置选项.

例:

````csharp

public override void ConfigureServices(ServiceConfigurationContext context)

{

context.Services.Configure<AbpAuditingOptions>(options =>

{

options.IsEnabled = false;

});

}

````

\* `AbpAuditingOptions` 是一个简单的类,定义了一些属性,例如这里使用的 `IsEnabled`.

\* `AbpModule` 基类定义 `Configure` 方法简化代码. 你可以直接使用 `Configure<...>`,而不是`context.Services.Configure <...>`.

如果你正在开发一个可重用的模块,你可能需要定义一个允许开发人员配置模块的选项类. 这时定义一个如下所示的普通类:

````csharp

public class MyOptions

{

public int Value1 { get; set; }

public bool Value2 { get; set; }

}

````

然后开发人员可以像上面 `AbpAuditingOptions` 示例一样配置你的选项:

````csharp

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<MyOptions>(options =>

{

options.Value1 = 42;

options.Value2 = true;

});

}

````

\* 在本示例中,使用了简化的 `Configure<...>`方法.

### 获取选项值

在你需要获得一个选项值时,将 `IOptions<TOption>` 服务[注入](Dependency-Injection.md)到你的类中,使用它的 `.Value` 属性得到值.

例:

````csharp

public class MyService : ITransientDependency

{

private readonly MyOptions \_options;

public MyService(IOptions<MyOptions> options)

{

\_options = options.Value; //Notice the options.Value usage!

}

public void DoIt()

{

var v1 = \_options.Value1;

var v2 = \_options.Value2;

}

}

````

阅读[微软文档](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/options)了解选择模式的所有细节.

## 预配置

选项模式的限制之一是你只能解析(注入) `IOptions <MyOptions>` 并在依赖注入配置完成(即所有模块的`ConfigureServices`方法完成)后获取选项值.

如果你正在开发一个模块,可能需要让开发者能够设置一些选项,并在依赖注入注册阶段使用这些选项. 你可能需要根据选项值配置其他服务或更改依赖注入的注册代码.

对于此类情况,ABP为 `IServiceCollection` 引入了 `PreConfigure<TOptions>` 和 `ExecutePreConfiguredActions<TOptions>` 扩展方法. 该模式的工作原理如下所述.

在你的模块中定义预先选项类. 例:

````csharp

public class MyPreOptions

{

public bool MyValue { get; set; }

}

````

然后任何依赖于模块的模块类都可以在其 `PreConfigureServices` 方法中使用 `PreConfigure<TOptions>` 方法.

例:

````csharp

public override void PreConfigureServices(ServiceConfigurationContext context)

{

PreConfigure<MyPreOptions>(options =>

{

options.MyValue = true;

});

}

````

> 多个模块可以预配置选项,并根据它们的依赖顺序覆盖选项值.

最后在你的模块 `ConfigureServices` 方法中执行 `ExecutePreConfiguredActions` 方法来获得配置的选项值.

例:

````csharp

public override void ConfigureServices(ServiceConfigurationContext context)

{

var options = context.Services.ExecutePreConfiguredActions<MyPreOptions>();

if (options.MyValue)

{

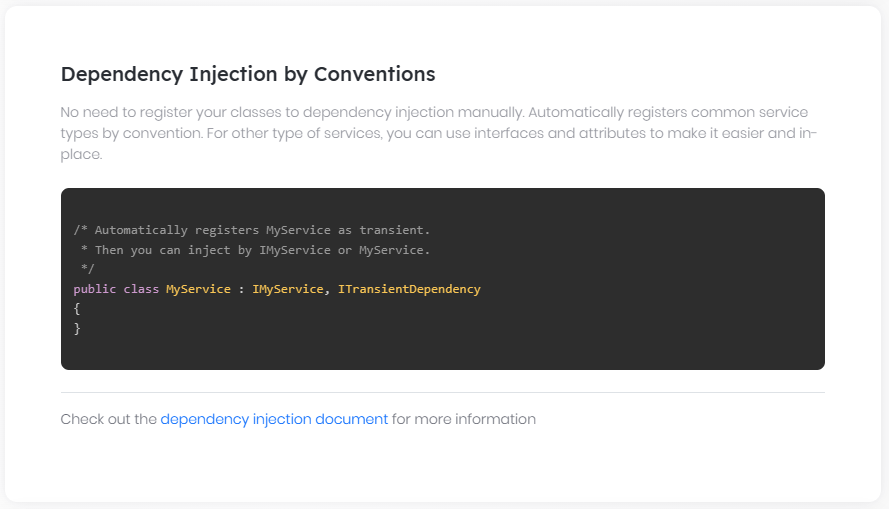
//...

}

}

````

## 依赖注入



## 依赖注入

ABP的依赖注入系统是基于Microsoft的[依赖注入扩展](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection)库（Microsoft.Extensions.DependencyInjection nuget包）开发的.因此,它的文档在ABP中也是有效的.

> 虽然ABP框架没有对任何第三方DI提供程序的核心依赖, 但它必须使用一个提供程序来支持动态代理(dynamic proxying)和一些高级特性以便ABP特性能正常工作.启动模板中已安装了Autofac. 更多信息请参阅 [Autofac 集成](Autofac-Integration.md) 文档.

### 模块化

由于ABP是一个模块化框架,因此每个模块都定义它自己的服务并在它自己的单独[模块类](Module-Development-Basics.md)中通过依赖注入进行注册.例:

````C#

public class BlogModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

//在此处注入依赖项

}

}

````

### 依照约定的注册

ABP引入了依照约定的服务注册.依照约定你无需做任何事,它会自动完成.如果要禁用它,你可以通过重写`PreConfigureServices`方法,设置`SkipAutoServiceRegistration`为`true`.

````C#

public class BlogModule : AbpModule

{

public override void PreConfigureServices(ServiceConfigurationContext context)

{

SkipAutoServiceRegistration = true;

}

}

````

一旦跳过自动注册,你应该手动注册你的服务.在这种情况下,`AddAssemblyOf`扩展方法可以帮助你依照约定注册所有服务.例:

````c#

public class BlogModule : AbpModule

{

public override void PreConfigureServices(ServiceConfigurationContext context)

{

SkipAutoServiceRegistration = true;

}

public override void ConfigureServices(ServiceConfigurationContext context)

{

context.Services.AddAssemblyOf<BlogModule>();

}

}

````

以下部分解释了约定和配置.

#### 固有的注册类型

一些特定类型会默认注册到依赖注入.例子:

\* 模块类注册为singleton.

\* MVC控制器（继承``Controller``或``AbpController``）被注册为transient.

\* MVC页面模型（继承``PageModel``或``AbpPageModel``）被注册为transient.

\* MVC视图组件（继承``ViewComponent``或``AbpViewComponent``）被注册为transient.

\* 应用程序服务（实现``IApplicationService``接口或继承``ApplicationService``类）注册为transient.

\* 存储库（实现``IRepository``接口）注册为transient.

\* 域服务（实现``IDomainService``接口）注册为transient.

示例:

````C#

public class BlogPostAppService : ApplicationService

{

}

````

``BlogPostAppService`` 由于它是从已知的基类派生的,因此会自动注册为transient生命周期.

#### 依赖接口

如果实现这些接口,则会自动将类注册到依赖注入:

\* ``ITransientDependency`` 注册为transient生命周期.

\* ``ISingletonDependency`` 注册为singleton生命周期.

\* ``IScopedDependency`` 注册为scoped生命周期.

示例:

````C#

public class TaxCalculator : ITransientDependency

{

}

````

``TaxCalculator``因为实现了``ITransientDependency``,所以它会自动注册为transient生命周期.

#### Dependency 特性

配置依赖注入服务的另一种方法是使用``DependencyAttribute``.它具有以下属性:

\* ``Lifetime``: 注册的生命周期:Singleton,Transient或Scoped.

\* ``TryRegister``: 设置``true``则只注册以前未注册的服务.使用IServiceCollection的TryAdd ... 扩展方法.

\* ``ReplaceServices``: 设置``true``则替换之前已经注册过的服务.使用IServiceCollection的Replace扩展方法.

示例:

````C#

[Dependency(ServiceLifetime.Transient, ReplaceServices = true)]

public class TaxCalculator

{

}

````

如果定义了``Lifetime``属性,则``Dependency``特性具有比其他依赖接口更高的优先级.

#### ExposeServices 特性

``ExposeServicesAttribute``用于控制相关类提供了什么服务.例:

````C#

[ExposeServices(typeof(ITaxCalculator))]

public class TaxCalculator: ICalculator, ITaxCalculator, ICanCalculate, ITransientDependency

{

}

````

``TaxCalculator``类只公开``ITaxCalculator``接口.这意味着你只能注入``ITaxCalculator``,但不能注入``TaxCalculator``或``ICalculator``到你的应用程序中.

#### 依照约定公开的服务

如果你未指定要公开的服务,则ABP依照约定公开服务.以上面定义的``TaxCalculator``为例:

\* 默认情况下,类本身是公开的.这意味着你可以按``TaxCalculator``类注入它.

\* 默认情况下,默认接口是公开的.默认接口是由命名约定确定.在这个例子中,``ICalculator``和``ITaxCalculator``是``TaxCalculator``的默认接口,但``ICanCalculate``不是.

#### 组合到一起

只要有意义,特性和接口是可以组合在一起使用的.

````C#

[Dependency(ReplaceServices = true)]

[ExposeServices(typeof(ITaxCalculator))]

public class TaxCalculator : ITaxCalculator, ITransientDependency

{

}

````

#### 手动注册

在某些情况下,你可能需要向``IServiceCollection``手动注册服务,尤其是在需要使用自定义工厂方法或singleton实例时.在这种情况下,你可以像[Microsoft文档](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection)描述的那样直接添加服务.例:

````C#

public class BlogModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

//注册一个singleton实例

context.Services.AddSingleton<TaxCalculator>(new TaxCalculator(taxRatio: 0.18));

//注册一个从IServiceProvider解析得来的工厂方法

context.Services.AddScoped<ITaxCalculator>(sp => sp.GetRequiredService<TaxCalculator>());

}

}

````

### 注入依赖关系

使用已注册的服务有三种常用方法.

#### 构造方法注入

这是将服务注入类的最常用方法.例如:

````C#

public class TaxAppService : ApplicationService

{

private readonly ITaxCalculator \_taxCalculator;

public TaxAppService(ITaxCalculator taxCalculator)

{

\_taxCalculator = taxCalculator;

}

public void DoSomething()

{

//...使用 \_taxCalculator...

}

}

````

``TaxAppService``在构造方法中得到``ITaxCalculator``.依赖注入系统在运行时自动提供所请求的服务.

构造方法注入是将依赖项注入类的首选方式.这样,除非提供了所有构造方法注入的依赖项,否则无法构造类.因此,该类明确的声明了它必需的服务.

#### 属性注入

Microsoft依赖注入库不支持属性注入.但是,ABP可以与第三方DI提供商（例如[Autofac](https://autofac.org/)）集成,以实现属性注入.例:

````C#

public class MyService : ITransientDependency

{

public ILogger<MyService> Logger { get; set; }

public MyService()

{

Logger = NullLogger<MyService>.Instance;

}

public void DoSomething()

{

//...使用 Logger 写日志...

}

}

````

对于属性注入依赖项,使用公开的setter声明公共属性.这允许DI框架在创建类之后设置它.

属性注入依赖项通常被视为可选依赖项.这意味着没有它们,服务也可以正常工作.``Logger``就是这样的依赖项,``MyService``可以继续工作而无需日志记录.

为了使依赖项成为可选的,我们通常会为依赖项设置默认/后备(fallback)值.在此示例中,NullLogger用作后备.因此,如果DI框架或你在创建``MyService``后未设置Logger属性,则``MyService``依然可以工作但不写日志.

属性注入的一个限制是你不能在构造函数中使用依赖项,因为它是在对象构造之后设置的.

当你想要设计一个默认注入了一些公共服务的基类时,属性注入也很有用.如果你打算使用构造方法注入,那么所有派生类也应该将依赖的服务注入到它们自己的构造方法中,这使得开发更加困难.但是,对于非可选服务使用属性注入要非常小心,因为它使得类的要求难以清楚地看到.

#### 从IServiceProvider解析服务

你可能希望直接从``IServiceProvider``解析服务.在这种情况下,你可以将``IServiceProvider``注入到你的类并使用``GetService``方法,如下所示:

````C#

public class MyService : ITransientDependency

{

private readonly IServiceProvider \_serviceProvider;

public MyService(IServiceProvider serviceProvider)

{

\_serviceProvider = serviceProvider;

}

public void DoSomething()

{

var taxCalculator = \_serviceProvider.GetService<ITaxCalculator>();

//...

}

}

````

#### 释放/处理（Releasing/Disposing）服务

如果你使用了构造函数或属性注入,则无需担心释放服务的资源.但是,如果你从``IServiceProvider``解析了服务,在某些情况下,你可能需要注意释放服务.

ASP.NET Core会在当前HTTP请求结束时释放所有服务,即使你直接从``IServiceProvider``解析了服务（假设你注入了IServiceProvider）.但是,在某些情况下,你可能希望释放/处理手动解析的服务:

\* 你的代码在AspNet Core请求之外执行,执行者没有处理服务范围.

\* 你只有对根服务提供者的引用.

\* 你可能希望立即释放和处理服务（例如,你可能会创建太多具有大量内存占用且不想过度使用内存的服务）.

在任何情况下,你都可以使用这样的`using`代码块来安全地立即释放服务:

````C#

using (var scope = \_serviceProvider.CreateScope())

{

var service1 = scope.ServiceProvider.GetService<IMyService1>();

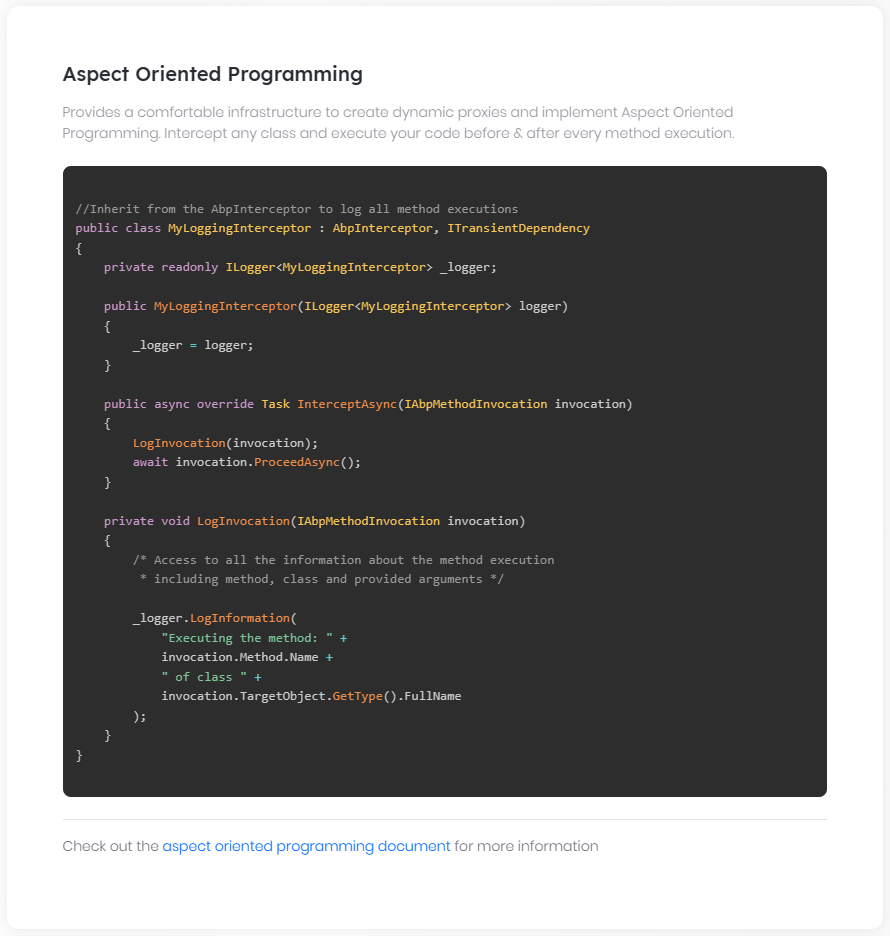
var service2 = scope.ServiceProvider.GetService<IMyService2>();

}

````

两个服务在创建的scope被处理时（在using块的末尾）释放.

## 高级特性



### IServiceCollection.OnRegistered 事件

你可能想在注册到依赖注入的每个服务上执行一个操作, 在你的模块的 `PreConfigureServices` 方法中, 使用 `OnRegistered` 方法注册一个回调(callback) , 如下所示:

````csharp

public class AppModule : AbpModule

{

public override void PreConfigureServices(ServiceConfigurationContext context)

{

context.Services.OnRegistered(ctx =>

{

var type = ctx.ImplementationType;

//...

});

}

}

````

`ImplementationType` 提供了服务类型. 该回调(callback)通常用于向服务添加拦截器. 例如:

````csharp

public class AppModule : AbpModule

{

public override void PreConfigureServices(ServiceConfigurationContext context)

{

context.Services.OnRegistered(ctx =>

{

if (ctx.ImplementationType.IsDefined(typeof(MyLogAttribute), true))

{

ctx.Interceptors.TryAdd<MyLogInterceptor>();

}

});

}

}

````

这个示例判断一个服务类是否具有 `MyLogAttribute` 特性, 如果有的话就添加一个 `MyLogInterceptor` 到拦截器集合中.

> 注意, 如果服务类公开了多于一个服务或接口, `OnRegistered` 回调(callback)可能被同一服务类多次调用. 因此, 较安全的方法是使用 `Interceptors.TryAdd` 方法而不是 `Interceptors.Add` 方法. 请参阅动态代理(dynamic proxying)/拦截器 [文档](Dynamic-Proxying-Interceptors.md).

## 第三方提供程序

虽然ABP框架没有对任何第三方DI提供程序的核心依赖, 但它必须使用一个提供程序来支持动态代理(dynamic proxying)和一些高级特性以便ABP特性能正常工作.

启动模板中已安装了Autofac. 更多信息请参阅 [Autofac 集成](Autofac-Integration.md) 文档.

### 请参阅

\* [ASP.NET Core依赖注入最佳实践,提示和技巧](https://blog.abp.io/asp-net-core-dependency-injection-best-practices-tips-tricks)

Autofac 集成

# 集成 Autofac

Autofac 是.Net世界中最常用的依赖注入框架之一. 相比.Net Core标准的依赖注入库, 它提供了更多高级特性, 比如动态代理和属性注入.

## 安装 Autofac

> 所有的启动模板和示例都已经集成了 Autofac. 所以, 多数时候你无需手动安装这个包.

安装 [Volo.Abp.Autofac](https://www.nuget.org/packages/Volo.Abp.Autofac) nuget 包到你的项目 (对于一个多项目应用程序, 建议安装到可执行项目或者Web项目中.)

````

Install-Package Volo.Abp.Autofac

````

然后为你的模块添加 `AbpAutofacModule` 依赖:

```csharp

using Volo.Abp.Modularity;

using Volo.Abp.Autofac;

namespace MyCompany.MyProject

{

[DependsOn(typeof(AbpAutofacModule))]

public class MyModule : AbpModule

{

//...

}

}

```

最后, 配置 `AbpApplicationCreationOptions` 用 Autofac 替换默认的依赖注入服务. 根据应用程序类型, 情况有所不同.

### ASP.NET Core 应用程序

如下所示, 在 **\*\*Program.cs\*\*** 文件中调用 `UseAutofac()`:

````csharp

public class Program

{

public static int Main(string[] args)

{

CreateHostBuilder(args).Build().Run();

}

internal static IHostBuilder CreateHostBuilder(string[] args) =>

Host.CreateDefaultBuilder(args)

.ConfigureWebHostDefaults(webBuilder =>

{

webBuilder.UseStartup<Startup>();

})

.UseAutofac(); //Integrate Autofac!

}

````

### 控制台应用程序

如下所示, 在 `AbpApplicationFactory.Create` 中用options调用 `UseAutofac()` 方法:

````csharp

using System;

using Microsoft.Extensions.DependencyInjection;

using Volo.Abp;

namespace AbpConsoleDemo

{

class Program

{

static void Main(string[] args)

{

using (var application = AbpApplicationFactory.Create<AppModule>(options =>

{

options.UseAutofac(); //Autofac integration

}))

{

//...

}

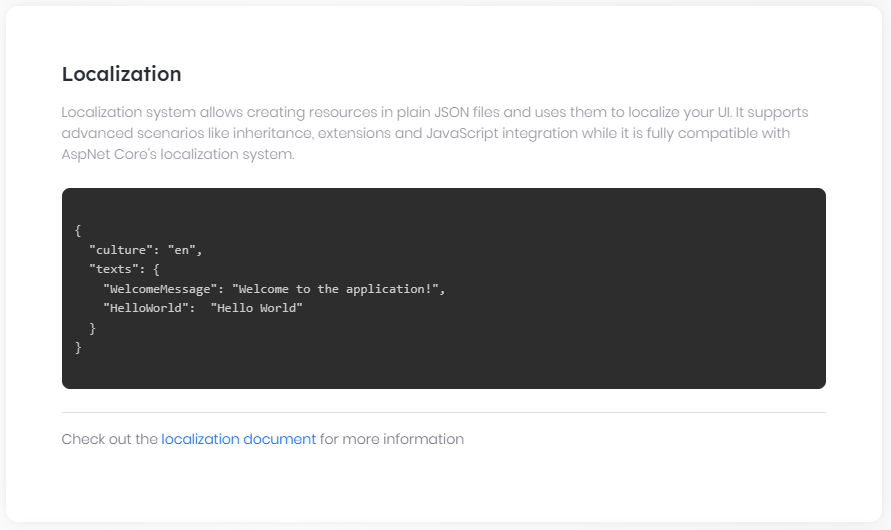
}

}

}

````

## 本地化



## 本地化

ABP的本地化系统与`Microsoft.Extensions.Localization`无缝集成,并与[AspnetCore的本地化文档](https://docs.microsoft.com/zh-cn/aspnet/core/fundamentals/localization?view=aspnetcore-2.1)兼容. 它添加了一些实用功能和增强功能, 使其更易于在实际开发中应用.

### Volo.Abp.Localization Package

> 启动模板默认已经安装了此nuget包, 所以在大多数情况下, 你不需要手动安装它.

Volo.Abp.Localization是本地化系统的核心包. 使用程序包管理器控制台 (PMC) 将其安装到项目中:

```

Install-Package Volo.Abp.Localization

```

然后, 你可以将 **\*\*AbpLocalizationModule\*\*** 依赖项添加到模块:

```c#

using Volo.Abp.Modularity;

using Volo.Abp.Localization;

namespace MyCompany.MyProject

{

[DependsOn(typeof(AbpLocalizationModule))]

public class MyModule : AbpModule

{

//...

}

}

```

#### 创建本地化资源

本地化资源用于将相关的本地化字符串组合在一起,并将它们与应用程序的其他本地化字符串分开.通常一个模块会定义自己的本地化资源. 本地化资源就是一个普通的类. 例如:

````C#

public class TestResource

{

}

````

然后应该使用 `AbpLocalizationOptions` 添加如下所示:

````C#

[DependsOn(typeof(AbpLocalizationModule))]

public class MyModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpVirtualFileSystemOptions>(options =>

{

// "YourRootNameSpace" 是项目的根命名空间名字. 如果你的项目的根命名空间名字为空,则无需传递此参数.

options.FileSets.AddEmbedded<MyModule>("YourRootNameSpace");

});

Configure<AbpLocalizationOptions>(options =>

{

options.Resources

.Add<TestResource>("en")

.AddVirtualJson("/Localization/Resources/Test");

});

}

}

````

在这个例子中;

\* 添加了一个新的本地化资源, 使用"en"（英语）作为默认的本地化.

\* 用JSON文件存储本地化字符串.

\* 使用[虚拟文件系统](Virtual-File-System.md) 将JSON文件嵌入到程序集中.

JSON文件位于 "/Localization/Resources/Test" 项目文件夹下, 如下图所示:

![localization-resource-json-files](images/localization-resource-json-files.png)

本地化文件内容如下所示:

````json

{

"culture": "en",

"texts": {

"HelloWorld": "Hello World!"

}

}

````

\* 每个本地化文件都需要定义 `culture` (文化) 代码 (例如 "en" 或 "en-US").

\* `texts` 部分只包含本地化字符串的键值集合 (键也可能有空格).

### 默认资源

可以将 `AbpLocalizationOptions.DefaultResourceType` 设置为资源类型,在未指定本地化资源时使用:

````csharp

Configure<AbpLocalizationOptions>(options =>

{

options.DefaultResourceType = typeof(TestResource);

});

````

> [启动模板]](Startup-Templates/Application.md) 设置 `DefaultResourceType` 为应用程序的本地化资源.

请参阅下面的*\*客户端\**部分获取用例

##### 简短的本地化资源名称

本地化资源也可以在客户端(JavaScript)使用. 因此, 为本地化资源设置一个简短的名称可以更方便的本地化文本. 例如:

````C#

[LocalizationResourceName("Test")]

public class TestResource

{

}

````

请参阅下面的获取本地化资源Test中客户端部分.

##### 继承其他资源

资源可以从其他资源继承,这使得可以在不引用现有资源的情况下重用现有的本地化字符串. 例如:

````C#

[InheritResource(typeof(AbpValidationResource))]

public class TestResource

{

}

````

也可以通过 `AbpLocalizationOptions` 配置:

````C#

services.Configure<AbpLocalizationOptions>(options =>

{

options.Resources

.Add<TestResource>("en") //Define the resource by "en" default culture

.AddVirtualJson("/Localization/Resources/Test") //Add strings from virtual json files

.AddBaseTypes(typeof(AbpValidationResource)); //Inherit from an existing resource

});

````

\* 资源可以从多个资源继承.

\* 如果新的本地化资源定义了相同的本地化字符串, 那么它会覆盖该字符串

##### 扩展现有资源

继承资源可以创建新的资源, 无需修改现有的资源. 但是在某些情况下, 你可能不想创建新资源,而是直接扩展现有资源. 例如:

````C#

services.Configure<AbpLocalizationOptions>(options =>

{

options.Resources

.Get<TestResource>()

.AddVirtualJson("/Localization/Resources/Test/Extensions");

});

````

\* 如果扩展文件定义了相同的本地化字符串, 那么它会覆盖该字符串.

#### 获取本地化文本

##### 服务器端

在服务端获取本地化文本的用法是非常标准的(它与AspNetCore提供的获取本地化资源方式无缝集成).

###### 在类中简单的用法

````C#

public class MyService

{

private readonly IStringLocalizer<TestResource> \_localizer;

public MyService(IStringLocalizer<TestResource> localizer)

{

\_localizer = localizer;

}

public void Foo()

{

var str = \_localizer["HelloWorld"];

}

}

````

##### 格式参数

格式参数可以在本地化Key参数后传递,如果你的消息是 `Hello {0}, welcome!`,可以将 `{0}` 传递给localizer,例如: `\_localizer["HelloMessage", "John"]`.

###### 在Razor视图/Page中简单的用法

````c#

@inject IHtmlLocalizer<TestResource> Localizer

<h1>@Localizer["HelloWorld"]</h1>

````

有关在服务器端使用本地化的详细使用方法, 请参阅[AspNetCore的本地化文档](https://docs.microsoft.com/zh-cn/aspnet/core/fundamentals/localization)

##### 客户端

ABP提供了JavaScript服务, 可以在客户端使用相同的本地化文本.

#### getResource

`abp.localization.getResource` 函数用于获取本地化资源:

````js

var testResource = abp.localization.getResource('Test');

````

本地化字符串:

````js

var str = testResource('HelloWorld');

````

#### 本地化

`abp.localization.localize` 函数用于获取本地化文本,你可以传递本地化Key和资源名称:

````js

var str = abp.localization.localize('HelloWorld', 'Test');

````

`HelloWorld` 是本地化文本的Key, `Test` 是本地化资源的名称.

如果未指定本地化资源名称,它使用 `AbpLocalizationOptions` 中定义的默认本地化资源(参见上面的*\*默认资源\**部分). 例:

````js

var str = abp.localization.localize('HelloWorld'); //uses the default resource

````

##### 格式参数

如果本地化字符串包含参数, 例如 `Hello {0}, welcome!`. 你可以将参数传递给本地化方法. 例:

````js

var str1 = abp.localization.getResource('Test')('HelloWelcomeMessage', 'John');

var str2 = abp.localization.localize('HelloWorld', 'Test', 'John');

````

上面的两个示例都会输出 `Hello John, welcome!`.

## 另请参阅

\* [Angular UI中的本地化](UI/Angular/Localization.md)

## 异常处理

## 异常处理

ABP提供了用于处理Web应用程序异常的标准模型.

\* 自动 **\*\*处理所有异常\*\*** .如果是API/AJAX请求,会向客户端返回一个**\*\*标准格式化后的错误消息\*\*** .

\* 自动隐藏 **\*\*内部详细错误\*\*** 并返回标准错误消息.

\* 为异常消息的 **\*\*本地化\*\*** 提供一种可配置的方式.

\* 自动为标准异常设置 **\*\*HTTP状态代码\*\*** ,并提供可配置选项,以映射自定义异常.

### 自动处理异常

当满足下面**\*\*任意一个条件\*\***时,`AbpExceptionFilter` 会处理此异常:

\* 当**\*\*controller action\*\***方法返回类型是**\*\*object result\*\***(而不是view result)并有异常抛出时.

\* 当一个请求为AJAX(Http请求头中`X-Requested-With`为`XMLHttpRequest`)时.

\* 当客户端接受的返回类型为`application/json`(Http请求头中`accept` 为`application/json`)时.

如果异常被处理过,则会自动**\*\*记录日志\*\***并将格式化的**\*\*JSON消息\*\***返回给客户端.

#### 错误消息格式

每个错误消息都是`RemoteServiceErrorResponse` 类的实例.最简单的错误JSON只有一个 **\*\*Message\*\*** 属性,如下所示:

````json

{

"error": {

"message": "This topic is locked and can not add a new message"

}

}

````

其它**\*\*可选字段\*\***可以根据已发生的异常来填充.

##### 错误代码

错误 **\*\*代码(code)\*\*** 是异常信息中一个有唯一值并可选的字符串值.抛出的异常应实现`IHasErrorCode` 接口来填充该字段.示例JSON如下:

````json

{

"error": {

"code": "App:010042",

"message": "This topic is locked and can not add a new message"

}

}

````

错误代码同样可用于异常信息的本地化及自定义HTTP状态代码(请参阅下面的相关部分).

##### 错误详细信息

错误的 **\*\*详细信息(Details)\*\*** 是可选属性.抛出的异常应实现`IHasErrorDetails` 接口来填充该字段.示例JSON如下:

```json

{

"error": {

"code": "App:010042",

"message": "This topic is locked and can not add a new message",

"details": "A more detailed info about the error..."

}

}

```

##### 验证错误

当抛出的异常实现`IHasValidationErrors` 接口时,**\*\*validationErrors\*\***是一个可被填充的标准字段.示例JSON如下:

````json

{

"error": {

"code": "App:010046",

"message": "Your request is not valid, please correct and try again!",

"validationErrors": [{

"message": "Username should be minimum length of 3.",

"members": ["userName"]

},

{

"message": "Password is required",

"members": ["password"]

}]

}

}

````

`AbpValidationException`已经实现了`IHasValidationErrors`接口,当请求输入无效时,框架会自动抛出此错误. 因此,除非你有自定义的验证逻辑,否则不需要处理验证错误.

#### 日志

被捕获的异常会被自动记录到日志中.

##### 日志级别

默认情况下,记录异常级别为`Error` .可以通过实现`IHasLogLevel` 接口来指定日志的级别,例如:

````C#

public class MyException : Exception, IHasLogLevel

{

public LogLevel LogLevel { get; set; } = LogLevel.Warning;

//...

}

````

##### 异常自定义日志

某些异常类型可能需要记录额外日志信息.可以通过实现`IExceptionWithSelfLogging` 接口来记录指定日志,例如:

````C#

public class MyException : Exception, IExceptionWithSelfLogging

{

public void Log(ILogger logger)

{

//...log additional info

}

}

````

> 扩展方法`ILogger.LogException` 用来记录异常日志. 在需要时可以使用相同的扩展方法.

### 业务异常

大多数异常都是业务异常.可以通过使用`IBusinessException` 接口来标记异常为业务异常.

`BusinessException` 除了实现`IHasErrorCode`,`IHasErrorDetails` ,`IHasLogLevel` 接口外,还实现了`IBusinessException` 接口.其默认日志级别为`Warning`.

通常你会将一个错误代码关联至特定的业务异常.例如:

````C#

throw new BusinessException(QaErrorCodes.CanNotVoteYourOwnAnswer);

````

`QaErrorCodes.CanNotVoteYourOwnAnswer` 是一个字符串常量. 建议使用下面的错误代码格式:

````

<code-namespace>:<error-code>

````

**\*\*code-namespace\*\***,应在指定的模块/应用层中保证其唯一.例如:

````

Volo.Qa:010002

````

`Volo.Qa`在这是作为`code-namespace`. `code-namespace` 同样可以在 **\*\*本地化\*\*** 异常信息时使用.

\* 你可以直接抛出一个 `BusinessException` 异常,或者需要时可以从该类派生你自己的Exception类型.

\* 对于`BusinessException` 类型,其所有属性都是可选的.但是通常会设置`ErrorCode`或`Message`属性.

### 异常本地化

这里有个问题,就是如何在发送错误消息到客户端时,对错误消息进行本地化.ABP提供了2个模型.

#### 用户友好异常

如果异常实现了 `IUserFriendlyException` 接口,那么ABP不会修改 `Message`和`Details`属性,而直接将它发送给客户端.

`UserFriendlyException` 类是内建的 `IUserFriendlyException` 接口的实现,示例如下:

````C#

throw new UserFriendlyException(

"Username should be unique!"

);

````

采用这种方式是不需要本地化的.如果需要本地化消息,则可以注入**\*\*string localizer\*\***( 请参阅[本地化文档](Localization.md) )来实现. 例:

````C#

throw new UserFriendlyException(\_stringLocalizer["UserNameShouldBeUniqueMessage"]);

````

再在本地化资源中为每种语言添加对应的定义.例如:

````json

{

"culture": "en",

"texts": {

"UserNameShouldBeUniqueMessage": "Username should be unique!"

}

}

````

**\*\*string localizer\*\*** 支持参数化信息.例如

````C#

throw new UserFriendlyException(\_stringLocalizer["UserNameShouldBeUniqueMessage", "john"]);

````

其本地化文本如下:

````json

"UserNameShouldBeUniqueMessage": "Username should be unique! '{0}' is already taken!"

````

\* `IUserFriendlyException`接口派生自`IBusinessException`,而 `UserFriendlyException`类派生自`BusinessException`类.

#### 使用错误代码

`UserFriendlyException`很好用,但是在一些高级用法里面,它存在以下问题:

\* 在抛出异常的地方必须注入**\*\*string localizer\*\*** 来实现本地化 .

\* 但是,在某些情况下,**\*\*可能注入不了string localizer\*\***(比如,在静态上下文或实体方法中)

那么这时就可以通过使用 **\*\*错误代码\*\*** 的方式来处理本地化,而不是在抛出异常的时候.

首先,在模块配置代码中将 **\*\*code-namespace\*\*** 映射至 **\*\*本地化资源\*\***:

````C#

services.Configure<AbpExceptionLocalizationOptions>(options =>

{

options.MapCodeNamespace("Volo.Qa", typeof(QaResource));

});

````

然后`Volo.Qa`命名空间下的所有异常都将被对应的本地化资源进行本地化处理. 本地化资源中应包含对应错误代码的文本. 例如:

````json

{

"culture": "en",

"texts": {

"Volo.Qa:010002": "You can not vote your own answer!"

}

}

````

最后就可以抛出一个包含错误代码的业务异常了:

````C#

throw new BusinessException(QaDomainErrorCodes.CanNotVoteYourOwnAnswer);

````

\* 抛出所有实现`IHasErrorCode` 接口的异常都具有相同的行为.因此,对错误代码的本地化,并不是`BusinessException`类所特有的.

\* 为错误消息定义本地化文本并不是必须的. 如果未定义,ABP会将默认的错误消息发送给客户端. 而不使用异常的`Message`属性. 如果你想要发送异常的`Message`,使用`UserFriendlyException`(或使用实现`IUserFriendlyException`接口的异常类型)

##### 使用消息的格式化参数

如果有参数化的错误消息,则可以使用异常的`Data`属性进行设置.例如:

````C#

throw new BusinessException("App:010046")

{

Data =

{

{"UserName", "john"}

}

};

````

另外有一种更为快捷的方式:

````C#

throw new BusinessException("App:010046")

.WithData("UserName", "john");

````

下面就是一个包含`UserName` 参数的错误消息:

````json

{

"culture": "en",

"texts": {

"App:010046": "Username should be unique. '{UserName}' is already taken!"

}

}

````

\* `WithData` 支持有多个参数的链式调用 (如`.WithData(...).WithData(...)`).

### HTTP状态代码映射

ABP尝试按照以下规则,自动映射常见的异常类型的HTTP状态代码:

\* 对于 `AbpAuthorizationException`:

\* 用户没有登录,返回 `401` (未认证).

\* 用户已登录,但是当前访问未授权,返回 `403` (未授权).

\* 对于 `AbpValidationException` 返回 `400` (错误的请求) .

\* 对于 `EntityNotFoundException`返回 `404` (未找到).

\* 对于 `IBusinessException` 和 `IUserFriendlyException` (它是`IBusinessException`的扩展) 返回`403` (未授权) .

\* 对于 `NotImplementedException` 返回 `501` (未实现) .

\* 对于其他异常 (基础架构中未定义的) 返回 `500` (服务器内部错误) .

`IHttpExceptionStatusCodeFinder` 是用来自动判断HTTP状态代码.默认的实现是`DefaultHttpExceptionStatusCodeFinder`.可以根据需要对其进行更换或扩展.

#### 自定义映射

可以重写HTTP状态代码的自动映射,示例如下:

````C#

services.Configure<AbpExceptionHttpStatusCodeOptions>(options =>

{

options.Map("Volo.Qa:010002", HttpStatusCode.Conflict);

});

````

### 内置的异常

框架会自动抛出以下异常类型:

- 当用户没有权限执行操作时,会抛出 `AbpAuthorizationException` 异常. 有关更多信息,请参阅授权文档[authorization](Authorization.md).

- 如果当前请求的输入无效,则抛出`AbpValidationException 异常`. 有关更多信息,请参阅[验证文档](Validation.md).

- 如果请求的实体不存在,则抛出`EntityNotFoundException` 异常. 此异常大多数由 [repositories](Repositories.md) 抛出.

你同样可以在代码中抛出这些类型的异常(虽然很少需要这样做)

## 发送异常详情到客户端

你可以通过 `AbpExceptionHandlingOptions` 类的 `SendExceptionsDetailsToClients` 属性异常发送到客户端:

````csharp

services.Configure<AbpExceptionHandlingOptions>(options =>

{

options.SendExceptionsDetailsToClients = true;

});

````

## 验证

# 验证

验证系统用于验证对于特定的控制器操作或服务的方法的用户输入或客户端请求.

ABP与ASP.NET Core模型验证系统系统兼容,[模型验证文档](https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation)中的内容对于基于ABP应用程序同样有效.所以本文主要集中在ABP特征,而不是重复微软文档.

ABP增加了以下优点:

\* 定义 `IValidationEnabled` 向任意类添加自动验证. 所有的[应用服务](Application-Services.md)都实现了该接口,所以它们会被自动验证.

\* 自动将数据注解属性的验证错误信息本地化.

\* 提供可扩展的服务来验证方法调用或对象的状态.

\* 提供[FluentValidation](https://fluentvalidation.net/)的集成.

## 验证DTO

本节简要介绍了验证系统.有关详细信息请参阅[ASP.NET Core验证文档](https://docs.microsoft.com/zh-cn/aspnet/core/mvc/models/validation).

### 数据注解 Attribute

使用数据注解是一种以声明式对[DTO](Data-Transfer-Objects.md)进行验证的简单方法.

示例 :

````csharp

public class CreateBookDto

{

[Required]

[StringLength(100)]

public string Name { get; set; }

[Required]

[StringLength(1000)]

public string Description { get; set; }

[Range(0, 999.99)]

public decimal Price { get; set; }

}

````

当使用该类作为[应用服务](Application-Services.md)或控制器的参数时,将对其自动验证并抛出本地化异常(由ABP框架[处理](Exception-Handling.md)).

### IValidatableObject

`IValidatableObject` can be implemented by a DTO to perform custom validation logic. `CreateBookDto` in the following example implements this interface and checks if the `Name` is equals to the `Description` and returns a validation error in this case.

可以将DTO实现 `IValidatableObject` 接口进行自定义验证逻辑. 下面的示例中 `CreateBookDto` 实现了这个接口,并检查 `Name` 是否等于 `Description` 并返回一个验证错误.

````csharp

using System.Collections.Generic;

using System.ComponentModel.DataAnnotations;

namespace Acme.BookStore

{

public class CreateBookDto : IValidatableObject

{

[Required]

[StringLength(100)]

public string Name { get; set; }

[Required]

[StringLength(1000)]

public string Description { get; set; }

[Range(0, 999.99)]

public decimal Price { get; set; }

public IEnumerable<ValidationResult> Validate(

ValidationContext validationContext)

{

if (Name == Description)

{

yield return new ValidationResult(

"Name and Description can not be the same!",

new[] { "Name", "Description" }

);

}

}

}

}

````

#### 解析服务

如果你需要从[依赖注入系统](Dependency-Injection.md)解析服务,可以使用 `ValidationContext` 对象.

例:

````csharp

var myService = validationContext.GetRequiredService<IMyService>();

````

> 虽然可以在 `Validate` 方法中解析服务实现任何可能性,但在DTO中实现领域验证逻辑不是一个很好的做法. 应保持简单的DTO,他们的目的是传输数据(DTO:数据传输对象).

## 验证基础设施

本节介绍了ABP框架提供的一些额外的服务.

### IValidationEnabled 接口

`IValidationEnabled` 是可以由任何类来实现的空标记接口(注册到[DI](Dependency-Injection.md)并从中解析),让ABP框架为该类执行验证系统.

示例 :

````csharp

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Validation;

namespace Acme.BookStore

{

public class MyService : ITransientDependency, IValidationEnabled

{

public virtual async Task DoItAsync(MyInput input)

{

//...

}

}

}

````

> ABP框架使用[动态代理/拦截](Dynamic-Proxying-Interceptors.md)系统来执行验证.为了使其工作,你的方法应该是 **\*\*virtual\*\*** 的,服务应该被注入并通过接口(如`IMyService`)使用.

### AbpValidationException

一旦ABP确定了一个验证错误,它就会抛出类型为 `AbpValidationException` 的异常. 你的应用程序代码可以抛出 `AbpValidationException`,但大多数情况不会使用它.

\* `ValidationErrors` 是 `AbpValidationException` 的属性,它包含了验证错误列表.

\* `AbpValidationException` 的日志级别设置为 `Warning`. 它将所有验证错误记录到[日志系统](Logging.md).

\* `AbpValidationException` 由ABP框架自动捕获并将HTTP状态码设置为400转换成可用的错误响应. 参阅[异常处理](Exception-Handling.md)文档了解更多.

## 高级主题

### IObjectValidator

除了自动验证你可能需要手动验证对象,这种情况下[注入](Dependency-Injection.md)并使用 `IObjectValidator` 服务:

\* `ValidateAsync` 方法根据验证​​规则验证给定对象,如果对象没有被验证通过会抛出 `AbpValidationException` 异常.

\* `GetErrorsAsync` 不会抛出异常,只返回验证错误.

`IObjectValidator` 默认由 `ObjectValidator` 实现. `ObjectValidator`是可扩展的; 可以实现`IObjectValidationContributor`接口提供自定义逻辑.

示例 :

````csharp

public class MyObjectValidationContributor

: IObjectValidationContributor, ITransientDependency

{

public Task AddErrorsAsync(ObjectValidationContext context)

{

//Get the validating object

var obj = context.ValidatingObject;

//Add the validation errors if available

context.Errors.Add(...);

return Task.CompletedTask;

}

}

````

\* 记录将类注册到[DI](Dependency-Injection.md)(实现`ITransientDependency` 如同本例)

\* ABP会自动发现验证类,并用于任何类型的对象验证(包括自动方法调用验证).

### IMethodInvocationValidator

`IMethodInvocationValidator` 用于验证方法调用. 它在内部使用 `IObjectValidator` 来验证传递给方法调用的对象. 由于框架会自动使用此服务,通常你并不需要此服务,但在少数情况下你可能在应用程序中重用或替换它.

## FluentValidation Integration

Volo.Abp.FluentValidation 包将FluentValidation库集成到了验证系统(通过实现 `IObjectValidationContributor`). 请参阅[FluentValidation集成文档](FluentValidation.md)了解更多信息.

FluentValidation集成

# FluentValidation 集成

ABP[验证](Validation.md)基础设施是可扩展的. [Volo.Abp.FluentValidation](https://www.nuget.org/packages/Volo.Abp.FluentValidation) NuGet 包扩展了验证系统使其与[FluentValidation](https://fluentvalidation.net/)库一起工作.

## 安装

建议使用[ABP CLI](CLI.md)安装包.

### 使用ABP CLI

在项目(.csproj文件)的文件夹中打开命令行窗口并输入以下命令:

````bash

abp add-package Volo.Abp.FluentValidation

````

### 手动安装

如果你想手动安装;

1. 添加 [Volo.Abp.FluentValidation](https://www.nuget.org/packages/Volo.Abp.FluentValidation) NuGet包到你的项目:

````

Install-Package Volo.Abp.FluentValidation

````

2. 添加 `AbpFluentValidationModule` 到你的模块的依赖列表:

````csharp

[DependsOn(

//...other dependencies

typeof(AbpFluentValidationModule) //Add the FluentValidation module

)]

public class YourModule : AbpModule

{

}

````

## 使用 FluentValidation

按照 [FluentValidation文档](https://fluentvalidation.net/) 创建验证器类.

例如:

````csharp

public class CreateUpdateBookDtoValidator : AbstractValidator<CreateUpdateBookDto>

{

public CreateUpdateBookDtoValidator()

{

RuleFor(x => x.Name).Length(3, 10);

RuleFor(x => x.Price).ExclusiveBetween(0.0f, 999.0f);

}

}

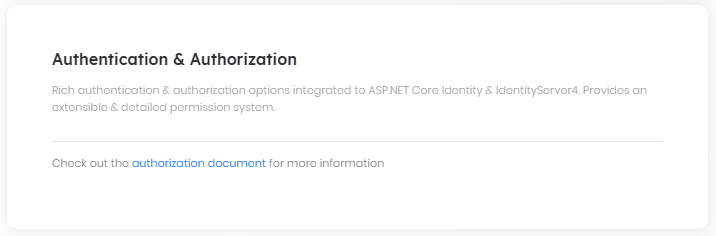
````

ABP会自动找到这个类并在对象验证时与 `CreateUpdateBookDto` 关联.

## 另请参阅

\* [验证系统](Validation.md)

## 授权



# 授权

授权用于在应用程序中判断是否允许用户执行某些特定的操作.

ABP扩展了[ASP.NET Core 授权](https://docs.microsoft.com/zh-cn/aspnet/core/security/authorization/introduction), 将 **\*\*权限\*\*** 添加为自动[策略](https://docs.microsoft.com/zh-cn/aspnet/core/security/authorization/policies)并且使授权系统在 **\*\*[**应用服务**](Application-Services.md)\*\*** 同样可用.

所以ASP.NET Core授权的功能特性和它的文档在基于ABP的应用程序是可用的.

本文中着重介绍在ASP.NET Core授权功能基础上添加的功能.

## Authorize Attribute

ASP.NET Core 定义了 [\*\*Authorize\*\*](https://docs.microsoft.com/zh-cn/aspnet/core/security/authorization/simple) attribute 用于在控制器,控制器方法以及页面上授权. 现在ABP将它带到了[应用服务](Application-Services.md).

示例:

````csharp

using System;

using System.Collections.Generic;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Authorization;

using Volo.Abp.Application.Services;

namespace Acme.BookStore

{

[Authorize]

public class AuthorAppService : ApplicationService, IAuthorAppService

{

public Task<List<AuthorDto>> GetListAsync()

{

...

}

[AllowAnonymous]

public Task<AuthorDto> GetAsync(Guid id)

{

...

}

[Authorize("BookStore\_Author\_Create")]

public Task CreateAsync(CreateAuthorDto input)

{

...

}

}

}

````

\* `Authorize` attribute 使用户必须登陆到应用程序才可以访问 `AuthorAppService` 中的方法. 所以`GetListAsync` 方法仅可用于通过身份验证的用户.

\* `AllowAnonymous` 禁用身份验证. 所以 `GetAsync` 方法任何人都可以访问,包括未授权的用户.

\* `[Authorize("BookStore\_Author\_Create")]` 定义了一个策略 (参阅 [基于策略的授权](https://docs.microsoft.com/zh-cn/aspnet/core/security/authorization/policies)),它用于检查当前用户的权限.

"BookStore\_Author\_Create" 是一个策略名称. 如果你想要使用策略的授权方式,需要在ASP.NET Core授权系统中预先定义它.

你可以按照ASP.NET Core文档进行实施策略授权,但对于简单的 `true/false` 条件(比如是否授予了用户策略) ABP定义了权限系统,在下一部分中会进行讲解.

## 权限系统

权限系统是为特定用户,角色或客户端授权或禁止的简单策略.

### 定义权限

创建一个继承自 `PermissionDefinitionProvider` 的类,如下所示:

````csharp

using Volo.Abp.Authorization.Permissions;

namespace Acme.BookStore.Permissions

{

public class BookStorePermissionDefinitionProvider : PermissionDefinitionProvider

{

public override void Define(IPermissionDefinitionContext context)

{

var myGroup = context.AddGroup("BookStore");

myGroup.AddPermission("BookStore\_Author\_Create");

}

}

}

````

> ABP会自动发现这个类,不需要进行配置!

你需要在 `Define` 方法中添加**\*\*权限组\*\***或者获取已存在的权限组,并向权限组中添加**\*\*权限\*\***.

在定义权限后就可以在ASP.NET Core权限系统中当做**\*\*策略\*\***名称使用. 在角色的权限管理模态框中同样可以看到:

![authorization-new-permission-ui](images/authorization-new-permission-ui.png)

\* 左边的选项卡显示了 "BookStore" 权限组.

\* 右侧的 "BookStore\_Author\_Create" 是权限名称,在这里可以为角色授权或禁止.

保存后更改会持久化到数据库并在授权系统使用.

> 只有在你安装了identity模块时,才会出现上图中的管理对话框. 该模块用于用户和角色的. 启动模板中已经预装identity模块.

#### 本地化权限名称

"BookStore\_Author\_Create" 名称对于权限系统来说很适合,但对于UI展示会让操作人员费解. 幸运的是 `AddPermission` 和 `AddGroup` 方法提供了 `LocalizableString` 参数:

````csharp

var myGroup = context.AddGroup(

"BookStore",

LocalizableString.Create<BookStoreResource>("BookStore")

);

myGroup.AddPermission(

"BookStore\_Author\_Create",

LocalizableString.Create<BookStoreResource>("Permission:BookStore\_Author\_Create")

);

````

然后在本地化文档中对 "BookStore" 和 "Permission:BookStore\_Author\_Create" 键添加本地化语言:

````json

"BookStore": "Book Store",

"Permission:BookStore\_Author\_Create": "Creating a new author"

````

> 有关更多信息请参阅[本地化系统文档](Localization.md).

下图展示了本地化后的效果:

![authorization-new-permission-ui-localized](images/authorization-new-permission-ui-localized.png)

#### 多租户

权限系统同样支持在ABP中做为一等公民的 [多租户](Multi-Tenancy.md). 在定义新权限时可以设置多租户选项. 有下面三个值:

\* **\*\*Host\*\***: 权限仅适用于宿主.

\* **\*\*Tenant\*\***: 权限仅适用于租户.

\* **\*\*Both\*\*** (默认): 权限适用与宿主和租户.

> 如果你的应用程序不是多租户的,可以忽略这个选项.

`AddPermission` 方法的第三个参数用于设置多租户选项:

````csharp

myGroup.AddPermission(

"BookStore\_Author\_Create",

LocalizableString.Create<BookStoreResource>("Permission:BookStore\_Author\_Create"),

multiTenancySide: MultiTenancySides.Tenant //set multi-tenancy side!

);

````

#### 启用/禁用权限

权限默认为启用. 它也可以被禁用,禁用权限所有的用户将无法使用它. 你仍然可以检查这个权限,但它总是会返回被禁止.

定义示例:

````csharp

myGroup.AddPermission("Author\_Management", isEnabled: false);

````

通常你不需要定义禁用权限(除非你暂时想要禁用应用程序的功能). 无论怎样,你可能想要禁用依赖模块中定义的权限,这样你可以禁用相关的功能. 参阅下面的 "*\*更改依赖模块的权限定义\**" 节,查看示例用法.

> 注意:检查一个未定义的权限会抛出异常,而被禁用的权限的返回禁止(false).

#### 子权限

权限可以具有子权限,当你想要创建一个层次结构的权限树时它特别有用. 在这个树中一个权限可能含有子权限,并且子权限只有在授权父权限时才可用.

定义示例:

````csharp

var authorManagement = myGroup.AddPermission("Author\_Management");

authorManagement.AddChild("Author\_Management\_Create\_Books");

authorManagement.AddChild("Author\_Management\_Edit\_Books");

authorManagement.AddChild("Author\_Management\_Delete\_Books");

````

在页面上如下所示 (你可能想要本地化权限名称):

![authorization-new-permission-ui-hierarcy](images/authorization-new-permission-ui-hierarcy.png)

下面的示例代码是一个典型的应用服务:

````csharp

[Authorize("Author\_Management")]

public class AuthorAppService : ApplicationService, IAuthorAppService

{

public Task<List<AuthorDto>> GetListAsync()

{

...

}

public Task<AuthorDto> GetAsync(Guid id)

{

...

}

[Authorize("Author\_Management\_Create\_Books")]

public Task CreateAsync(CreateAuthorDto input)

{

...

}

[Authorize("Author\_Management\_Edit\_Books")]

public Task UpdateAsync(CreateAuthorDto input)

{

...

}

[Authorize("Author\_Management\_Delete\_Books")]

public Task DeleteAsync(CreateAuthorDto input)

{

...

}

}

````

\* 拥有`Author\_Management`权限的用户可以访问`GetListAsync`和 `GetAsync`方法.

\* 示例中的其他方法需要额外的权限.

### 自定义策略覆盖已有权限

有时我们需要扩展扩展应用程序中预构建模块的权限,可以定义并注册一个与权限名称相同的策略到ASP.Net Core授权系统,策略会覆盖已有权限.

参阅 [基于策略的授权](https://docs.microsoft.com/zh-cn/aspnet/core/security/authorization/policies) 文档了解如何自定义策略.

### 更改依赖模块的权限定义

从 `PermissionDefinitionProvider` 派生的类(就像上面的示例一样) 可以获取现有的权限定义(由依赖[模块](Module-Development-Basics.md)定义)并更改其定义.

示例:

````csharp

context

.GetPermissionOrNull(IdentityPermissions.Roles.Delete)

.IsEnabled = false;

````

当你在权限提供程序编写了这行代码,它会找到[身份模块](Modules/Identity.md)的 "role deletion" 权限并且禁用它,因此没有人可以在应用程序中删除角色.

> 提供: 更好的方式应该检查 `GetPermissionOrNull` 返回值,如果权限未定义,它会返回null值.

## IAuthorizationService

ASP.NET Core 提供了 `IAuthorizationService` 用于检查权限. 注入后使用它进行条件控制权限.

示例:

````csharp

public async Task CreateAsync(CreateAuthorDto input)

{

var result = await AuthorizationService

.AuthorizeAsync("Author\_Management\_Create\_Books");

if (result.Succeeded == false)

{

//throw exception

throw new AbpAuthorizationException("...");

}

//continue to the normal flow...

}

````

> 因为应用服务会经常检查权限,`ApplicationService` 已经属性注入了`AuthorizationService`, 所有继承自 `ApplicationService` 的类都可以直接使用. 你也可以直接 [注入](Dependency-Injection.md)到类中.

上面的示例代码是检查权限的标准代码,ABP提供了一种简化的方式来编写它.

示例:

````csharp

public async Task CreateAsync(CreateAuthorDto input)

{

await AuthorizationService.CheckAsync("Author\_Management\_Create\_Books");

//continue to the normal flow...

}

````

如果未授权 `CheckAsync` 扩展方法会抛出 `AbpAuthorizationException` 异常. 还有一个 `IsGrantedAsync` 扩展方法会返回 `true` 或 `false`.

`IAuthorizationService` 中有多个 `AuthorizeAsync` 方法重载. [ASP.NET Core 授权文档](https://docs.microsoft.com/zh-cn/aspnet/core/security/authorization/introduction)中有详细的解释.

> 提示: 尽可能使用声明式的 `Authorize` attribute,因为它比较简单不会侵入方法内部. 如果你需要在业务代码中有条件的检查权限,那么请使用 `IAuthorizationService`.

### 在JavaScript中检查权限

有时你会需要在客户端检查策略/权限. 在ASP.NET Core MVC/Razor页面应用程序可以使用 `abp.auth` API. 示例:

````js

abp.auth.isGranted('MyPermissionName');

````

## 权限管理

通常权限管理是管理员用户使用权限管理模态框进行授权:

![authorization-new-permission-ui-localized](images/authorization-new-permission-ui-localized.png)

如果你想要通过代码管理权限, 可以注入使用 `IPermissionManager`. 如下所示:

````csharp

public class MyService : ITransientDependency

{

private readonly IPermissionManager \_permissionManager;

public MyService(IPermissionManager permissionManager)

{

\_permissionManager = permissionManager;

}

public async Task GrantPermissionForUserAsync(Guid userId, string permissionName)

{

await \_permissionManager.SetForUserAsync(userId, permissionName, true);

}

public async Task ProhibitPermissionForUserAsync(Guid userId, string permissionName)

{

await \_permissionManager.SetForUserAsync(userId, permissionName, false);

}

}

````

`SetForUserAsync` 方法用于设置用户的权限 (true/false). 类似的还有 `SetForRoleAsync` 和 `SetForClientAsync` 扩展方法.

`IPermissionManager` 由权限管理模块定义, 更多信息请参阅 [权限管理模块文档](Modules/Permission-Management.md).

## 高级主题

### Permission Value Providers

权限检查是可扩展的. 继承自 `PermissionValueProvider` (或实现 `IPermissionValueProvider`) 的任何类都可以参与权限检查. 有三个预定义的Provider:

\* `UserPermissionValueProvider` 从当前的声明中拿到当前用户ID并检查用户授权. 用户声明由 `AbpClaimTypes.UserId` 静态属性定义.

\* `RolePermissionValueProvider` 从当前的声明中拿到授予当前用户的角色集合并且判断角色是否具有指定的权限. 角色声明由 `AbpClaimTypes.Role` 静态属性定义.

\* `ClientPermissionValueProvider` 从当前声明中拿到当前客户端并检查客户端是否具有指定的权限. 这在没有当前登录用户的客户端交互特别有用. 客户端声明由 `AbpClaimTypes.ClientId` 静态属性定义.

你可以定义自己的`PermissionValueProvider`扩展权限检查系统.

示例:

````csharp

public class SystemAdminPermissionValueProvider : PermissionValueProvider

{

public SystemAdminPermissionValueProvider(IPermissionStore permissionStore)

: base(permissionStore)

{

}

public override string Name => "SystemAdmin";

public async override Task<PermissionGrantResult>

CheckAsync(PermissionValueCheckContext context)

{

if (context.Principal?.FindFirst("User\_Type")?.Value == "SystemAdmin")

{

return PermissionGrantResult.Granted;

}

return PermissionGrantResult.Undefined;

}

}

````

示例`SystemAdminPermissionValueProvider`允许声明`User\_Type`值为`SystemAdmin`的用户授予所有权限. 通常在`Provider`中使用当前声明和 `IPermissionStore`.

`PermissionValueProvider` 的 `CheckAsync` 应该返回下面三个值之一:

\* `PermissionGrantResult.Granted` 授予用户权限,如果没有其他的授权值提供程序返回 `Prohibited`, 那么最后会返回 `Granted`.

\* `PermissionGrantResult.Prohibited` 禁止授权用户,任何一个授权值提供程序返回了 `Prohibited`, 那么其他的提供程序返回的值都不再重要.

\* `PermissionGrantResult.Undefined` 代表当前无法确定是否授予或禁止权限, 返回`UnDefined`由其他权限值提供程序检查权限.

定义`Provider`后将其添加到 `AbpPermissionOptions`,如下所示:

````csharp

Configure<AbpPermissionOptions>(options =>

{

options.ValueProviders.Add<SystemAdminPermissionValueProvider>();

});

````

### Permission Store

`IPermissionStore` 是唯一需要从持久化源(通常是数据库)中读取权限值的接口. 它的实现在权限管理模块. 参见 [权限管理模块](Modules/Permission-Management.md) 了解更多信息

### AlwaysAllowAuthorizationService

`AlwaysAllowAuthorizationService` 类可以绕过授权服务. 通常用于在需要禁用授权系统的集成测试中.

使用 `IServiceCollection.AddAlwaysAllowAuthorization()` 扩展方法将 `AlwaysAllowAuthorizationService` 注册到 [依赖注入](Dependency-Injection.md) 系统中:

````csharp

public override void ConfigureServices(ServiceConfigurationContext context)

{

context.Services.AddAlwaysAllowAuthorization();

}

````

启动模板的集成测试已经禁用了授权服务.

### Claims Principal Factory

声明是认证和授权的重要组成部分. ABP 使用 `IAbpClaimsPrincipalFactory` 来服务创建身份认证声明. 该服务被设计为可扩展的. 如果你需要将自定义声明添加到身份认证票证中, 可以在你的应用程序中实现 `IAbpClaimsPrincipalContributor`.

**\*\*示例:添加一个** `SocialSecurityNumber ` **声明并获取它:\*\***

```csharp

public class SocialSecurityNumberClaimsPrincipalContributor : IAbpClaimsPrincipalContributor, ITransientDependency

{

public async Task ContributeAsync(AbpClaimsPrincipalContributorContext context)

{

var identity = context.ClaimsPrincipal.Identities.FirstOrDefault();

var userId = identity?.FindUserId();

if (userId.HasValue)

{

var userService = context.ServiceProvider.GetRequiredService<IUserService>(); //Your custom service

var socialSecurityNumber = await userService.GetSocialSecurityNumberAsync(userId.Value);

if (socialSecurityNumber != null)

{

identity.AddClaim(new Claim("SocialSecurityNumber", socialSecurityNumber));

}

}

}

}

Configure<AbpClaimsServiceOptions>(options=>

{

options.RequestedClaims.Add("SocialSecurityNumber")

})

public static class CurrentUserExtensions

{

public static string GetSocialSecurityNumber(this ICurrentUser currentUser)

{

return currentUser.FindClaimValue("SocialSecurityNumber");

}

}

```

## 接下来

\* [权限管理模块](Modules/Permission-Management.md)

\* [ASP.NET Core MVC / Razor 页面 JavaScript Auth API](API/JavaScript-API/Auth.md)

\* [Angular界面中的权限管理](UI/Angular/Permission-Management.md)

## 缓存

# 缓存

ABP框架扩展了 [ASP.NET Core的分布式缓存系统](https://docs.microsoft.com/en-us/aspnet/core/performance/caching/distributed).

## 安装

> 默认情况下 [启动模板](Startup-Templates/Application.md) 已经安装了这个包. 所以大部分情况下你不需要手动安装.

[Volo.Abp.Caching](https://www.nuget.org/packages/Volo.Abp.Caching)是缓存系统的核心包. 可以使用 [ABP CLI](CLI.md) 的add-package命令将其安装到项目中：

```

abp add-package Volo.Abp.Caching

```

你需要在包含 `csproj` 文件的文件夹中的命令行终端上运行此命令(请参阅 [其他选项](https://abp.io/package-detail/Volo.Abp.Caching) 安装).

## 使用方式

### `IDistributedCache` 接口

ASP.NET Core 定义了 `IDistributedCache` 接口用于 get/set 缓存值. 但是会有以下问题:

\* 它适用于 **\*\*byte 数组\*\*** 而不是 .NET 对象. 因此你需要对缓存的对象进行**\*\*序列化/反序列化\*\***.

\* 它为所有的缓存项提供了 **\*\*单个 key 池\*\*** , 因此;

\* 你需要注意键区分 **\*\*不同类型的对象\*\***.

\* 你需要注意**\*\*不同租户\*\***(参见[多租户](Multi-Tenancy.md))的缓存项.

> `IDistributedCache` 定义在 `Microsoft.Extensions.Caching.Abstractions` 包中. 这使它不仅适用于ASP.NET Core应用程序, 也可用于**\*\*任何类型的程序\*\***.

> `IDistributedCache` 接口的默认实现是 `MemoryDistributedCache` 它使用**\*\*内存\*\***工作. 参见 [ASP.NET Core文档](https://docs.microsoft.com/zh-cn/aspnet/core/performance/caching/distributed) 了解如何切换到 **\*\*Redis\*\*** 或其他缓存提供程序. 此外, 如果要将Redis用作分布式缓存服务器, [Redis缓存](Redis-Cache.md) 文档.

有关更多信息, 参见 [ASP.NET Core 分布式缓存文档](https://docs.microsoft.com/zh-cn/aspnet/core/performance/caching/distributed).

### `IDistributedCache<TCacheItem>` 接口

ABP框架在[Volo.Abp.Caching](https://www.nuget.org/packages/Volo.Abp.Caching/)包定义了通用的泛型 `IDistributedCache<TCacheItem>` 接口. `TCacheItem` 是存储在缓存中的对象类型.

`IDistributedCache<TCacheItem>` 接口解决了上述中的问题;

\* 它在内部 **\*\*序列化/反序列化\*\*** 缓存对象. 默认使用 **\*\*JSON\*\*** 序列化, 但可以替换[依赖注入](Dependency-Injection.md)系统中 `IDistributedCacheSerializer` 服务的实现来覆盖默认的处理.

\* 它根据缓存中对象类型自动向缓存key添加 **\*\*缓存名称\*\*** 前缀. 默认缓存名是缓存对象类的全名(如果你的类名以`CacheItem` 结尾, 那么`CacheItem` 会被忽略,不应用到缓存名称上). 你也可以在缓存类上使用 **\*\***`CacheName` **特性\*\*** 设置缓存的名称.

\* 它自动将**\*\*当前的租户id\*\***添加到缓存键中, 以区分不同租户的缓存项 (只有在你的应用程序是[多租户](Multi-Tenancy.md)的情况下生效). 如果要在多租户应用程序中的所有租户之间共享缓存对象, 请在缓存项类上定义`IgnoreMultiTenancy`特性以禁用此选项.

\* 允许为每个应用程序定义 **\*\*全局缓存键前缀\*\*** , 不同的应用程序可以在共享的分布式缓存中拥有自己的隔离池.

\* 它可以在任何可能绕过缓存的情况下 **\*\*容忍错误\*\*** 的发生. 这在缓存服务器出现临时问题时非常有用.

\* 它有 `GetManyAsync` 和 `SetManyAsync` 等方法, 可以显著提高**\*\*批处理\*\***的性能.

**\*\*示例: 在缓存中存储图书名称和价格\*\***

````csharp

namespace MyProject

{

public class BookCacheItem

{

public string Name { get; set; }

public float Price { get; set; }

}

}

````

你可以注入 `IDistributedCache<BookCacheItem>` 服务用于 get/set `BookCacheItem` 对象.

````csharp

using System;

using System.Threading.Tasks;

using Microsoft.Extensions.Caching.Distributed;

using Volo.Abp.Caching;

using Volo.Abp.DependencyInjection;

namespace MyProject

{

public class BookService : ITransientDependency

{

private readonly IDistributedCache<BookCacheItem> \_cache;

public BookService(IDistributedCache<BookCacheItem> cache)

{

\_cache = cache;

}

public async Task<BookCacheItem> GetAsync(Guid bookId)

{

return await \_cache.GetOrAddAsync(

bookId.ToString(), //缓存键

async () => await GetBookFromDatabaseAsync(bookId),

() => new DistributedCacheEntryOptions

{

AbsoluteExpiration = DateTimeOffset.Now.AddHours(1)

}

);

}

private Task<BookCacheItem> GetBookFromDatabaseAsync(Guid bookId)

{

//TODO: 从数据库获取数据

}

}

}

````

\* 示例服务代码中的 `GetOrAddAsync()` 方法从缓存中获取图书项. `GetOrAddAsync`是ABP框架在 ASP.NET Core 分布式缓存方法中添增的附加方法.

\* 如果没有在缓存中找到图书,它会调用工厂方法 (本示例中是 `GetBookFromDatabaseAsync`)从原始数据源中获取图书项.

\* `GetOrAddAsync` 有一个可选参数 `DistributedCacheEntryOptions` , 可用于设置缓存的生命周期.

`IDistributedCache<BookCacheItem>` 与ASP.NET Core的`IDistributedCache` 接口拥有相同的方法, 你可以参考 [ASP.NET Core文档](https://docs.microsoft.com/zh-cn/aspnet/core/performance/caching/distributed).

### `IDistributedCache<TCacheItem, TCacheKey>` 接口

`IDistributedCache<TCacheItem>` 接口默认了**\*\*缓存键\*\***是 `string` 类型 (如果你的键不是string类型需要进行手动类型转换). 但当缓存键的类型不是`string`时, 可以使用`IDistributedCache<TCacheItem, TCacheKey>`.

**\*\*示例: 在缓存中存储图书名称和价格\*\***

示例缓存项

````csharp

using Volo.Abp.Caching;

namespace MyProject

{

[CacheName("Books")]

public class BookCacheItem

{

public string Name { get; set; }

public float Price { get; set; }

}

}

````

\* 在本例中使用`CacheName`特性给`BookCacheItem`类设置缓存名称.

你可以注入 `IDistributedCache<BookCacheItem, Guid>` 服务用于 get/set `BookCacheItem` 对象.

````csharp

using System;

using System.Threading.Tasks;

using Microsoft.Extensions.Caching.Distributed;

using Volo.Abp.Caching;

using Volo.Abp.DependencyInjection;

namespace MyProject

{

public class BookService : ITransientDependency

{

private readonly IDistributedCache<BookCacheItem, Guid> \_cache;

public BookService(IDistributedCache<BookCacheItem, Guid> cache)

{

\_cache = cache;

}

public async Task<BookCacheItem> GetAsync(Guid bookId)

{

return await \_cache.GetOrAddAsync(

bookId, //Guid类型作为缓存键

async () => await GetBookFromDatabaseAsync(bookId),

() => new DistributedCacheEntryOptions

{

AbsoluteExpiration = DateTimeOffset.Now.AddHours(1)

}

);

}

private Task<BookCacheItem> GetBookFromDatabaseAsync(Guid bookId)

{

//TODO: 从数据库获取数据

}

}

}

````

\* 示例服务中 `GetOrAddAsync()` 方法获取缓存的图书项.

\* 我们采用了 `Guid` 做为键, 在 `\_cache\_GetOrAddAsync()` 方法中传入 `Guid` 类型的bookid.

#### 复杂类型的缓存键

`IDistributedCache<TCacheItem, TCacheKey>` 在内部使用键对象的 `ToString()` 方法转换类型为string. 如果你的将复杂对象做为缓存键,那么需要重写类的 `ToString` 方法.

举例一个作为缓存键的类:

````csharp

public class UserInOrganizationCacheKey

{

public Guid UserId { get; set; }

public Guid OrganizationId { get; set; }

//构建缓存key

public override string ToString()

{

return $"{UserId}\_{OrganizationId}";

}

}

````

用法示例:

````csharp

public class BookService : ITransientDependency

{

private readonly IDistributedCache<UserCacheItem, UserInOrganizationCacheKey> \_cache;

public BookService(

IDistributedCache<UserCacheItem, UserInOrganizationCacheKey> cache)

{

\_cache = cache;

}

...

}

````

## 配置

### AbpDistributedCacheOptions

`AbpDistributedCacheOptions` 是配置缓存的主要[Option类](Options.md).

**\*\*示例：为应用程序设置缓存键前缀\*\***

```csharp

Configure<AbpDistributedCacheOptions>(options =>

{

options.KeyPrefix = "MyApp1";

});

```

> 在[模块类](Module-Development-Basics.md)的`ConfigureServices`方法中添加代码.

#### 可用选项

\* `HideErrors` (`bool`, 默认: `true`): 启用/禁用隐藏从缓存服务器写入/读取值时的错误.

\* `KeyPrefix` (`string`, 默认: `null`): 如果你的缓存服务器由多个应用程序共同使用, 则可以为应用程序的缓存键设置一个前缀. 在这种情况下, 不同的应用程序不能覆盖彼此的缓存内容.

\* `GlobalCacheEntryOptions` (`DistributedCacheEntryOptions`): 用于设置保存缓内容却没有指定选项时, 默认的分布式缓存选项 (例如 `AbsoluteExpiration` 和 `SlidingExpiration`). `SlidingExpiration`的默认值设置为20分钟.

## 错误处理

当为你的对象设计缓存时, 通常会首先尝试从缓存中获取值. 如果在缓存中找不到该值, 则从**\*\*来源\*\***查询对象. 它可能在**\*\*数据库\*\***中, 或者可能需要通过HTTP调用远程服务器.

在大多数情况下, 你希望**\*\*容忍缓存错误\*\***; 如果缓存服务器出现错误, 也不希望取消该操作. 相反, 你可以默默地隐藏(并记录)错误并**\*\*从来源查询\*\***. 这就是ABP框架默认的功能.

ABP的分布式缓存 [异常处理](Exception-Handling.md), 默认记录并隐藏错误. 有一个全局修改该功能的选项(参见下面的选项内容).

所有的`IDistributedCache<TCacheItem>` (和 `IDistributedCache<TCacheItem, TCacheKey>`)方法都有一个可选的参数`hideErrors`, 默认值为`null`. 如果此参数设置为`null`, 则全局生效, 否则你可以选择单个方法调用时隐藏或者抛出异常.

## 批量操作

ABP的分布式缓存接口定义了以下批量操作方法,当你需要在一个方法中调用多次缓存操作时,这些方法可以提高性能

\* `SetManyAsync` 和 `SetMany` 方法可以用来向缓存中设置多个值.

\* `GetManyAsync` 和 `GetMany` 方法可以用来从缓存中获取多个值.

\* `GetOrAddManyAsync` 和 `GetOrAddMany` 方法可以用来从缓存中获取并添加缺少的值.

\* `RefreshManyAsync` 和 `RefreshMany` 方法可以来用重置多个值的滚动过期时间.

\* `RemoveManyAsync` 和 `RemoveMany` 方法可以用来从缓存中删除多个值.

> 这些不是标准的ASP.NET Core缓存方法, 所以某些提供程序可能不支持. [ABP Redis集成包](Redis-Cache.md)实现了它们. 如果提供程序不支持,会回退到 `SetAsync` 和 `GetAsync` ... 方法(循环调用).

## 高级主题

### 工作单元级别的缓存

分布式缓存服务提供了一个有趣的功能. 假设你已经更新了数据库中某本书的价格, 然后将新价格设置到缓存中, 以便以后使用缓存的值. 如果设置缓存后出现异常, 并且更新图书价格的**\*\*事务被回滚了\*\***, 该怎么办？在这种情况下, 缓存值是错误的.

`IDistributedCache<..>`方法提供一个可选参数, `considerUow`, 默认为`false`. 如果将其设置为`true`, 则你对缓存所做的更改不会应用于真正的缓存存储, 而是与当前的[工作单元](Unit-Of-Work.md)关联. 你将获得在同一工作单元中设置的缓存值, 但**\*\*仅当前工作单元成功时\*\***更改才会生效.

### IDistributedCacheSerializer

`IDistributedCacheSerializer`服务用于序列化和反序列化缓存内容. 默认实现是`Utf8JsonDistributedCacheSerializer`类, 它使用`IJsonSerializer`服务将对象转换为[JSON](Json-Serialization.md), 反之亦然. 然后, 它使用UTC8编码将JSON字符串转换为分布式缓存接受的字节数组.

如果你想实现自己的序列化逻辑, 可以自己实现并[替换](Dependency-Injection.md) 此服务.

### IDistributedCacheKeyNormalizer

默认情况下, `IDistributedCacheKeyNormalizer`是由`DistributedCacheKeyNormalizer`类实现的. 它将缓存名称、应用程序缓存前缀和当前租户id添加到缓存键中. 如果需要更高级的键规范化, 可以自己实现并[替换](Dependency-Injection.md)此服务.

## 另请参阅

\* [Redis 缓存](Redis-Cache.md)

Redis 缓存

# Redis 缓存

ABP Framework [缓存系统](Caching.md) 拓展了 [ASP.NET Core 分布式缓存](https://docs.microsoft.com/zh-cn/aspnet/core/performance/caching/distributed). 因此, 标准 ASP.NET Core 分布式缓存支持的 **\*\*任何提供程序\*\*** 都可以在你的应用程序中使用，并且可以像 **\*\*微软文档\*\*** 那样进行配置.

不过, ABP为 Redis Cache 还提供了一个集成包: [Volo.Abp.Caching.StackExchangeRedis](https://www.nuget.org/packages/Volo.Abp.Caching.StackExchangeRedis).

为何使用中这个包而不是标准的[Microsoft.Extensions.Caching.StackExchangeRedis](https://www.nuget.org/packages/Microsoft.Extensions.Caching.StackExchangeRedis/)包有两个原因.

1. 它实现 `SetManyAsync` 和 `GetManyAsync` 方法. 这些都不是 Microsoft 缓存库的标准方法，而是由ABP框架[缓存](Caching.md)系统添加的. 当你需要通过单个方法调用设置/获取多个缓存项时，它们**\*\*显著提高了性能\*\***.

2. 它 **\*\*简化\*\*** 了 Redis 缓存 **\*\*配置\*\*** (将在下面解释).

> Volo.Abp.Caching.StackExchangeRedis 已经使用了 Microsoft.Extensions.Caching.StackExchangeRedis 包，但对其进行了扩展和改进.

## 安装

> 如果使用Redis，则此软件包已安装在应用程序启动模板中.

在项目`.csproj`的文件夹中打开命令行窗口并输入命令:

````bash

abp add-package Volo.Abp.Caching.StackExchangeRedis

````

## 配置

Volo.Abp.Caching.StackExchangeRedis 包自动从`IConfiguration`获取Redis[配置](Configuration.md). 因此，你可以在以下位置设置配置 `appsettings.json`:

````js

"Redis": {

"IsEnabled": "true",

"Configuration": "127.0.0.1"

}

````

设置`IsEnabled`为可选的，如果未设置将默认视为`true`.

或者，你可以在[模块](Module-Development-Basics.md)的ConfigureServices方法中配置标准的[RedisCacheOptions](https://docs.microsoft.com/zh-cn/dotnet/api/microsoft.extensions.caching.stackexchangeredis.rediscacheoptions)类：

````csharp

Configure<RedisCacheOptions>(options =>

{

//...

});

````

## 另请参阅

\* [缓存](Caching.md)

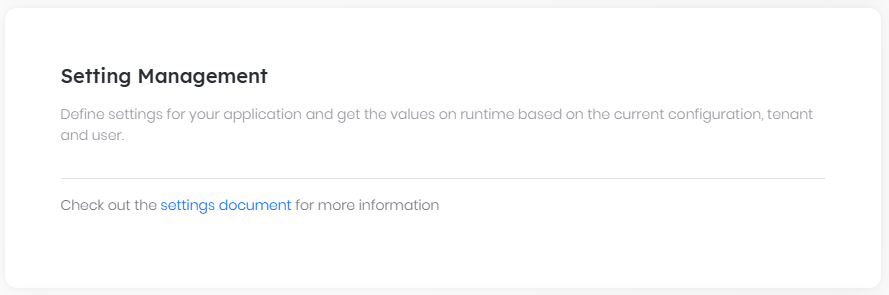
## 日志

# 日志

ABP框架没有实现任何日志基础设施. 它使用[ASP.NET Core日志系统](https://docs.microsoft.com/zh-cn/aspnet/core/fundamentals/logging).

> .NET Core 的日志系统是独立于ASP.NET Core的,它可以在任何类型的应用程序中使用.

## 设置管理



# Settings

[配置系统](Configuration.md) 是在启动时配置应用程序很好的方式. 除了配置之外, ABP提供了另外一种设置和获取应用程序设置的方式.

设置存储在动态数据源(通常是数据库)中的键值对. 设置系统预构建了用户,租户,全局和默认设置方法并且可以进行扩展.

## 定义设置

使用设置之前需要定义它. ABP是 [模块化](Module-Development-Basics.md)的, 不同的模块可以拥有不同的设置. 模块中派生 `SettingDefinitionProvider` 类定义模块内的配置. 示例如下:

````csharp

public class EmailSettingProvider : SettingDefinitionProvider

{

public override void Define(ISettingDefinitionContext context)

{

context.Add(

new SettingDefinition("Smtp.Host", "127.0.0.1"),

new SettingDefinition("Smtp.Port", "25"),

new SettingDefinition("Smtp.UserName"),

new SettingDefinition("Smtp.Password", isEncrypted: true),

new SettingDefinition("Smtp.EnableSsl", "false")

);

}

}

````

ABP会自动发现并注册设置的定义.

### SettingDefinition

`SettingDefinition` 类具有以下属性:

\* **\*\*Name\*\***: 应用程序中设置的唯一名称. 是**\*\*具有约束的唯一属性\*\***, 在应用程序获取/设置此设置的值 (设置名称定义为常量而不是`magic`字符串是个好主意).

\* **\*\*DefaultValue\*\***: 设置的默认值.

\* **\*\*DisplayName\*\***: 本地化的字符串,用于在UI上显示名称.

\* **\*\*Description\*\***: 本地化的字符串,用于在UI上显示描述.

\* **\*\*IsVisibleToClients\*\***: 布尔值,表示此设置是否在客户端可用. 默认为false,避免意外暴漏内部关键设置.

\* **\*\*IsInherited\*\***: 布尔值,此设置值是否从其他提供程序继承. 如果没有为请求的提供程序设置设定值,那么默认值是true并回退到下一个提供程序 (参阅设置值提供程序部分了解更多).

\* **\*\*IsEncrypted\*\***: 布尔值,表示是否在保存值是加密,读取时解密. 在数据库中存储加密的值.

\* **\*\*Providers\*\***: 限制可用于特定的设置值提供程序(参阅设置值提供程序部分了解更多).

\* **\*\*Properties\*\***: 设置此值的自定义属性 名称/值 集合,可以在之后的应用程序代码中使用.

### 更改依赖模块的设置定义

在某些情况下,你可能希望更改应用程序/模块所依赖的其他模块中定义的设置的某些属性. 设置定义提供程序可以查询和更新设置定义.

下面的示例中获取了由 [Volo.Abp.Emailing](Emailing.md) 包定义的设置并将其更改:

````csharp

public class MySettingDefinitionProvider : SettingDefinitionProvider

{

public override void Define(ISettingDefinitionContext context)

{

var smtpHost = context.GetOrNull("Abp.Mailing.Smtp.Host");

if (smtpHost != null)

{

smtpHost.DefaultValue = "mail.mydomain.com";

smtpHost.DisplayName =

new LocalizableString(

typeof(MyLocalizationResource),

"SmtpServer\_DisplayName"

);

}

}

}

````

> 使用常量作为设置名称是一种好习惯,ABP的包就是这样做的. `Abp.Mailing.Smtp.Host`设置名称是在`EmailSettingNames`类(在Volo.Abp.Emailing命名空间中)定义的常量.

## 读取设置值

### ISettingProvider

`ISettingProvider` 用于获取指定设置的值或所有设置的值. 示例用法:

````csharp

public class MyService

{

private readonly ISettingProvider \_settingProvider;

//Inject ISettingProvider in the constructor

public MyService(ISettingProvider settingProvider)

{

\_settingProvider = settingProvider;

}

public async Task FooAsync()

{

//Get a value as string.

string userName = await \_settingProvider.GetOrNullAsync("Smtp.UserName");

//Get a bool value and fallback to the default value (false) if not set.

bool enableSsl = await \_settingProvider.GetAsync<bool>("Smtp.EnableSsl");

//Get a bool value and fallback to the provided default value (true) if not set.

bool enableSsl = await \_settingProvider.GetAsync<bool>(

"Smtp.EnableSsl", defaultValue: true);

//Get a bool value with the IsTrueAsync shortcut extension method

bool enableSsl = await \_settingProvider.IsTrueAsync("Smtp.EnableSsl");

//Get an int value or the default value (0) if not set

int port = (await \_settingProvider.GetAsync<int>("Smtp.Port"));

//Get an int value or null if not provided

int? port = (await \_settingProvider.GetOrNullAsync("Smtp.Port"))?.To<int>();

}

}

````

> `ISettingProvider` 是非常常用的服务,一些基类中(如`IApplicationService`)已经将其属性注入. 这种情况下可以直接使用`SettingProvider`.

### 在客户端读取设置值

如果允许在客户端显示某个设置,可以使用 JavaScript 代码读取设置值. 示例:

````js

//Gets a value as string.

var language = abp.setting.get('Abp.Localization.DefaultLanguage');

//Gets an integer value.

var requiredLength = abp.setting.getInt('Abp.Identity.Password.RequiredLength');

//Gets a boolean value.

var requireDigit = abp.setting.getBoolean('Abp.Identity.Password.RequireDigit');

````

使用 `abp.setting.values` 可以读取所有设置值的字典.

## 设置值提供程序

设置系统是可扩展的, 你可以定义设置值提供程序扩展它,根据任何条件从任何来源获取设置值.

`ISettingProvider` 使用设置值提供程序来获取设置值. 如果值提供程序无法获取设置值,则会回退到下一个值提供程序.

有五个预构建设置值提供程序按以下顺序注册:

\* `DefaultValueSettingValueProvider`: 从设置定义的默认值中获取值(参见上面的SettingDefinition部分).

\* `ConfigurationSettingValueProvider`: 从[IConfiguration服务](Configuration.md)中获取值.

\* `GlobalSettingValueProvider`: 获取设置的全局(系统范围)值.

\* `TenantSettingValueProvider`: 获取当前租户的设置值(参阅 [多租户](Multi-Tenancy.md)文档).

\* `UserSettingValueProvider`: 获取当前用户的设置值(参阅 [当前用户](CurrentUser.md) 文档).

> 设置回退系统从底部 (用户) 到 顶部(默认) 方向起作用.

全局,租户和用户设置值提供程序使用 `ISettingStore` 从数据源读取值(参见下面的小节).

### 在应用程序配置中设置值

上一节提到 `ConfigurationSettingValueProvider` 从 `IConfiguration` 服务中读取设置, 该服务默认从 `appsettings.json` 中读取值. 所以在 `appsettings.json` 文件中配置设置值是最简单的方式.

例如你可以像以下方式一样配置 [IEmailSender](Emailing.md) 设置:

````json

{

"Settings": {

"Abp.Mailing.DefaultFromAddress": "noreply@mydomain.com",

"Abp.Mailing.DefaultFromDisplayName": "My Application",

"Abp.Mailing.Smtp.Host": "mail.mydomain.com",

"Abp.Mailing.Smtp.Port": "547",

"Abp.Mailing.Smtp.UserName": "myusername",

"Abp.Mailing.Smtp.Password": "mySecretPassW00rd",

"Abp.Mailing.Smtp.EnableSsl": "True"

}

}

````

设置值应该在 `Settings` 部分配置,如本例所示.

> `IConfiguration`是.NET Core的服务,它不仅可以从 `appsettings.json` 中读取值,还可以从环境,用户机密...等中读取值. 有关更多信息请参阅[微软文档](https://docs.microsoft.com/zh-cn/aspnet/core/fundamentals/configuration/).

### 自定义设置值提供程序

扩展设置系统的方式是定义一个派生自 `SettingValueProvider` 的类. 示例:

````csharp

public class CustomSettingValueProvider : SettingValueProvider

{

public override string Name => "Custom";

public CustomSettingValueProvider(ISettingStore settingStore)

: base(settingStore)

{

}

public override Task<string> GetOrNullAsync(SettingDefinition setting)

{

/\* Return the setting value or null

Use the SettingStore or another data source \*/

}

}

````

> 或者你直接可以实现 `ISettingValueProvider` 接口. 这时需要记得将其注册到 [依赖注入](Dependency-Injection.md).

每一个提供程序都应该具有唯一的名称 (这里的名称是 "Custom" ). 内置提供程序使用给定的名称:

\* `DefaultValueSettingValueProvider`: "**\*\*D\*\***".

\* `ConfigurationSettingValueProvider`: "**\*\*C\*\***".

\* `GlobalSettingValueProvider`: "**\*\*G\*\***".

\* `TenantSettingValueProvider`: "**\*\*T\*\***".

\* `UserSettingValueProvider`: "**\*\*U\*\***".

最好使用一个字母的名称来减少数据库中的数据大小(提供者名称在每行中重复).

定义自定义设置值提供程序后,需要将其显式注册到 `AbpSettingOptions`:

````csharp

Configure<AbpSettingOptions>(options =>

{

options.ValueProviders.Add<CustomSettingValueProvider>();

});

````

本示例将其添加到最后一项,因此它将成为`ISettingProvider`使用的第一个值提供程序. 你也可以将其添加到`options.ValueProviders`列表的另一个位置.

### ISettingStore

尽管设置值提供程序可以自由使用任何来源来获取设置值,但 `ISettingStore` 服务是设置值的默认来源. 全局,租户和用户设置值提供者都使用它.

## ISettingEncryptionService

`ISettingEncryptionService` 用于在设置定义的 `isencryption` 属性设置为 `true` 时加密/解密设置值.

你可以在依赖注入系统中替换此服务,自定义实现加密/解密过程. 默认实现 `StringEncryptionService` 使用AES算法(参见字符串[加密文档](String-Encryption.md)学习更多).

## 设置管理模块

设置系统核心是相当独立的,不做任何关于如何管理(更改)设置值的假设. 默认的`ISettingStore`实现也是`NullSettingStore`,它为所有设置值返回null.

设置管理模块通过管理数据库中的设置值来完成逻辑(实现`ISettingStore`).有关更多信息参阅[设置管理模块](Modules/Setting-Management.md)学习更多.

## 链接字符串

# 连接字符串

ABP框架的设计是[模块化](Module-Development-Basics.md), [微服务兼容](Microservice-Architecture.md) 和 [多租户](Multi-Tenancy.md). 同时设计了连接字符串管理来支持这些场景;

\* 允许为每个模块设置单独的连接字符串,这样每个模块都可以有自己的物理数据库. 甚至可以将模块配置为使用不同的DBMS.

\* 允许为每个租户设置单独的连接字符串使用单独的数据库(在SaaS应用程序中).

它还支持混合场景;

\* 允许将模块分组到数据库 (所有的模块分组到一个共享数据库, 两个模块使用数据库A, 三个模块使用数据库B, 一个模块使用数据库C其余的模块使用数据库D...等.)

\* 允许将租户分组到数据库中,像模块一样.

\* 允许为每个租户每个模块分离数据库 (数据库过多会增加维护成本,但ABP框架支持这种需求).

所有[预构建应用模块](Modules/Index.md)已设计为与以上场景兼容.

## 配置连接字符串

参见以下配置:

````json

"ConnectionStrings": {

"Default": "Server=localhost;Database=MyMainDb;Trusted\_Connection=True;",

"AbpIdentityServer": "Server=localhost;Database=MyIdsDb;Trusted\_Connection=True;",

"AbpPermissionManagement": "Server=localhost;Database=MyPermissionDb;Trusted\_Connection=True;"

}

````

> ABP使用 `IConfiguration` 服务获取应用程序配置. 虽然在 `appsettings.json` 文件中写入配置是最简单的方法, 但它不仅限于此文件. 你可以使用环境变量, user secrets, Azure Key Vault... 等. 更多信息参阅 [配置](Configuration.md) 文档.

以上配置定义了三个不同的连接字符串:

\* `MyMainDb` (`Default` 连接字符串)是应用程序的主连接字符串. 如果没有为模块指定连接字符串,则回退到 `Default` 连接字符串. [应用程序启动模板](Startup-Templates/Application.md) 配置为使用单个字符串, 所以所有的模块都使用单个数据库.

\* `MyIdsDb` 由 [IdentityServer](Modules/IdentityServer.md) 模块使用.

\* `MyPermissionDb` 由 [权限管理](Modules/Permission-Management.md) 模块使用.

[预构建的应用程序模块](Modules/Index.md) 为连接字符串名称定义常量. 例如IdentityServer模块在 `AbpIdentityServerDbProperties` 类(位于 `Volo.Abp.IdentityServer` 命名空间)定义了 `ConnectionStringName` 常量 . 其他的模块类似的定义常量,你可以查看连接字符串的名称.

### AbpDbConnectionOptions

ABP实际上使用 `AbpDbConnectionOptions` 获取连接字符串. 如果如上所述设置了连接字符串, `AbpDbConnectionOptions` 会被自动填充. 但是你也可以使用[选项模式](Options.md)设置或覆盖连接字符串. 你可以在[模块](Module-Development-Basics.md)的 `ConfigureServices` 方法配置`AbpDbConnectionOptions`）.

如下所示:

````csharp

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpDbConnectionOptions>(options =>

{

options.ConnectionStrings.Default = "...";

options.ConnectionStrings["AbpPermissionManagement"] = "...";

});

}

````

## 设置连接字符串名称

模块通常使用 `ConnectionStringName` attribute 为 `DbContext` 类关联一个唯一的连接字符串名称. 示例:

````csharp

[ConnectionStringName("AbpIdentityServer")]

public class IdentityServerDbContext

: AbpDbContext<IdentityServerDbContext>, IIdentityServerDbContext

{

}

````

对于 [Entity Framework Core](Entity-Framework-Core.md) 和 [MongoDB](MongoDB.md), 写入到 `DbContext` 类 (和接口,如果有的话).

> 如果你开发的是与数据库提供程序无关的可重用模块, 请参见 [最佳实践指南](Best-Practices/Index.md).

## Entity Framework Core的数据库迁移

关系数据库需要在使用数据库之前创建数据库和数据库架构 (表, 视图...等).

启动模板(使用 EF Core ORM) 带有一个数据库和一个 `.EntityFrameworkCore.DbMigrations` 项目,其中包含数据库的迁移文件. 该项目主要定义了一个*\*YourProjectName\**MigrationsDbContext,它调用所有模块的 `Configure...()` 方法,例如 `builder.ConfigurePermissionManagement()`.

一旦要分离模块的数据库,通常需要创建第二个迁移路径. 请参阅[EF Core迁移文档](Entity-Framework-Core-Migrations.md)了解如何为所需模块创建和使用其他数据库.

## 多租户

参阅 [多租户文档](Multi-Tenancy.md)了解如何为租户使用单独的数据库.

## 替换连接字符串解析器

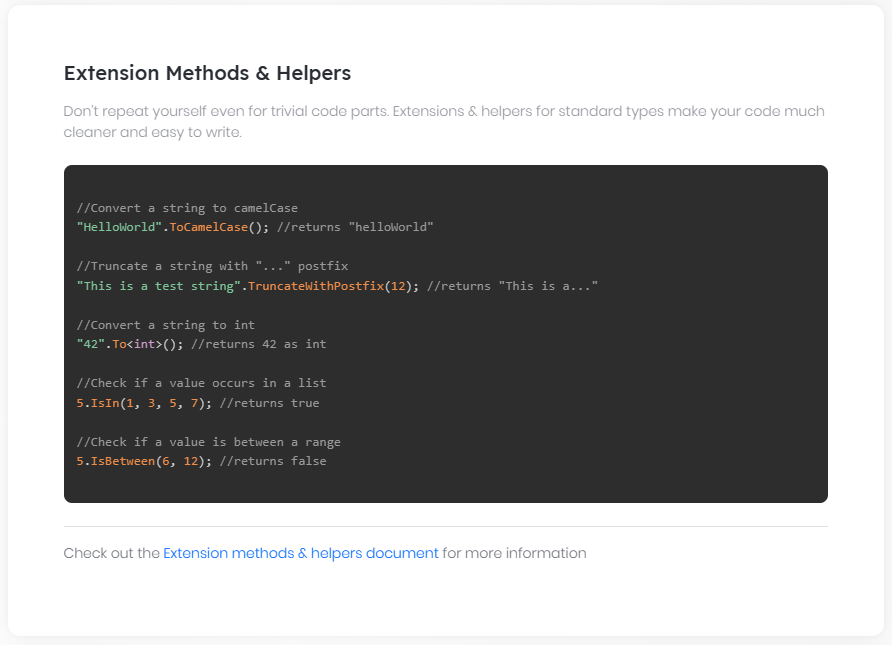
ABP定义了 `IConnectionStringResolver`,并在需要连接字符串时使用它. 有两个预构建的实现:

\* `DefaultConnectionStringResolver` 根据上面"配置连接字符串"一节中定义的规则,使用 `AbpDbConnectionOptions` 选择连接字符串.

\* `MultiTenantConnectionStringResolver` 用于多租户应用程序,并尝试获取当前租户的已配置连接字符串(如果有). 它使用 `ITenantStore` 查找连接字符串. 它继承了 `DefaultConnectionStringResolver`, 如果没有为当前租户指定连接字符串则回退到基本逻辑.

如果需要自定义逻辑来确定连接字符串,可以实现 `IConnectionStringResolver` 接口(也可以从现有类派生)并使用[依赖注入](Dependency-Injection.md)系统替换现有实现.

## 对象扩展



# 对象扩展

ABP框架提供了 **\*\*实体扩展系统\*\*** 允许你 **\*\*添加额外属性\*\*** 到已存在的对象 **\*\*无需修改相关类\*\***. 它允许你扩展[应用程序依赖模块](Modules/Index.md)实现的功能,尤其是当你要扩展[模块定义的实体](Customizing-Application-Modules-Extending-Entities.md)和[DTO](Customizing-Application-Modules-Overriding-Services.md)时.

> 你自己的对象通常不需要对象扩展系统,因为你可以轻松的添加常规属性到你的类中.

## IHasExtraProperties 接口

这是一个使类可扩展的接口. 它定义了 `Dictionary` 属性:

````csharp

Dictionary<string, object> ExtraProperties { get; }

````

然后你可以使用此字典添加或获取其他属性.

### 基类

默认以下基类实现了 `IHasExtraProperties` 接口:

\* 由 `AggregateRoot` 类实现 (参阅 [entities](Entities.md)).

\* 由 `ExtensibleEntityDto`, `ExtensibleAuditedEntityDto`... [DTO](Data-Transfer-Objects.md)基类实现.

\* 由 `ExtensibleObject` 实现, 它是一个简单的基类,任何类型的对象都可以继承.

如果你的类从这些类继承,那么你的类也是可扩展的,如果没有,你也可以随时手动继承.

### 基本扩展方法

虽然可以直接使用类的 `ExtraProperties` 属性,但建议使用以下扩展方法使用额外属性.

#### SetProperty

用于设置额外属性值:

````csharp

user.SetProperty("Title", "My Title");

user.SetProperty("IsSuperUser", true);

````

`SetProperty` 返回相同的对象, 你可以使用链式编程:

````csharp

user.SetProperty("Title", "My Title")

.SetProperty("IsSuperUser", true);

````

#### GetProperty

用于读取额外属性的值:

````csharp

var title = user.GetProperty<string>("Title");

if (user.GetProperty<bool>("IsSuperUser"))

{

//...

}

````

\* `GetProperty` 是一个泛型方法,对象类型做为泛型参数.

\* 如果未设置给定的属性,则返回默认值 (`int` 的默认值为 `0` , `bool` 的默认值是 `false` ... 等).

##### 非基本属性类型

如果你的属性类型不是原始类型(int,bool,枚举,字符串等),你需要使用 `GetProperty` 的非泛型版本,它会返回 `object`.

#### HasProperty

用于检查对象之前是否设置了属性.

#### RemoveProperty

用于从对象中删除属性. 使用此方法代替为属性设置 `null` 值.

### 一些最佳实践

为属性名称使用魔术字符串很危险,因为你很容易输入错误的属性名称-这并不安全;

\* 为你的额外属性名称定义一个常量.

\* 使用扩展方法轻松设置你的属性.

示例:

````csharp

public static class IdentityUserExtensions

{

private const string TitlePropertyName = "Title";

public static void SetTitle(this IdentityUser user, string title)

{

user.SetProperty(TitlePropertyName, title);

}

public static string GetTitle(this IdentityUser user)

{

return user.GetProperty<string>(TitlePropertyName);

}

}

````

然后, 你可以很容易地设置或获取 `Title` 属性:

````csharp

user.SetTitle("My Title");

var title = user.GetTitle();

````

## Object Extension Manager

你可以为可扩展对象(实现 `IHasExtraProperties`接口)设置任意属性, `ObjectExtensionManager` 用于显式定义可扩展类的其他属性.

显式定义额外的属性有一些用例:

\* 允许控制如何在对象到对象的映射上处理额外的属性 (参阅下面的部分).

\* 允许定义属性的元数据. 例如你可以在使用[EF Core](Entity-Framework-Core.md)时将额外的属性映射到数据库中的表字段.

> `ObjectExtensionManager` 实现单例模式 (`ObjectExtensionManager.Instance`) ,你应该在应用程序启动之前定义对象扩展. [应用程序启动模板](Startup-Templates/Application.md) 有一些预定义的静态类,可以安全在内部定义对象扩展.

### AddOrUpdate

`AddOrUpdate` 是定义对象额外属性或更新对象额外属性的主要方法.

示例: 为 `IdentityUser` 实体定义额外属性:

````csharp

ObjectExtensionManager.Instance

.AddOrUpdate<IdentityUser>(options =>

{

options.AddOrUpdateProperty<string>("SocialSecurityNumber");

options.AddOrUpdateProperty<bool>("IsSuperUser");

}

);

````

### AddOrUpdateProperty

虽然可以如上所示使用 `AddOrUpdateProperty`, 但如果要定义单个额外的属性,也可以使用快捷的扩展方法:

````csharp

ObjectExtensionManager.Instance

.AddOrUpdateProperty<IdentityUser, string>("SocialSecurityNumber");

````

有时将单个额外属性定义为多种类型是可行的. 你可以使用以下代码,而不是一个一个地定义:

````csharp

ObjectExtensionManager.Instance

.AddOrUpdateProperty<string>(

new[]

{

typeof(IdentityUserDto),

typeof(IdentityUserCreateDto),

typeof(IdentityUserUpdateDto)

},

"SocialSecurityNumber"

);

````

#### 属性配置

`AddOrUpdateProperty` 还可以为属性定义执行其他配置的操作.

Example:

````csharp

ObjectExtensionManager.Instance

.AddOrUpdateProperty<IdentityUser, string>(

"SocialSecurityNumber",

options =>

{

options.CheckPairDefinitionOnMapping = false;

});

````

> 参阅 "对象到对象映射" 部分了解 `CheckPairDefinitionOnMapping` 选项.

`options` 有一个名为 `Configuration` 的字典,该字典存储对象扩展定义甚至可以扩展. EF Core使用它来将其他属性映射到数据库中的表字段. 请参阅[扩展实体文档](Customizing-Application-Modules-Extending-Entities.md).

#### 默认值

自动为新属性设置默认值,默认值是属性类型的自然默认值,例如: `string`: `null` , `bool`: `false` 或 `int`: `0`.

有两种方法可以覆盖默认值:

##### DefaultValue 选项

`DefaultValue` 选项可以设置任何值:

````csharp

ObjectExtensionManager.Instance

.AddOrUpdateProperty<IdentityUser, int>(

"MyIntProperty",

options =>

{

options.DefaultValue = 42;

});

````

##### DefaultValueFactory 选项

`DefaultValueFactory` 可以设置返回默认值的函数:

````csharp

ObjectExtensionManager.Instance

.AddOrUpdateProperty<IdentityUser, DateTime>(

"MyDateTimeProperty",

options =>

{

options.DefaultValueFactory = () => DateTime.Now;

});

````

`options.DefaultValueFactory` 比 `options.DefaultValue` 优先级要高.

> 提示: 只有在默认值可能发生变化时(如示例中的`DateTime.Now;`) 才使用 `DefaultValueFactory`,如果是一个常量请使用 `DefaultValue` 选项.

#### CheckPairDefinitionOnMapping

控制在映射两个可扩展对象时如何检查属性定义. 请参阅*\*对象到对象映射\**部分,了解 `CheckPairDefinitionOnMapping` 选项.

## Validation

你可能要为你定义的额外属性添加一些 **\*\*验证规则\*\***. `AddOrUpdateProperty` 方法选项允许进行验证的方法有两种:

1. 你可以为属性添加 **\*\*数据注解 attributes\*\***.

2. 你可以给定一个action(代码块)执行 **\*\*自定义验证\*\***.

当你在**\*\*自动验证\*\***的方法(例如:控制器操作,页面处理程序方法,应用程序服务方法...)中使用对象时,验证会工作. 因此,每当扩展对象被验证时,所有额外的属性都会被验证.

### 数据注解 Attributes

所有标准的数据注解Attributes对于额外属性都是有效的. 例:

````csharp

ObjectExtensionManager.Instance

.AddOrUpdateProperty<IdentityUserCreateDto, string>(

"SocialSecurityNumber",

options =>

{

options.Attributes.Add(new RequiredAttribute());

options.Attributes.Add(

new StringLengthAttribute(32) {

MinimumLength = 6

}

);

});

````

使用以上配置,如果没有提供有效的 `SocialSecurityNumber` 值, `IdentityUserCreateDto` 对象将是无效的.

### 自定义验证

如果需要,可以添加一个自定义action验证额外属性. 例:

````csharp

ObjectExtensionManager.Instance

.AddOrUpdateProperty<IdentityUserCreateDto, string>(

"SocialSecurityNumber",

options =>

{

options.Validators.Add(context =>

{

var socialSecurityNumber = context.Value as string;

if (socialSecurityNumber == null ||

socialSecurityNumber.StartsWith("X"))

{

context.ValidationErrors.Add(

new ValidationResult(

"Invalid social security number: " + socialSecurityNumber,

new[] { "SocialSecurityNumber" }

)

);

}

});

});

````

`context.ServiceProvider` 可以解析服务.

除了为单个属性添加自定义验证逻辑外,还可以添加在对象级执行的自定义验证逻辑. 例:

````csharp

ObjectExtensionManager.Instance

.AddOrUpdate<IdentityUserCreateDto>(objConfig =>

{

//Define two properties with their own validation rules

objConfig.AddOrUpdateProperty<string>("Password", propertyConfig =>

{

propertyConfig.Attributes.Add(new RequiredAttribute());

});

objConfig.AddOrUpdateProperty<string>("PasswordRepeat", propertyConfig =>

{

propertyConfig.Attributes.Add(new RequiredAttribute());

});

//Write a common validation logic works on multiple properties

objConfig.Validators.Add(context =>

{

if (context.ValidatingObject.GetProperty<string>("Password") !=

context.ValidatingObject.GetProperty<string>("PasswordRepeat"))

{

context.ValidationErrors.Add(

new ValidationResult(

"Please repeat the same password!",

new[] { "Password", "PasswordRepeat" }

)

);

}

});

});

````

## 对象到对象映射

假设你已向可扩展的实体对象添加了额外的属性并使用了自动[对象到对象的映射](Object-To-Object-Mapping.md)将该实体映射到可扩展的DTO类. 在这种情况下你需要格外小心,因为额外属性可能包含**\*\*敏感数据\*\***,这些数据对于客户端不可用.

本节提供了一些**\*\*好的做法\*\***,可以控制对象映射的额外属性.

### MapExtraPropertiesTo

`MapExtraPropertiesTo` 是ABP框架提供的扩展方法,用于以受控方式将额外的属性从一个对象复制到另一个对象. 示例:

````csharp

identityUser.MapExtraPropertiesTo(identityUserDto);

````

`MapExtraPropertiesTo` 需要在**\*\*两侧\*\***(本例中是`IdentityUser` 和 `IdentityUserDto`)**\*\*定义属性\*\***. 以将值复制到目标对象. 否则即使源对象(在此示例中为 `identityUser` )中确实存在该值,它也不会复制. 有一些重载此限制的方法.

#### MappingPropertyDefinitionChecks

`MapExtraPropertiesTo` 获取一个附加参数来控制单个映射操作的定义检查:

````csharp

identityUser.MapExtraPropertiesTo(

identityUserDto,

MappingPropertyDefinitionChecks.None

);

````

> 要小心,因为 `MappingPropertyDefinitionChecks.None` 会复制所有的额外属性而不进行任何检查. `MappingPropertyDefinitionChecks` 枚举还有其他成员.

如果要完全禁用属性的定义检查,可以在定义额外的属性(或更新现有定义)时进行,如下所示:

````csharp

ObjectExtensionManager.Instance

.AddOrUpdateProperty<IdentityUser, string>(

"SocialSecurityNumber",

options =>

{

options.CheckPairDefinitionOnMapping = false;

});

````

#### 忽略属性

你可能要在映射操作忽略某些属性:

````csharp

identityUser.MapExtraPropertiesTo(

identityUserDto,

ignoredProperties: new[] {"MySensitiveProp"}

);

````

忽略的属性不会复制到目标对象.

#### AutoMapper集成

如果你使用的是[AutoMapper](https://automapper.org/)库,ABP框架还提供了一种扩展方法来利用上面定义的 `MapExtraPropertiesTo` 方法.

你可以在映射配置文件中使用 `MapExtraProperties()` 方法.

````csharp

public class MyProfile : Profile

{

public MyProfile()

{

CreateMap<IdentityUser, IdentityUserDto>()

.MapExtraProperties();

}

}

````

它与 `MapExtraPropertiesTo()` 方法具有相同的参数.

## Entity Framework Core 数据库映射

如果你使用的是EF Core,可以将额外的属性映射到数据库中的表字段. 例:

````csharp

ObjectExtensionManager.Instance

.AddOrUpdateProperty<IdentityUser, string>(

"SocialSecurityNumber",

options =>

{

options.MapEfCore(b => b.HasMaxLength(32));

}

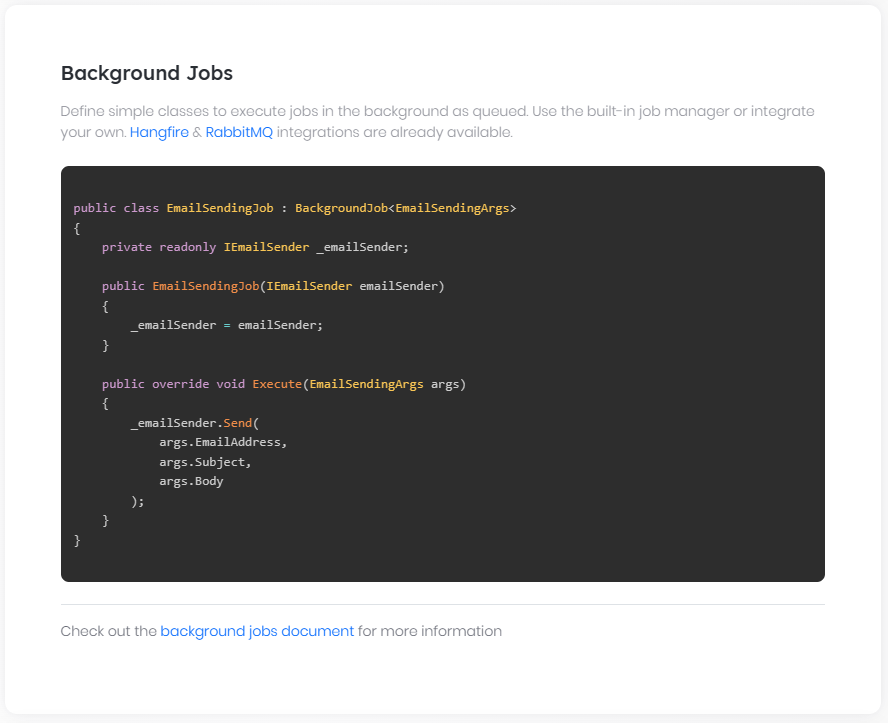
);

````

参阅 [Entity Framework Core 集成文档](Entity-Framework-Core.md) 了解更多内容.

# 基础设施

## 后台服务



后台作业

# 后台作业

## 介绍

后台作业用来在后台里执行应用里的一些任务, 出于几个原因, 你可能需要后台工作, 以下是一些例子:

- 为执行**\*\*长时间运行的任务\*\***而用户无需等待, 例如:用户按了一下"报告"按钮开始一个长时间运行的报告任务, 你把这个任务添加到**\*\*队列\*\***里,并在完成后通过电子邮件将报告的结果发送给你的用户.

- 创建**\*\*可重试\*\***和**\*\*持久的任务\*\***以**\*\*确保\*\***代码将**\*\*成功执行\*\***. 例如, 你可以在后台作业中发送电子邮件以克服**\*\*临时故障\*\***并**\*\*保证\*\***最终发送. 这样用户不需要在发送电子邮件时等待.

后台作业是**\*\*持久性的\*\***这意味着即使你的应用程序崩溃了, 后台作业也会在稍后**\*\*重试\*\***并**\*\*执行\*\***.

ABP为后台作业提供了一个**\*\*抽象\*\***模块和几个后台作业**\*\*实现\*\***. 它具有内置/默认的实现以及与Hangfire和RabbitMQ的集成.

## 抽象模块

ABP为后台作业提供了一个 **\*\*抽象\*\*** 模块和 **\*\*多个实现\*\***. 它有一个内置/默认实现以及Hangfire,RabbitMQ与Quartz集成.

`Volo.Abp.BackgroundJobs.Abstractions` nuget package 提供了创建后台作业和队列作业所需要的服务. 如果你的模块只依赖这个包,那么它可以独立于其实现/集成.

> `Volo.Abp.BackgroundJobs.Abstractions` package 默认在启动模板中已经安装.

### 创建后台作业

后台作业是一个实现`IBackgroundJob<TArgs>`接口或继承自`BackgroundJob<TArgs>`类的类.`TArgs`是一个简单的C#类, 用于存储作业数据.

在示例中使用后台作业发送电子邮件,首先定义一个类来存储后台作业的参数

````csharp

public class EmailSendingArgs

{

public string EmailAddress { get; set; }

public string Subject { get; set; }

public string Body { get; set; }

}

````

然后创建后台作业类,它使用 `EmailSendingArgs` 对象发送电子邮件:

````csharp

using Volo.Abp.BackgroundJobs;

using Volo.Abp.Emailing;

namespace MyProject

{

public class EmailSendingJob : BackgroundJob<EmailSendingArgs>, ITransientDependency

{

private readonly IEmailSender \_emailSender;

public EmailSendingJob(IEmailSender emailSender)

{

\_emailSender = emailSender;

}

public override void Execute(EmailSendingArgs args)

{

\_emailSender.Send(

args.EmailAddress,

args.Subject,

args.Body

);

}

}

}

````

这个作业简单的使用了 `IEmailSender` 发送电子邮件 (请参阅 [邮件发送文档](Emailing.md)).

#### 异常处理

后台作业不应该隐藏异常. 如果它抛出一个异常, 在稍后后台作业将会自动重试. 只有在你不想为当前参数重新运行后台作业时才隐藏异常.

### 队列作业

现在, 你可以使用 `IBackgroundJobManager` 服务向队列中添加一个发送电子邮件作业:

````csharp

public class RegistrationService : ApplicationService

{

private readonly IBackgroundJobManager \_backgroundJobManager;

public RegistrationService(IBackgroundJobManager backgroundJobManager)

{

\_backgroundJobManager = backgroundJobManager;

}

public async Task RegisterAsync(string userName, string emailAddress, string password)

{

//TODO: 创建一个新用户到数据库中...

await \_backgroundJobManager.EnqueueAsync(

new EmailSendingArgs

{

EmailAddress = emailAddress,

Subject = "You've successfully registered!",

Body = "..."

}

);

}

}

````

刚才我们注入 `IBackgroundJobManager` 服务了并且使用它的 `EnqueueAsync` 方法添加一个新的作业到队列中.

Enqueue方法接收一些可选参数用于控制后台作业:

\* **\*\*priority\*\*** 用于控制作业项的优先级. 它接收一个 `BackgroundJobPriority` 类型的枚举,它有 `Low`, `BelowNormal`, `Normal` (默认), `AboveNormal` 和 `Hight` 字段.

\* **\*\*delay\*\*** 用于作业第一次重试之前的等待时间 (`TimeSpan`)类型.

### 禁用作业执行

你可能希望在你的应用程序中禁用后台作业执行. 如果你希望在另一个进程中执行后台作业并在当前进程中禁用它,通常可以使用以下命令.

使用 `AbpBackgroundJobOptions` 配置作业执行:

````csharp

[DependsOn(typeof(AbpBackgroundJobsModule))]

public class MyModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpBackgroundJobOptions>(options =>

{

options.IsJobExecutionEnabled = false; //禁用作业执行

});

}

}

````

> 默认后台管理器(见下文)不支持多进程执行相同的作业队列. 所以, 如果你的应用程序中有多个正在运行的实现,并且使用的是默认的后台管理器, 你应该只在一个应用程序实例进程中启用作业队列.

## 默认后台作业管理器

ABP framework 包含一个简单的 `IBackgroundJobManager` 实现;

- 在**\*\*单线程\*\***中**\*\*FIFO(先入先出)\*\***.

- **\*\*重试\*\***作业执行直到作业**\*\*执行成功\*\***或**\*\*超时\*\***. 默认作业超时时间是2天. 记录所有异常 .

- 作业执行成功时从存储中(数据库)**\*\*删除\*\***作业. 如果超时, 作业会在数据库中被设置为**\*\*abandoned\*\***.

- 作业的**\*\*重试等待时间会越来越长\*\***. 作业第一次重试等待1分钟, 第二次重试等待2分钟, 第三次重试等待4分钟,以此类推.

- 以固定的时间间隔轮询存储中的作业. 查询作业, 按优先级排序(asc)然后按尝试次数排序(asc).

> `Volo.Abp.BackgroundJobs` nuget package 包含默认的后台作业管理器并且在默认在启动模板中已经安装.

### 配置

在你的[模块类](Module-Development-Basics.md)中使用 `AbpBackgroundJobWorkerOptions` 配置默认作业管理器.

示例中更改后台作业的的超时时间:

````csharp

[DependsOn(typeof(AbpBackgroundJobsModule))]

public class MyModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpBackgroundJobWorkerOptions>(options =>

{

options.DefaultTimeout = 864000; //10 days (as seconds)

});

}

}

````

### 数据存储

默认的后台作业管理器需要数据存储用来保存和读取作业. 它将 `IBackgroundJobStore` 定义为抽象的. 所以, 如果你想要的话你可以替换它的实现.

后台作业模块使用各种数据访问提供程序实现 `IBackgroundJobStore`. 参阅 [后台工作模块文档](Modules/Background-Jobs.md).

> 默认情况下,后台作业模块已经安装到启动模板中,它基于你的ORM/数据访问选项.

## 集成

后台作业系统是可扩展的,你可以使用自己的实现或预先构建的集成更改默认后台作业管理器.

请参阅预构建的作业管理器备选方案:

\* [Hangfire 后台作业管理器](Background-Jobs-Hangfire.md)

\* [RabbitMQ 后台作业管理器](Background-Jobs-RabbitMq.md)

\* [Quartz 后台作业管理器](Background-Jobs-Quartz.md)

## 另请参阅

\* [后台工作者](Background-Workers.md)

Hangfire 集成

# Hangfire后台作业管理

[Hangfire](https://www.hangfire.io/)是一个高级的后台作业管理. 你可以用ABP框架集成Hangfire代替[默认后台作业管理](Background-Jobs.md). 通过这种方式你可以使用相同的后台作业API,将你的代码独立于Hangfire. 如果你喜欢也可以直接使用Hangfire的API.

> 参阅[后台作业文档](Background-Jobs.md),学习如何使用后台作业系统. 本文只介绍了如何安装和配置Hangfire集成.

## 安装

建议使用[ABP CLI](CLI.md)安装包.

### 使用ABP CLI

在项目的文件夹(.csproj文件)中打开命令行窗口输入以下命令:

````bash

abp add-package Volo.Abp.BackgroundJobs.HangFire

````

### 手动安装

如果你想手动安装;

1. 添加 [Volo.Abp.BackgroundJobs.HangFire](https://www.nuget.org/packages/Volo.Abp.BackgroundJobs.HangFire) NuGet包添加到你的项目:

````

Install-Package Volo.Abp.BackgroundJobs.HangFire

````

2. 添加 `AbpBackgroundJobsHangfireModule` 到你的模块的依赖列表:

````csharp

[DependsOn(

//...other dependencies

typeof(AbpBackgroundJobsHangfireModule) //Add the new module dependency

)]

public class YourModule : AbpModule

{

}

````

## 配置

你可以安装任何Hangfire存储. 最常用的是SQL Server(参阅[Hangfire.SqlServer](https://www.nuget.org/packages/Hangfire.SqlServer)NuGet包).

当你安装NuGet包后,你需要为你的项目配置Hangfire.

1.首先, 我们需要更改 `Module` 类 (例如: `<YourProjectName>HttpApiHostModule`) 的 `ConfigureServices` 方法去配置Hangfire存储和连接字符串:

````csharp

public override void ConfigureServices(ServiceConfigurationContext context)

{

var configuration = context.Services.GetConfiguration();

var hostingEnvironment = context.Services.GetHostingEnvironment();

//... other configarations.

ConfigureHangfire(context, configuration);

}

private void ConfigureHangfire(ServiceConfigurationContext context, IConfiguration configuration)

{

context.Services.AddHangfire(config =>

{

config.UseSqlServerStorage(configuration.GetConnectionString("Default"));

});

}

````

> 你必须为Hangfire配置一个存储

### 指定队列

你可以使用 [`QueueAttribute`](https://docs.hangfire.io/en/latest/background-processing/configuring-queues.html) 来指定队列.

````csharp

using System.Threading.Tasks;

using Volo.Abp.BackgroundJobs;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Emailing;

namespace MyProject

{

[Queue("alpha")]

public class EmailSendingJob

: AsyncBackgroundJob<EmailSendingArgs>, ITransientDependency

{

private readonly IEmailSender \_emailSender;

public EmailSendingJob(IEmailSender emailSender)

{

\_emailSender = emailSender;

}

public override async Task ExecuteAsync(EmailSendingArgs args)

{

await \_emailSender.SendAsync(

args.EmailAddress,

args.Subject,

args.Body

);

}

}

}

````

1. 如果你想要使用Hangfire的面板,你可以在 `Module` 类的 `OnApplicationInitialization` 方法添加: `UseHangfireDashboard`

````csharp

public override void OnApplicationInitialization(ApplicationInitializationContext context)

{

var app = context.GetApplicationBuilder();

// ... others

app.UseHangfireDashboard();

}

````

RabbitMQ 集成

# RabbitMQ 后台作业管理

RabbitMQ 是一个标准的消息队列中间件,虽然它常用于消息传递/分布式事件,但也非常适合存储 FIFO(先进先出) 顺序的后台作业.

ABP Framework 提供了 [Volo.Abp.BackgroundJobs.RabbitMQ](https://www.nuget.org/packages/Volo.Abp.BackgroundJobs.RabbitMQ) 包,将使用 RabbitMQ 来执行后台作业.

> 参阅 [后台作业文档](Background-Jobs.md) 学习如何使用后台作业系统,本文只介绍了如何安装和配置 RabbitMQ 集成.

## 安装

使用 ABP CLI 将 [Volo.Abp.BackgroundJobs.RabbitMQ](https://www.nuget.org/packages/Volo.Abp.BackgroundJobs.RabbitMQ) 包添加到你的项目:

- 如果之前没有安装过 [ABP CLI](https://docs.abp.io/en/abp/latest/CLI),请先安装它.

- 跳转到待安装后台作业管理的项目目录中(包含 `.csproj` 文件的目录),打开终端管理器.

- 执行 `abp add-package Volo.Abp.BackgroundJobs.RabbitMQ` 命令.

如果你想要手动安装,请先用 NuGet 包管理器安装 [Volo.Abp.BackgroundJobs.RabbitMQ](https://www.nuget.org/packages/Volo.Abp.BackgroundJobs.RabbitMQ) 包到指定项目,之后使在你的 [模块](Module-Development-Basics.md) 上面添加 `[DependsOn(typeof(AbpBackgroundJobsRabbitMqModule))]` 配置依赖.

## 配置

### 默认配置

默认配置将会使用标准端口和主机名(localhost)连接到 RabbitMQ 服务,**\*\*你不需要进行额外配置\*\***.

### RabbitMQ 连接

你可以使用 ASP.NET Core 的 [标准配置系统](Configuration.md) 对 RabbitMQ 进行详细配置,比如 `appsettings.json` 或者是 [选项类](Options.md).

#### 通过 `appsettings.json` 文件配置

这种方式是配置 RabbitMQ 连接最简单的方式,你可以使用其他的配置源(例如环境变量).这些强大的功能都是由 [ASP.NET Core](https://docs.microsoft.com/zh-cn/aspnet/core/fundamentals/configuration/) 提供的支持.

**\*\*示例: 配置默认的 RabbitMQ 连接\*\***

```json

{

"RabbitMQ": {

"Connections": {

"Default": {

"HostName": "123.123.123.123",

"Port": "5672"

}

}

}

}

```

你可以在配置文件使用所有 [ConnectionFactry](http://rabbitmq.github.io/rabbitmq-dotnet-client/api/RabbitMQ.Client.ConnectionFactory.html#properties) 的属性,关于这些属性的具体含义,可以查看 RabbitMQ 的 [官方文档](https://www.rabbitmq.com/dotnet-api-guide.html#exchanges-and-queues).

目前我们允许定义多个连接,多连接的情况适用于不同的后台作业,具体配置信息可以参考下面的 RabbitMQ 后台作业配置说明.

**\*\*示例: 定义两个 RabbitMQ 连接\*\***

```json

{

"RabbitMQ": {

"Connections": {

"Default": {

"HostName": "123.123.123.123"

},

"SecondConnection": {

"HostName": "321.321.321.321"

}

}

}

}

```

如果需要连接到 RabbitMQ 集群,你可以指定多个 HostName.

**\*\*示例: 连接到 RabbitMQ 集群\*\***

```json

{

"RabbitMQ": {

"Connections": {

"Default": {

"HostName": "123.123.123.123;234.234.234.234"

}

},

"EventBus": {

"ClientName": "MyClientName",

"ExchangeName": "MyExchangeName"

}

}

}

```

#### 使用选项类

`AbpRabbitMqOptions` 类型用于配置 RabbitMQ 的连接字符串,你可以在 [模块](Module-Development-Basics.md) 的 `ConfigureService` 方法中进行配置.

**\*\*示例: 配置 RabbitMQ 连接\*\***

```csharp

Configure<AbpRabbitMqOptions>(options =>

{

options.Connections.Default.UserName = "user";

options.Connections.Default.Password = "pass";

options.Connections.Default.HostName = "123.123.123.123";

options.Connections.Default.Port = 5672;

});

```

关于选项类,可以结合 `appsettings.json` 文件一起使用.针对同一个属性,在选项类里面对该值进行了设定,会覆盖掉 `appsettings.json` 的值.

### RabbitMQ 后台作业配置说明

#### 后台作业队列的名称

默认情况下,每个后台作业都会使用一个单独的队列,结合标准前缀和作业名称来构造一个完整的队列名称.默认的前缀为 `AbpBackgroundJobs`,所以有一个作业的名称是 `EmailSending` 的话,在 RabbitMQ 的队列名称就是 `AbpBackgroundJobs.EmailSending`.

> 在后台作业的参数类上,可以使用 `BackgroundJobName` 特性指定后台作业的名称.否则的话,后台作业的名称将会是后台作业类的全名(也包含命名空间).

#### 后台作业使用的连接

默认情况下,后台作业都会使用 `Default` 作为默认连接.

#### 自定义

`AbpRabbitMqBackgroundJobOptions` 可以自定义队列名和作业使用的 RabbitMQ 连接.

\*\*示例: \*\*

```csharp

Configure<AbpRabbitMqBackgroundJobOptions>(options =>

{

options.DefaultQueueNamePrefix = "my\_app\_jobs.";

options.DefaultDelayedQueueNamePrefix = "my\_app\_jobs.delayed"

options.PrefetchCount = 1;

options.JobQueues[typeof(EmailSendingArgs)] =

new JobQueueConfiguration(

typeof(EmailSendingArgs),

queueName: "my\_app\_jobs.emails",

connectionName: "SecondConnection",

delayedQueueName:"my\_app\_jobs.emails.delayed"

);

});

```

- 这个示例将默认的队列名前缀设置为 `my\_app\_jobs.`并且设置默认的延迟队列名为 `my\_app\_jobs.delayed`,如果多个项目都使用的同一个 RabbitMQ 服务,设置不同的前缀可以避免执行其他项目的后台作业.

- 设置了预取数量, 用于所有队列.

- 这里还设置了 `EmailSendingArgs` 绑定的 RabbitMQ 连接.

`JobQueueConfiguration` 类的构造函数中,还有一些其他的可选参数.

- `queueName`: 指定后台作业对应的队列名称(全名).

\* `DelayedQueueName`: 指定后台延迟执行的作业对于的队列名称(全名).

- `connectionName`: 后台作业对应的 RabbitMQ 连接名称,默认是 `Default`.

- `durable`: 可选参数,默认为 `true`.

- `exclusive`: 可选参数,默认为 `false`.

- `autoDelete`: 可选参数,默认为 `false`.

\* `PrefetchCount` (可选参数, 默认为: null)

如果你想要更多地了解 `durable`,`exclusive`,`autoDelete` 的用法,请阅读 RabbitMQ 提供的文档.

## 另请参阅

- [后台作业](Background-Jobs.md)

Quartz 集成

# Quartz 后台作业管理

[Quartz](https://www.quartz-scheduler.net/)是一个高级的作业管理. 你可以用ABP框架集成Quartz代替[默认后台作业管理](Background-Jobs.md). 通过这种方式你可以使用相同的后台作业API,将你的代码独立于Quartz. 如果你喜欢也可以直接使用Quartz的API.

> 参阅[后台作业文档](Background-Jobs.md),学习如何使用后台作业系统. 本文只介绍了如何安装和配置Quartz集成.

## 安装

建议使用[ABP CLI](CLI.md)安装包.

### 使用ABP CLI

在项目的文件夹(.csproj文件)中打开命令行窗口输入以下命令:

````bash

abp add-package Volo.Abp.BackgroundJobs.Quartz

````

### 手动安装

如果你想手动安装;

1. 添加 [Volo.Abp.BackgroundJobs.Quartz](https://www.nuget.org/packages/Volo.Abp.BackgroundJobs.Quartz) NuGet包添加到你的项目:

````

Install-Package Volo.Abp.BackgroundJobs.Quartz

````

2. 添加 `AbpBackgroundJobsQuartzModule` 到你的模块的依赖列表:

````csharp

[DependsOn(

//...other dependencies

typeof(AbpBackgroundJobsQuartzModule) //Add the new module dependency

)]

public class YourModule : AbpModule

{

}

````

## 配置

Quartz是一个可配置的类库,对此ABP框架提供了 `AbpQuartzOptions`. 你可以在模块预配置此选项,ABP在初始化Quartz模块时将使用它. 例:

````csharp

[DependsOn(

//...other dependencies

typeof(AbpBackgroundJobsQuartzModule) //Add the new module dependency

)]

public class YourModule : AbpModule

{

public override void PreConfigureServices(ServiceConfigurationContext context)

{

var configuration = context.Services.GetConfiguration();

PreConfigure<AbpQuartzOptions>(options =>

{

options.Properties = new NameValueCollection

{

["quartz.jobStore.dataSource"] = "BackgroundJobsDemoApp",

["quartz.jobStore.type"] = "Quartz.Impl.AdoJobStore.JobStoreTX, Quartz",

["quartz.jobStore.tablePrefix"] = "QRTZ\_",

["quartz.serializer.type"] = "json",

["quartz.dataSource.BackgroundJobsDemoApp.connectionString"] = configuration.GetConnectionString("Quartz"),

["quartz.dataSource.BackgroundJobsDemoApp.provider"] = "SqlServer",

["quartz.jobStore.driverDelegateType"] = "Quartz.Impl.AdoJobStore.SqlServerDelegate, Quartz",

};

});

}

}

````

从ABP3.1版本开始,我们在 `AbpQuartzOptions` 添加了 `Configurator` 用于配置Quartz. 例:

````csharp

[DependsOn(

//...other dependencies

typeof(AbpBackgroundJobsQuartzModule) //Add the new module dependency

)]

public class YourModule : AbpModule

{

public override void PreConfigureServices(ServiceConfigurationContext context)

{

var configuration = context.Services.GetConfiguration();

PreConfigure<AbpQuartzOptions>(options =>

{

options.Configurator = configure =>

{

configure.UsePersistentStore(storeOptions =>

{

storeOptions.UseProperties = true;

storeOptions.UseJsonSerializer();

storeOptions.UseSqlServer(configuration.GetConnectionString("Quartz"));

storeOptions.UseClustering(c =>

{

c.CheckinMisfireThreshold = TimeSpan.FromSeconds(20);

c.CheckinInterval = TimeSpan.FromSeconds(10);

});

});

};

});

}

}

````

> 你可以选择你喜爱的方式来配置Quaratz.

Quartz**\*\*默认\*\***将作业与调度信息存储在**\*\*内存\*\***中,示例中我们使用[选项模式](Options.md)的预配置将其更改为存储到数据库中. 有关Quartz的更多配置请参阅[Quartz文档](https://www.quartz-scheduler.net/).

## 异常处理

### 默认异常处理策略

当后台作业发生异常时ABP提供了**\*\*默认\*\***异常处理策略,它会为你的作业重试3次,每次间隔3秒. 你可以通过 `AbpBackgroundJobQuartzOptions` 更改默认重试次数与间隔时间:

```csharp

[DependsOn(

//...other dependencies

typeof(AbpBackgroundJobsQuartzModule) //Add the new module dependency

)]

public class YourModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpBackgroundJobQuartzOptions>(options =>

{

options.RetryCount = 1;

options.RetryIntervalMillisecond = 1000;

});

}

}

```

### 自定义异常处理策略

你可以通过 `AbpBackgroundJobQuartzOptions` 选项自定义异常处理策略:

```csharp

[DependsOn(

//...other dependencies

typeof(AbpBackgroundJobsQuartzModule) //Add the new module dependency

)]

public class YourModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpBackgroundJobQuartzOptions>(options =>

{

options.RetryStrategy = async (retryIndex, executionContext, exception) =>

{

// customize exception handling

};

});

}

}

```

# RabbitMQ 后台作业管理

RabbitMQ 是一个标准的消息队列中间件,虽然它常用于消息传递/分布式事件,但也非常适合存储 FIFO(先进先出) 顺序的后台作业.

ABP Framework 提供了 [Volo.Abp.BackgroundJobs.RabbitMQ](https://www.nuget.org/packages/Volo.Abp.BackgroundJobs.RabbitMQ) 包,将使用 RabbitMQ 来执行后台作业.

> 参阅 [后台作业文档](Background-Jobs.md) 学习如何使用后台作业系统,本文只介绍了如何安装和配置 RabbitMQ 集成.

## 安装

使用 ABP CLI 将 [Volo.Abp.BackgroundJobs.RabbitMQ](https://www.nuget.org/packages/Volo.Abp.BackgroundJobs.RabbitMQ) 包添加到你的项目:

- 如果之前没有安装过 [ABP CLI](https://docs.abp.io/en/abp/latest/CLI),请先安装它.

- 跳转到待安装后台作业管理的项目目录中(包含 `.csproj` 文件的目录),打开终端管理器.

- 执行 `abp add-package Volo.Abp.BackgroundJobs.RabbitMQ` 命令.

如果你想要手动安装,请先用 NuGet 包管理器安装 [Volo.Abp.BackgroundJobs.RabbitMQ](https://www.nuget.org/packages/Volo.Abp.BackgroundJobs.RabbitMQ) 包到指定项目,之后使在你的 [模块](Module-Development-Basics.md) 上面添加 `[DependsOn(typeof(AbpBackgroundJobsRabbitMqModule))]` 配置依赖.

## 配置

### 默认配置

默认配置将会使用标准端口和主机名(localhost)连接到 RabbitMQ 服务,**\*\*你不需要进行额外配置\*\***.

### RabbitMQ 连接

你可以使用 ASP.NET Core 的 [标准配置系统](Configuration.md) 对 RabbitMQ 进行详细配置,比如 `appsettings.json` 或者是 [选项类](Options.md).

#### 通过 `appsettings.json` 文件配置

这种方式是配置 RabbitMQ 连接最简单的方式,你可以使用其他的配置源(例如环境变量).这些强大的功能都是由 [ASP.NET Core](https://docs.microsoft.com/zh-cn/aspnet/core/fundamentals/configuration/) 提供的支持.

**\*\*示例: 配置默认的 RabbitMQ 连接\*\***

```json

{

"RabbitMQ": {

"Connections": {

"Default": {

"HostName": "123.123.123.123",

"Port": "5672"

}

}

}

}

```

你可以在配置文件使用所有 [ConnectionFactry](http://rabbitmq.github.io/rabbitmq-dotnet-client/api/RabbitMQ.Client.ConnectionFactory.html#properties) 的属性,关于这些属性的具体含义,可以查看 RabbitMQ 的 [官方文档](https://www.rabbitmq.com/dotnet-api-guide.html#exchanges-and-queues).

目前我们允许定义多个连接,多连接的情况适用于不同的后台作业,具体配置信息可以参考下面的 RabbitMQ 后台作业配置说明.

**\*\*示例: 定义两个 RabbitMQ 连接\*\***

```json

{

"RabbitMQ": {

"Connections": {

"Default": {

"HostName": "123.123.123.123"

},

"SecondConnection": {

"HostName": "321.321.321.321"

}

}

}

}

```

如果需要连接到 RabbitMQ 集群,你可以指定多个 HostName.

**\*\*示例: 连接到 RabbitMQ 集群\*\***

```json

{

"RabbitMQ": {

"Connections": {

"Default": {

"HostName": "123.123.123.123;234.234.234.234"

}

},

"EventBus": {

"ClientName": "MyClientName",

"ExchangeName": "MyExchangeName"

}

}

}

```

#### 使用选项类

`AbpRabbitMqOptions` 类型用于配置 RabbitMQ 的连接字符串,你可以在 [模块](Module-Development-Basics.md) 的 `ConfigureService` 方法中进行配置.

**\*\*示例: 配置 RabbitMQ 连接\*\***

```csharp

Configure<AbpRabbitMqOptions>(options =>

{

options.Connections.Default.UserName = "user";

options.Connections.Default.Password = "pass";

options.Connections.Default.HostName = "123.123.123.123";

options.Connections.Default.Port = 5672;

});

```

关于选项类,可以结合 `appsettings.json` 文件一起使用.针对同一个属性,在选项类里面对该值进行了设定,会覆盖掉 `appsettings.json` 的值.

### RabbitMQ 后台作业配置说明

#### 后台作业队列的名称

默认情况下,每个后台作业都会使用一个单独的队列,结合标准前缀和作业名称来构造一个完整的队列名称.默认的前缀为 `AbpBackgroundJobs`,所以有一个作业的名称是 `EmailSending` 的话,在 RabbitMQ 的队列名称就是 `AbpBackgroundJobs.EmailSending`.

> 在后台作业的参数类上,可以使用 `BackgroundJobName` 特性指定后台作业的名称.否则的话,后台作业的名称将会是后台作业类的全名(也包含命名空间).

#### 后台作业使用的连接

默认情况下,后台作业都会使用 `Default` 作为默认连接.

#### 自定义

`AbpRabbitMqBackgroundJobOptions` 可以自定义队列名和作业使用的 RabbitMQ 连接.

\*\*示例: \*\*

```csharp

Configure<AbpRabbitMqBackgroundJobOptions>(options =>

{

options.DefaultQueueNamePrefix = "my\_app\_jobs.";

options.DefaultDelayedQueueNamePrefix = "my\_app\_jobs.delayed"

options.PrefetchCount = 1;

options.JobQueues[typeof(EmailSendingArgs)] =

new JobQueueConfiguration(

typeof(EmailSendingArgs),

queueName: "my\_app\_jobs.emails",

connectionName: "SecondConnection",

delayedQueueName:"my\_app\_jobs.emails.delayed"

);

});

```

- 这个示例将默认的队列名前缀设置为 `my\_app\_jobs.`并且设置默认的延迟队列名为 `my\_app\_jobs.delayed`,如果多个项目都使用的同一个 RabbitMQ 服务,设置不同的前缀可以避免执行其他项目的后台作业.

- 设置了预取数量, 用于所有队列.

- 这里还设置了 `EmailSendingArgs` 绑定的 RabbitMQ 连接.

`JobQueueConfiguration` 类的构造函数中,还有一些其他的可选参数.

- `queueName`: 指定后台作业对应的队列名称(全名).

\* `DelayedQueueName`: 指定后台延迟执行的作业对于的队列名称(全名).

- `connectionName`: 后台作业对应的 RabbitMQ 连接名称,默认是 `Default`.

- `durable`: 可选参数,默认为 `true`.

- `exclusive`: 可选参数,默认为 `false`.

- `autoDelete`: 可选参数,默认为 `false`.

\* `PrefetchCount` (可选参数, 默认为: null)

如果你想要更多地了解 `durable`,`exclusive`,`autoDelete` 的用法,请阅读 RabbitMQ 提供的文档.

## 另请参阅

- [后台作业](Background-Jobs.md)

# 后台工作者

## 介绍

后台工作者在应用程序后台运行的简单的独立线程,一般来说它们定期运行执行一些任务.例如;

\* 后台工作者可以定期**\*\*删除过时的日志\*\***.

\* 后台工作者可以定期检查**\*\*不活跃的用户\*\***并且向其**\*\*发送邮件\*\***使用户继续使用你的应用程序.

## 创建一个后台工作者

后台工作者应该直接或间接的继承 `IBackgroundWorker` 接口.

> 后台工作者是[单例](Dependency-Injection.md)的. 所以实例化运行你的工作者类的单个实例.

### BackgroundWorkerBase

`BackgroundWorkerBase` 是创建后台工作者的简单方法.

````csharp

public class MyWorker : BackgroundWorkerBase

{

public override Task StartAsync(CancellationToken cancellationToken = default)

{

//...

}

public override Task StopAsync(CancellationToken cancellationToken = default)

{

//...

}

}

````

`StartAsync` 开始你的工作者(在应用程序启动时),`StopAsync` 停止它(在应用程序关闭时).

> 你可以直接实现 `IBackgroundWorker`, 但 `BackgroundWorkerBase` 提供了一些像 `Logger` 的常用属性.

### AsyncPeriodicBackgroundWorkerBase

假设我们要设置用户为不活跃用户(如果用户最近30天未登录应用程序).`AsyncPeriodicBackgroundWorkerBase` 类简化了创建定期工作者的过程,我们在下面的示例中使用它:

````csharp

public class PassiveUserCheckerWorker : AsyncPeriodicBackgroundWorkerBase

{

public PassiveUserCheckerWorker(

AbpTimer timer,

IServiceScopeFactory serviceScopeFactory

) : base(

timer,

serviceScopeFactory)

{

Timer.Period = 600000; //10 minutes

}

protected async override Task DoWorkAsync(

PeriodicBackgroundWorkerContext workerContext)

{

Logger.LogInformation("Starting: Setting status of inactive users...");

//Resolve dependencies

var userRepository = workerContext

.ServiceProvider

.GetRequiredService<IUserRepository>();

//Do the work

await userRepository.UpdateInactiveUserStatusesAsync();

Logger.LogInformation("Completed: Setting status of inactive users...");

}

}

````

\* `AsyncPeriodicBackgroundWorkerBase` 使用 `AbpTimer`(线程安全定时器)对象来确定**\*\*时间段\*\***. 我们可以在构造函数中设置了`Period` 属性.

\* 它需要实现 `DoWorkAsync` 方法**\*\*执行\*\***定期任务.

\* 最好使用 `PeriodicBackgroundWorkerContext` **\*\*解析依赖\*\*** 而不是构造函数. 因为 `AsyncPeriodicBackgroundWorkerBase` 使用 `IServiceScope` 在你的任务执行结束时会对其 **\*\*disposed\*\***.

\* `AsyncPeriodicBackgroundWorkerBase` **\*\*捕获并记录\*\*** 由 `DoWorkAsync` 方法抛出的 **\*\*异常\*\***.

## 注册后台工作者

创建一个后台工作者后,你应该将其添加到 `IBackgroundWorkerManager`. 最常见的地方是模块类的 `OnApplicationInitializationAsync` 方法:

````csharp

[DependsOn(typeof(AbpBackgroundWorkersModule))]

public class MyModule : AbpModule

{

public override async Task OnApplicationInitialization(

ApplicationInitializationContext context)

{

await context.AddBackgroundWorkerAsync<PassiveUserCheckerWorker>();

}

}

````

`context.AddBackgroundWorkerAsync(...)` 是以下代码的简化扩展方法:

````csharp

context.ServiceProvider

.GetRequiredService<IBackgroundWorkerManager>()

.AddAsync(

context

.ServiceProvider

.GetRequiredService<PassiveUserCheckerWorker>()

);

````

所以,它解析了给定的后台工作者并添加到 `IBackgroundWorkerManager`.

如果我们通常在 `OnApplicationInitializationAsync` 添加工作者,但并不是强制的. 你可以在应用程序的任何地方注入 `IBackgroundWorkerManager` 并在运行时添加工作者. 在你的应用程序关闭时Background worker manager会释放所有已注册的后台工作者.

## Options

`AbpBackgroundWorkerOptions` 是用于设置后台工作者的选择. 目前只有一个选项:

\* `IsEnabled` (默认值: true): 用于为你的应用程序启动或禁用后台工作者系统.

## 让应用程序始终运行

后台工作者只有在你的应用程序运行时才会工作. 如果你将后台作业托管在web应用程序中(这是默认行为),那么你应该确保你的web应用程序被配置为始终运行. 否则只有在你的应用程序正在运行时后台作业才会工作.

## 在集群运行

如果你在集群环境中运行同时运行应用程序的多个实现,这种情况下要小心,每个应用程序都运行相同的后台工作者,如果你的工作者在相同的资源上运行(例如处理相同的数据),那么可能会产生冲突.

如果这对你的工作者是一个问题,你有两个选项:

\* 使用上面提到的 `AbpBackgroundWorkerOptions` 禁用其他的后台工作者系统,只保留一个实例.

\* 所有的应用程序都禁用后台工作者系统,创建一个特殊的应用程序在一个服务上运行执行工作者.

## 集成

后台工作者系统是可扩展的,你可以更改默认的后台工作者管理器为你自己的实现,或者使用以下预构建的集成:

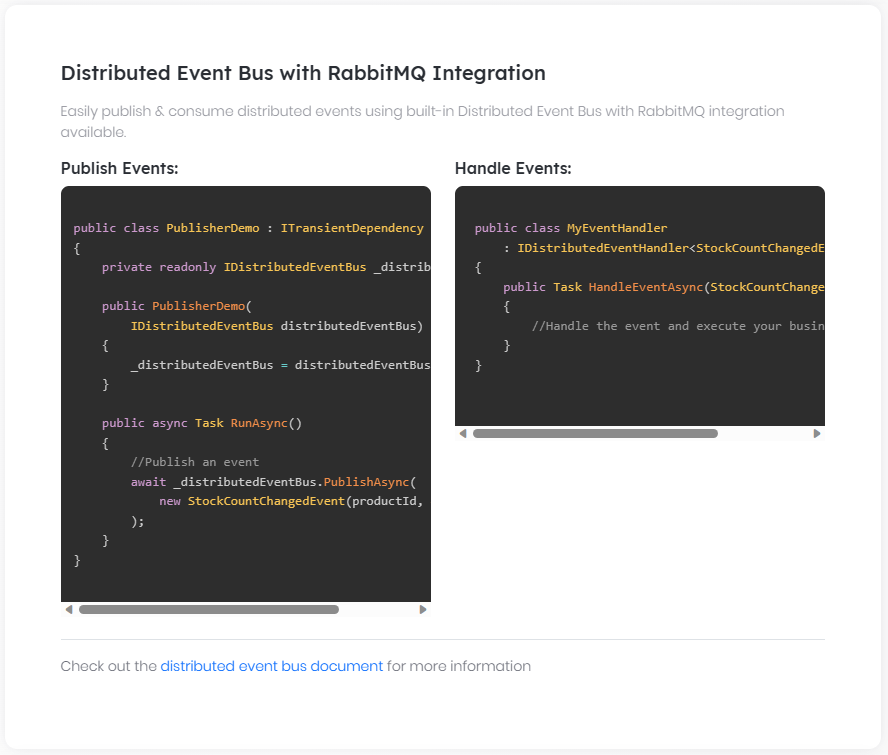
\* [Quartz 后台工作者管理器](Background-Workers-Quartz.md)

\* [Hangfire 后台工作者管理器](Background-Workers-Hangfire.md)

## 另请参阅

\* [后台作业](Background-Jobs.md)

## 事件总线



# 事件总线

事件总线是将消息从发送方传输到接收方的中介. 它在对象,服务和应用程序之间提供了一种松散耦合的通信方式.

## 事件总线类型

ABP框架提供了两种事件总线类型;

\* **\*\*[**本地事件总线**](Local-Event-Bus.md)\*\*** 适合进程内消息传递.

\* **\*\*[**分布式事件总线**](Distributed-Event-Bus.md)\*\*** 适合进程间消息传递,如微服务发布和订阅分布式事件.

本地 Event Bus

# 本地事件总线

本地事件总线允许服务发布和订阅**\*\*进程内事件\*\***. 这意味着如果两个服务(发布者和订阅者)在同一个进程中运行,那么它是合适的.

## 发布事件

以下介绍了两种发布本地事件的方法.

### ILocalEventBus

可以[注入](Dependency-Injection.md) `ILocalEventBus` 并且使用发布本地事件.

**\*\*示例: 产品的存货数量发生变化时发布本地事件\*\***

````csharp

using System;

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.EventBus.Local;

namespace AbpDemo

{

public class MyService : ITransientDependency

{

private readonly ILocalEventBus \_localEventBus;

public MyService(ILocalEventBus localEventBus)

{

\_localEventBus = localEventBus;

}

public virtual async Task ChangeStockCountAsync(Guid productId, int newCount)

{

//TODO: IMPLEMENT YOUR LOGIC...

//PUBLISH THE EVENT

await \_localEventBus.PublishAsync(

new StockCountChangedEvent

{

ProductId = productId,

NewCount = newCount

}

);

}

}

}

````

`PublishAsync` 方法需要一个参数:事件对象,它负责保持与事件相关的数据,是一个简单的普通类:

````csharp

using System;

namespace AbpDemo

{

public class StockCountChangedEvent

{

public Guid ProductId { get; set; }

public int NewCount { get; set; }

}

}

````

即使你不需要传输任何数据也需要创建一个类(在这种情况下为空类).

### 实体/聚合根类

[实体](Entities.md)不能通过依赖注入注入服务,但是在实体/聚合根类中发布本地事件是非常常见的.

**\*\*示例: 在聚合根方法内发布本地事件\*\***

````csharp

using System;

using Volo.Abp.Domain.Entities;

namespace AbpDemo

{

public class Product : AggregateRoot<Guid>

{

public string Name { get; set; }

public int StockCount { get; private set; }

private Product() { }

public Product(Guid id, string name)

: base(id)

{

Name = name;

}

public void ChangeStockCount(int newCount)

{

StockCount = newCount;

//ADD an EVENT TO BE PUBLISHED

AddLocalEvent(

new StockCountChangedEvent

{

ProductId = Id,

NewCount = newCount

}

);

}

}

}

````

`AggregateRoot` 类定义了 `AddLocalEvent` 来添加一个新的本地事件,事件在聚合根对象保存(创建,更新或删除)到数据库时发布.

> 如果实体发布这样的事件,以可控的方式更改相关属性是一个好的实践,就像上面的示例一样 - `StockCount`只能由保证发布事件的 `ChangeStockCount` 方法来更改.

#### IGeneratesDomainEvents 接口

实际上添加本地事件并不是 `AggregateRoot` 类独有的. 你可以为任何实体类实现 `IGeneratesDomainEvents`. 但是 `AggregateRoot` 默认实现了它简化你的工作.

> 不建议为不是聚合根的实体实现此接口,因为它可能不适用于此类实体的某些数据库提供程序. 例如它适用于EF Core,但不适用于MongoDB.

#### 它是如何实现的?

调用 `AddLocalEvent` 不会立即发布事件. 当你将更改保存到数据库时发布该事件;

\* 对于 EF Core, 它在 `DbContext.SaveChanges` 中发布.

\* 对于 MongoDB, 它在你调用仓储的 `InsertAsync`, `UpdateAsync` 或 `DeleteAsync` 方法时发由 (因为MongoDB没有更改跟踪系统).

## 订阅事件

一个服务可以实现 `ILocalEventHandler<TEvent>` 来处理事件.

**\*\*示例: 处理上面定义的**`StockCountChangedEvent`**\*\***

````csharp

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.EventBus;

namespace AbpDemo

{

public class MyHandler

: ILocalEventHandler<StockCountChangedEvent>,

ITransientDependency

{

public async Task HandleEventAsync(StockCountChangedEvent eventData)

{

//TODO: your code that does somthing on the event

}

}

}

````

这就是全部,`MyHandler` 由ABP框架**\*\*自动发现\*\***,并在发生 `StockCountChangedEvent` 事件时调用 `HandleEventAsync`.

\* 事件可以由**\*\*0个或多个处理程序\*\***订阅.

\* 一个事件处理程序可以**\*\*订阅多个事件\*\***,但是需要为每个事件实现 `ILocalEventHandler<TEvent>` 接口.

> 事件处理程序类必须注册到依赖注入(DI),示例中使用了 `ITransientDependency`. 参阅[DI文档](Dependency-Injection.md)了解更多选项.

如果您执行**\*\*数据库操作\*\***并在事件处理程序中使用[仓储](Repositories.md),那么您可能需要创建一个[工作单元](Unit-Of-Work.md),因为一些存储库方法需要在**\*\*活动的工作单元\*\***中工作. 确保处理方法设置为 `virtual`,并为该方法添加一个 `[UnitOfWork]` attribute. 或者手动使用 `IUnitOfWorkManager` 创建一个工作单元范围.

### 事务和异常行为

当一个事件发布,订阅的事件处理程序将立即执行.所以;

\* 如果处理程序**\*\*抛出一个异常\*\***,它会影响发布该事件的代码. 这意味着它在 `PublishAsync` 调用上获得异常. 因此如果你想隐藏错误,在事件处理程序中**\*\*使用try-catch\*\***.

\*如果在一个[工作单元](Unit-Of-Work.md)范围内执行的事件发布的代码,该事件处理程序也由工作单元覆盖. 这意味着,如果你的UOW是事务和处理程序抛出一个异常,事务会回滚.

## 预定义的事件

**\*\*发布实体创建,更新,删除事件\*\***是常见的操作. ABP框架为所有的实体**\*\*自动\*\***发布这些事件. 你只需要订阅相关的事件.

**\*\*示例: 订阅用户创建事件\*\***

````csharp

using System.Threading.Tasks;

using Microsoft.AspNetCore.Identity;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Domain.Entities.Events;

using Volo.Abp.EventBus;

namespace AbpDemo

{

public class MyHandler

: ILocalEventHandler<EntityCreatedEventData<IdentityUser>>,

ITransientDependency

{

public async Task HandleEventAsync(

EntityCreatedEventData<IdentityUser> eventData)

{

var userName = eventData.Entity.UserName;

var email = eventData.Entity.Email;

//...

}

}

}

````

这个类订阅 `EntityCreatedEventData<IdentityUser>`,它在用户创建后发布. 你可能需要向新用户发送一封"欢迎"电子邮件.

这些事件有两种类型:过去时态的事件和进行时态的事件.

### 用过去时态事件

当相关工作单元完成且实体更改成功保存到数据库时,将发布带有过去时态的事件. 如果在这些事件处理程序上抛出异常,则**\*\*无法回滚\*\***事务,因为事务已经提交.

事件类型;

\* `EntityCreatedEventData<T>` 当实体创建成功后发布.

\* `EntityUpdatedEventData<T>` 当实体更新成功后发布.

\* `EntityDeletedEventData<T>` 当实体删除成功后发布.

\* `EntityChangedEventData<T>` 当实体创建,更新,删除后发布. 如果你需要监听任何类型的更改,它是一种快捷方式 - 而不是订阅单个事件.

### 用于进行时态事件

带有进行时态的事件在完成事务之前发布(如果数据库事务由所使用的数据库提供程序支持). 如果在这些事件处理程序上抛出异常,它**\*\*会回滚\*\***事务,因为事务还没有完成,更改也没有保存到数据库中.

事件类型;

\* `EntityCreatingEventData<T>` 当新实体保存到数据库前发布.

\* `EntityUpdatingEventData<T>` 当已存在实体更新到数据库前发布.

\* `EntityDeletingEventData<T>` 删除实体前发布.

\* `EntityChangingEventData<T>` 当实体创建,更新,删除前发布. 如果你需要监听任何类型的更改,它是一种快捷方式 - 而不是订阅单个事件.

#### 它是如何实现的?

在将更改保存到数据库时发布预构建事件;

\* 对于 EF Core, 他们在 `DbContext.SaveChanges` 发布.

\* 对于 MongoDB, 在你调用仓储的 `InsertAsync`, `UpdateAsync` 或 `DeleteAsync` 方法发布(因为MongoDB没有更改追踪系统).

分布式 Event Bus

# 分布式事件总线

分布式事件总线系统允许**\*\*发布\*\***和**\*\*订阅跨应用/服务边界\*\***传输的事件. 你可以使用分布式事件总线在**\*\*微服务\*\***或**\*\*应用程序\*\***之间异步发送和接收消息.

## 提供程序

分布式事件总线系统提供了一个可以被任何提供程序实现的**\*\*抽象\*\***. 有四种开箱即用的提供程序:

\* `LocalDistributedEventBus` 是默认实现,实现作为进程内工作的分布式事件总线. 是的!如果没有配置真正的分布式提供程序,**\*\*默认实现的工作方式与[**本地事件总线**](Local-Event-Bus.md)一样\*\***.

\* `RabbitMqDistributedEventBus` 通过[RabbitMQ](https://www.rabbitmq.com/)实现分布式事件总线. 请参阅[RabbitMQ集成文档](Distributed-Event-Bus-RabbitMQ-Integration.md)了解如何配置它.

\* `KafkaDistributedEventBus` 通过[Kafka](https://kafka.apache.org/)实现分布式事件总线. 请参阅[Kafka集成文档](Distributed-Event-Bus-Kafka-Integration.md)了解如何配置它.

\* `RebusDistributedEventBus` 通过[Rebus](http://mookid.dk/category/rebus/)实现分布式事件总线. 请参阅[Rebus集成文档](Distributed-Event-Bus-Rebus-Integration.md)了解如何配置它.

使用本地事件总线作为默认具有一些重要的优点. 最重要的是:它允许你编写与分布式体系结构兼容的代码. 您现在可以编写一个整体应用程序,以后可以拆分成微服务. 最好通过分布式事件而不是本地事件在边界上下文之间(或在应用程序模块之间)进行通信.

例如,[预构建的应用模块](Modules/Index.md)被设计成在分布式系统中作为服务工作,同时它们也可以在独立应用程序中作为模块工作,而不依赖于外部消息代理.

## 发布事件

以下介绍了两种发布分布式事件的方法.

### IDistributedEventBus

可以[注入](Dependency-Injection.md) `IDistributedEventBus` 并且使用发布分布式事件.

**\*\*示例: 产品的存货数量发生变化时发布分布式事件\*\***

````csharp

using System;

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.EventBus.Distributed;

namespace AbpDemo

{

public class MyService : ITransientDependency

{

private readonly IDistributedEventBus \_distributedEventBus;

public MyService(IDistributedEventBus distributedEventBus)

{

\_distributedEventBus = distributedEventBus;

}

public virtual async Task ChangeStockCountAsync(Guid productId, int newCount)

{

await \_distributedEventBus.PublishAsync(

new StockCountChangedEto

{

ProductId = productId,

NewCount = newCount

}

);

}

}

}

````

`PublishAsync` 方法需要一个参数:事件对象,它负责保持与事件相关的数据,是一个简单的普通类:

````csharp

using System;

namespace AbpDemo

{

[EventName("MyApp.Product.StockChange")]

public class StockCountChangedEto

{

public Guid ProductId { get; set; }

public int NewCount { get; set; }

}

}

````

即使你不需要传输任何数据也需要创建一个类(在这种情况下为空类).

> `Eto` 是我们按照约定使用的**\*\*E\*\***vent **\*\*T\*\***ransfer **\*\*O\*\***bjects(事件传输对象)的后缀. s虽然这不是必需的,但我们发现识别这样的事件类很有用(就像应用层上的[DTO](Data-Transfer-Objects.md) 一样).

#### 事件名称

`EventName`attribute是可选的,但建议使用. 如果不声明,事件名将事件名称将是事件类的全名. 这里是 `AbpDemo.StockCountChangedEto`.

#### 关于序列化的事件对象

事件传输对象**\*\*必须是可序列化\*\***的,因为将其传输到进程外时，它们将被序列化/反序列化为JSON或其他格式.

避免循环引用,多态,私有setter,并提供默认(空)构造函数,如果你有其他的构造函数.(虽然某些序列化器可能会正常工作),就像DTO一样.

### 实体/聚合根类

[实体](Entities.md)不能通过依赖注入注入服务,但是在实体/聚合根类中发布分布式事件是非常常见的.

**\*\*示例: 在聚合根方法内发布分布式事件\*\***

````csharp

using System;

using Volo.Abp.Domain.Entities;

namespace AbpDemo

{

public class Product : AggregateRoot<Guid>

{

public string Name { get; set; }

public int StockCount { get; private set; }

private Product() { }

public Product(Guid id, string name)

: base(id)

{

Name = name;

}

public void ChangeStockCount(int newCount)

{

StockCount = newCount;

//ADD an EVENT TO BE PUBLISHED

AddDistributedEvent(

new StockCountChangedEto

{

ProductId = Id,

NewCount = newCount

}

);

}

}

}

````

`AggregateRoot` 类定义了 `AddDistributedEvent` 来添加一个新的分布式事件,事件在聚合根对象保存(创建,更新或删除)到数据库时发布.

> 如果实体发布这样的事件,以可控的方式更改相关属性是一个好的实践,就像上面的示例一样 - `StockCount`只能由保证发布事件的 `ChangeStockCount` 方法来更改.

#### IGeneratesDomainEvents 接口

实际上添加分布式事件并不是 `AggregateRoot` 类独有的. 你可以为任何实体类实现 `IGeneratesDomainEvents`. 但是 `AggregateRoot` 默认实现了它简化你的工作.

> 不建议为不是聚合根的实体实现此接口,因为它可能不适用于此类实体的某些数据库提供程序. 例如它适用于EF Core,但不适用于MongoDB.

#### 它是如何实现的?

调用 `AddDistributedEvent` 不会立即发布事件. 当你将更改保存到数据库时发布该事件;

\* 对于 EF Core, 它在 `DbContext.SaveChanges` 中发布.

\* 对于 MongoDB, 它在你调用仓储的 `InsertAsync`, `UpdateAsync` 或 `DeleteAsync` 方法时发由 (因为MongoDB没有更改跟踪系统).

## 订阅事件

一个服务可以实现 `IDistributedEventHandler<TEvent>` 来处理事件.

**\*\*示例: 处理上面定义的**`StockCountChangedEto`**\*\***

````csharp

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.EventBus.Distributed;

namespace AbpDemo

{

public class MyHandler

: IDistributedEventHandler<StockCountChangedEto>,

ITransientDependency

{

public async Task HandleEventAsync(StockCountChangedEto eventData)

{

var productId = eventData.ProductId;

}

}

}

````

这就是全部.

\* `MyHandler` 由ABP框架**\*\*自动发现\*\***,并在发生 `StockCountChangedEto` 事件时调用 `HandleEventAsync`.

\* 如果你使用的是分布式消息代理,比如RabbitMQ,ABP会自动**\*\*订阅消息代理上的事件\*\***,获取消息执行处理程序.

\* 如果事件处理程序成功执行(没有抛出任何异常),它将向消息代理发送**\*\*确认(ACK)\*\***.

你可以在处理程序注入任何服务来执行所需的逻辑. 一个事件处理程序可以**\*\*订阅多个事件\*\***,但是需要为每个事件实现 `IDistributedEventHandler<TEvent>` 接口.

> 事件处理程序类必须注册到依赖注入(DI),示例中使用了 `ITransientDependency`. 参阅[DI文档](Dependency-Injection.md)了解更多选项.

## 预定义的事件

如果你配置,ABP框架会为[实体](Entities.md)**\*\*自动发布创建,更新和删除\*\***分布式事件.

### 事件类型

有三种预定义的事件类型:

\* `EntityCreatedEto<T>` 是实体 `T` 创建后发布.

\* `EntityUpdatedEto<T>` 是实体 `T` 更新后发布.

\* `EntityDeletedEto<T>` 是实体 `T` 删除后发布.

这些都是泛型的, `T` 实际上是**\*\*E\*\***vent **\*\*T\*\***ransfer **\*\*O\*\***bject (ETO)的类型,而不是实体的类型,因为实体对象不能做为事件数据传输,所以通常会为实体类定义一个ETO类,如为 `Product` 实体定义 `ProductEto`.

### 订阅事件

订阅自动事件与订阅常规分布式事件相同.

**\*\*示例: 产品更新后获取通知\*\***

````csharp

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Domain.Entities.Events.Distributed;

using Volo.Abp.EventBus.Distributed;

namespace AbpDemo

{

public class MyHandler :

IDistributedEventHandler<EntityUpdatedEto<ProductEto>>,

ITransientDependency

{

public async Task HandleEventAsync(EntityUpdatedEto<ProductEto> eventData)

{

var productId = eventData.Entity.Id;

//TODO

}

}

}

````

\* `MyHandler` 实现了 `IDistributedEventHandler<EntityUpdatedEto<ProductEto>>`.

### 配置

你可以在[模块](Module-Development-Basics.md)的 `ConfigureServices` 中配置 `AbpDistributedEntityEventOptions`添加选择器.

**\*\*示例: 配置示例\*\***

````csharp

Configure<AbpDistributedEntityEventOptions>(options =>

{

//Enable for all entities

options.AutoEventSelectors.AddAll();

//Enable for a single entity

options.AutoEventSelectors.Add<IdentityUser>();

//Enable for all entities in a namespace (and child namespaces)

options.AutoEventSelectors.AddNamespace("Volo.Abp.Identity");

//Custom predicate expression that should return true to select a type

options.AutoEventSelectors.Add(

type => type.Namespace.StartsWith("MyProject.")

);

});

````

\* 最后一个提供了灵活性来决定是否应该针对给定的实体类型发布事件. 返回 `true` 代表为该 `Type` 发布事件.

你可以添加多个选择器. 如果选择器之一与实体类型匹配,则将其选中.

### 事件传输对象

一旦你为一个实体启用了**\*\*自动事件\*\***,ABP框架就会为实体上的更改发布事件. 如果你没有为实体指定对应的**\*\*E\*\***vent **\*\*T\*\***ransfer **\*\*O\*\***bject(ETO), ABP框架会使用一个标准类型 `EntityEto`,它只有两个属性:

\* `EntityType` (`string`): 实体类的全名(包括命令空间).

\* `KeysAsString` (`string`): 已更改实体的主键.如果它只有一个主键,这个属性将是主键值. 对于复合键,它包含所有用`,`(逗号)分隔的键.

因此可以实现 `IDistributedEventHandler<EntityUpdatedEto<EntityEto>>` 订阅事件. 但是订阅这样的通用事件不是一个好方法,你可以为实体类型定义对应的ETO.

**\*\*示例: 为** `Product` **声明使用** `ProductEto`**\*\***

````csharp

Configure<AbpDistributedEntityEventOptions>(options =>

{

options.AutoEventSelectors.Add<Product>();

options.EtoMappings.Add<Product, ProductEto>();

});

````

在这个示例中;

\* 添加选择器允许发布 `Product` 实体的创建,更新和删除事件.

\* 配置为使用 `ProductEto` 作为事件传输对象来发布与 `Product` 相关的事件.

分布式事件系统使用[对象到对象的映射](Object-To-Object-Mapping.md)系统来映射 `Product` 对象到 `ProductEto` 对象,你需要配置映射. 请参阅可以对象到对象映射文档了解所有选项,下面的示例展示了如何使用[AutoMapper](https://automapper.org/)库配置它.

**\*\*示例: 使用AutoMapper配置** `Product` **到** `ProductEto` **映射\*\***

````csharp

using System;

using AutoMapper;

using Volo.Abp.Domain.Entities.Events.Distributed;

namespace AbpDemo

{

[AutoMap(typeof(Product))]

public class ProductEto : EntityEto

{

public Guid Id { get; set; }

public string Name { get; set; }

}

}

````

此示例使用AutoMapper的 `AutoMap` 属性配置的映射. 你可以创建一个配置文件类代替. 请参阅AutoMapper文档了解更多选项.

#### RabbitMQ 集成

# 分布式事件总线RabbitMQ集成

> 本文解释了 **\*\*如何配置[**RabbitMQ**](https://www.rabbitmq.com/)\*\*** 做为分布式总线提供程序. 参阅[分布式事件总线文档](Distributed-Event-Bus.md)了解如何使用分布式事件总线系统.

## 安装

使用ABP CLI添加[Volo.Abp.EventBus.RabbitMQ](https://www.nuget.org/packages/Volo.Abp.EventBus.RabbitMQ)NuGet包到你的项目:

\* 安装[ABP CLI](https://docs.abp.io/en/abp/latest/CLI),如果你还没有安装.

\* 在你想要安装 `Volo.Abp.EventBus.RabbitMQ` 包的 `.csproj` 文件目录打开命令行(终端).

\* 运行 `abp add-package Volo.Abp.EventBus.RabbitMQ` 命令.

如果你想要手动安装,安装[Volo.Abp.EventBus.RabbitMQ](https://www.nuget.org/packages/Volo.Abp.EventBus.RabbitMQ) NuGet 包到你的项目然后添加 `[DependsOn(typeof(AbpEventBusRabbitMqModule))]` 到你的项目[模块](Module-Development-Basics.md)类.

## 配置

可以使用配置使用标准的[配置系统](Configuration.md),如 `appsettings.json` 文件,或[选项](Options.md)类.

### `appsettings.json` 文件配置

这是配置RabbitMQ设置最简单的方法. 它也非常强大,因为你可以使用[由AspNet Core支持](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/)的任何其他配置源(如环境变量).

**\*\*示例：最小化配置与默认配置连接到本地的RabbitMQ服务器\*\***

````json

{

"RabbitMQ": {

"EventBus": {

"ClientName": "MyClientName",

"ExchangeName": "MyExchangeName"

}

}

}

````

\* `ClientName` 是应用程序的名称,用于RabbitMQ的**\*\*队列名称\*\***.

\* `ExchangeName` 是 **\*\*交换机名称\*\***.

参阅[RabbitMQ文档](https://www.rabbitmq.com/dotnet-api-guide.html#exchanges-and-queues)更好的了解这些选项.

#### 连接

如果需要连接到本地主机以外的另一台服务器,需要配置连接属性.

**\*\*示例: 指定主机名 (如IP地址)\*\***

````json

{

"RabbitMQ": {

"Connections": {

"Default": {

"HostName": "123.123.123.123"

}

},

"EventBus": {

"ClientName": "MyClientName",

"ExchangeName": "MyExchangeName"

}

}

}

````

允许定义多个连接. 在这种情况下,你可以指定用于事件总线的连接.

**\*\*示例: 声明两个连接并将其中一个用于事件总线\*\***

````json

{

"RabbitMQ": {

"Connections": {

"Default": {

"HostName": "123.123.123.123"

},

"SecondConnection": {

"HostName": "321.321.321.321"

}

},

"EventBus": {

"ClientName": "MyClientName",

"ExchangeName": "MyExchangeName",

"ConnectionName": "SecondConnection"

}

}

}

````

这允许你可以在你的应用程序使用多个RabbitMQ服务器,但将其中一个做为事件总线.

你可以使用任何[ConnectionFactry](http://rabbitmq.github.io/rabbitmq-dotnet-client/api/RabbitMQ.Client.ConnectionFactory.html#properties)属性作为连接属性.

**\*\*示例: 指定连接端口\*\***

````csharp

{

"RabbitMQ": {

"Connections": {

"Default": {

"HostName": "123.123.123.123",

"Port": "5672"

}

}

}

}

````

如果需要连接到 RabbitMQ 集群，你可以指定多个 HostName。

**\*\*示例: 连接到 RabbitMQ 集群\*\***

```json

{

"RabbitMQ": {

"Connections": {

"Default": {

"HostName": "123.123.123.123;234.234.234.234"

}

},

"EventBus": {

"ClientName": "MyClientName",

"ExchangeName": "MyExchangeName"

}

}

}

```

### 选项类

`AbpRabbitMqOptions` 和 `AbpRabbitMqEventBusOptions` 类用于配置RabbitMQ的连接字符串和事件总线选项.

你可以在你的[模块](Module-Development-Basics.md)的 `ConfigureServices` 方法配置选项.

**\*\*示例: 配置连接\*\***

````csharp

Configure<AbpRabbitMqOptions>(options =>

{

options.Connections.Default.UserName = "user";

options.Connections.Default.Password = "pass";

options.Connections.Default.HostName = "123.123.123.123";

options.Connections.Default.Port = 5672;

});

````

**\*\*示例: 配置客户端,交换机名称和预取数量\*\***

````csharp

Configure<AbpRabbitMqEventBusOptions>(options =>

{

options.ClientName = "TestApp1";

options.ExchangeName = "TestMessages";

options.PrefetchCount = 1;

});

````

使用这些选项类可以与 `appsettings.json` 组合在一起. 在代码中配置选项属性会覆盖配置文件中的值.

#### Kafka 集成

# 分布式事件总线Kafka集成

> 本文解释了 **\*\*如何配置[**Kafka**](https://kafka.apache.org/)\*\*** 做为分布式总线提供程序. 参阅[分布式事件总线文档](Distributed-Event-Bus.md)了解如何使用分布式事件总线系统.

## 安装

使用ABP CLI添加[Volo.Abp.EventBus.Kafka](https://www.nuget.org/packages/Volo.Abp.EventBus.Kafka)NuGet包到你的项目:

\* 安装[ABP CLI](https://docs.abp.io/en/abp/latest/CLI),如果你还没有安装.

\* 在你想要安装 `Volo.Abp.EventBus.Kafka` 包的 `.csproj` 文件目录打开命令行(终端).

\* 运行 `abp add-package Volo.Abp.EventBus.Kafka` 命令.

如果你想要手动安装,安装[Volo.Abp.EventBus.Kafka](https://www.nuget.org/packages/Volo.Abp.EventBus.Kafka) NuGet 包到你的项目然后添加 `[DependsOn(typeof(AbpEventBusKafkaModule))]` 到你的项目[模块](Module-Development-Basics.md)类.

## 配置

可以使用配置使用标准的[配置系统](Configuration.md),如 `appsettings.json` 文件,或[选项](Options.md)类.

### `appsettings.json` 文件配置

这是配置Kafka设置最简单的方法. 它也非常强大,因为你可以使用[由AspNet Core支持](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/)的任何其他配置源(如环境变量).

**\*\*示例：使用默认配置连接到本地Kafka服务器的最小配置\*\***

````json

{

"Kafka": {

"EventBus": {

"GroupId": "MyGroupId",

"TopicName": "MyTopicName"

}

}

}

````

\* `MyGroupId` 是应用程序的名称,用于Kafka的**\*\*GroupId\*\***.

\* `MyTopicName` 是**\*\*topic名称\*\***.

参阅[Kafka文档](https://docs.confluent.io/current/clients/confluent-kafka-dotnet/api/Confluent.Kafka.html)更好的了解这些选项.

#### 连接

如果需要连接到本地主机以外的另一台服务器,需要配置连接属性.

**\*\*示例: 指定主机名 (如IP地址)\*\***

````json

{

"Kafka": {

"Connections": {

"Default": {

"BootstrapServers": "123.123.123.123:9092"

}

},

"EventBus": {

"GroupId": "MyGroupId",

"TopicName": "MyTopicName"

}

}

}

````

允许定义多个连接. 在这种情况下,你可以指定用于事件总线的连接.

**\*\*示例: 声明两个连接并将其中一个用于事件总线\*\***

````json

{

"Kafka": {

"Connections": {

"Default": {

"BootstrapServers": "123.123.123.123:9092"

},

"SecondConnection": {

"BootstrapServers": "321.321.321.321:9092"

}

},

"EventBus": {

"GroupId": "MyGroupId",

"TopicName": "MyTopicName",

"ConnectionName": "SecondConnection"

}

}

}

````

这允许你可以在你的应用程序使用多个Kafka服务器,但将其中一个做为事件总线.

你可以使用任何[ClientConfig](https://docs.confluent.io/current/clients/confluent-kafka-dotnet/api/Confluent.Kafka.ClientConfig.html)属性作为连接属性.

**\*\*示例: 指定socket超时时间\*\***

````json

{

"Kafka": {

"Connections": {

"Default": {

"BootstrapServers": "123.123.123.123:9092",

"SocketTimeoutMs": 60000

}

}

}

}

````

### 选项类

`AbpKafkaOptions` 和 `AbpKafkaEventBusOptions` 类用于配置Kafka的连接字符串和事件总线选项.

你可以在你的[模块](Module-Development-Basics.md)的 `ConfigureServices` 方法配置选项.

**\*\*示例: 配置连接\*\***

````csharp

Configure<AbpKafkaOptions>(options =>

{

options.Connections.Default.BootstrapServers = "123.123.123.123:9092";

options.Connections.Default.SaslUsername = "user";

options.Connections.Default.SaslPassword = "pwd";

});

````

**\*\*示例: 配置 consumer config\*\***

````csharp

Configure<AbpKafkaOptions>(options =>

{

options.ConfigureConsumer = config =>

{

config.GroupId = "MyGroupId";

config.EnableAutoCommit = false;

};

});

````

**\*\*示例: 配置 producer config\*\***

````csharp

Configure<AbpKafkaOptions>(options =>

{

options.ConfigureProducer = config =>

{

config.MessageTimeoutMs = 6000;

config.Acks = Acks.All;

};

});

````

**\*\*示例: 配置 topic specification\*\***

````csharp

Configure<AbpKafkaOptions>(options =>

{

options.ConfigureTopic = specification =>

{

specification.ReplicationFactor = 3;

specification.NumPartitions = 3;

};

});

````

使用这些选项类可以与 `appsettings.json` 组合在一起. 在代码中配置选项属性会覆盖配置文件中的值.

#### Rebus 集成

# 分布式事件总线Rebus集成

> 本文解释了 **\*\*如何配置[**Rebus**](http://mookid.dk/category/rebus/)\*\*** 做为分布式总线提供程序. 参阅[分布式事件总线文档](Distributed-Event-Bus.md)了解如何使用分布式事件总线系统.

## 安装

使用ABP CLI添加[Volo.Abp.EventBus.Rebus](https://www.nuget.org/packages/Volo.Abp.EventBus.Rebus)NuGet包到你的项目:

\* 安装[ABP CLI](https://docs.abp.io/en/abp/latest/CLI),如果你还没有安装.

\* 在你想要安装 `Volo.Abp.EventBus.Rebus` 包的 `.csproj` 文件目录打开命令行(终端).

\* 运行 `abp add-package Volo.Abp.EventBus.Rebus` 命令.

如果你想要手动安装,安装[Volo.Abp.EventBus.Rebus](https://www.nuget.org/packages/Volo.Abp.EventBus.Rebus) NuGet 包到你的项目然后添加 `[DependsOn(typeof(AbpEventBusRebusModule))]` 到你的项目[模块](Module-Development-Basics.md)类.

## 配置

可以使用配置使用标准的[配置系统](Configuration.md),如[选项](Options.md)类.

`AbpRebusEventBusOptions` 类用于配置事件总线选项.

你可以在你的[模块](Module-Development-Basics.md)的 `PreConfigureServices` 方法配置选项.

**\*\*示例: 最小化配置\*\***

```csharp

PreConfigure<AbpRebusEventBusOptions>(options =>

{

options.InputQueueName = "eventbus";

});

```

Rebus 有很多选项,你可以使用 `AbpRebusEventBusOptions` 的 `Configurer` 属性来配置.

默认事件**\*\*存储在内存中\*\***. 参阅[rebus文档](https://github.com/rebus-org/Rebus/wiki/Transport)了解更多信息.

**\*\*示例: 配置存储\*\***

````csharp

PreConfigure<AbpRebusEventBusOptions>(options =>

{

options.InputQueueName = "eventbus";

options.Configurer = rebusConfigurer =>

{

rebusConfigurer.Transport(t => t.UseMsmq("eventbus"));

rebusConfigurer.Subscriptions(s => s.UseJsonFile(@"subscriptions.json"));

};

});

````

你可以使用 `AbpRebusEventBusOptions` 的 `Publish` 属性来更改发布方法.

**\*\*示例: 配置事件发布\*\***

````csharp

PreConfigure<AbpRebusEventBusOptions>(options =>

{

options.InputQueueName = "eventbus";

options.Publish = async (bus, type, data) =>

{

await bus.Publish(data);

};

});

````

## 种子数据

# 种子数据

## 介绍

使用数据库的某些应用程序(或模块),可能需要有一些**\*\*初始数据\*\***才能​​够正常启动和运行. 例如**\*\*管理员用户\*\***和角色必须在一开始就可用. 否则你就无法**\*\*登录\*\***到应用程序创建新用户和角色.

数据种子也可用于[测试](Testing.md)的目的,你的自动测试可以假定数据库中有一些可用的初始数据.

### 为什么要有种子数据系统?

尽管EF Core Data Seeding系统提供了一种方法,但它非常有限,不包括生产场景. 此外它仅适用于EF Core.

ABP框架提供了种子数据系统;

\* **\*\*模块化\*\***: 任何[模块](Module-Development-Basics.md)都可以无声地参与数据播种过程,而不相互了解和影响. 通过这种方式模块将种子化自己的初始数据.

\* **\*\*数据库独立\*\***: 它不仅适用于 EF Core, 也使用其他数据库提供程序(如 [MongoDB](MongoDB.md)).

\* **\*\*生产准备\*\***: 它解决了生产环境中的问题. 参见下面的*\*On Production\**部分.

\* **\*\*依赖注入\*\***: 它充分利用了依赖项注入,你可以在播种初始数据时使用任何内部或外部服务. 实际上你可以做的不仅仅是数据播种.

## IDataSeedContributor

将数据种子化到数据库需要实现 `IDataSeedContributor` 接口.

**\*\*示例: 如果没有图书,则向数据库播种一个初始图书\*\***

````csharp

using System;

using System.Threading.Tasks;

using Volo.Abp.Data;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Domain.Repositories;

using Volo.Abp.Guids;

namespace Acme.BookStore

{

public class BookStoreDataSeedContributor

: IDataSeedContributor, ITransientDependency

{

private readonly IRepository<Book, Guid> \_bookRepository;

private readonly IGuidGenerator \_guidGenerator;

private readonly ICurrentTenant \_currentTenant;

public BookStoreDataSeedContributor(

IRepository<Book, Guid> bookRepository,

IGuidGenerator guidGenerator,

ICurrentTenant currentTenant)

{

\_bookRepository = bookRepository;

\_guidGenerator = guidGenerator;

\_currentTenant = currentTenant;

}

public async Task SeedAsync(DataSeedContext context)

{

using (\_currentTenant.Change(context?.TenantId))

{

if (await \_bookRepository.GetCountAsync() > 0)

{

return;

}

var book = new Book(

id: \_guidGenerator.Create(),

name: "The Hitchhiker's Guide to the Galaxy",

type: BookType.ScienceFiction,

publishDate: new DateTime(1979, 10, 12),

price: 42

);

await \_bookRepository.InsertAsync(book);

}

}

}

}

````

\* `IDataSeedContributor` 定义了 `SeedAsync` 方法用于执行 **\*\*数据种子逻辑\*\***.

\* 通常**\*\*检查数据库\*\***是否已经存在种子数据.

\* 你可以**\*\*注入\*\***服务,检查数据播种所需的任何逻辑.

> 数据种子贡献者由ABP框架自动发现,并作为数据播种过程的一部分执行.

### DataSeedContext

如果你的应用程序是[多租户](Multi-Tenancy.md), `DataSeedContext` 包含 `TenantId`,因此你可以在插入数据或基于租户执行自定义逻辑时使用该值.

`DataSeedContext` 还包含用于从 `IDataSeeder` 传递到种子提供者的name-value配置参数.

## 模块化

一个应用程序可以具有多个种子数据贡献者(`IDataSeedContributor`)类. 任何可重用模块也可以实现此接口播种其自己的初始数据.

例如[Identity模块](Modules/Identity.md)有一个种子数据贡献者,它创建一个管理角色和管理用户并分配所有权限.

## IDataSeeder

> 通常你不需要直接使用 `IDataSeeder` 服务,因为如果你从[应用程序启动模板](Startup-Templates/Application.md)开始,该服务已经完成. 但是建议阅读以了解种子数据系统背后的设计.

`IDataSeeder` 是用于生成初始数据的主要服务. 使用它很容易;

````csharp

public class MyService : ITransientDependency

{

private readonly IDataSeeder \_dataSeeder;

public MyService(IDataSeeder dataSeeder)

{

\_dataSeeder = dataSeeder;

}

public async Task FooAsync()

{

await \_dataSeeder.SeedAsync();

}

}

````

你可以[注入](Dependency-Injection.md) `IDataSeeder` 并且在你需要时使用它初始化种子数据. 它内部调用 `IDataSeedContributor` 的实现去完成数据播种.

可以将命名的配置参数发送到 `SeedAsync` 方法,如下所示:

````csharp

await \_dataSeeder.SeedAsync(

new DataSeedContext()

.WithProperty("MyProperty1", "MyValue1")

.WithProperty("MyProperty2", 42)

);

````

然后种子数据提供者可以通过前面解释的 `DataSeedContext` 访问这些属性.

如果模块需要参数,应该在[模块文档](Modules/Index.md)中声明它. 例如[Identity Module](Modules/Identity.md)使用 `AdminEmail` 和 `AdminPassword` 参数,如果你提供了(默认使用默认值).

### 在何处以及如何播种数据?

重要的是要了解在何处以及如何执行 `IDataSeeder.SeedAsync()`.

#### On Production

[应用程序启动模板](Startup-Templates/Application.md)带有一个*\*YourProjectName\**\*\*.DbMigrator\*\* 项目(图中的Acme.BookStore.DbMigrator). 这是一个**\*\*控制台应用程序\*\***,负责**\*\*迁移\*\***数据库架构(关系数据库)和初始种子数据:

![bookstore-visual-studio-solution-v3](images/bookstore-visual-studio-solution-v3.png)

控制台应用程序已经为你正确配置,它甚至支持**\*\*多租户\*\***场景,其中每个租户拥有自己的数据库(迁移和必须的数据库).

当你将解决方案的**\*\*新版本部署到服务器\*\***时,都需要运行这个DbMigrator应用程序. 它会迁移你的**\*\*数据库架构\*\***(创建新的表/字段…)和播种正确运行解决方案的新版本所需的**\*\*新初始数据\*\***. 然后就可以部署/启动实际的应用程序了.

即使你使用的是MongoDB或其他NoSQL数据库(不需要进行架构迁移),也建议使用DbMigrator应用程序为你的数据添加种子或执行数据迁移.

有这样一个单独的控制台应用程序有几个优点;

\* 你可以在更新你的应用程序**\*\*之前运行它\*\***,所以你的应用程序可以在准备就绪的数据库上运行.

\* 与本身初始化种子数据相比,你的应用程序**\*\*启动速度更快\*\***.

\* 应用程序可以在**\*\*集群环境\*\***中正确运行(其中应用程序的多个实例并发运行). 在这种情况下如果在应用程序启动时播种数据就会有冲突.

#### On Development

我们建议以相同的方式进行开发. 每当你[创建数据库迁移](https://docs.microsoft.com/en-us/ef/ef6/modeling/code-first/migrations/)(例如使用EF Core `Add-Migration` 命令)或更改数据种子代码(稍后说明)时,请运行DbMigrator控制台应用程序.

> 你可以使用EF Core继续执行标准的 `Update-Database` 命令,但是它不会初始化种子数据.

#### On Testing

你可能想为自动[测试](Testing.md)初始化数据种子, 这需要使用 `IDataSeeder.SeedAsync()`. 在[应用程序启动模板](Startup-Templates/Application.md)中,它在TestBase项目的*\*YourProjectName\**TestBaseModule类的[OnApplicationInitialization](Module-Development-Basics.md)方法中完成.

除了标准种子数据(也在生产中使用)之外,你可能还希望为自动测试添加其他种子数据. 你可以在测试项目中创建一个新的数据种子贡献者以处理更多数据.

## 分布式锁

# 分布式锁

分布式锁是一种管理多个应用程序访问同一资源的技术. 主要目的是同一时间只允许多个应用程序中的一个访问资源. 否则, 从不同的应用程序访问同一对象可能会破坏资源.

> ABP当前的分布式锁实现基于[DistributedLock](https://github.com/madelson/DistributedLock)库.

## 安装

你可以打开一个命令行终端并输入以下命令来安装[Volo.Abp.DistributedLocking](https://www.nuget.org/packages/Volo.Abp.DistributedLocking)到你的项目中:

````bash

abp add-package Volo.Abp.DistributedLocking

````

这个库提供了使用分布式锁系统所需的API, 但是, 在使用它之前, 你应该配置一个提供程序.

### 配置一个提供程序

[DistributedLock](https://github.com/madelson/DistributedLock)库对[Redis](https://github.com/madelson/DistributedLock/blob/master/docs/DistributedLock.Redis.md)和[ZooKeeper](https://github.com/madelson/DistributedLock/blob/master/docs/DistributedLock.ZooKeeper.md)提供[多种实现](https://github.com/madelson/DistributedLock#implementations).

例如, 如果你想使用[Redis provider](https://github.com/madelson/DistributedLock/blob/master/docs/DistributedLock.Redis.md), 你应该将[DistributedLock.Redis](https://www.nuget.org/packages/DistributedLock.Redis) NuGet包添加到项目中, 然后将以下代码添加到ABP[模块](Module-Development-Basics.md)类的`ConfigureServices`方法中:

````csharp

using Medallion.Threading;

using Medallion.Threading.Redis;

namespace AbpDemo

{

[DependsOn(

typeof(AbpDistributedLockingModule)

//If you have the other dependencies, you should do here

)]

public class MyModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

var configuration = context.Services.GetConfiguration();

context.Services.AddSingleton<IDistributedLockProvider>(sp =>

{

var connection = ConnectionMultiplexer

.Connect(configuration["Redis:Configuration"]);

return new

RedisDistributedSynchronizationProvider(connection.GetDatabase());

});

}

}

}

````

此代码从[配置](Configuration.md)获取Redis连接字符串, 因此你可以将以下行添加到`appsettings.json`文件:

````json

"Redis": {

"Configuration": "127.0.0.1"

}

````

## 使用

有两种方法可以使用分布式锁API: ABP的`IAbpDistributedLock`抽象和[DistributedLock](https://github.com/madelson/DistributedLock)库的API.

### 使用IAbpDistributedLock服务

`IAbpDistributedLock`是ABP框架提供的一个用于简单使用分布式锁的服务.

**\*\*实例: 使用**`IAbpDistributedLock.TryAcquireAsync`**方法\*\***

````csharp

using Volo.Abp.DistributedLocking;

namespace AbpDemo

{

public class MyService : ITransientDependency

{

private readonly IAbpDistributedLock \_distributedLock;

public MyService(IAbpDistributedLock distributedLock)

{

\_distributedLock = distributedLock;

}

public async Task MyMethodAsync()

{

await using (var handle =

await \_distributedLock.TryAcquireAsync("MyLockName"))

{

if (handle != null)

{

// your code that access the shared resource

}

}

}

}

}

````

`TryAcquireAsync`可能无法获取锁. 如果无法获取锁, 则返回`null`. 在这种情况下, 你不应该访问资源. 如果句柄不为`null`, 则表示你已获得锁, 并且可以安全地访问资源.

`TryAcquireAsync`方法拥有以下参数:

\* `name` (`string`, 必须): 锁的唯一名称. 不同的锁命名用于访问不同的资源.

\* `timeout` (`TimeSpan`): 等待获取锁的超时值. 默认值为`TimeSpan.Zero`, 这意味着如果锁已经被另一个应用程序拥有, 它不会等待.

\* `cancellationToken`: 取消令牌可在触发后取消操作.

### 配置

#### AbpDistributedLockOptions

`AbpDistributedLockOptions` 是配置分布式锁的主要选项类.

**\*\*示例: 设置应用程序的分布式锁Key前缀\*\***

```csharp

Configure<AbpDistributedLockOptions>(options =>

{

options.KeyPrefix = "MyApp1";

});

```

> 在你的[模块类](Module-Development-Basics.md)中的 `ConfigureServices` 方法进行配置.

##### 可用选项

\* KeyPrefix (string, 默认值: null): 指定分布式锁名称前缀.

### 使用DistributedLock库的API

ABP的`IAbpDistributedLock`服务非常有限, 主要用于ABP框架的内部使用. 对于你自己的应用程序, 可以使用DistributedLock库自己的API. 参见[文档](https://github.com/madelson/DistributedLock)详细信息.

## Volo.Abp.DistributedLocking.Abstractions库

如果你正在构建一个可重用的库或应用程序模块, 那么对于作为单个实例运行的简单应用程序, 你可能不希望为模块带来额外的依赖关系. 在这种情况下, 你的库可以依赖于[Volo.Abp.DistributedLocking.Abstractions](https://nuget.org/packages/Volo.Abp.DistributedLocking.Abstractions)库, 它定义了`IAbpDistributedLock`服务, 并将其在进程内实现(实际上不是分布式的). 通过这种方式, 你的库可以在作为单个实例运行的应用程序中正常运行(没有分布式锁提供程序依赖项). 如果应用程序部署到[集群环境](Deployment/Clustered-Environment.md), 那么应用程序开发人员应该安装一个真正的分布式提供程序, 如*\*安装\**部分所述.

## 虚拟文件系统

## 虚拟文件系统

虚拟文件系统使得管理物理上不存在于文件系统中(磁盘)的文件成为可能. 它主要用于将(js, css, image...)文件嵌入到程序集中, 并在运行时将它们像物理文件一样使用.

## 安装

> 大多数情况下你不需要手动安装这个包,因为[应用程序启动模板](Startup-Templates/Application.md)已经预先安装.

[Volo.Abp.VirtualFileSystem](https://www.nuget.org/packages/Volo.Abp.VirtualFileSystem) 是虚拟文件系统的NuGet主页.

使用ABP CLIi添加包到你的项目:

\* 安装[ABP CLI](https://docs.abp.io/en/abp/latest/CLI),如果你还没有安装.

\* 在你想要添加 `Volo.Abp.VirtualFileSystem` 包的项目的 `.csproj` 文件目录打开命令行(终端).

\* 运行 `abp add-package Volo.Abp.VirtualFileSystem` 命令.

如果你想要手动安装,安装[Volo.Abp.VirtualFileSystem](https://www.nuget.org/packages/Volo.Abp.VirtualFileSystem)NuGet包到你的项目并且添加`[DependsOn(typeof(AbpVirtualFileSystemModule))]`到你项目的[ABP Module](Module-Development-Basics.md)类.

## 与嵌入式文件工作

### 嵌入文件

要将文件嵌入到程序集中, 首先需要把该文件标记为**\*\*嵌入式资源\*\***. 最简单的方式是在 **\*\*解决方案管理器\*\*** 中选择文件, 然后找到 **\*\*"属性"\*\*** 窗口将 **\*\*"生成操作"\*\*** 设置为 **\*\*"嵌入式资源"\*\***.

例如:

![build-action-embedded-resource-sample](images/build-action-embedded-resource-sample.png)

如果需要添加多个文件, 这样做会很乏味. 作为选择, 你可以直接编辑 **\*\*.csproj\*\*** 文件:

````C#

<ItemGroup>

<EmbeddedResource Include="MyResources\\*\*\\*.\*" />

<Content Remove="MyResources\\*\*\\*.\*" />

</ItemGroup>

````

此配置以递归方式添加项目的 **\*\*MyResources\*\*** 文件夹下的所有文件(包括将来新添加的文件).

如果文件名包含一些特殊字符,在项目/程序集中嵌入文件可能会导致问题. 为了克服这个限制;

1. 将[Microsoft.Extensions.FileProviders.Embedded](https://www.nuget.org/packages/Microsoft.Extensions.FileProviders.Embedded) NuGet包添加到包含嵌入式资源的项目中.

2. 添加 `<GenerateEmbeddedFilesManifest>true</GenerateEmbeddedFilesManifest>` 到 `.csproj` 文件的 `<PropertyConfig>...</PropertyConfig>` 部分中.

> 尽管这两个步骤是可选的,并且ABP无需这些配置即可工作,但强烈建议你这样做.

### 配置AbpVirtualFileSystemOptions

使用 `AbpVirtualFileSystemOptions` [选项类](Options.md)可以在[模块](Module-Development-Basics.md)的 `ConfigureServices` 方法中将嵌入式文件注册到虚拟文件系统.

**\*\*示例: 添加嵌入式文件到虚拟文件系统\*\***

````csharp

Configure<AbpVirtualFileSystemOptions>(options =>

{

options.FileSets.AddEmbedded<MyModule>();

});

````

`AddEmbedded` 扩展方法需要一个类, 从给定**\*\*类的程序集中查找所有嵌入文件\*\***, 并将它们注册到虚拟文件系统. 通常将模块类作为通用参数传递.

`AddEmbedded` 有两个可选参数;

\* `baseNamespace`: 仅在你没有配置上面说明的 `GenerateEmbeddedFilesManifest` 并且你的根名称空间不为空时,才可能需要这样做. 在这种情况下,请在此处设置你的根名称空间.

\* `baseFolder`: 如果你不想公开项目中的所有嵌入式文件,只希望公开特定的文件夹(和子文件夹/文件),可以相对于项目根页面设置基本文件夹.

**\*\*示例: 添加项目中** `MyFiles` **目录下的文件\*\***

````csharp

Configure<AbpVirtualFileSystemOptions>(options =>

{

options.FileSets.AddEmbedded<MyModule>(

baseNamespace: "Acme.BookStore.MyFiles",

baseFolder: "/MyFiles"

);

});

````

这个例子假设;

\* 你的项目根(default)命令空间是 `Acme.BookStore`.

\* 你的项目有一个名为 `MyFiles` 的目录.

\* 你只想添加 `MyFiles` 目录到虚拟文件系统.

## IVirtualFileProvider

将文件嵌入到程序集中并注册到虚拟文件系统后,可以使用 `IVirtualFileProvider` 接口来获取文件或目录内容:

````C#

public class MyService : ITransientDependency

{

private readonly IVirtualFileProvider \_virtualFileProvider;

public MyService(IVirtualFileProvider virtualFileProvider)

{

\_virtualFileProvider = virtualFileProvider;

}

public void Test()

{

//Getting a single file

var file = \_virtualFileProvider

.GetFileInfo("/MyResources/js/test.js");

var fileContent = file.ReadAsString();

//Getting all files/directories under a directory

var directoryContents = \_virtualFileProvider

.GetDirectoryContents("/MyResources/js");

}

}

````

### ASP.NET Core 集成

虚拟文件系统与 ASP.NET Core 无缝集成:

\* 虚拟文件可以像Web应用程序上的物理(静态)文件一样使用.

\* Js, css, 图像文件和所有其他Web内容可以嵌入到程序集中并像物理文件一样使用.

\* 应用程序(或其他模块)可以**\*\*覆盖模块的虚拟文件\*\***, 就像将具有相同名称和扩展名的文件放入虚拟文件的同一文件夹中一样.

#### 静态虚拟文件夹

默认情况下,ASP.NET Core仅允许 `wwwroot` 文件夹包含客户端使用的静态文件. 当你使用虚拟文件系统时以下文件夹也可以包含静态文件:

\* Pages

\* Views

\* Components

\* Themes

这允许你可以在 `.cshtml` 文件附近添加 `.js`, `.css`... 文件,更易于开发和维护你的项目.

#### 在开发过程中处理嵌入式文件

将文件嵌入到模块程序集中并能够通过引用程序集(或添加nuget包)在另一个项目中使用它对于创建可重用模块非常有价值. 但是, 这使得开发模块本身变得有点困难.

假设你正在开发一个包含嵌入式JavaScript文件的模块. 当你更改文件时, 你必须重新编译项目, 重新启动应用程序并刷新浏览器页面以使更改生效. 显然, 这是非常耗时和乏味的.

我们需要的是应用程序在开发时直接使用物理文件的能力, 让浏览器刷新时同步JavaScript文件的任何更改. `ReplaceEmbeddedByPhysical` 方法使其成为可能.

下面的示例展示了应用程序依赖于包含嵌入文件的模块("MyModule"), 并且应用程序可以在开发过程中直接使用模块的源代码.

````C#

[DependsOn(typeof(MyModule))]

public class MyWebAppModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

var hostingEnvironment = context.Services.GetHostingEnvironment();

if (hostingEnvironment.IsDevelopment()) //only for development time

{

Configure<AbpVirtualFileSystemOptions>(options =>

{

options.FileSets.ReplaceEmbeddedByPhysical<MyModule>(

Path.Combine(

hostingEnvironment.ContentRootPath,

string.Format(

"..{0}MyModuleProject",

Path.DirectorySeparatorChar

)

)

);

});

}

}

}

````

上面的代码假设`MyWebAppModule`和`MyModule`是Visual Studio解决方案中的两个不同的项目, `MyWebAppModule`依赖于`MyModule`.

> [应用程序启动模板]已经为本地化文件应用这个方法,所以当你更改一个本地化文件时,它会自动检测到更改.

## 替换/重写虚拟文件

虚拟文件系统在运行时创建一个统一的文件系统,其中实际的文件在开发时被分配到不同的模块中.

如果两个模块将文件添加到相同的虚拟路径(如`my-path/my-file.css`),之后添加的模块将替换/替换前一个([模块依赖](Module-Development-Basics.md)顺序决定了添加文件的顺序).

此功能允许你的应用程序可以覆盖/替换定义应用程序所使用的模块的任何虚拟文件. 这是ABP框架的基本可扩展性功能之一.

因此,如果需要替换模块的文件,只需在模块/应用程序中完全相同的路径中创建该文件.

### 物理文件

物理文件总是覆盖虚拟文件. 这意味着如果你把一个文件放在 `/wwwroot/my-folder/my-file.css`,它将覆盖虚拟文件系统相同位置的文件.因此你需要知道在模块中定义的文件路径来覆盖它们.

## 审计日志

# 审计日志

[维基百科](https://en.wikipedia.org/wiki/Audit\_trail): "*\*审计跟踪(也称为****\*\*审计日志\*\*****)是一种安全相关的按时间顺序记录,记录集或记录目的和来源. 这种记录提供了在任何特定时间的操作,过程或事件产生影响活动顺序的文件证据\** ".

ABP框架提供一个可扩展的**\*\*审计日志系统\*\***,自动化的根据**\*\*约定\*\***记录审计日志,并提供**\*\*配置\*\***控制审计日志的级别.

一个**\*\*审计日志对象\*\***(参见下面的审计日志对象部分)通常是针对每个web请求创建和保存的.包括;

\* **\*\*请求和响应的细节\*\*** (如URL,HTTP方法,浏览器信息,HTTP状态代码...等).

\* **\*\*执行的动作\*\*** (控制器操作和应用服务方法调用及其参数).

\* **\*\*实体的变化\*\*** (在Web请求中).

\* **\*\*异常信息\*\*** (如果在执行请求发生操作).

\* **\*\*请求时长\*\*** (测量应用程序的性能).

> [启动模板](Startup-Templates/Index.md)已经将审计日志系统配置为适用于大多数应用程序. 本文档介绍了对审计日志系统更精细的控制.

## 数据库提供程序支持

\* [Entity Framework Core](Entity-Framework-Core.md)提供程序完全支持.

\* [MongoDB](MongoDB.md)提供程序不支持实体更改审计记录. 其他功能按预期工作.

## UseAuditing()

`UseAuditing()` 中间件应该被添加到ASP.NET Core请求管道,用于创建和保存审计日志. 如果你使用[启动模板](Startup-Templates/Index.md)创建的应用程序,它已经默认添加.

## AbpAuditingOptions

`AbpAuditingOptions` 是配置审计日志系统的主要[options对象](Options.md). 你可以在[模块](Module-Development-Basics.md)的 `ConfigureServices` 方法中进行配置:

````csharp

Configure<AbpAuditingOptions>(options =>

{

options.IsEnabled = false; //Disables the auditing system

});

````

这里是你可以配置的选项列表:

\* `IsEnabled` (默认值: `true`): 启用或禁用审计系统的总开关. 如果值为 `false`,则不使用其他选项.

\* `HideErrors` (默认值: `true`): 在保存审计日志对象时如果发生任何错误,审计日志系统会将错误隐藏并写入常规[日志](Logging.md). 如果保存审计日志对系统非常重要那么将其设置为 `false` 以便在隐藏错误时抛出异常.

\* `IsEnabledForAnonymousUsers` (默认值: `true`): 如果只想为经过身份验证的用户记录审计日志,请设置为 `false`.如果为匿名用户保存审计日志,你将看到这些用户的 `UserId` 值为 `null`.

\* `AlwaysLogOnException`(默认值: `true`): 如果设置为 `true`,将始终在异常/错误情况下保存审计日志,不检查其他选项(`IsEnabled` 除外,它完全禁用了审计日志).

\* `IsEnabledForGetRequests` (默认值: `false`): HTTP GET请求通常不应该在数据库进行任何更改,审计日志系统不会为GET请求保存审计日志对象. 将此值设置为 `true` 可为GET请求启用审计日志系统.

\* `ApplicationName`: 如果有多个应用程序保存审计日志到单一的数据库,使用此属性设置为你的应用程序名称区分不同的应用程序日志.

\* `DisableLogActionInfo` (默认值: `false`): 如果设置为 `true`, 将不再记录 `AuditLogActionInfo`.

\* `IgnoredTypes`: 审计日志系统忽略的 `Type` 列表. 如果它是实体类型,则不会保存此类型实体的更改. 在序列化操作参数时也使用此列表.

\* `EntityHistorySelectors`:选择器列表,用于确定是否选择了用于保存实体更改的实体类型. 有关详细信息请参阅下面的部分.

\* `Contributors`: `AuditLogContributor` 实现的列表. 贡献者是扩展审计日志系统的一种方式. 有关详细信息请参阅下面的"审计日志贡献者"部分.

### 实体历史选择器

保存你的所有实体的所有变化将需要大量的数据库空间. 出于这个原因**\*\*审计日志系统不保存为实体的任何改变,除非你明确地对其进行配置\*\***.

要保存的所有实体的所有更改,只需使用 `AddAllEntities()` 扩展方法.

````csharp

Configure<AbpAuditingOptions>(options =>

{

options.EntityHistorySelectors.AddAllEntities();

});

````

`options.EntityHistorySelectors` 实际上是一个类型谓词的列表,你可以写一个lambda表达式定义过滤器.

下面的示例中与使用 `AddAllEntities()` 扩展方法效果相同:

````csharp

Configure<AbpAuditingOptions>(options =>

{

options.EntityHistorySelectors.Add(

new NamedTypeSelector(

"MySelectorName",

type =>

{

if (typeof(IEntity).IsAssignableFrom(type))

{

return true;

}

else

{

return false;

}

}

)

);

});

````

条件 `typeof(IEntity).IsAssignableFrom(type)` 对于任何实现 `IEntity` 接口的类(从技术上来这些都是你应用程序中的实体) 结果都为 `true` . 你可以根据自己的逻辑编写条件并返回 `true` 或 `false`.

`options.EntityHistorySelectors` 是一种灵活动态的选择实体进行审计日志记录的方法. 另一种方法是为每个实体使用 `Audited` 和 `DisableAuditing` attribute.

## 启用/禁用审计日志服务

### 启用/禁用 Controllers & Actions

默认所有的控制器动作都会被记录下来(有关GET请求,请参阅上面的 `IsEnabledForGetRequests` ).

你可以使用 `[DisableAuditing]` 来禁用特定的控制器:

````csharp

[DisableAuditing]

public class HomeController : AbpController

{

//...

}

````

使用 `[DisableAuditing]` 在action级别控制:

````csharp

public class HomeController : AbpController

{

[DisableAuditing]

public async Task<ActionResult> Home()

{

//...

}

public async Task<ActionResult> OtherActionLogged()

{

//...

}

}

````

### 启用/禁用 应用服务&方法

[应用服务](Application-Services.md)也默认包含在审计日志中. 你可在服务或方法级别使用 `[DisableAuditing]`．

#### 启用/禁用 其他服务

可以为任何类型的类(注册到[依赖注入](Dependency-Injection.md)并从依赖注入解析)启用审计日志,默认情况下仅对控制器和应用程序服务启用.

对于任何需要被审计记录的类或方法都可以使用 `[Audited]` 和`IAuditingEnabled`.此外,你的类可以(直接或固有的)实现 `IAuditingEnabled` 接口以认启用该类的审计日志记录.

### 启用/禁用 实体 & 属性

以下情况下实体在实体更改审计日志记录中忽略实体;

\* 如果将实体类型添加到 `AbpAuditingOptions.IgnoredTypes`(如前所述),它在审计日志系统中被完全忽略.

\* 如果对象不是[实体](Entities.md)(没有直接或固有的实现 `IEntity` - 所有实体默认实现这个接口).

\* 如果实体访问级别不是public的.

你可以使用 `Audited` 来启用实体更改审计日志:

````csharp

[Audited]

public class MyEntity : Entity<Guid>

{

//...

}

````

或者禁用实体:

````csharp

[DisableAuditing]

public class MyEntity : Entity<Guid>

{

//...

}

````

只有前面提到的 `AbpAuditingOptions.EntityHistorySelector` 选择实体时才有必要禁用审计日志记录.

你可以仅禁用实体的某些属性的审计,以审计日志记录进行精细控制:

````csharp

[Audited]

public class MyUser : Entity<Guid>

{

public string Name { get; set; }

public string Email { get; set; }

[DisableAuditing] //Ignore the Passoword on audit logging

public string Password { get; set; }

}

````

审计日志系统保存 `MyUser` 实体的更改,出于安全的目的忽略 `Password` 属性.

在某些情况下你可能要保存一些属性,但忽略所有其他属性. 为忽略的属性编写 `[DisableAuditing]` 将很乏味. 这种情况下将 `[Audited]` 用于所需的属性,使用 `[DisableAuditing]` 属性标记该实体:

````csharp

[DisableAuditing]

public class MyUser : Entity<Guid>

{

[Audited] //Only log the Name change

public string Name { get; set; }

public string Email { get; set; }

public string Password { get; set; }

}

````

## IAuditingStore

`IAuditingStore` 是一个接口,用于保存ABP框架的审计日志对象(下面说明). 如果需要将审计日志对象保存到自定义数据存储中,可以在自己的应用程序中实现 `IAuditingStore` 并在[依赖注入系统](Dependency-Injection.md)替换.

如果没有注册审计存储,则使用 `SimpleLogAuditingStore`. 它只是将审计对象写入标准[日志系统](Logging.md).

[审计日志模块](Modules/Audit-Logging.md)已在[启动模板](Startup-Templates/Index.md)中配置,它将审计日志对象保存到数据库中(支持多个数据库提供程序). 所以大多数时候你并不需要关心 `IAuditingStore` 是如何实现和使用的.

## 审计日志对象

默认为每个**\*\*web请求\*\***创建一个**\*\*审计日志对象\*\***,审计日志对象可以由以下关系图表示:

![\*\*auditlog-object-diagram\*\*](images/auditlog-object-diagram.png)

\* **\*\*AuditLogInfo\*\***: 具有以下属性:

\* `ApplicationName`: 当你保存不同的应用审计日志到同一个数据库,这个属性用来区分应用程序.

\* `UserId`:当前用户的Id,用户未登录为 `null`.

\* `UserName`:当前用户的用户名,如果用户已经登录(这里的值不依赖于标识模块/系统进行查找).

\* `TenantId`: 当前租户的Id,对于多租户应用.

\* `TenantName`: 当前租户的名称,对于多租户应用.

\* `ExecutionTime`: 审计日志对象创建的时间.

\* `ExecutionDuration`: 请求的总执行时间,以毫秒为单位. 可以用来观察应用程序的性能.

\* `ClientId`: 当前客户端的Id,如果客户端已经通过认证.客户端通常是使用HTTP API的第三方应用程序.

\* `ClientName`: 当前客户端的名称,如果有的话.

\* `ClientIpAddress`: 客户端/用户设备的IP地址.

\* `CorrelationId`: 当前[相关Id](CorrelationId.md). 相关Id用于在单个逻辑操作中关联由不同应用程序(或微服务)写入的审计日志.

\* `BrowserInfo`: 当前用户的浏览器名称/版本信息,如果有的话.

\* `HttpMethod`: 当前HTTP请求的方法(GET,POST,PUT,DELETE ...等).

\* `HttpStatusCode`: HTTP响应状态码.

\* `Url`: 请求的URL.

\* **\*\*AuditLogActionInfo\*\***: 一个 审计日志动作通常是web请求期间控制器动作或[应用服务](Application-Services.md)方法调用. 一个审计日志可以包含多个动作. 动作对象具有以下属性:

\* `ServiceName`:执行的控制器/服务的名称.

\* `MethodName`:控制器/服务执行的方法的名称.

\* `Parameters`:传递给方法的参数的JSON格文本.

\* `ExecutionTime`: 执行的时间.

\* `ExecutionDuration`: 方法执行时长,以毫秒为单位. 可以用来观察方法的性能.

\* **\*\*EntityChangeInfo\*\***: 表示一个实体在Web请求中的变更. 审计日志可以包含0个或多个实体的变更. 实体变更具有以下属性:

\* `ChangeTime`: 当实体被改变的时间.

\* `ChangeType`:具有以下字段的枚举: `Created`(0), `Updated`(1)和 `Deleted`(2).

\* `EntityId`: 更改实体的Id.

\* `EntityTenantId`:实体所属的租户Id.

\* `EntityTypeFullName`: 实体的类型(类)的完整命名空间名称(例如Book实体的*\*Acme.BookStore.Book\**.

\* **\*\*EntityPropertyChangeInfo\*\***: 表示一个实体的属性的更改.一个实体的更改信息(上面已说明)可含有具有以下属性的一个或多个属性的更改:

\* `NewValue`: 属性的新值. 如果实体已被删除为 `null`.

\* `OriginalValue`:变更前旧/初始值. 如果实体是新创建为 `null`.

\* `PropertyName`: 实体类的属性名称.

\* `PropertyTypeFullName`:属性类型的完整命名空间名称.

\* **\*\*Exception\*\***: 审计日志对象可能包含零个或多个异常. 可以得到失败请求的异常信息.

\* **\*\*Comment\*\***:用于将自定义消息添加到审计日志条目的任意字符串值. 审计日志对象可能包含零个或多个注释.

除了上面说明的标准属性之外,`AuditLogInfo`, `AuditLogActionInfo` 和 `EntityChangeInfo` 对象还实现了`IHasExtraProperties` 接口,你可以向这些对象添加自定义属性.

## 审计日志贡献者

你可以创建类继承 `AuditLogContributor`类 来扩展审计系统,该类定义了 `PreContribute` 和 `PostContribute` 方法.

唯一预构建的贡献者是 `AspNetCoreAuditLogContributor` 类,它设置HTTP请求的相关属性.

贡献者可以设置 `AuditLogInfo` 类的属性和集合来添加更多信息.

例:

````csharp

public class MyAuditLogContributor : AuditLogContributor

{

public override void PreContribute(AuditLogContributionContext context)

{

var currentUser = context.ServiceProvider.GetRequiredService<ICurrentUser>();

context.AuditInfo.SetProperty(

"MyCustomClaimValue",

currentUser.FindClaimValue("MyCustomClaim")

);

}

public override void PostContribute(AuditLogContributionContext context)

{

context.AuditInfo.Comments.Add("Some comment...");

}

}

````

\* `context.ServiceProvider` 可以从[依赖注入系统](Dependency-Injection.md)中解析服务.

\* `context.AuditInfo` 可以用来访问当前审计日志的对象并进行操作.

创建贡献者后,需要将其添加到 `AbpAuditingOptions.Contributors` 列表中:

````csharp

Configure<AbpAuditingOptions>(options =>

{

options.Contributors.Add(new MyAuditLogContributor());

});

````

## IAuditLogScope & IAuditingManager

本节介绍用于高级用例的 `IAuditLogScope` 和 `IAuditingManager` 服务.

**\*\*审计日志范围\*\***是**\*\*构建\*\***和**\*\*保存\*\***审计日志对象的[环境范围](Ambient-Context-Pattern.md)(前面解释过). 默认审计日志中间件会为Web请求创建审计日志范围(请参阅上面的 `UseAuditing()` 部分).

### 获取当前审计日志范围

上面提到,审计日志贡献者是操作审计日志对象的全局方法. 你可从服务中获得值.

如果需要在应用程序的任意位置上操作审计日志对象,可以访问当前审计日志范围并获取当前审计日志对象(与范围的管理方式无关).

例:

````csharp

public class MyService : ITransientDependency

{

private readonly IAuditingManager \_auditingManager;

public MyService(IAuditingManager auditingManager)

{

\_auditingManager = auditingManager;

}

public async Task DoItAsync()

{

var currentAuditLogScope = \_auditingManager.Current;

if (currentAuditLogScope != null)

{

currentAuditLogScope.Log.Comments.Add(

"Executed the MyService.DoItAsync method :)"

);

currentAuditLogScope.Log.SetProperty("MyCustomProperty", 42);

}

}

}

````

总是检查 `\_auditingManager.Current` 是否为空,因为它是在外部范围中控制的,在调用方法之前你不知道是否创建了审计日志范围.

### 手动创建审计日志范围

你很少需要手动创建审计日志的范围,但如果你需要,可以使用 `IAuditingManager` 创建审计日志的范围.

例:

````csharp

public class MyService : ITransientDependency

{

private readonly IAuditingManager \_auditingManager;

public MyService(IAuditingManager auditingManager)

{

\_auditingManager = auditingManager;

}

public async Task DoItAsync()

{

using (var auditingScope = \_auditingManager.BeginScope())

{

try

{

//Call other services...

}

catch (Exception ex)

{

//Add exceptions

\_auditingManager.Current.Log.Exceptions.Add(ex);

}

finally

{

//Always save the log

await auditingScope.SaveAsync();

}

}

}

}

````

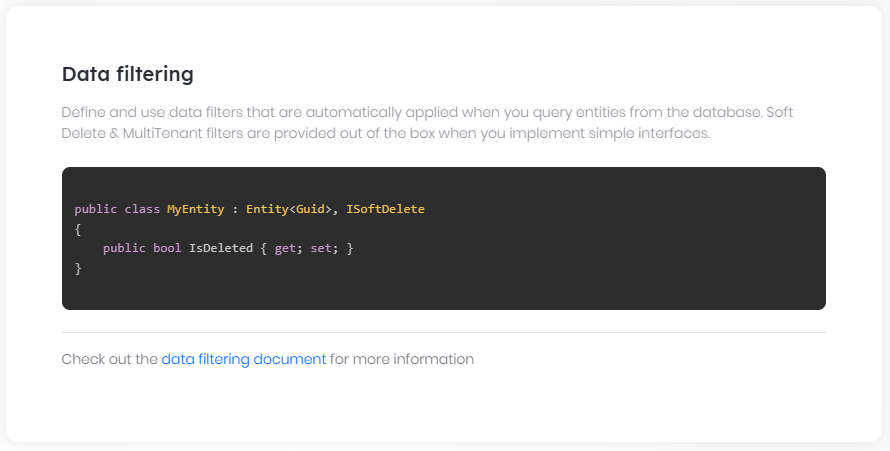
你可以调用其他服务,它们可能调用其他服务,它们可能更改实体,等等. 所有这些交互都保存为finally块中的一个审计日志对象.

## 审计日志模块

审计日志模块基本上实现了 `IAuditingStore`, 将审计日志对象保存到数据库中并支持多个数据库提供程序. 默认此模块已添加到启动模板中.

参见[审计日志模块文档](Modules/Audit-Logging.md)了解更多.

## 数据过滤



# 数据过滤

[Volo.Abp.Data](https://www.nuget.org/packages/Volo.Abp.Data) 包定义了在查询数据库时自动过滤数据的服务.

## 预定义的过滤

ABP定义了一些开箱即用的过滤.

### ISoftDelete

将[实体](Entities.md)标记为已删除,并不是物理删除. 实现 `ISoftDelete` 接口将你的实体"软删除".

示例:

````csharp

using System;

using Volo.Abp;

using Volo.Abp.Domain.Entities;

namespace Acme.BookStore

{

public class Book : AggregateRoot<Guid>, ISoftDelete

{

public string Name { get; set; }

public bool IsDeleted { get; set; } //Defined by ISoftDelete

}

}

````

`ISoftDelete` 定义了 `IsDeleted` 属性. 当你使用[仓储](Repositories.md)删除一条记录时, ABP会自动将 `IsDeleted` 设置为true,并将删除操作替换为修改操作(如果需要,也可以手动将 `IsDeleted` 设置为true). 在查询数据库时会**\*\*自动过滤软删除的实体\*\***.

> `ISoftDelete` 过滤默认启用, 想要真正的从数据库删除实体需要显示的禁用过滤. 参见下面提到的 `IDataFilter` 服务.

### IMultiTenant

[多租户](Multi-Tenancy.md) 是创建 SaaS 应用程序的有效方法. 多租户应用程序通常需要在租户间隔离数据. 实现 `IMultiTenant` 接口使你的实体支持 "多租户".

示例:

````csharp

using System;

using Volo.Abp;

using Volo.Abp.Domain.Entities;

using Volo.Abp.MultiTenancy;

namespace Acme.BookStore

{

public class Book : AggregateRoot<Guid>, ISoftDelete, IMultiTenant

{

public string Name { get; set; }

public bool IsDeleted { get; set; } //Defined by ISoftDelete

public Guid? TenantId { get; set; } //Defined by IMultiTenant

}

}

````

`IMultiTenant` 接口定义了 `TenantId` 属性用于自动过滤当前租户实体. 更多信息参见[多租户](Multi-Tenancy.md)文档.

## IDataFilter 服务: 启用/禁用 数据过滤

你可以使用 `IDataFilter` 服务控制数据过滤.

示例:

````csharp

using System;

using System.Collections.Generic;

using System.Threading.Tasks;

using Volo.Abp;

using Volo.Abp.Data;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Domain.Repositories;

namespace Acme.BookStore

{

public class MyBookService : ITransientDependency

{

private readonly IDataFilter \_dataFilter;

private readonly IRepository<Book, Guid> \_bookRepository;

public MyBookService(

IDataFilter dataFilter,

IRepository<Book, Guid> bookRepository)

{

\_dataFilter = dataFilter;

\_bookRepository = bookRepository;

}

public async Task<List<Book>> GetAllBooksIncludingDeletedAsync()

{

//Temporary disable the ISoftDelete filter

using (\_dataFilter.Disable<ISoftDelete>())

{

return await \_bookRepository.GetListAsync();

}

}

}

}

````

\* [注入](Dependency-Injection.md) `IDataFilter` 服务到你的类中.

\* 在 `using` 语句中使用 `Disable` 方法创建一个代码块,其中禁用了 `ISoftDelete` 过滤器(始终与 `using` 搭配使用,确保代码块执行后将过滤重置为之前的状态).

`IDataFilter.Enable` 方法可以启用过滤. 可以嵌套使用 `Enable` 和 `Disable` 方法定义内部作用域.

## AbpDataFilterOptions

`AbpDataFilterOptions` 用于设置数据过滤系统[选项](Options.md).

下面的示例代码在默认情况下禁用了 `ISoftDelete` 过滤,除非显示启用,在查询数据库时会包含标记为已删除的实体:

````csharp

Configure<AbpDataFilterOptions>(options =>

{

options.DefaultStates[typeof(ISoftDelete)] = new DataFilterState(isEnabled: false);

});

````

> 更改全局过滤的默认值需要小心,特别是在你使用预构建的模块时该模块可能是在默认启用软删除过滤的情况下开发的. 但你可以安全的为自己定义的数据过滤执行此操作.

## 自定义数据过滤

定义和实现新的过滤很大程序上取决与数据库提供者. ABP为所有的数据库提供者实现了预构建的过滤.

首先为过滤定义一个接口 (如 `ISoftDelete` 和 `IMultiTenant`) 然后用实体实现它.

示例:

````csharp

public interface IIsActive

{

bool IsActive { get; }

}

````

`IIsActive` 接口可以过滤活跃/消极数据,任何[实体](Entities.md)都可以实现它:

````csharp

public class Book : AggregateRoot<Guid>, IIsActive

{

public string Name { get; set; }

public bool IsActive { get; set; } //Defined by IIsActive

}

````

### EntityFramework Core

ABP使用[EF Core的全局过滤](https://docs.microsoft.com/en-us/ef/core/querying/filters)系统用于[EF Core 集成](Entity-Framework-Core.md). 所以它很好的集成到EF Core中,即使你直接使用 `DbContext` 它也可以正常工作.

实现自定义过滤的最佳方法是为重写你的 `DbContext` 的 `ShouldFilterEntity` 和 `CreateFilterExpression` 方法. 示例:

````csharp

protected bool IsActiveFilterEnabled => DataFilter?.IsEnabled<IIsActive>() ?? false;

protected override bool ShouldFilterEntity<TEntity>(IMutableEntityType entityType)

{

if (typeof(IIsActive).IsAssignableFrom(typeof(TEntity)))

{

return true;

}

return base.ShouldFilterEntity<TEntity>(entityType);

}

protected override Expression<Func<TEntity, bool>> CreateFilterExpression<TEntity>()

{

var expression = base.CreateFilterExpression<TEntity>();

if (typeof(IIsActive).IsAssignableFrom(typeof(TEntity)))

{

Expression<Func<TEntity, bool>> isActiveFilter =

e => !IsActiveFilterEnabled || EF.Property<bool>(e, "IsActive");

expression = expression == null

? isActiveFilter

: CombineExpressions(expression, isActiveFilter);

}

return expression;

}

````

\* 添加 `IsActiveFilterEnabled` 属性用于检查是否启用了 `IIsActive` . 内部使用了之前介绍到的 `IDataFilter` 服务.

\* 重写 `ShouldFilterEntity` 和 `CreateFilterExpression` 方法检查给定实体是否实现 `IIsActive` 接口,在必要时组合表达式.

### MongoDB

ABP抽象了 `IMongoDbRepositoryFilterer` 接口为[MongoDB 集成](MongoDB.md)实现数据过滤, 只有正确的使用仓储,它才会工作. 否则你需要手动过滤数据.

目前为MongoDB集成实现数据过滤的最佳方法是重写派生自 `MongoDbRepositoryFilterer` 基类的 `FilterQueryable` 方法:

````csharp

[ExposeServices(typeof(IMongoDbRepositoryFilterer<Book, Guid>))]

public class BookMongoDbRepositoryFilterer : MongoDbRepositoryFilterer<Book, Guid> , ITransientDependency

{

public BookMongoDbRepositoryFilterer(

IDataFilter dataFilter,

ICurrentTenant currentTenant) :

base(dataFilter, currentTenant)

{

}

public override TQueryable FilterQueryable<TQueryable>(TQueryable query)

{

if (DataFilter.IsEnabled<IIsActive>())

{

return (TQueryable)query.Where(x => x.IsActive);

}

return base.FilterQueryable(query);

}

}

````

示例中仅为 `Book` 实体实现了过滤. 如果你想要为所有的实体实现过滤 (实现了 `IIsActive` 接口的实体),可以创建自己的 `MongoDbRepositoryFilterer` 基类并重 写 `AddGlobalFilters` 方法. 如下所示:

````csharp

public abstract class MyMongoRepository<TMongoDbContext, TEntity, TKey> : MongoDbRepository<TMongoDbContext, TEntity, TKey>

where TMongoDbContext : IAbpMongoDbContext

where TEntity : class, IEntity<TKey>

{

protected MyMongoRepository(IMongoDbContextProvider<TMongoDbContext> dbContextProvider)

: base(dbContextProvider)

{

}

protected override void AddGlobalFilters(List<FilterDefinition<TEntity>> filters)

{

base.AddGlobalFilters(filters);

if (typeof(IIsActive).IsAssignableFrom(typeof(TEntity))

&& DataFilter.IsEnabled<IIsActive>())

{

filters.Add(Builders<TEntity>.Filter.Eq(e => ((IIsActive)e).IsActive, true));

}

}

}

public class MyMongoDbModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

//.......

context.Services

.Replace(ServiceDescriptor.Transient(typeof(IMongoDbRepositoryFilterer<,>),typeof(MyMongoDbRepositoryFilterer<,>)));

}

}

````

> 参阅[MongoDb集成文档](MongoDB.md)的 "设置默认仓储" 部分了解如何使用自定义类替换默认仓储.

## 当前用户

# 当前用户

在Web应用程序中检索有关已登录用户的信息是很常见的. 当前用户是与Web应用程序中的当前请求相关的活动用户.

## ICurrentUser

`ICurrentUser` 是主要的服务,用于获取有关当前活动的用户信息.

示例: [注入](Dependency-Injection.md) `ICurrentUser` 到服务中:

````csharp

using System;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Users;

namespace AbpDemo

{

public class MyService : ITransientDependency

{

private readonly ICurrentUser \_currentUser;

public MyService(ICurrentUser currentUser)

{

\_currentUser = currentUser;

}

public void Foo()

{

Guid? userId = \_currentUser.Id;

}

}

}

````

公共基类已经将此服务作为基本属性注入. 例如你可以直接在[应用服务](Application-Services.md)中使用 `CurrentUser` 属性:

````csharp

using System;

using Volo.Abp.Application.Services;

namespace AbpDemo

{

public class MyAppService : ApplicationService

{

public void Foo()

{

Guid? userId = CurrentUser.Id;

}

}

}

````

### 属性

以下是 `ICurrentUser` 接口的基本属性:

\* **\*\*IsAuthenticated\*\*** 如果当前用户已登录(已认证),则返回 `true`. 如果用户尚未登录,则 `Id` 和 `UserName` 将返回 `null`.

\* **\*\*Id\*\*** (Guid?): 当前用户的Id,如果用户未登录,返回 `null`.

\* **\*\*UserName\*\*** (string): 当前用户的用户名称. 如果用户未登录,返回 `null`.

\* **\*\*TenantId\*\*** (Guid?): 当前用户的租户Id. 对于[多租户](Multi-Tenancy.md) 应用程序很有用. 如果当前用户未分配给租户,返回 `null`.

\* **\*\*Email\*\*** (string): 当前用户的电子邮件地址. 如果当前用户尚未登录或未设置电子邮件地址,返回 `null`.

\* **\*\*EmailVerified\*\*** (bool): 如果当前用户的电子邮件地址已经过验证,返回 `true`.

\* **\*\*PhoneNumber\*\*** (string): 当前用户的电话号码. 如果当前用户尚未登录或未设置电话号码,返回 `null`.

\* **\*\*PhoneNumberVerified\*\*** (bool): 如果当前用户的电话号码已经过验证,返回 `true`.

\* **\*\*Roles\*\*** (string[]): 当前用户的角色. 返回当前用户角色名称的字符串数组.

### Methods

`ICurrentUser` 是在 `ICurrentPrincipalAccessor` 上实现的(请参阅以下部分),并可以处理声明. 实际上所有上述属性都是从当前经过身份验证的用户的声明中检索的.

如果你有自定义声明或获取其他非常见声明类型, `ICurrentUser` 有一些直接使用声明的方法.

\* **\*\*FindClaim\*\***: 获取给定名称的声明,如果未找到返回 `null`.

\* **\*\*FindClaims\*\***: 获取具有给定名称的所有声明(允许具有相同名称的多个声明值).

\* **\*\*GetAllClaims\*\***: 获取所有声明.

\* **\*\*IsInRole\*\***: 一种检查当前用户是否在指定角色中的简化方法.

除了这些标准方法,还有一些扩展方法:

\* **\*\*FindClaimValue\*\***: 获取具有给定名称的声明的值,如果未找到返回 `null`. 它有一个泛型重载将值强制转换为特定类型.

\* **\*\*GetId\*\***: 返回当前用户的 `Id`. 如果当前用户没有登录它会抛出一个异常(而不是返回`null`). 仅在你确定用户已经在你的代码上下文中进行了身份验证时才使用此选项.

### 验证和授权

`ICurrentUser` 的工作方式与用户的身份验证或授权方式无关. 它可以与使用当前主体的任何身份验证系统无缝地配合使用(请参阅下面的部分).

## ICurrentPrincipalAccessor

`ICurrentPrincipalAccessor` 是当需要当前用户的Principal时使用的服务(由ABP框架和你的应用程序代码使用).

对于Web应用程序, 它获取当前 `HttpContext` 的 `User` 属性,对于非Web应用程序它将返回 `Thread.CurrentPrincipal`.

> 通常你不需要这种低级别的 `ICurrentPrincipalAccessor` 服务,直接使用上述的 `ICurrentUser` 即可.

### 基本用法

你可以注入 `ICurrentPrincipalAccessor` 并且使用 `Principal` 属性获取当前principal:

````csharp

public class MyService : ITransientDependency

{

private readonly ICurrentPrincipalAccessor \_currentPrincipalAccessor;

public MyService(ICurrentPrincipalAccessor currentPrincipalAccessor)

{

\_currentPrincipalAccessor = currentPrincipalAccessor;

}

public void Foo()

{

var allClaims = \_currentPrincipalAccessor.Principal.Claims.ToList();

//...

}

}

````

### 更改当前Principal

除了某些高级场景外,你不需要设置或更改当前Principal. 如果需要可以使用 `ICurrentPrincipalAccessor` 的 `Change` 方法. 它接受一个 `ClaimsPrincipal` 对象并使其成为作用域的"当前"对象.

示例:

````csharp

public class MyAppService : ApplicationService

{

private readonly ICurrentPrincipalAccessor \_currentPrincipalAccessor;

public MyAppService(ICurrentPrincipalAccessor currentPrincipalAccessor)

{

\_currentPrincipalAccessor = currentPrincipalAccessor;

}

public void Foo()

{

var newPrincipal = new ClaimsPrincipal(

new ClaimsIdentity(

new Claim[]

{

new Claim(AbpClaimTypes.UserId, Guid.NewGuid().ToString()),

new Claim(AbpClaimTypes.UserName, "john"),

new Claim("MyCustomCliam", "42")

}

)

);

using (\_currentPrincipalAccessor.Change(newPrincipal))

{

var userName = CurrentUser.UserName; //returns "john"

//...

}

}

}

````

始终在 `using` 语句中使用 `Change` 方法,在 `using` 范围结束后它将恢复为原始值.

这可以是一种模拟用户登录的应用程序代码范围的方法,但是请尝试谨慎使用它.

## AbpClaimTypes

`AbpClaimTypes` 是一个静态类它定义了标准声明的名称被ABP框架使用.

\* `UserName`, `UserId`, `Role` 和 `Email` 属性的默认值是通常[System.Security.Claims.ClaimTypes](https://docs.microsoft.com/en-us/dotnet/api/system.security.claims.claimtypes)类设置的, 但你可以改变它们.

\* 其他属性,如 `EmailVerified`, `PhoneNumber`, `TenantId` ...是由ABP框架通过尽可能遵循标准名称来定义的.

建议使用这个类的属性来代替声明名称的魔术字符串.

## 对象到对象映射



# 对象到对象映射

将对象映射到另一个对象是常用并且繁琐重复的工作,大部分情况下两个类都具有相同或相似的属性. 例如下面的 [应用服务](Application-Services.md)方法:

```csharp

public class UserAppService : ApplicationService

{

private readonly IRepository<User, Guid> \_userRepository;

public UserAppService(IRepository<User, Guid> userRepository)

{

\_userRepository = userRepository;

}

public async Task CreateUser(CreateUserInput input)

{

//Manually creating a User object from the CreateUserInput object

var user = new User

{

Name = input.Name,

Surname = input.Surname,

EmailAddress = input.EmailAddress,

Password = input.Password

};

await \_userRepository.InsertAsync(user);

}

}

```

`CreateUserInput` 和 `User` 是一个简单的[DTO](Data-Transfer-Objects.md)和[实体](Entities.md)类. 上面的代码使用input对象创建了一个 `User` 实体. 上面的代码很简单,但在实际应用程序中 `User` 实体会拥有很多属性,手动创建实体乏味且容易出错. `User` 和 `CreateUserInput` 添加新属性时还需要再去修改代码.

我们需要一个库自动处理类到类的映射. ABP提供了对象到对象映射的抽象并集成了[AutoMapper](http://automapper.org/)做为对象映射器.

## IObjectMapper

`IObjectMapper` 接口 (在 [Volo.Abp.ObjectMapping](https://www.nuget.org/packages/Volo.Abp.ObjectMapping)包中) 定义了一个简单的 `Map` 方法. 上面的手动映射示例可以用以下方式重写:

````csharp

public class UserAppService : ApplicationService

{

private readonly IRepository<User, Guid> \_userRepository;

public UserAppService(IRepository<User, Guid> userRepository)

{

\_userRepository = userRepository;

}

public async Task CreateUser(CreateUserInput input)

{

//Automatically creating a new User object using the CreateUserInput object

var user = ObjectMapper.Map<CreateUserInput, User>(input);

await \_userRepository.InsertAsync(user);

}

}

````

> 示例中的 `ObjectMapper` 属性在 `ApplicationService` 基类中属性注入. 在其他地方也可以直接注入 `IObjectMapper` 接口.

Map方法有两个泛型参数: 第一个是源对象类型,第二个是目标对象类型.

如果想要设置现有对象属性,可以使用 `Map` 的重载方法:

````csharp

public class UserAppService : ApplicationService

{

private readonly IRepository<User, Guid> \_userRepository;

public UserAppService(IRepository<User, Guid> userRepository)

{

\_userRepository = userRepository;

}

public async Task UpdateUserAsync(Guid id, UpdateUserInput input)

{

var user = await \_userRepository.GetAsync(id);

//Automatically set properties of the user object using the UpdateUserInput

ObjectMapper.Map<UpdateUserInput, User>(input, user);

await \_userRepository.UpdateAsync(user);

}

}

````

必须先定义映射,然后才能映射对象. 请参阅AutoMapper集成部分了解如何定义映射.

## AutoMapper 集成

[AutoMapper](http://automapper.org/) 是最流行的对象到对象映射库之一. [Volo.Abp.AutoMapper](https://www.nuget.org/packages/Volo.Abp.AutoMapper)程序包使用AutoMapper实现了 `IObjectMapper`.

定义了以下部分的映射后就可以使 `IObjectMapper` 接口.

### 定义映射

AutoMapper提供了多种定义类之间映射的方法. 有关详细信息请参阅[AutoMapper的文档](https://docs.automapper.org).

其中定义一种映射的方法是创建一个[Profile](https://docs.automapper.org/en/stable/Configuration.html#profile-instances) 类. 例如:

````csharp

public class MyProfile : Profile

{

public MyProfile()

{

CreateMap<User, UserDto>();

}

}

````

然后使用`AbpAutoMapperOptions`注册配置文件:

````csharp

[DependsOn(typeof(AbpAutoMapperModule))]

public class MyModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpAutoMapperOptions>(options =>

{

//Add all mappings defined in the assembly of the MyModule class

options.AddMaps<MyModule>();

});

}

}

````

`AddMaps` 注册给定类的程序集中所有的配置类,通常使用模块类. 它还会注册 [attribute 映射](https://docs.automapper.org/en/stable/Attribute-mapping.html).

### 配置验证

`AddMaps` 使用可选的 `bool` 参数控制[模块](Module-Development-Basics.md)的[配置验证](https://docs.automapper.org/en/stable/Configuration-validation.html):

````csharp

options.AddMaps<MyModule>(validate: true);

````

如果此选项默认是 `false` , 但最佳实践建议启用.

可以使用 `AddProfile` 而不是 `AddMaps` 来控制每个配置文件类的配置验证:

````csharp

options.AddProfile<MyProfile>(validate: true);

````

> 如果你有多个配置文件,并且只需要为其中几个启用验证,那么首先使用`AddMaps`而不进行验证,然后为你想要验证的每个配置文件使用`AddProfile`.

### 映射对象扩展

[对象扩展系统](Object-Extensions.md) 允许为已存在的类定义额外属性. ABP 框架提供了一个映射定义扩展可以正确的映射两个对象的额外属性.

````csharp

public class MyProfile : Profile

{

public MyProfile()

{

CreateMap<User, UserDto>()

.MapExtraProperties();

}

}

````

如果两个类都是可扩展对象(实现了 `IHasExtraProperties` 接口),建议使用 `MapExtraProperties` 方法. 更多信息请参阅[对象扩展文档](Object-Extensions.md).

### 其他有用的扩展方法

有一些扩展方法可以简化映射代码.

#### 忽视审计属性

当你将一个对象映射到另一个对象时,通常会忽略审核属性.

假设你需要将 `ProductDto` ([DTO](Data-Transfer-Objects.md))映射到Product[实体](Entities.md),该实体是从 `AuditedEntity` 类继承的(该类提供了 `CreationTime`, `CreatorId`, `IHasModificationTime` 等属性).

从DTO映射时你可能想忽略这些基本属性,可以使用 `IgnoreAuditedObjectPropertie()` 方法忽略所有审计属性(而不是手动逐个忽略它们):

````csharp

public class MyProfile : Profile

{

public MyProfile()

{

CreateMap<ProductDto, Product>()

.IgnoreAuditedObjectProperties();

}

}

````

还有更多扩展方法, 如 `IgnoreFullAuditedObjectProperties()` 和 `IgnoreCreationAuditedObjectProperties()`,你可以根据实体类型使用.

> 请参阅[实体文档](Entities.md)中的"*\*基类和接口的审计属性\**"部分了解有关审计属性的更多信息。

#### 忽视其他属性

在AutoMapper中,通常可以编写这样的映射代码来忽略属性:

````csharp

public class MyProfile : Profile

{

public MyProfile()

{

CreateMap<SimpleClass1, SimpleClass2>()

.ForMember(x => x.CreationTime, map => map.Ignore());

}

}

````

我们发现它的长度是不必要的并且创建了 `Ignore()` 扩展方法:

````csharp

public class MyProfile : Profile

{

public MyProfile()

{

CreateMap<SimpleClass1, SimpleClass2>()

.Ignore(x => x.CreationTime);

}

}

````

## 高级主题

### IObjectMapper<TContext> 接口

假设你已经创建了一个**\*\*可重用的模块\*\***,其中定义了AutoMapper配置文件,并在需要映射对象时使用 `IObjectMapper`. 根据[模块化](Module-Development-Basics.md)的性质,你的模块可以用于不同的应用程序.

`IObjectMapper` 是一个抽象,可以由最终应用程序替换使用另一个映射库. 这里的问题是你的可重用模块设计为使用AutoMapper,因为它为其定义映射配置文件. 这种情况下即使最终应用程序使用另一个默认对象映射库,你也要保证模块始终使用AutoMapper.

`IObjectMapper<TContext>`将对象映射器上下文化,你可以为不同的 模块/上下文 使用不同的库.

用法示例:

````csharp

public class UserAppService : ApplicationService

{

private readonly IRepository<User, Guid> \_userRepository;

private readonly IObjectMapper<MyModule> \_objectMapper;

public UserAppService(

IRepository<User, Guid> userRepository,

IObjectMapper<MyModule> objectMapper) //Inject module specific mapper

{

\_userRepository = userRepository;

\_objectMapper = objectMapper;

}

public async Task CreateUserAsync(CreateUserInput input)

{

//Use the module specific mapper

var user = \_objectMapper.Map<CreateUserInput, User>(input);

await \_userRepository.InsertAsync(user);

}

}

````

`UserAppService` 注入 `IObjectMapper<MyModule>`, 它是模块的特定对象映射器,用法与 `IObjectMapper` 完全相同.

上面的示例代码未使用 `ApplicationService` 中定义的 `ObjectMapper` 属性,而是注入了 `IObjectMapper<MyModule>`. 但是 `ApplicationService` 定义了可以在类构造函数中设置的 `ObjectMapperContext` 属性, 因此仍然可以使用基类属性. 示例可以进行以下重写:

````csharp

public class UserAppService : ApplicationService

{

private readonly IRepository<User, Guid> \_userRepository;

public UserAppService(IRepository<User, Guid> userRepository)

{

\_userRepository = userRepository;

//Set the object mapper context

ObjectMapperContext = typeof(MyModule);

}

public async Task CreateUserAsync(CreateUserInput input)

{

var user = ObjectMapper.Map<CreateUserInput, User>(input);

await \_userRepository.InsertAsync(user);

}

}

````

虽然使用上下文化的对象映射器与普通的对象映射器相同, 但是也应该在模块的 `ConfigureServices` 方法中注册上下文化的映射器:

````csharp

[DependsOn(typeof(AbpAutoMapperModule))]

public class MyModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

//Use AutoMapper for MyModule

context.Services.AddAutoMapperObjectMapper<MyModule>();

Configure<AbpAutoMapperOptions>(options =>

{

options.AddMaps<MyModule>(validate: true);

});

}

}

````

`IObjectMapper<MyModule>`是可重用模块的一项基本功能,可在多个应用程序中使用,每个模块可以使用不同的库进行对象到对象的映射. 所有预构建的ABP模块都在使用它. 但是对于最终应用程序,你可以忽略此接口,始终使用默认的 `IObjectMapper` 接口.

### IObjectMapper<TSource, TDestination> 接口

ABP允许自定义特定类的映射代码. 假设你要创建一个自定义类从 `User` 映射到 `UserDto`. 这种情况下,你可以创建一个实现 `IObjectMapper<User,UserDto>`的类 :

````csharp

public class MyCustomUserMapper : IObjectMapper<User, UserDto>, ITransientDependency

{

public UserDto Map(User source)

{

//TODO: Create a new UserDto

}

public UserDto Map(User source, UserDto destination)

{

//TODO: Set properties of an existing UserDto

return destination;

}

}

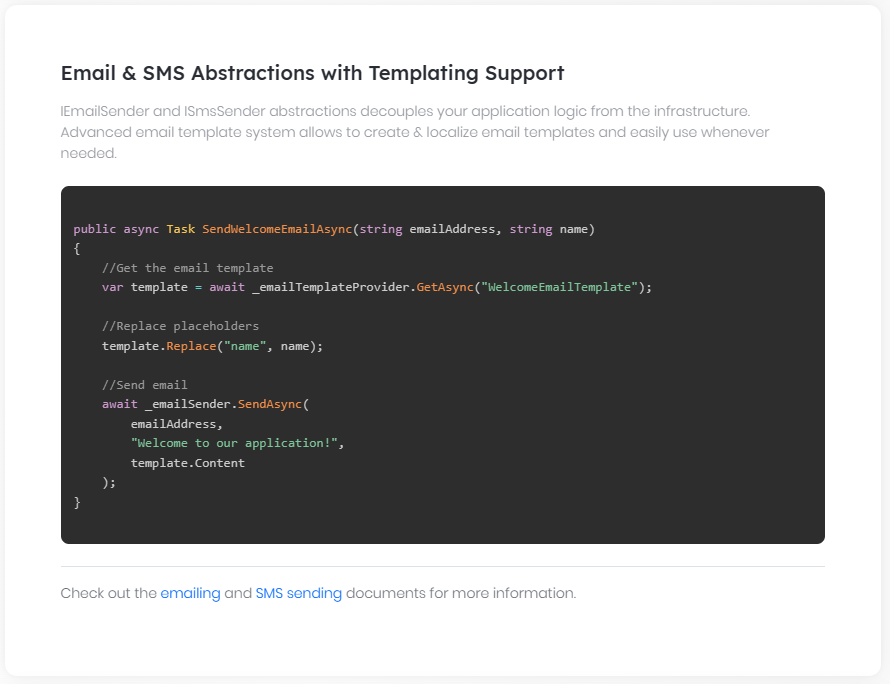
````

ABP会自动发现注册 `MyCustomUserMapper`, 在你使用IObjectMapper将用户映射到UserDto时会自动使用自定义映射.

一个类可以为不同的对象实现多个 `IObjectMapper<TSource,TDestination>`.

> 这种方法功能强大, `MyCustomUserMapper`可以注入任何其他服务并在Map方法中使用.

## 邮件发送



邮件发送系统

# 邮件发送系统

ABP 框架为发送电子邮件提供各种服务、设置和集成;

\* 提供用于发送电子邮件的`IEmailSender`服务.

\* 定义 [settings](Settings.md)来配置电子邮件发送.

\* 集成到[后台作业系统](Background-Jobs.md)以通过后台作业发送电子邮件.

\* 提供[MailKit 集成](MailKit.md)包.

## 安装

> 如果你使用的是[应用程序启动模板](Startup-Templates/Application.md)，则该软件包已安装。

>

建议使用 [ABP CLI](CLI.md) 安装此包。在项目文件夹（.csproj 文件）中打开命令行窗口并键入以下命令：

````bash

abp add-package Volo.Abp.Emailing

````

如果你还没有做到这一点，你首先需要安装 ABP CLI。有关其他安装选项，请参阅 [包描述页面](https://abp.io/package-detail/Volo.Abp.Emailing)。

## 发送电子邮件

### IEmailSender

[注入](Dependency-Injection.md) `IEmailSender` 到任何服务并使用 `SendAsync` 方法发送电子邮件。

**\*\*示例\*\***

````csharp

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Emailing;

namespace MyProject

{

public class MyService : ITransientDependency

{

private readonly IEmailSender \_emailSender;

public MyService(IEmailSender emailSender)

{

\_emailSender = emailSender;

}

public async Task DoItAsync()

{

await \_emailSender.SendAsync(

"target@domain.com", // target email address

"Email subject", // subject

"This is email body..." // email body

);

}

}

}

````

`SendAsync` 方法具有重载以提供更多参数，例如；

\* **\*\*from\*\***: 你可以将其设置为设置发件人电子邮件地址。如果未提供，则使用默认发件人地址（请参阅下面的电子邮件设置）

\* **\*\*isBodyHtml\*\***: 表示邮件正文是否可以包含HTML标签。**\*\*默认：true\*\***。

> `IEmailSender` 是建议的发送邮件的方式，因为它使你的代码提供者独立。

#### 邮件消息

除了原始参数之外，你还可以传递一个**\*\*标准的** `MailMessage` **对象\*\***（[参见](https://docs.microsoft.com/en-us/dotnet/api/system.net.mail.mailmessage) ) 到 `SendAsync` 方法以设置更多选项，例如添加附件。

### ISmtpEmailSender

默认情况下，发送电子邮件由标准的 `SmtpClient` 类（[参见](https://docs.microsoft.com/en-us/dotnet/api/system.net.mail.smtpclient)）实现。实现类是`SmtpEmailSender`。此类还公开了 `ISmtpEmailSender` 服务（除了 `IEmailSender`）。

大多数时候你想直接使用`IEmailSender`来让你的代码提供者独立。但是，如果要创建具有相同电子邮件设置的 `SmtpClient` 对象，可以注入 `ISmtpEmailSender` 并使用其 `BuildClientAsync` 方法获取 `SmtpClient` 对象并自己发送电子邮件。

## 发送邮件任务队列/后台作业

`IEmailSender`有一个`QueueAsync`方法，可以用来将邮件添加到后台作业队列中，在后台线程中发送。通过这种方式，你不会因为等待发送邮件而占用用户的时间。`QueueAsync`方法得到的参数与`SendAsync`方法相同。

发送邮件任务队列可以容忍错误，因为后台作业系统具有重试机制来克服临时网络/服务器问题。

有关后台作业系统的更多信息，请参阅[后台作业文档](Background-Jobs.md)。

## 电子邮件设置

电子邮件发送使用 [设置系统](Settings.md) 来定义设置并在运行时获取这些设置的值。 `Volo.Abp.Emailing.EmailSettingNames` 定义了设置名称的常量，如下所示：

\* **\*\*Abp.Mailing.DefaultFromAddress\*\***: 当你在发送电子邮件时未指定发件人时,用作发件人的电子邮件地址（就像上面的示例一样）.

\* **\*\*Abp.Mailing.DefaultFromDisplayName\*\***: 当你在发送电子邮件时未指定发件人时,用作发件人的显示名称（就像在上面的示例中一样）.

\* **\*\*Abp.Mailing.Smtp.Host\*\***: SMTP 服务器的 IP/域（默认值：127.0.0.1）。

\* **\*\*Abp.Mailing.Smtp.Port\*\***: SMTP 服务器的端口（默认值：25）.

\* **\*\*Abp.Mailing.Smtp.UserName\*\***: 用户名，如果 SMTP 服务器需要身份验证需要。

\* **\*\*Abp.Mailing.Smtp.Password\*\***: 密码，如果 SMTP 服务器需要身份验证需要。 **\*\*此值已加密\*\***（请参阅下面的部分）.

\* **\*\*Abp.Mailing.Smtp.Domain\*\***: 账号域，如果 SMTP 服务器需要身份验证需要.

\* **\*\*Abp.Mailing.Smtp.EnableSsl\*\***: 指示 SMTP 服务器是否使用 SSL 的值（“true”或“false”。默认值：“false”）.

\* **\*\*Abp.Mailing.Smtp.UseDefaultCredentials\*\***:如果为 true，则使用默认凭据,而不是提供的用户名和密码（“true”或“false”。默认值：“true”）。.

可以从[设置管理](Modules/Setting-Management.md)模块的*\*设置页面\**管理电子邮件设置：

![email-settings](images/email-settings.png)

>如果你已从 ABP 启动模板创建解决方案，则已安装设置管理模块。

如果你不使用设置管理模块，你可以简单地在 `appsettings.json` 文件中定义设置：

````json

"Settings": {

"Abp.Mailing.Smtp.Host": "127.0.0.1",

"Abp.Mailing.Smtp.Port": "25",

"Abp.Mailing.Smtp.UserName": "",

"Abp.Mailing.Smtp.Password": "",

"Abp.Mailing.Smtp.Domain": "",

"Abp.Mailing.Smtp.EnableSsl": "false",

"Abp.Mailing.Smtp.UseDefaultCredentials": "true",

"Abp.Mailing.DefaultFromAddress": "noreply@abp.io",

"Abp.Mailing.DefaultFromDisplayName": "ABP application"

}

````

You can set/change these settings programmatically using the `ISettingManager` and store values in a database. See the [setting system document](Settings.md) to understand the setting system better.

你可以使用 `ISettingManager` 以编程方式设置/更改这些设置，并将值存储在数据库中。请参阅 [设置系统文档](Settings.md)更好地了解设置系统。

### 加密 SMTP 密码

*\*Abp.Mailing.Smtp.Password\** 必须是一个**\*\*加密\*\***值。如果你使用 `ISettingManager` 设置密码，你不必担心。它在内部加密 set 上的值并在 get 上解密。

如果使用 `appsettings.json` 存储密码，则应手动注入 `ISettingEncryptionService` 并使用其 `Encrypt` 方法获取加密值。这可以通过在你的应用程序中创建一个简单的代码来完成。然后你可以删除代码。更好的是，你可以在应用程序中创建一个 UI 来配置电子邮件设置。在这种情况下，你可以直接使用 `ISettingManager` 而不用担心加密。

### ISmtpEmailSenderConfiguration

如果你不想使用设置系统来存储电子邮件发送配置，你可以将 `ISmtpEmailSenderConfiguration` 服务替换为你自己的实现，以从任何其他来源获取配置。 `ISmtpEmailSenderConfiguration` 默认由 `SmtpEmailSenderConfiguration` 实现，如上所述，它从设置系统中获取配置。

## 文本模板集成

ABP 框架提供了一个强大而灵活的[文本模板系统](Text-Templating.md)。你可以使用文本模板系统来创建动态电子邮件内容。注入 `ITemplateRenderer` 并使用 `RenderAsync` 渲染模板。然后将结果用作电子邮件正文。

虽然你可以定义和使用自己的文本模板，但电子邮件发送系统提供了两个简单的内置文本模板。

**\*\*示例：使用标准和简单的消息模板发送电子邮件\*\***

````csharp

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Emailing;

using Volo.Abp.Emailing.Templates;

using Volo.Abp.TextTemplating;

namespace Acme.BookStore.Web

{

public class MyService : ITransientDependency

{

private readonly IEmailSender \_emailSender;

private readonly ITemplateRenderer \_templateRenderer;

public MyService(

IEmailSender emailSender,

ITemplateRenderer templateRenderer)

{

\_emailSender = emailSender;

\_templateRenderer = templateRenderer;

}

public async Task DoItAsync()

{

var body = await \_templateRenderer.RenderAsync(

StandardEmailTemplates.Message,

new

{

message = "This is email body..."

}

);

await \_emailSender.SendAsync(

"target-address@domain.com",

"Email subject",

body

);

}

}

}

````

生成的电子邮件正文将如下所示：

````html

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">

<head>

<meta charset="utf-8" />

</head>

<body>

This is email body...

</body>

</html>

````

电子邮件系统定义了具有给定名称的内置文本模板：

"**\*\*Abp.StandardEmailTemplates.Message\*\***" 是最简单的带有文本消息的模板：

````html

{%{{{model.message}}}%}

````

此模板使用“Abp.StandardEmailTemplates.Layout”作为其布局.

"**\*\*Abp.StandardEmailTemplates.Layout\*\***" 是一个提供 HTML 文档布局的简单模板：

````html

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">

<head>

<meta charset="utf-8" />

</head>

<body>

{%{{{content}}}%}

</body>

</html>

````

最终呈现的消息如上所示。

> 这些模板名称是在 `Volo.Abp.Emailing.Templates.StandardEmailTemplates` 类中定义的。

### 覆盖/替换标准模板

You typically want to replace the standard templates with your own ones, so you can prepare a branded email messages. To do that, you can use the power of the [virtual file system](Virtual-File-System.md) (VFS) or replace them in your own template definition provider.

你希望用自己的模板替换标准模板，这样你就可以准备电子邮件模板文件。你可以使用 [虚拟文件系统](Virtual-File-System.md) (VFS),或在你自己的模板定义提供程序中替换它们。

虚拟文件系统中模板的路径如下图所示：

\* `/Volo/Abp/Emailing/Templates/Layout.tpl`

\* `/Volo/Abp/Emailing/Templates/Message.tpl`

如果你将文件添加到虚拟文件系统中的相同位置，你的文件将覆盖它们。

模板是内联本地化的，这意味着你可以利用 [本地化系统](Localization.md) 的强大功能使你的模板具有多元文化。

详见[文本模板系统](Text-Templating.md) 文档。

> 请注意，你可以为应用程序定义和使用自己的模板，而不是使用标准的简单模板。这些标准模板主要用于可重用的模块，它们不定义自己的模板，而是依赖内置的模板。只需覆盖标准的电子邮件布局模板，就可以轻松自定义使用的模块发送的电子邮件。

## NullEmailSender

`NullEmailSender` 是实现 `IEmailSender` 的内置类，但将电子邮件内容写入 [标准日志系统](Logging.md)，而不是实际发送电子邮件。

这个类特别有用，尤其是在你开发是不想发送真实电子邮件。 [应用启动模板](Startup-Templates/Application.md)已经在**\*\*DEBUG模式\*\***中使用了这个类，在领域层配置如下：

````csharp

#if DEBUG

context.Services.Replace(ServiceDescriptor.Singleton<IEmailSender, NullEmailSender>());

#endif

````

因此，你在 DEBUG 模式下不会收到电子邮件。电子邮件将在生产时按预期发送（RELEASE 模式）。如果你也想在 DEBUG 上发送电子邮件，请删除这些行。

## 其他

\* [用于发送电子邮件的MailKit集成](MailKit.md)

MailKit集成

# MailKit 集成

[MailKit](http://www.mimekit.net/) 是一个用于 .net 的跨平台、流行的开源邮件客户端库.ABP 框架提供了一个集成包来使用 MailKit 作为[邮件发送系统](Emailing.md)的收发组件.

## 安装

建议使用 [ABP CLI](CLI.md) 安装包.在项目文件夹（.csproj 文件）中打开命令行窗口并键入以下命令：

````bash

abp add-package Volo.Abp.MailKit

````

如果执行失败,你首先需要安装 ABP CLI.有关其他安装选项,请参阅 [包描述页面](https://abp.io/package-detail/Volo.Abp.MailKit).

## 发送电子邮件

### IEmailSender

[注入](Dependency-Injection.md) 标准的 `IEmailSender` 到任何服务并使用 `SendAsync` 方法发送电子邮件.详见 [邮件发送文档](Emailing.md).

> `IEmailSender` 是建议的发送邮件的方式,即使你使用MailKit,因为它使你的代码独立.

### IMailKitSmtpEmailSender

`BuildClientAsync()` 方法扩展了`IEmailSender`.此方法可用于获取可用于执行 MailKit 特定操作的`MailKit.Net.Smtp.SmtpClient`对象.

## 配置

MailKit 集成包使用电子邮件发送系统相同配置选项.请参阅[电子邮件发送文档](Emailing.md) 进行配置.

除了标准设置之外,这个包还定义了 `AbpMailKitOptions` 作为一个简单的 [选项](Options.md) 类.此类仅定义一个选项：

\* **\*\*SecureSocketOption\*\***:用于设置“SecureSocketOptions” . Default：`null`（使用默认值）

**\*\*示例: 使用 *\*SecureSocketOptions.SslOnConnect\**\*\***

````csharp

Configure<AbpMailKitOptions>(options =>

{

options.SecureSocketOption = SecureSocketOptions.SslOnConnect;

});

````

请参阅 [MailKit 文档](http://www.mimekit.net/) 了解更多信息.

## 也可以看看

\* [电子邮件发送系统](Emailing.md)

## BLOB存储



BLOB存储系统

# BLOB 存储

通常将文件内容存储在应用程序中并根据需要读取这些文件内容. 不仅是文件你可能还需要将各种类型的[BLOB](https://en.wikipedia.org/wiki/Binary\_large\_object)(大型二进制对象)保存到存储中. 例如你可能要保存用户个人资料图片.

BLOB通常是一个**\*\*字节数组\*\***. 有很多地方可以存储BLOB项. 可以选择将其存储在本地文件系统中,共享数据库中或[Azure BLOB存储](https://azure.microsoft.com/zh-cn/services/storage/blobs/)中.

ABP框架为BLOB提供了抽象,并提供了一些可以轻松集成到的预构建存储提供程序. 抽象有一些好处;

\* 你可以通过几行配置**\*\*轻松的集成\*\***你喜欢的BLOB存储提供程序.

\* 你可以**\*\*轻松的更改\*\***BLOB存储,而不用改变你的应用程序代码.

\* 如果你想创建**\*\*可重用的应用程序模块\*\***,无需假设BLOB的存储方式.

ABP BLOB存储系统兼容ABP框架其他功能,如[多租户](Multi-Tenancy.md).

## BLOB 存储提供程序

ABP框架已经有以下存储提供程序的实现;

\* [File System](Blob-Storing-File-System.md):将BLOB作为标准文件存储在本地文件系统的文件夹中.

\* [Database](Blob-Storing-Database.md): 将BLOB存储在数据库中.

\* [Azure](Blob-Storing-Azure.md): 将BLOB存储在 [Azure BLOB storage](https://azure.microsoft.com/en-us/services/storage/blobs/)中.

\* [Aliyun](Blob-Storing-Aliyun.md): 将BLOB存储在[Aliyun Storage Service](https://help.aliyun.com/product/31815.html)中.

\* [Minio](Blob-Storing-Minio.md): 将BLOB存储在[MinIO Object storage](https://min.io/)中.

\* [Aws](Blob-Storing-Aws.md): 将BLOB存储在[Amazon Simple Storage Service](https://aws.amazon.com/s3/)中.

以后会实现更多的提供程序,你可以为自己喜欢的提供程序创建[请求](https://github.com/abpframework/abp/issues/new),或者你也可以[自己实现](Blob-Storing-Custom-Provider.md)它并[贡献](Contribution/Index.md)到ABP框架.

可以在**\*\*容器系统\*\***的帮助下一起**\*\*使用多个提供程序\*\***,其中每个容器可以使用不同的提供程序.

> 除非你**\*\*配置存储提供程序\*\***否则BLOB存储系统无法工作. 有关存储提供程序配置请参考链接的文档.

## 安装

[Volo.Abp.BlobStoring](https://www.nuget.org/packages/Volo.Abp.BlobStoring)是定义BLOB存储服务的主要包. 你可以用此包使用BLOB存储系统而不依赖特定存储提供程序.

使用ABP CLI这个包添加到你的项目:

\* 安装 [ABP CLI](https://docs.abp.io/en/abp/latest/CLI), 如果你还没有安装.

\* 在要添加 `Volo.Abp.BlobStoring` 包的 `.csproj` 文件目录打开命令行.

\* 运行 `abp add-package Volo.Abp.BlobStoring` 命令.

如果要手动安装,在你的项目中安装 `Volo.Abp.BlobStoring` NuGet包然后将`[DependsOn(typeof(AbpBlobStoringModule))]`添加到项目内的[ABP模块](Module-Development-Basics.md)类中.

## IBlobContainer

`IBlobContainer` 是存储和读取BLOB的主要接口. 应用程序可能有多个容器,每个容器都可以单独配置. 有一个**\*\*默认容器\*\***可以通过注入 `IBlobContainer` 来简单使用.

**\*\*示例: 简单地保存和读取命名BLOB的字节\*\***

````csharp

using System.Threading.Tasks;

using Volo.Abp.BlobStoring;

using Volo.Abp.DependencyInjection;

namespace AbpDemo

{

public class MyService : ITransientDependency

{

private readonly IBlobContainer \_blobContainer;

public MyService(IBlobContainer blobContainer)

{

\_blobContainer = blobContainer;

}

public async Task SaveBytesAsync(byte[] bytes)

{

await \_blobContainer.SaveAsync("my-blob-1", bytes);

}

public async Task<byte[]> GetBytesAsync()

{

return await \_blobContainer.GetAllBytesOrNullAsync("my-blob-1");

}

}

}

````

该服务用 `my-blob-1` 名称保存给定的字节,然后以相同的名称获取先前保存的字节.

> 一个BLOB是一个命名对象,**\*\*每个BLOB都应该有一个唯一的名称\*\***,它是一个任意的字符串.

`IBlobContainer` 可以处理 `Stream` 和 `byte[]` 对象,在下一节中将详细介绍.

### 保存 BLOB

`SaveAsync` 方法用于保存新的或替换现有的BLOB. 默认情况下,它可以保存 `Stream`,但是有一个快捷的扩展方法来保存字节数组.

`SaveAsync` 有以下参数:

\* **\*\*name\*\*** (string): 唯一的BLOB名称.

\* **\*\*stream\*\*** (Stream) or **\*\*byte\*\*** (byte[]): 读取BLOB内容或字节数组的流.

\* **\*\*overrideExisting\*\*** (bool): 设置为 `true`,如果BLOB内容已经存在,则替换它. 默认值为 `false`,则抛出 `BlobAlreadyExistsException` 异常.

### 读取/获取 BLOB

\* `GetAsync`: 返回给定BLOB名称可用于读取BLOB内容的 `Stream` 对象. 使用后始终要**\*\*dispose流\*\***. 如果找不到具有给定名称的BLOB,则抛出异常.

\* `GetOrNullAsync`: 与 `GetAsync` 方法相反,如果未找到给定名称的BLOB,则返回 `null`.

\* `GetAllBytesAsync`: 返回 `byte[]` 而不是 `Stream`. 如果找不到具有给定名称的BLOB,则抛出异常.

\* `GetAllBytesOrNullAsync`: 与 `GetAllBytesAsync` 方法相反,如果未找到给定名称的BLOB,则返回 `null`.

### 删除 BLOB

`DeleteAsync` 使用给定BLOB名称删除BLOB数据. 如果找不到给定的BLOB不会引发任何异常. 相反如果你关心BLOB,它会返回一个 `bool`,表示BLOB实际上是否已删除.

### 其他方法

\* `ExistsAsync` 方法简单的检查容器中是否存在具有给定名称的BLOB.

### 关于命名BLOB

没有命名BLOB的规则. BLOB名称只是每个容器(和每个租户-参见"*\*多租户\**"部分)唯一的字符串. 但是不同的存储提供程序可能会按惯例实施某些做法. 例如[文件系统提供程序](Blob-Storing-File-System.md)在BLOB名称中使用目录分隔符 (`/`) 和文件扩展名(如果BLOB名称为 `images/common/x.png` ,则在根容器文件夹下的 `images/common` 文件夹中存储 `x.png`).

## 类型化 IBlobContainer

类型化BLOB容器系统是一种在应用程序中创建和管理**\*\*多个容器\*\***的方法;

\* **\*\*每个容器分别存储\*\***. 这意味着BLOB名称在一个容器中应该是唯一的,两个具有相同名称的BLOB可以存在不同的容器中不会互相影响.

\* **\*\*每个容器可以单独配置\*\***,因此每个容器可以根据你的配置使用不同的存储提供程序.

要创建类型化容器,需要创建一个简单的用 `BlobContainerName` 属性装饰的类:

````csharp

using Volo.Abp.BlobStoring;

namespace AbpDemo

{

[BlobContainerName("profile-pictures")]

public class ProfilePictureContainer

{

}

}

````

> 如果不使用 `BlobContainerName` attribute,ABP Framework将使用类的全名(带有名称空间),但是始终建议使用稳定的容器名称,即使重命名该类也不会被更改.

创建容器类后,可以为容器类型注入 `IBlobContainer<T>`.

**\*\*示例: 用于保存和读取[**当前用户**](CurrentUser.md)的个人资料图片的[**应用服务**](Application-Services.md)\*\***

````csharp

[Authorize]

public class ProfileAppService : ApplicationService

{

private readonly IBlobContainer<ProfilePictureContainer> \_blobContainer;

public ProfileAppService(IBlobContainer<ProfilePictureContainer> blobContainer)

{

\_blobContainer = blobContainer;

}

public async Task SaveProfilePictureAsync(byte[] bytes)

{

var blobName = CurrentUser.GetId().ToString();

await \_blobContainer.SaveAsync(blobName, bytes);

}

public async Task<byte[]> GetProfilePictureAsync()

{

var blobName = CurrentUser.GetId().ToString();

return await \_blobContainer.GetAllBytesOrNullAsync(blobName);

}

}

````

`IBlobContainer<T>` 有与 `IBlobContainer` 相同的方法.

> 在开发可重复使用的模块时,**\*\*始终使用类型化的容器是一个好习惯\*\***,这样最终的应用程序就可以为你的容器配置提供程序,而不会影响其他容器.

### 默认容器

如果不使用泛型参数,直接注入 `IBlobContainer` (如上所述),会得到默认容器. 注入默认容器的另一种方法是使用 `IBlobContainer<DefaultContainer>`,它返回完全相同的容器.

默认容器的名称是 `default`.

### 命令容器

类型容器只是命名容器的快捷方式. 你可以注入并使用 `IBlobContainerFactory` 来获得一个BLOB容器的名称:

````csharp

public class ProfileAppService : ApplicationService

{

private readonly IBlobContainer \_blobContainer;

public ProfileAppService(IBlobContainerFactory blobContainerFactory)

{

\_blobContainer = blobContainerFactory.Create("profile-pictures");

}

//...

}

````

## IBlobContainerFactory

`IBlobContainerFactory` 是用于创建BLOB容器的服务. 上面提供了一个示例.

**\*\*示例: 通过名称创建容器\*\***

````csharp

var blobContainer = blobContainerFactory.Create("profile-pictures");

````

**\*\*示例: 通过类型创建容器\*\***

````csharp

var blobContainer = blobContainerFactory.Create<ProfilePictureContainer>();

````

> 通常你不需要使用 `IBlobContainerFactory`, 因为在注入 `IBlobContainer` 或`IBlobContainer<T>` 时会在内部使用它.

## 配置容器

在使用容器之前应先对其进行配置. 最基本的配置是选择一个 **\*\*BLOB存储提供程序\*\***(请参阅上面的"*\*BLOB存储提供程序\**"部分).

`AbpBlobStoringOptions` 是用于配置容器的[选项类](Options.md). 你可以在[模块](Module-Development-Basics.md)的 `ConfigureServices` 方法中配置选项.

### 配置单个容器

````csharp

Configure<AbpBlobStoringOptions>(options =>

{

options.Containers.Configure<ProfilePictureContainer>(container =>

{

//TODO...

});

});

````

这个例子配置 `ProfilePictureContainer`. 你还可以通过容器名称进行配置:

````csharp

Configure<AbpBlobStoringOptions>(options =>

{

options.Containers.Configure("profile-pictures", container =>

{

//TODO...

});

});

````

### 配置默认容器

````csharp

Configure<AbpBlobStoringOptions>(options =>

{

options.Containers.ConfigureDefault(container =>

{

//TODO...

});

});

````

> 默认容器有一个特殊情况;如果不为容器指定配置,则**\*\*返回到默认容器配置\*\***. 这是一种为所有容器配置默认值并在需要时专门针对特定容器进行配置的好方法.

### 配置所有容器

````csharp

Configure<AbpBlobStoringOptions>(options =>

{

options.Containers.ConfigureAll((containerName, containerConfiguration) =>

{

//TODO...

});

});

````

这是配置所有容器的方式.

> 与配置默认容器的主要区别在于, `ConfigureAll` 会覆盖配置,即使它是专门为特定容器配置的.

## 多租户

如果你的应用程序是多租户的,BLOB存储系统可以**\*\*与[**多租户**](Multi-Tenancy.md)无缝协作\*\***. 所有提供程序都将多租户实现为标准功能. 它们将不同租户的**\*\*BLOB彼此隔离\*\***,因此它们只能访问自己的BLOB. 这意味着你可以**\*\*为不同的租户使用相同的BLOB名称\*\***.

如果应用程序是多租户的,则可能需要单独控制容器的**\*\*多租户行为\*\***. 例如你可能希望**\*\*禁用特定容器的多租户\*\***,这样容器中的BLOB将对**\*\*所有租户可用\*\***. 这是在所有租户之间共享BLOB的一种方法.

**\*\*示例: 禁用特定容器的多租户\*\***

````csharp

Configure<AbpBlobStoringOptions>(options =>

{

options.Containers.Configure<ProfilePictureContainer>(container =>

{

container.IsMultiTenant = false;

});

});

````

> 如果你的应用程序不是多租户的,不用担心,它会正常工作. 你不需要配置 `IsMultiTenant` 选项.

## 扩展BLOB存储系统

大多数时候除了创建定制的BLOB存储提供程序外,你不需要[自定义BLOB存储系统](Blob-Storing-Custom-Provider.md).但是如果需要,你可以替换任何服务(通过[依赖注入](Dependency-Injection.md)). 这里有一些上面没有提到的其他服务,但你可能想知道:

\* `IBlobProviderSelector` 用于通过容器名称获取 `IBlobProvider` 实例. 默认实现(`DefaultBlobProviderSelector`)使用配置选择提供程序.

\* `IBlobContainerConfigurationProvider` 用于获取给定容器名称的`BlobContainerConfiguration`. 默认实现(`DefaultBlobContainerConfigurationProvider`)从上述 `AbpBlobStoringOptions` 获取配置.

## BLOB 存储 vs 文件管理系统

注意BLOB存储不是一个文件管理系统. 它是一个用于保存,获取和删除命名BLOB的低级别系统. 它不提供目录那样的层次结构,这是典型文件系统所期望的.

如果你想创建文件夹并在文件夹之间移动文件,为文件分配权限并在用户之间共享文件,那么你需要在BLOB存储系统上实现你自己的应用程序.

## 另请参阅

\* [创建自定义BLOB存储提供程序](Blob-Storing-Custom-Provider.md)

存储提供程序

#### 文件系统提供程序

# BLOB存储文件系统提供程序

文件系统存储提供程序用于将BLOB作为文件夹中的标准文件存储在本地文件系统中.

> 阅读[BLOB存储文档](Blob-Storing.md)了解如何使用BLOB存储系统. 本文档仅介绍如何为容器配置文件系统.

## 介绍

使用ABP CLI添加[Volo.Abp.BlobStoring.FileSystem](https://www.nuget.org/packages/Volo.Abp.BlobStoring.FileSystem)NuGet包到你的项目:

\* 安装 [ABP CLI](https://docs.abp.io/en/abp/latest/CLI), 如果你还没有安装.

\* 在要添加 `Volo.Abp.BlobStoring.FileSystem` 包的 `.csproj` 文件目录打开命令行.

\* 运行 `abp add-package Volo.Abp.BlobStoring.FileSystem` 命令.

如果要手动安装,在你的项目中安装 `Volo.Abp.BlobStoring.FileSystem` NuGet包然后将`[DependsOn(typeof(AbpBlobStoringFileSystemModule))]`添加到项目内的[ABP模块](Module-Development-Basics.md)类中.

## 配置

如同[BLOB存储文档](Blob-Storing.md)所述,配置是在[模块](Module-Development-Basics.md)类的 `ConfigureServices` 方法完成的.

**\*\*示例: 配置为默认使用文件系统存储提供程序\*\***

````csharp

Configure<AbpBlobStoringOptions>(options =>

{

options.Containers.ConfigureDefault(container =>

{

container.UseFileSystem(fileSystem =>

{

fileSystem.BasePath = "C:\\my-files";

});

});

});

````

`UseFileSystem` 扩展方法用于为容器设置文件系统提供程序并配置文件系统选项.

> 参阅[BLOB存储文档](Blob-Storing.md) 学习如何为指定容器配置提供程序.

### 选项

\* **\*\*BasePath\*\*** (string): 存储BLOB的基本文件夹路径,它是必选的.

\* **\*\*AppendContainerNameToBasePath\*\*** (bool; 默认: `true`): 指定是否在基本文件夹中创建具有容器名称的文件夹. 如果你在同一个 `BaseFolder` 中存储多个容器,请将其保留为`true`. 如果你不喜欢不必要的更深层次的文件夹,你可以将它设置为 `false`.

## 文件路径计算

文件系统提供程序在文件夹中组织BLOB文件并实现一些约定. 默认情况下,BLOB文件的完整路径由以下规则确定:

\* 它以如上所述配置的 `BasePath` 开始.

\* 如果当前租户为 `null`(或容器禁用多租户 - 请参阅[BLOB存储文档](Blob-Storing.md) 了解如何禁用容器的多租户),则追加 `host` 文件夹.

\* 如果当前租户不为 `null`,则追加 `tenants/<tenant-id>` 文件夹.

\* 如果 `AppendContainerNameToBasePath` 为`true`,则追加容器的名称. 如果容器名称包含 `/`,将导致文件夹嵌套.

\* 追加BLOB名称,如果BLOB名称包含 `/` 它创建文件夹. 如果BLOB名称包含 `.` 它将有一个文件扩展名.

## 扩展文件系统提供程序

\* `FileSystemBlobProvider` 是实现文件系统存储的主要服务. 你可以从这个类继承并[覆盖](Customizing-Application-Modules-Overriding-Services.md)方法进行自定义.

\* `IBlobFilePathCalculator` 服务用于计算文件路径. 默认实现是 `DefaultBlobFilePathCalculator` . 如果你想自定义文件路径计算,可以替换/覆盖它.

#### 数据库系统提供程序

# BLOB存储数据库提供程序

BLOB存储数据库提供程序可以将BLOB存储在关系或非关系数据库中.

有两个数据库提供程序实现;

\* [Volo.Abp.BlobStoring.Database.EntityFrameworkCore](https://www.nuget.org/packages/Volo.Abp.BlobStoring.Database.EntityFrameworkCore) 包实现[EF Core](Entity-Framework-Core.md), 它可以通过EF Core存储BLOB在[任何支持的DBMS](https://docs.microsoft.com/en-us/ef/core/providers/)中.

\* [Volo.Abp.BlobStoring.Database.MongoDB](https://www.nuget.org/packages/Volo.Abp.BlobStoring.Database.MongoDB) 包实现了[MongoDB](MongoDB.md).

> 阅读[BLOB存储文档](Blob-Storing.md)了解如何使用BLOB存储系统. 本文档仅介绍如何为容器配置数据库提供程序.

## 安装

### 自动安装

如果你已基于[应用程序启动模板](Startup-Templates/Application.md)创建了解决方案,则可以使用 `abp add-module` [CLI](CLI.md)命令将相关软件包自动添加到解决方案中.

在包含解决方案(`.sln`)文件的文件夹中打开命令行运行以下命令:

````bash

abp add-module Volo.Abp.BlobStoring.Database

````

此命令将所有NuGet软件包添加到解决方案的相应层. 如果使用的是EF Core,它会添加必要的配置,添加新的数据库迁移并更新数据库.

### 手动安装

这里是此提供程序定义的所有包:

\* [Volo.Abp.BlobStoring.Database.Domain.Shared](https://www.nuget.org/packages/Volo.Abp.BlobStoring.Database.Domain.Shared)

\* [Volo.Abp.BlobStoring.Database.Domain](https://www.nuget.org/packages/Volo.Abp.BlobStoring.Database.Domain)

\* [Volo.Abp.BlobStoring.Database.EntityFrameworkCore](https://www.nuget.org/packages/Volo.Abp.BlobStoring.Database.EntityFrameworkCore)

\* [Volo.Abp.BlobStoring.Database.MongoDB](https://www.nuget.org/packages/Volo.Abp.BlobStoring.Database.MongoDB)

你可以只安装 `Volo.Abp.BlobStoring.Database.EntityFrameworkCore` 或 `Volo.Abp.BlobStoring.Database.MongoDB` (根据你的偏好),因为它们依赖其他包.

安装完成后,添加 `DepenedsOn` 属性到相关[模块](Module-Development-Basics.md).下面是由上面列出的相关NuGet包定义的模块类列表:

\* `BlobStoringDatabaseDomainModule`

\* `BlobStoringDatabaseDomainSharedModule`

\* `BlobStoringDatabaseEntityFrameworkCoreModule`

\* `BlobStoringDatabaseMongoDbModule`

如果你正在使用EF Core,还需要配置你的**\*\*Migration DbContext\*\***将BLOB存储表添加到你的数据库. 在 `OnModelCreating` 方法中调用 `builder.ConfigureBlobStoring()` 扩展方法来包含到DbContext的映射. 你可以使用标准的 `Add-Migration` 和 `Update-Database` [命令](https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/)在数据库中创建必要的表.

## 配置

### 连接字符串

如果你要使用你的 `Default` 连接字符串,则不需要做任何其他配置.

如果要将BLOB存储到单独的数据库,请在配置文件(`appsettings.json`)中将 `AbpBlobStoring` 用作连接字符串名称. 请阅读[EF Core Migrations](Entity-Framework-Core-Migrations.md)文档了解如何为所需模块创建和使用其他数据库.

### 配置容器

如果只使用数据库存储提供程序,则不需要手动配置,因为它是自动完成的. 如果使用多个存储提供程序,可能需要对其进行配置.

如同[BLOB存储文档](Blob-Storing.md)所述,配置是在[模块](Module-Development-Basics.md)类的 `ConfigureServices` 方法完成的.

**\*\*示例: 配置为默认使用数据库系统存储提供程序\*\***

````csharp

Configure<AbpBlobStoringOptions>(options =>

{

options.Containers.ConfigureDefault(container =>

{

container.UseDatabase();

});

});

````

> 参阅[BLOB存储文档](Blob-Storing.md) 学习如何为指定容器配置提供程序.

## 附加信息

它需要使用[BLOB存储服务](Blob-Storing.md)来使用BLOB存储系统. 但是如果要处理数据库表/实体,可以使用以下信息.

### 实体

此模块定义的实体:

\* `DatabaseBlobContainer` (aggregate root) 表示存储在数据库中的容器.

\* `DatabaseBlob` (aggregate root) 表示数据库中的BLOB.

参阅[实体文档](Entities.md)了解什么是实体和聚合根.

### 仓储

\* `IDatabaseBlobContainerRepository`

\* `IDatabaseBlobRepository`

你还可以使用 `IRepository` 和 `IRepository` 来获得 `IQueryable` 能力. 更多信息请参阅[仓储文档](Repositories.md).

### 其他服务

\* `DatabaseBlobProvider` 是实现数据库BLOB存储提供程序的主要服务,如果你想要通过[依赖注入](Dependency-Injection.md)覆盖/替换它(不要替换 `IBlobProvider` 接口,而是替换 `DatabaseBlobProvider` 类).

#### Azure提供程序

# BLOB Storing Azure提供程序

BLOB存储Azure提供程序可以将BLOB存储在[Azure Blob storage](https://azure.microsoft.com/en-us/services/storage/blobs/)中.

> 阅读[BLOB存储文档](Blob-Storing.md)了解如何使用BLOB存储系统. 本文档仅介绍如何为容器配置Azure提供程序.

## 安装

使用ABP CLI添加[Volo.Abp.BlobStoring.Azure](https://www.nuget.org/packages/Volo.Abp.BlobStoring.Azure)NuGet包到你的项目:

\* 安装 [ABP CLI](https://docs.abp.io/en/abp/latest/CLI), 如果你还没有安装.

\* 在要添加 `Volo.Abp.BlobStoring.Azure` 包的 `.csproj` 文件目录打开命令行.

\* 运行 `Volo.Abp.BlobStoring.Azure` 命令.

如果要手动安装,在你的项目中安装 `Volo.Abp.BlobStoring.Azure` NuGet包然后将`[DependsOn(typeof(AbpBlobStoringAzureModule))]`添加到项目内的[ABP模块](Module-Development-Basics.md)类中.

## 配置

如同[BLOB存储文档](Blob-Storing.md)所述,配置是在[模块](Module-Development-Basics.md)类的 `ConfigureServices` 方法完成的.

**\*\*示例: 配置为默认使用Azure存储提供程序\*\***

````csharp

Configure<AbpBlobStoringOptions>(options =>

{

options.Containers.ConfigureDefault(container =>

{

container.UseAzure(azure =>

{

azure.ConnectionString = "your azure connection string";

azure.ContainerName = "your azure container name";

azure.CreateContainerIfNotExists = true;

});

});

});

````

> 参阅[BLOB存储文档](Blob-Storing.md) 学习如何为指定容器配置提供程序.

### 选项

\* **\*\*ConnectionString\*\*** (string): 连接字符串包括应用程序在运行时使用共享密钥授权访问Azure存储帐户中的数据所需的授权信息. 请参考[Azure文档](https://docs.microsoft.com/en-us/azure/storage/common/storage-configure-connection-string).

\* **\*\*ContainerName\*\*** (string): 你可以在azure中指定容器名称. 如果没有指定它将使用 `BlobContainerName` 属性定义的BLOB容器的名称(请参阅[BLOB存储文档](Blob-Storing.md)). 请注意Azure有一些**\*\*命名容器的规则\*\***,容器名称必须是有效的DNS名称,[符合以下命名规则](https://docs.microsoft.com/en-us/rest/api/storageservices/naming-and-referencing-containers--blobs--and-metadata#container-names):

\* 容器名称必须以字母或数字开头或结尾,并且只能包含字母,数字和破折号(-)字符.

\* 每个破折号(-)必须紧跟在字母或数字之后;容器名称中不允许使用连续的破折号.

\* 容器名称中的所有字母都必须**\*\*小写\*\***.

\* 容器名称的长度必须在**\*\*3\*\***到**\*\*63\*\***个字符之间.

\* **\*\*CreateContainerIfNotExists\*\*** (bool): 默认值为 `false`, 如果azure中不存在容器, `AzureBlobProvider` 将尝试创建它.

## Azure BLOB 名称计算器

Azure BLOB提供程序组织BLOB名称并实现一些约定. 默认情况下BLOB的全名由以下规则确定:

\* 如果当前租户为 `null`(或容器禁用多租户 - 请参阅[BLOB存储文档](Blob-Storing.md) 了解如何禁用容器的多租户),则追加 `host` 字符串.

\* 如果当前租户不为 `null`,则追加 `tenants/<tenant-id>` 字符串.

\* 追加 BLOB 名称.

## 其他服务

\* `AzureBlobProvider` 是实现Azure BLOB存储提供程序的主要服务,如果你想要通过[依赖注入](Dependency-Injection.md)覆盖/替换它(不要替换 `IBlobProvider` 接口,而是替换 `AzureBlobProvider` 类).

\* `IAzureBlobNameCalculator` 服务用于计算文件路径. 默认实现是 `DefaultAzureBlobNameCalculator` . 如果你想自定义文件路径计算,可以替换/覆盖它.

#### Aliyun提供程序

# BLOB Storing Aliyun提供程序

BLOB存储Aliyun提供程序可以将BLOB存储在[Aliyun Blob storage](https://help.aliyun.com/product/31815.html)中.

> 阅读[BLOB存储文档](Blob-Storing.md)了解如何使用BLOB存储系统. 本文档仅介绍如何为容器配置Aliyun提供程序.

## 安装

使用ABP CLI添加[Volo.Abp.BlobStoring.Aliyun](https://www.nuget.org/packages/Volo.Abp.BlobStoring.Aliyun)NuGet包到你的项目:

\* 安装 [ABP CLI](https://docs.abp.io/en/abp/latest/CLI), 如果你还没有安装.

\* 在要添加 `Volo.Abp.BlobStoring.Aliyun` 包的 `.csproj` 文件目录打开命令行.

\* 运行 `Volo.Abp.BlobStoring.Aliyun` 命令.

如果要手动安装,在你的项目中安装 `Volo.Abp.BlobStoring.Aliyun` NuGet包然后将`[DependsOn(typeof(AbpBlobStoringAliyunModule))]`添加到项目内的[ABP模块](Module-Development-Basics.md)类中.

## 配置

如同[BLOB存储文档](Blob-Storing.md)所述,配置是在[模块](Module-Development-Basics.md)类的 `ConfigureServices` 方法完成的.

**\*\*示例: 配置为默认使用Aliyun存储提供程序\*\***

````csharp

Configure<AbpBlobStoringOptions>(options =>

{

options.Containers.ConfigureDefault(container =>

{

container.UseAliyun(aliyun =>

{

aliyun.AccessKeyId = "your aliyun access key id";

aliyun.AccessKeySecret = "your aliyun access key secret";

aliyun.Endpoint = "your oss endpoint";

aliyun.RegionId = "your sts region id";

aliyun.RoleArn = "the arn of ram role";

aliyun.RoleSessionName = "the name of the certificate";

aliyun.Policy = "policy";

aliyun.DurationSeconds = "expiration date";

aliyun.ContainerName = "your aliyun container name";

aliyun.CreateContainerIfNotExists = true;

});

});

});

````

> 参阅[BLOB存储文档](Blob-Storing.md) 学习如何为指定容器配置提供程序.

### 选项

\* **\*\*AccessKeyId\*\*** ([NotNull]string): 云账号AccessKey是访问阿里云API的密钥,具有该账户完全的权限,请你务必妥善保管!强烈建议遵循[阿里云安全最佳实践](https://help.aliyun.com/document\_detail/102600.html),使用RAM子用户AccessKey来进行API调用.

\* **\*\*AccessKeySecret\*\*** ([NotNull]string): 同上.

\* **\*\*Endpoint\*\*** ([NotNull]string): Endpoint表示OSS对外服务的访问域名. [访问域名和数据中心](https://help.aliyun.com/document\_detail/31837.html)

\* **\*\*UseSecurityTokenService\*\*** (bool): 是否使用STS临时授权访问OSS,默认false. [STS临时授权访问OSS](https://help.aliyun.com/document\_detail/100624.html)

\* **\*\*RegionId\*\*** (string): STS服务的接入地址,每个地址的功能都相同,请尽量在同地域进行调用. [接入地址](https://help.aliyun.com/document\_detail/66053.html)

\* **\*\*RoleArn\*\*** ([NotNull]string): STS所需角色ARN.

\* **\*\*RoleSessionName\*\*** ([NotNull]string): 用来标识临时访问凭证的名称,建议使用不同的应用程序用户来区分.

\* **\*\*Policy\*\*** (string): 在扮演角色的时候额外添加的权限限制. 请参见[基于RAM Policy的权限控制](https://help.aliyun.com/document\_detail/100680.html).

\* **\*\*DurationSeconds\*\*** (int): 设置临时访问凭证的有效期,单位是s,最小为900,最大为3600.

\* **\*\*ContainerName\*\*** (string): 你可以在aliyun中指定容器名称. 如果没有指定它将使用 `BlobContainerName` 属性定义的BLOB容器的名称(请参阅[BLOB存储文档](Blob-Storing.md)). 请注意Aliyun有一些**\*\*命名容器的规则\*\***,容器名称必须是有效的DNS名称,[符合以下命名规则](https://help.aliyun.com/knowledge\_detail/39668.html):

\* 只能包含小写字母,数字和短横线(-)

\* 必须以小写字母和数字开头和结尾

\* Bucket名称的长度限制在**\*\*3\*\***到**\*\*63\*\***个字符之间

\* **\*\*CreateContainerIfNotExists\*\*** (bool): 默认值为 `false`, 如果aliyun中不存在容器, `AliyunBlobProvider` 将尝试创建它.

\* **\*\*TemporaryCredentialsCacheKey\*\*** (bool): STS凭证缓存Key,默认Guid.NewGuid().ToString("N").

## Aliyun BLOB 名称计算器

Aliyun BLOB提供程序组织BLOB名称并实现一些约定. 默认情况下BLOB的全名由以下规则确定:

\* 如果当前租户为 `null`(或容器禁用多租户 - 请参阅[BLOB存储文档](Blob-Storing.md) 了解如何禁用容器的多租户),则追加 `host` 字符串.

\* 如果当前租户不为 `null`,则追加 `tenants/<tenant-id>` 字符串.

\* 追加 BLOB 名称.

## 其他服务

\* `AliyunBlobProvider` 是实现Aliyun BLOB存储提供程序的主要服务,如果你想要通过[依赖注入](Dependency-Injection.md)覆盖/替换它(不要替换 `IBlobProvider` 接口,而是替换 `AliyunBlobProvider` 类).

\* `IAliyunBlobNameCalculator` 服务用于计算文件路径. 默认实现是 `DefaultAliyunBlobNameCalculator`. 如果你想自定义文件路径计算,可以替换/覆盖它.

\* `IOssClientFactory` 服务用于生成OSS客户端. 默认实现是 `DefaultOssClientFactory` . 如果你想自定义OSS客户端生成,可以替换/覆盖它.

#### Minio提供程序

# BLOB Storing Minio 提供程序

BLOB Storing Minio提供程序帮助你存储对象到 [MinIO Object storage](https://min.io/),

> 阅读[BLOB存储文档](Blob-Storing.md)了解如何使用BLOB存储系统, 本文档仅介绍如何为容器配置Minio提供程序,

## 安装

使用 ABP CLI 来安装 [Volo.Abp.BlobStoring.Minio](https://www.nuget.org/packages/Volo.Abp.BlobStoring.Minio) NuGet 包到你的项目：

\* 如果你没有安装ABP CLI,请先安装 [ABP CLI](https://docs.abp.io/en/abp/latest/CLI),

\* 在要添加 `Volo.Abp.BlobStoring.Minio` 包的 `.csproj` 文件目录打开命令行,

\* 执行 `abp add-package Volo.Abp.BlobStoring.Minio` 命令,

如果你要手动安装, 通过NuGet安装 [Volo.Abp.BlobStoring.Minio](https://www.nuget.org/packages/Volo.Abp.BlobStoring.Minio) 到你的项目,然后添加 `[DependsOn(typeof(AbpBlobStoringMinioModule))]` 特性到你的 [ABP module](Module-Development-Basics.md) 类上,

## 配置

配置在你的[module](Module-Development-Basics.md)类中的`ConfigureServices`方法中完成,

**\*\*例: 配置使用Minio存储\*\***

````csharp

Configure<AbpBlobStoringOptions>(options =>

{

options.Containers.ConfigureDefault(container =>

{

container.UseMinio(minio =>

{

minio.EndPoint = "你的 minio endPoint";

minio.AccessKey = "你的 minio accessKey";

minio.SecretKey = "你的 minio secretKey";

minio.BucketName = "你的 minio bucketName";

});

});

});

````

> 参阅[BLOB存储文档](Blob-Storing.md) 学习如何为指定容器配置提供程序,

### 选项

\* **\*\*EndPoint\*\*** (string): 你的Minio对象存储服务的URL, 查看文档：https://docs.min.io/docs/dotnet-client-quickstart-guide.html

\* **\*\*AccessKey\*\*** (string): Access key是唯一标识你的账户的用户ID,

\* **\*\*SecretKey\*\*** (string): Secret Key是你的账户的密码

\* **\*\*BucketName\*\*** (string):你可以指定bucket名称,如果没有指定,将使用 `BlobContainerName` 属性定义的BLOB容器的名称(查阅[BLOB storing document](Blob-Storing.md)),MinIO完全兼容S3标准,所以有一些 **\*\*bucket命名规则\*\***,必须符合[规则](https://docs.aws.amazon.com/AmazonS3/latest/dev/BucketRestrictions.html):

\* Bucket名称必须介于 3 到 63 个字符之间.

\* Bucket名称只能由小写字母、数字、句点 (.) 和连字符 (-) 组成.

\* Bucket名称必须以字母或数字开头和结尾.

\* Bucket名称不能是ip (例如, 192.168.5.4).

\* Bucket名称不能以 **\*\*xn--\*\*** 开头, (对于2020年2月以后创建的 Bucket).

\* Bucket名称在分区中必须唯一 .

\* Bucket 与 Amazon S3 Transfer Acceleration 一起使用时名称中不能有句点 (.).

\* **\*\*WithSSL\*\*** (bool): 默认 `false`,代表使用HTTPS,

\* **\*\*CreateContainerIfNotExists\*\*** (bool): 默认 `false`,如果不存在bucket, `MinioBlobProvider` 将会创建一个,

## Minio BLOB 名称计算器

默认情况下BLOB的全名由以下规则确定:

\* 如果当前租户为 `null`(或容器禁用多租户 - 请参阅[BLOB存储文档](Blob-Storing.md) 了解如何禁用容器的多租户),则追加 `host` 字符串,

\* 如果当前租户不为 `null`,则追加 `tenants/<tenant-id>` 字符串,

\* 追加 BLOB 名称,

## 其他服务

\* `MinioBlobProvider` 是实现Minio BLOB存储提供程序的主要服务,如果你想要通过[依赖注入](Dependency-Injection.md)覆盖/替换它(不要替换 `IBlobProvider` 接口,而是替换 `MinioBlobProvider` 类).

\* `IMinioBlobNameCalculator` 服务用于计算文件路径. 默认实现是 `DefaultMinioBlobNameCalculator`. 如果你想自定义文件路径计算,可以替换/覆盖它.

#### AWS提供程序

# BLOB Storing Aws提供程序

BLOB存储Aws提供程序可以将BLOB存储在[Amazon Simple Storage Service](https://aws.amazon.com/cn/s3/)中.

> 阅读[BLOB存储文档](Blob-Storing.md)了解如何使用BLOB存储系统. 本文档仅介绍如何为容器配置Aws提供程序.

## 安装

使用ABP CLI添加[Volo.Abp.BlobStoring.Aws](https://www.nuget.org/packages/Volo.Abp.BlobStoring.Aws)NuGet包到你的项目:

\* 安装 [ABP CLI](https://docs.abp.io/en/abp/latest/CLI), 如果你还没有安装.

\* 在要添加 `Volo.Abp.BlobStoring.Aws` 包的 `.csproj` 文件目录打开命令行.

\* 运行 `Volo.Abp.BlobStoring.Aws` 命令.

如果要手动安装,在你的项目中安装 `Volo.Abp.BlobStoring.Aws` NuGet包然后将`[DependsOn(typeof(AbpBlobStoringAwsModule))]`添加到项目内的[ABP模块](Module-Development-Basics.md)类中.

## 配置

如同[BLOB存储文档](Blob-Storing.md)所述,配置是在[模块](Module-Development-Basics.md)类的 `ConfigureServices` 方法完成的.

**\*\*示例: 配置为默认使用Aws存储提供程序\*\***

````csharp

Configure<AbpBlobStoringOptions>(options =>

{

options.Containers.ConfigureDefault(container =>

{

container.UseAws(Aws =>

{

Aws.AccessKeyId = "your Aws access key id";

Aws.SecretAccessKey = "your Aws access key secret";

Aws.UseCredentials = "set true to use credentials";

Aws.UseTemporaryCredentials = "set true to use temporary credentials";

Aws.UseTemporaryFederatedCredentials = "set true to use temporary federated credentials";

Aws.ProfileName = "the name of the profile to get credentials from";

Aws.ProfilesLocation = "the path to the aws credentials file to look at";

Aws.Region = "the system name of the service";

Aws.Name = "the name of the federated user";

Aws.Policy = "policy";

Aws.DurationSeconds = "expiration date";

Aws.ContainerName = "your Aws container name";

Aws.CreateContainerIfNotExists = true;

});

});

});

````

> 参阅[BLOB存储文档](Blob-Storing.md) 学习如何为指定容器配置提供程序.

### 选项

\* **\*\*AccessKeyId\*\*** (string): AWS Access Key ID.

\* **\*\*SecretAccessKey\*\*** (string): AWS Secret Access Key.

\* **\*\*UseCredentials\*\*** (bool): 使用[本地凭证](https://docs.aws.amazon.com/zh\_cn/AmazonS3/latest/dev/AuthUsingAcctOrUserCredentials.html)访问AWS服务,默认: `false`.

\* **\*\*UseTemporaryCredentials\*\*** (bool): 使用[临时凭证](https://docs.aws.amazon.com/zh\_cn/AmazonS3/latest/dev/AuthUsingTempSessionToken.html)访问AWS服务,默认: `false`.

\* **\*\*UseTemporaryFederatedCredentials\*\*** (bool): 使用[联合身份用户临时凭证](https://docs.aws.amazon.com/zh\_cn/AmazonS3/latest/dev/AuthUsingTempFederationToken.html)访问AWS服务, 默认: `false`.

\* **\*\*ProfileName\*\*** (string): [本地凭证配置文件](https://docs.aws.amazon.com/zh\_cn/sdk-for-net/v3/developer-guide/net-dg-config-creds.html)名称.

\* **\*\*ProfilesLocation\*\*** (string): 本地配置文件位置.

\* **\*\*Region\*\*** (string): 服务的地区名称.

\* **\*\*Policy\*\*** (string): JSON格式的IAM策略.

\* **\*\*DurationSeconds\*\*** (int): 设置临时访问凭证的有效期,单位是s,最小为900,最大为129600.

\* **\*\*ContainerName\*\*** (string): 你可以在Aws中指定容器名称. 如果没有指定它将使用 `BlobContainerName` 属性定义的BLOB容器的名称(请参阅[BLOB存储文档](Blob-Storing.md)). 请注意Aws有一些**\*\*命名容器的规则\*\***,容器名称必须是有效的DNS名称,[符合以下命名规则](https://docs.aws.amazon.com/AmazonS3/latest/dev/BucketRestrictions.html):

\* Bucket名称必须介于 3 到 63 个字符之间.

\* Bucket名称只能由小写字母、数字、句点 (.) 和连字符 (-) 组成.

\* Bucket名称必须以字母或数字开头和结尾.

\* Bucket名称不能是ip (例如, 192.168.5.4).

\* Bucket名称不能以 **\*\*xn--\*\*** 开头, (对于2020年2月以后创建的 Bucket).

\* Bucket名称在分区中必须唯一 .

\* Bucket 与 Amazon S3 Transfer Acceleration 一起使用时名称中不能有句点 (.).

\* **\*\*CreateContainerIfNotExists\*\*** (bool): 默认值为 `false`, 如果Aws中不存在容器, `AwsBlobProvider` 将尝试创建它.

## Aws BLOB 名称计算器

Aws BLOB提供程序组织BLOB名称并实现一些约定. 默认情况下BLOB的全名由以下规则确定:

\* 如果当前租户为 `null`(或容器禁用多租户 - 请参阅[BLOB存储文档](Blob-Storing.md) 了解如何禁用容器的多租户),则追加 `host` 字符串.

\* 如果当前租户不为 `null`,则追加 `tenants/<tenant-id>` 字符串.

\* 追加 BLOB 名称.

## 其他服务

\* `AwsBlobProvider` 是实现Aws BLOB存储提供程序的主要服务,如果你想要通过[依赖注入](Dependency-Injection.md)覆盖/替换它(不要替换 `IBlobProvider` 接口,而是替换 `AwsBlobProvider` 类).

\* `IAwsBlobNameCalculator` 服务用于计算文件路径. 默认实现是 `DefaultAwsBlobNameCalculator`. 如果你想自定义文件路径计算,可以替换/覆盖它.

\* `IAmazonS3ClientFactory` 服务用于生成AWS S3客户端. 默认实现是 `DefaultAmazonS3ClientFactory` . 如果你想自定义AWS S3客户端生成,可以替换/覆盖它.

#### 创建自定义提供程序

# BLOB 存储: 创建自定义提供程序

本文档通过一个示例说明如何为BLOB存储系统创建新的存储提供程序.

> 阅读[BLOB存储文档](Blob-Storing.md)了解如何使用BLOB存储系统. 本文档仅介绍如何创建新存储提供程序.

## 示例实现

第一步是创建一个实现 `IBlobProvider` 接口或 `BlobProviderBase` 抽象类继承的类.

````csharp

using System.IO;

using System.Threading.Tasks;

using Volo.Abp.BlobStoring;

using Volo.Abp.DependencyInjection;

namespace AbpDemo

{

public class MyCustomBlobProvider : BlobProviderBase, ITransientDependency

{

public override Task SaveAsync(BlobProviderSaveArgs args)

{

//TODO...

}

public override Task<bool> DeleteAsync(BlobProviderDeleteArgs args)

{

//TODO...

}

public override Task<bool> ExistsAsync(BlobProviderExistsArgs args)

{

//TODO...

}

public override Task<Stream> GetOrNullAsync(BlobProviderGetArgs args)

{

//TODO...

}

}

}

````

\* `MyCustomBlobProvider` 继承 `BlobProviderBase` 并覆盖 `abstract` 方法. 实际的实现取决于你.

\* 实现 `ITransientDependency` 接口将这个类注做为瞬态服务注册到[依赖注入](Dependency-Injection.md)系统.

> **\*\*注意: 命名约定很重要\*\***. 如果类名没有以 `BlobProvider` 结尾,则必须手动注册/公开你的服务为 `IBlobProvider`.

这是所有. 现在你可以配置容器(在[模块](Module-Development-Basics.md)的 `ConfigureServices` 方法中)使用 `MyCustomBlobProvider` 类:

````csharp

Configure<AbpBlobStoringOptions>(options =>

{

options.Containers.ConfigureDefault(container =>

{

container.ProviderType = typeof(MyCustomBlobProvider);

});

});

````

> 如果你想配置特定的容器,请参阅[BLOB存储文档](Blob-Storing.md).

### BlobContainerConfiguration 扩展方法

如果你想提供一个更简单的配置方式,可以为 `BlobContainerConfiguration` 类创建一个扩展方法:

````csharp

public static class MyBlobContainerConfigurationExtensions

{

public static BlobContainerConfiguration UseMyCustomBlobProvider(

this BlobContainerConfiguration containerConfiguration)

{

containerConfiguration.ProviderType = typeof(MyCustomBlobProvider);

return containerConfiguration;

}

}

````

然后你可以使用扩展方法更容易地配置容器:

````csharp

Configure<AbpBlobStoringOptions>(options =>

{

options.Containers.ConfigureDefault(container =>

{

container.UseMyCustomBlobProvider();

});

});

````

### 额外的配置选项

`BlobContainerConfiguration` 允许添加/删除提供程序特定的配置对象. 如果你的提供者需要额外的配置,你可以为 `BlobContainerConfiguration` 创建一个包装类提供的类型安全配置选项:

````csharp

public class MyCustomBlobProviderConfiguration

{

public string MyOption1

{

get => \_containerConfiguration

.GetConfiguration<string>("MyCustomBlobProvider.MyOption1");

set => \_containerConfiguration

.SetConfiguration("MyCustomBlobProvider.MyOption1", value);

}

private readonly BlobContainerConfiguration \_containerConfiguration;

public MyCustomBlobProviderConfiguration(

BlobContainerConfiguration containerConfiguration)

{

\_containerConfiguration = containerConfiguration;

}

}

````

然后你可以这样更改 `MyBlobContainerConfigurationExtensions` 类:

````csharp

public static class MyBlobContainerConfigurationExtensions

{

public static BlobContainerConfiguration UseMyCustomBlobProvider(

this BlobContainerConfiguration containerConfiguration,

Action<MyCustomBlobProviderConfiguration> configureAction)

{

containerConfiguration.ProviderType = typeof(MyCustomBlobProvider);

configureAction.Invoke(

new MyCustomBlobProviderConfiguration(containerConfiguration)

);

return containerConfiguration;

}

public static MyCustomBlobProviderConfiguration GetMyCustomBlobProviderConfiguration(

this BlobContainerConfiguration containerConfiguration)

{

return new MyCustomBlobProviderConfiguration(containerConfiguration);

}

}

````

\* 向 `UseMyCustomBlobProvider` 方法添加了一个参数,允许开发人员设置其他选项.

\* 添加了一个新的 `GetMyCustomBlobProviderConfiguration` 方法,该方法将在 `MyCustomBlobProvider` 类内使用获取配置的值.

然后任何人都可以如下设置 `MyOption1`:

````csharp

Configure<AbpBlobStoringOptions>(options =>

{

options.Containers.ConfigureDefault(container =>

{

container.UseMyCustomBlobProvider(provider =>

{

provider.MyOption1 = "my value";

});

});

});

````

最后你可以使用 `GetMyCustomBlobProviderConfiguration` 方法访问额外的选项:

````csharp

public class MyCustomBlobProvider : BlobProviderBase, ITransientDependency

{

public override Task SaveAsync(BlobProviderSaveArgs args)

{

var config = args.Configuration.GetMyCustomBlobProviderConfiguration();

var value = config.MyOption1;

//...

}

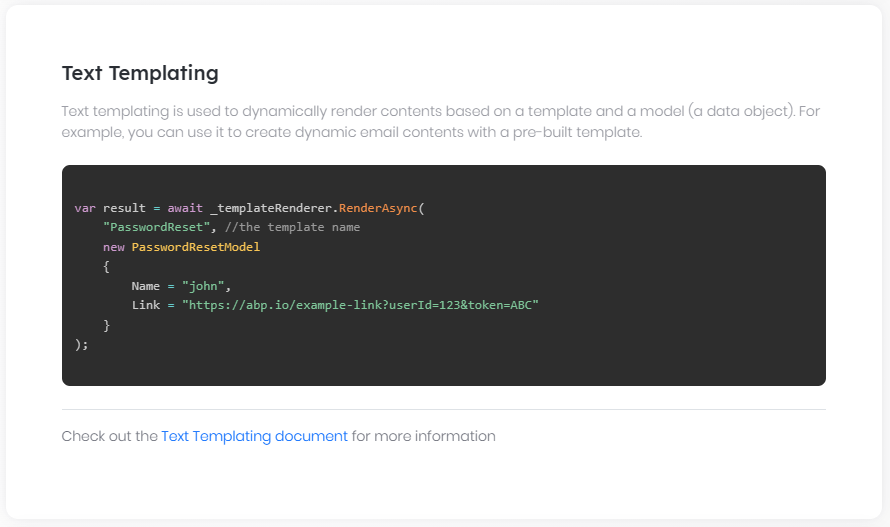
}

````

## 贡献?

如果你创建了一个新的提供程序,并且认为它对其他开发者有用,请考虑为GitHub上的ABP框架做出[贡献](Contribution/Index.md).

## 文本模版



# 文本模板

## 介绍

ABP框架提供了一个简单有效的文本模板系统,文本模板用于动态渲染基于模板和模型(数据对象)内容:

Template + Model =renderer=> Rendered Content

它非常类似于 ASP.NET Core Razor View (或 Page):

*\*RAZOR VIEW (或 PAGE) + MODEL* ***==render==****> HTML CONTENT\**

你可以将渲染的输出用于任何目的,例如发送电子邮件或准备一些报告.

模板渲染引擎非常强大:

\* 它支持**\*\*条件逻辑\*\***, **\*\*循环\*\***等等.

\* 模板内容**\*\*可以本地化\*\***.

\* 你可以为其他渲染模板定义**\*\*布局模板\*\***。

\* 对于高级场景,你可以传递任何对象到模板上下文.

ABP框架提供了两个模板引擎:

\* **\*\*[**Razor**](Text-Templating-Razor.md)\*\***

\* **\*\*[**Scriban**](Text-Templating-Scriban.md)\*\***

你可以在同一个应用应用程序中使用不同的模板引擎, 或者创建一个新的自定义模板引擎.

## 源码

查看开发和引用的[应用程序示例源码](https://github.com/abpframework/abp-samples/tree/master/TextTemplateDemo).

## 另请参阅

\* 本文开发和引用的[应用程序示例源码](https://github.com/abpframework/abp-samples/tree/master/TextTemplateDemo).

\* [本地化系统](Localization.md).

\* [虚拟文件系统](Virtual-File-System.md).

Razor 集成

# Razor 集成

Razor模板是标准的C#类, 所以你可以使用任何C#的功能, 例如`依赖注入`, 使用`LINQ`, 自定义方法甚至使用`仓储`

## 安装

建议使用[ABP CLI](CLI.md)安装此包．

### 使用ABP CLI

在项目文件夹(.csproj 文件)中打开命令行窗口并输入以下命令:

````bash

abp add-package Volo.Abp.TextTemplating.Razor

````

### 手动安装

如果你想要手动安装:

1. 添加 [Volo.Abp.TextTemplating.Razor](https://www.nuget.org/packages/Volo.Abp.TextTemplating.Razor) NuGet 包到你的项目:

````

Install-Package Volo.Abp.TextTemplating.Razor

````

2.添加 `AbpTextTemplatingRazorModule` 到你的模块的依赖列表:

````csharp

[DependsOn(

//...other dependencies

typeof(AbpTextTemplatingRazorModule) //Add the new module dependency

)]

public class YourModule : AbpModule

{

}

````

## 添加 MetadataReference到CSharpCompilerOptions

你需要将添加`MetadataReference`模板中使用的类型添加到 `CSharpCompilerOptions` 的 `References`.

```csharp

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpRazorTemplateCSharpCompilerOptions>(options =>

{

options.References.Add(MetadataReference.CreateFromFile(typeof(YourModule).Assembly.Location));

});

}

```

## 添加MetadataReference到模板

你可以添加一些`MetadataReference`到模板

```csharp

public override void ConfigureServices(ServiceConfigurationContext context)

{

services.Configure<AbpCompiledViewProviderOptions>(options =>

{

//Hello is template name.

options.TemplateReferences.Add("Hello", new List<Assembly>()

{

Assembly.Load("Microsoft.Extensions.Logging.Abstractions"),

Assembly.Load("Microsoft.Extensions.Logging")

}

.Select(x => MetadataReference.CreateFromFile(x.Location))

.ToList());

});

}

```

## 定义模板

在渲染模板之前,需要定义它. 创建一个继承自 `TemplateDefinitionProvider` 的类:

````csharp

public class DemoTemplateDefinitionProvider : TemplateDefinitionProvider

{

public override void Define(ITemplateDefinitionContext context)

{

context.Add(

new TemplateDefinition("Hello") //template name: "Hello"

.WithRazorEngine()

.WithVirtualFilePath(

"/Demos/Hello/Hello.cshtml", //template content path

isInlineLocalized: true

)

);

}

}

````

\* `context` 对象用于添加新模板或获取依赖模块定义的模板. 使用 `context.Add(...)` 定义新模板.

\* `TemplateDefinition` 是代表模板的类,每个模板必须有唯一的名称(在渲染模板时使用).

\* `/Demos/Hello/Hello.cshtml` 是模板文件的路径.

\* `isInlineLocalized` 声明针对所有语言使用一个模板(`true` 还是针对每种语言使用不同的模板(`false`). 更多内容参阅下面的本地化部分.

\* `WithRenderEngine` 方法为模板设置渲染引擎.

### 模板基类

每个 `cshtml` 模板页面都需要继承`RazorTemplatePageBase` 或 `RazorTemplatePageBase<Model>`. 基类提供了一些使用实用的属性可以在模板中使用. 例如: `Localizer`, `ServiceProvider`.

### 模板内容

`WithVirtualFilePath` 表示我们使用[虚拟文件系统](Virtual-File-System.md)存储模板内容. 在项目内创建一个 `Hello.cshtml` 文件,并在属性窗口中将其标记为"**\*\*嵌入式资源\*\***":

![hello-template-razor](images/hello-template-razor.png)

示例 `Hello.cshtml` 内容如下所示:

```csharp

namespace HelloModelNamespace

{

public class HelloModel

{

public string Name { get; set; }

}

}

```

[虚拟文件系统](Virtual-File-System.md) 需要在[模块](Module-Development-Basics.md)类的 `ConfigureServices` 方法添加你的文件:

````csharp

Configure<AbpVirtualFileSystemOptions>(options =>

{

options.FileSets.AddEmbedded<TextTemplateDemoModule>("TextTemplateDemo");

});

````

\* `TextTemplateDemoModule`是模块类.

\* `TextTemplateDemo` 是你的项目的根命名空间.

## 渲染模板

`ITemplateRenderer` 服务用于渲染模板内容.

### 示例: 渲染一个简单的模板

````csharp

public class HelloDemo : ITransientDependency

{

private readonly ITemplateRenderer \_templateRenderer;

public HelloDemo(ITemplateRenderer templateRenderer)

{

\_templateRenderer = templateRenderer;

}

public async Task RunAsync()

{

var result = await \_templateRenderer.RenderAsync(

"Hello", //the template name

new HelloModel

{

Name = "John"

}

);

Console.WriteLine(result);

}

}

````

\* `HelloDemo` 是一个简单的类,在构造函数注入了 `ITemplateRenderer` 并在 `RunAsync` 方法中使用它.

\* `RenderAsync` 有两个基本参数:

\* `templateName`: 要渲染的模板名称 (本示例中是 `Hello`).

\* `model`: 在模板内部用做 `model` 的对象 (本示例中是 `HelloModel` 对象).

示例会返回以下结果:

````csharp

Hello John :)

````

## 本地化

可以基于当前文化对模板内容进行本地化. 以下部分描述了两种类型的本地化选项.

### 内联本地化

内联本地化使用[本地化系统](Localization.md)本地化模板内的文本.

#### 示例: 重置密码链接

假设你需要向用户发送电子邮件重置密码. 模板内容:

```csharp

namespace ResetMyPasswordModelNamespace

{

public class ResetMyPasswordModel

{

public string Link { get; set; }

public string Name { get; set; }

}

}

```

```csharp

@inherits Volo.Abp.TextTemplating.Razor.RazorTemplatePageBase<ResetMyPasswordModelNamespace.ResetMyPasswordModel>

<a title="@Localizer["ResetMyPasswordTitle"]" href="@Model.Link">@Localizer["ResetMyPassword", Model.Name]</a>

```

`Localizer` 函数用于根据当前用户的文化来定位给定的Key,你需要在本地化文件中定义 `ResetMyPassword` 键:

````json

"ResetMyPasswordTitle": "Reset my password",

"ResetMyPassword": "Hi {0}, Click here to reset your password"

````

你还需要在模板定义提供程序类中声明要与此模板一起使用的本地化资源:

````csharp

context.Add(

new TemplateDefinition(

"PasswordReset", //Template name

typeof(DemoResource) //LOCALIZATION RESOURCE

)

.WithRazorEngine()

.WithVirtualFilePath(

"/Demos/PasswordReset/PasswordReset.cshtml", //template content path

isInlineLocalized: true

)

);

````

当你这样渲染模板时:

````csharp

var result = await \_templateRenderer.RenderAsync(

"PasswordReset", //the template name

new PasswordResetModel

{

Name = "john",

Link = "https://abp.io/example-link?userId=123&token=ABC"

}

);

````

你可以看到以下本地化结果:

````csharp

<a title="Reset my password" href="https://abp.io/example-link?userId=123&token=ABC">Hi john, Click here to reset your password</a>

````

> 如果你为应用程序定义了 [默认本地化资源](Localization.md), 则无需声明模板定义的资源类型.

### 多个内容本地化

你可能希望为每种语言创建不同的模板文件,而不是使用本地化系统本地化单个模板. 如果模板对于特定的文化(而不是简单的文本本地化)应该是完全不同的,则可能需要使用它.

#### 示例: 欢迎电子邮件模板

假设你要发送电子邮件欢迎用户,但要定义基于用户的文化完全不同的模板.

首先创建一个文件夹,将模板放在里面，像 `en.cshtml`, `tr.cshtml` 每一个你支持的文化:

![multiple-file-template-razor](images/multiple-file-template-razor.png)

然后在模板定义提供程序类中添加模板定义:

````csharp

context.Add(

new TemplateDefinition(

name: "WelcomeEmail",

defaultCultureName: "en"

)

.WithRazorEngine()

.WithVirtualFilePath(

"/Demos/WelcomeEmail/Templates", //template content folder

isInlineLocalized: false

)

);

````

\* 设置 **\*\*默认文化名称\*\***, 当没有所需的文化模板,回退到缺省文化.

\* 指定 **\*\*模板文件夹\*\*** 而不是单个模板文件.

\* 设置 `isInlineLocalized` 为 `false`.

就这些,你可以渲染当前文化的模板:

````csharp

var result = await \_templateRenderer.RenderAsync("WelcomeEmail");

````

> 为了简单我们跳过了模型,但是你可以使用前面所述的模型.

### 指定文化

`ITemplateRenderer` 服务如果没有指定则使用当前文化 (`CultureInfo.CurrentUICulture`). 如果你需要你可以使用 `cultureName` 参数指定文化.

````csharp

var result = await \_templateRenderer.RenderAsync(

"WelcomeEmail",

cultureName: "en"

);

````

## 布局模板

布局模板用于在其他模板之间创建共享布局. 它类似于ASP.NET Core MVC / Razor Pages中的布局系统.

### 示例: 邮件HTML布局模板

例如,你想为所有电子邮件模板创建一个布局.

首先像之前一样创建一个模板文件:

```csharp

@inherits Volo.Abp.TextTemplating.Razor.RazorTemplatePageBase

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">

<head>

<meta charset="utf-8" />

</head>

<body>

@Body

</body>

</html>

```

\* 布局模板必须具有 **\*\*Body\*\*** 部分作为渲染的子内容的占位符.

在模板定义提供程序中注册模板:

````csharp

context.Add(

new TemplateDefinition(

"EmailLayout",

isLayout: true //SET isLayout!

)

.WithRazorEngine()

.WithVirtualFilePath(

"/Demos/EmailLayout/EmailLayout.cshtml",

isInlineLocalized: true

)

);

````

现在你可以将此模板用作任何其他模板的布局:

````csharp

context.Add(

new TemplateDefinition(

name: "WelcomeEmail",

defaultCultureName: "en",

layout: "EmailLayout" //Set the LAYOUT

)

.WithRazorEngine()

.WithVirtualFilePath(

"/Demos/WelcomeEmail/Templates",

isInlineLocalized: false

)

);

````

## 全局上下文

ABP传递 `model`,可用于访问模板内的模型. 如果需要,可以传递更多的全局变量.

示例模板内容:

````csharp

@inherits Volo.Abp.TextTemplating.Razor.RazorTemplatePageBase

A global object value: @GlobalContext["myGlobalObject"]

````

模板假定它渲染上下文中的 `myGlobalObject` 对象. 你可以如下所示提供它:

````csharp

var result = await \_templateRenderer.RenderAsync(

"GlobalContextUsage",

globalContext: new Dictionary<string, object>

{

{"myGlobalObject", "TEST VALUE"}

}

);

````

渲染的结果将是:

````

A global object value: TEST VALUE

````

## 替换存在的模板

通过替换应用程序中使用的模块定义的模板. 这样你可以根据自己的需求自定义模板,而无需更改模块代码.

### 选项-1: 使用虚拟文件系统

[虚拟文件系统](Virtual-File-System.md)允许你通过将相同文件放入项目中的相同路径来覆盖任何文件.

#### 示例: 替换标准电子邮件布局模板

ABP框架提供了一个[邮件发送系统](Emailing.md), 它在内部使用文本模板来渲染邮件内容. 它在 `/Volo/Abp/Emailing/Templates/Layout.cshtml` 路径定义了一个标准邮件布局模板. 模板的唯一名称是 `Abp.StandardEmailTemplates.Layout` 并且这个字符中`Volo.Abp.Emailing.Templates.StandardEmailTemplates`静态类上定义为常量.

执行以下步骤将替换模板替换成你自定义的;

**\*\*1)\*\*** 在你的项目中相同的路径添加一个新文件 (`/Volo/Abp/Emailing/Templates/Layout.cshtml`):

![replace-email-layout-razor](images/replace-email-layout-razor.png)

**\*\*2)\*\*** 准备你的邮件布局模板:

````html

@inherits Volo.Abp.TextTemplating.Razor.RazorTemplatePageBase

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">

<head>

<meta charset="utf-8" />

</head>

<body>

<h1>This my header</h1>

@Body

<footer>

This is my footer...

</footer>

</body>

</html>

````

此示例只是向模板添加页眉和页脚并呈现它们之间的内容(请参阅上面的布局模板部分).

**\*\*3)\*\*** 在`.csproj`文件配置嵌入式资源e

\* 添加 [Microsoft.Extensions.FileProviders.Embedded](https://www.nuget.org/packages/Microsoft.Extensions.FileProviders.Embedded) NuGet 包到你的项目.

\* 在 `.csproj` 中添加 `<GenerateEmbeddedFilesManifest>true</GenerateEmbeddedFilesManifest>` 到 `<PropertyGroup>...</PropertyGroup>` 部分.

\* 添加以下代码到你的 `.csproj` 文件:

````xml

<ItemGroup>

<None Remove="Volo\Abp\Emailing\Templates\\*.cshtml" />

<EmbeddedResource Include="Volo\Abp\Emailing\Templates\\*.cshtml" />

</ItemGroup>

````

这将模板文件做为"嵌入式资源".

**\*\*4)\*\*** 配置虚拟文件系统

在[模块](Module-Development-Basics.md) `ConfigureServices` 方法配置 `AbpVirtualFileSystemOptions` 将嵌入的文件添加到虚拟文件系统中:

```csharp

Configure<AbpVirtualFileSystemOptions>(options =>

{

options.FileSets.AddEmbedded<BookStoreDomainModule>();

});

```

`BookStoreDomainModule` 应该是你的模块名称

> 确保你的模块(支持或间接)[依赖](Module-Development-Basics.md) `AbpEmailingModule`. 因为VFS可以基于依赖顺序覆盖文件.

现在渲染邮件布局模板时会使用你的模板.

### 选项-2: 使用模板定义提供者

你可以创建一个模板定义提供者类来获取邮件布局模板来更改它的虚拟文件路径.

**\*\*示例: 使用** `/MyTemplates/EmailLayout.cshtml` **文件而不是标准模板\*\***

```csharp

using Volo.Abp.DependencyInjection;

using Volo.Abp.Emailing.Templates;

using Volo.Abp.TextTemplating;

namespace MyProject

{

public class MyTemplateDefinitionProvider

: TemplateDefinitionProvider, ITransientDependency

{

public override void Define(ITemplateDefinitionContext context)

{

var emailLayoutTemplate = context.GetOrNull(StandardEmailTemplates.Layout);

emailLayoutTemplate

.WithVirtualFilePath(

"/MyTemplates/EmailLayout.cshtml",

isInlineLocalized: true

);

}

}

}

```

你依然应该添加 `/MyTemplates/EmailLayout.cshtml` 到虚拟文件系统. 这种方法允许你在任何文件夹中找到模板,而不是在依赖模块定义的文件夹中.

除了模板内容之外你还可以操作模板定义属性, 例如`DisplayName`, `Layout`或`LocalizationSource`.

## 高级功能

本节介绍文本模板系统的一些内部知识和高级用法.

### 模板内容Provider

`TemplateRenderer` 用于渲染模板,这是大多数情况下所需的模板. 但是你可以使用 `ITemplateContentProvider` 获取原始(未渲染的)模板内容.

> `ITemplateRenderer` 内部使用 `ITemplateContentProvider` 获取原始模板内容.

示例:

````csharp

public class TemplateContentDemo : ITransientDependency

{

private readonly ITemplateContentProvider \_templateContentProvider;

public TemplateContentDemo(ITemplateContentProvider templateContentProvider)

{

\_templateContentProvider = templateContentProvider;

}

public async Task RunAsync()

{

var result = await \_templateContentProvider

.GetContentOrNullAsync("Hello");

Console.WriteLine(result);

}

}

````

结果是原始模板内容:

````

@inherits Volo.Abp.TextTemplating.Razor.RazorTemplatePageBase<HelloModelNamespace.HelloModel>

Hello @Model.Name

````

\* `GetContentOrNullAsync` 如果没有为请求的模板定义任何内容,则返回 `null`.

\* 它可以获取 `cultureName` 参数,如果模板针对不同的文化具有不同的文件,则可以使用该参数(请参见上面的"多内容本地化"部分).

### 模板内容贡献者

`ITemplateContentProvider` 服务使用 `ITemplateContentContributor` 实现来查找模板内容. 有一个预实现的内容贡献者 `VirtualFileTemplateContentContributor`,它从上面描述的虚拟文件系统中获取模板内容.

你可以实现 `ITemplateContentContributor` 从另一个源读取原始模板内容.

示例:

````csharp

public class MyTemplateContentProvider

: ITemplateContentContributor, ITransientDependency

{

public async Task<string> GetOrNullAsync(TemplateContentContributorContext context)

{

var templateName = context.TemplateDefinition.Name;

//TODO: Try to find content from another source

return null;

}

}

````

如果源无法找到内容, 则返回 `null`, `ITemplateContentProvider` 将回退到下一个贡献者.

### Template Definition Manager

`ITemplateDefinitionManager` 服务可用于获取模板定义(由模板定义提供程序创建).

## 另请参阅

\* 本文开发和引用的[应用程序示例源码](https://github.com/abpframework/abp-samples/tree/master/TextTemplateDemo).

\* [本地化系统](Localization.md).

\* [虚拟文件系统](Virtual-File-System.md).

Scriban 集成

# Scriban 集成

## 安装

建议使用[ABP CLI](CLI.md)安装此包．

### 使用ABP CLI

在项目文件夹(.csproj 文件)中打开命令行窗口并输入以下命令:

````bash

abp add-package Volo.Abp.TextTemplating.Scriban

````

### 手动安装

如果你想要手动安装:

1. 添加 [Volo.Abp.TextTemplating.Scriban](https://www.nuget.org/packages/Volo.Abp.TextTemplating.Scriban) NuGet 包到你的项目:

````

Install-Package Volo.Abp.TextTemplating.Scriban

````

2.添加 `AbpTextTemplatingScribanModule` 到你的模块的依赖列表:

````csharp

[DependsOn(

//...other dependencies

typeof(AbpTextTemplatingScribanModule) //Add the new module dependency

)]

public class YourModule : AbpModule

{

}

````

## 定义模板

在渲染模板之前,需要定义它. 创建一个继承自 `TemplateDefinitionProvider` 的类:

````csharp

public class DemoTemplateDefinitionProvider : TemplateDefinitionProvider

{

public override void Define(ITemplateDefinitionContext context)

{

context.Add(

new TemplateDefinition("Hello") //template name: "Hello"

.WithVirtualFilePath(

"/Demos/Hello/Hello.tpl", //template content path

isInlineLocalized: true

)

.WithScribanEngine()

);

}

}

````

\* `context` 对象用于添加新模板或获取依赖模块定义的模板. 使用 `context.Add(...)` 定义新模板.

\* `TemplateDefinition` 是代表模板的类,每个模板必须有唯一的名称(在渲染模板时使用).

\* `/Demos/Hello/Hello.tpl` 是模板文件的路径.

\* `isInlineLocalized` 声明针对所有语言使用一个模板(`true` 还是针对每种语言使用不同的模板(`false`). 更多内容参阅下面的本地化部分.

\* `WithScribanEngine` 方法为模板设置渲染引擎.

### 模板内容

`WithVirtualFilePath` 表示我们使用[虚拟文件系统](Virtual-File-System.md)存储模板内容. 在项目内创建一个 `Hello.tpl` 文件,并在属性窗口中将其标记为"**\*\*嵌入式资源\*\***":

![hello-template](images/hello-template.png)

示例 `Hello.tpl` 内容如下所示:

````

Hello {%{{{model.name}}}%} :)

````

[虚拟文件系统](Virtual-File-System.md) 需要在[模块](Module-Development-Basics.md)类的 `ConfigureServices` 方法添加你的文件:

````csharp

Configure<AbpVirtualFileSystemOptions>(options =>

{

options.FileSets.AddEmbedded<TextTemplateDemoModule>("TextTemplateDemo");

});

````

\* `TextTemplateDemoModule`是模块类.

\* `TextTemplateDemo` 是你的项目的根命名空间.

## 渲染模板

`ITemplateRenderer` 服务用于渲染模板内容.

### 示例: 渲染一个简单的模板

````csharp

public class HelloDemo : ITransientDependency

{

private readonly ITemplateRenderer \_templateRenderer;

public HelloDemo(ITemplateRenderer templateRenderer)

{

\_templateRenderer = templateRenderer;

}

public async Task RunAsync()

{

var result = await \_templateRenderer.RenderAsync(

"Hello", //the template name

new HelloModel

{

Name = "John"

}

);

Console.WriteLine(result);

}

}

````

\* `HelloDemo` 是一个简单的类,在构造函数注入了 `ITemplateRenderer` 并在 `RunAsync` 方法中使用它.

\* `RenderAsync` 有两个基本参数:

\* `templateName`: 要渲染的模板名称 (本示例中是 `Hello`).

\* `model`: 在模板内部用做 `model` 的对象 (本示例中是 `HelloModel` 对象).

示例会返回以下结果:

````csharp

Hello John :)

````

### 匿名模型

虽然建议为模板创建模型类,但在简单情况下使用匿名对象也是可行的:

````csharp

var result = await \_templateRenderer.RenderAsync(

"Hello",

new

{

Name = "John"

}

);

````

示例中我们并没有创建模型类,但是创建了一个匿名对象模型.

### PascalCase 与 snake\_case

PascalCase 属性名(如 `UserName`) 在模板中使用蛇形命名(如 `user\_name`).

## 本地化

可以基于当前文化对模板内容进行本地化. 以下部分描述了两种类型的本地化选项.

### 内联本地化

内联本地化使用[本地化系统](Localization.md)本地化模板内的文本.

#### 示例: 重置密码链接

假设你需要向用户发送电子邮件重置密码. 模板内容:

````

<a title="{%{{{L "ResetMyPasswordTitle"}}}%}" href="{%{{{model.link}}}%}">{%{{{L "ResetMyPassword" model.name}}}%}</a>

````

`L` 函数用于根据当前用户的文化来定位给定的Key,你需要在本地化文件中定义 `ResetMyPassword` 键:

````json

"ResetMyPasswordTitle": "Reset my password",

"ResetMyPassword": "Hi {0}, Click here to reset your password"

````

你还需要在模板定义提供程序类中声明要与此模板一起使用的本地化资源:

````csharp

context.Add(

new TemplateDefinition(

"PasswordReset", //Template name

typeof(DemoResource) //LOCALIZATION RESOURCE

)

.WithScribanEngine()

.WithVirtualFilePath(

"/Demos/PasswordReset/PasswordReset.tpl", //template content path

isInlineLocalized: true

)

);

````

当你这样渲染模板时:

````csharp

var result = await \_templateRenderer.RenderAsync(

"PasswordReset", //the template name

new PasswordResetModel

{

Name = "john",

Link = "https://abp.io/example-link?userId=123&token=ABC"

}

);

````

你可以看到以下本地化结果:

````csharp

<a title="Reset my password" href="https://abp.io/example-link?userId=123&token=ABC">Hi john, Click here to reset your password</a>

````

> 如果你为应用程序定义了 [默认本地化资源](Localization.md), 则无需声明模板定义的资源类型.

### 多个内容本地化

你可能希望为每种语言创建不同的模板文件,而不是使用本地化系统本地化单个模板. 如果模板对于特定的文化(而不是简单的文本本地化)应该是完全不同的,则可能需要使用它.

#### 示例: 欢迎电子邮件模板

假设你要发送电子邮件欢迎用户,但要定义基于用户的文化完全不同的模板.

首先创建一个文件夹,将模板放在里面，像 `en.tpl`, `tr.tpl` 每一个你支持的文化:

![multiple-file-template](images/multiple-file-template.png)

然后在模板定义提供程序类中添加模板定义:

````csharp

context.Add(

new TemplateDefinition(

name: "WelcomeEmail",

defaultCultureName: "en"

)

.WithScribanEngine()

.WithVirtualFilePath(

"/Demos/WelcomeEmail/Templates", //template content folder

isInlineLocalized: false

)

);

````

\* 设置 **\*\*默认文化名称\*\***, 当没有所需的文化模板,回退到缺省文化.

\* 指定 **\*\*模板文件夹\*\*** 而不是单个模板文件.

\* 设置 `isInlineLocalized` 为 `false`.

就这些,你可以渲染当前文化的模板:

````csharp

var result = await \_templateRenderer.RenderAsync("WelcomeEmail");

````

> 为了简单我们跳过了模型,但是你可以使用前面所述的模型.

### 指定文化

`ITemplateRenderer` 服务如果没有指定则使用当前文化 (`CultureInfo.CurrentUICulture`). 如果你需要你可以使用 `cultureName` 参数指定文化.

````csharp

var result = await \_templateRenderer.RenderAsync(

"WelcomeEmail",

cultureName: "en"

);

````

## 布局模板

布局模板用于在其他模板之间创建共享布局. 它类似于ASP.NET Core MVC / Razor Pages中的布局系统.

### 示例: 邮件HTML布局模板

例如,你想为所有电子邮件模板创建一个布局.

首先像之前一样创建一个模板文件:

````xml

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">

<head>

<meta charset="utf-8" />

</head>

<body>

{%{{{content}}}%}

</body>

</html>

````

\* 布局模板必须具有 **\*\*{%{{{content}}}%}\*\*** 部分作为渲染的子内容的占位符.

在模板定义提供程序中注册模板:

````csharp

context.Add(

new TemplateDefinition(

"EmailLayout",

isLayout: true //SET isLayout!

)

.WithScribanEngine()

.WithVirtualFilePath(

"/Demos/EmailLayout/EmailLayout.tpl",

isInlineLocalized: true

)

);

````

现在你可以将此模板用作任何其他模板的布局:

````csharp

context.Add(

new TemplateDefinition(

name: "WelcomeEmail",

defaultCultureName: "en",

layout: "EmailLayout" //Set the LAYOUT

)

.WithScribanEngine()

.WithVirtualFilePath(

"/Demos/WelcomeEmail/Templates",

isInlineLocalized: false

)

);

````

## 全局上下文

ABP传递 `model`,可用于访问模板内的模型. 如果需要,可以传递更多的全局变量.

示例模板内容:

````

A global object value: {%{{{myGlobalObject}}}%}

````

模板假定它渲染上下文中的 `myGlobalObject` 对象. 你可以如下所示提供它:

````csharp

var result = await \_templateRenderer.RenderAsync(

"GlobalContextUsage",

globalContext: new Dictionary<string, object>

{

{"myGlobalObject", "TEST VALUE"}

}

);

````

渲染的结果将是:

````

A global object value: TEST VALUE

````

## 替换存在的模板

通过替换应用程序中使用的模块定义的模板. 这样你可以根据自己的需求自定义模板,而无需更改模块代码.

### 选项-1: 使用虚拟文件系统

[虚拟文件系统](Virtual-File-System.md)允许你通过将相同文件放入项目中的相同路径来覆盖任何文件.

#### 示例: 替换标准电子邮件布局模板

ABP框架提供了一个[邮件发送系统](Emailing.md), 它在内部使用文本模板来渲染邮件内容. 它在 `/Volo/Abp/Emailing/Templates/Layout.tp` 路径定义了一个标准邮件布局模板. 模板的唯一名称是 `Abp.StandardEmailTemplates.Layout` 并且这个字符中`Volo.Abp.Emailing.Templates.StandardEmailTemplates`静态类上定义为常量.

执行以下步骤将替换模板替换成你自定义的;

**\*\*1)\*\*** 在你的项目中相同的路径添加一个新文件 (`/Volo/Abp/Emailing/Templates/Layout.tpl`):

![replace-email-layout](images/replace-email-layout.png)

**\*\*2)\*\*** 准备你的邮件布局模板:

````html

<!DOCTYPE html>

<html lang="en" xmlns="http://www.w3.org/1999/xhtml">

<head>

<meta charset="utf-8" />

</head>

<body>

<h1>This my header</h1>

{%{{{content}}}%}

<footer>

This is my footer...

</footer>

</body>

</html>

````

此示例只是向模板添加页眉和页脚并呈现它们之间的内容(请参阅上面的布局模板部分).

**\*\*3)\*\*** 在`.csproj`文件配置嵌入式资源e

\* 添加 [Microsoft.Extensions.FileProviders.Embedded](https://www.nuget.org/packages/Microsoft.Extensions.FileProviders.Embedded) NuGet 包到你的项目.

\* 在 `.csproj` 中添加 `<GenerateEmbeddedFilesManifest>true</GenerateEmbeddedFilesManifest>` 到 `<PropertyGroup>...</PropertyGroup>` 部分.

\* 添加以下代码到你的 `.csproj` 文件:

````xml

<ItemGroup>

<None Remove="Volo\Abp\Emailing\Templates\\*.tpl" />

<EmbeddedResource Include="Volo\Abp\Emailing\Templates\\*.tpl" />

</ItemGroup>

````

这将模板文件做为"嵌入式资源".

**\*\*4)\*\*** 配置虚拟文件系统

在[模块](Module-Development-Basics.md) `ConfigureServices` 方法配置 `AbpVirtualFileSystemOptions` 将嵌入的文件添加到虚拟文件系统中:

```csharp

Configure<AbpVirtualFileSystemOptions>(options =>

{

options.FileSets.AddEmbedded<BookStoreDomainModule>();

});

```

`BookStoreDomainModule` 应该是你的模块名称

> 确保你的模块(支持或间接)[依赖](Module-Development-Basics.md) `AbpEmailingModule`. 因为VFS可以基于依赖顺序覆盖文件.

现在渲染邮件布局模板时会使用你的模板.

### 选项-2: 使用模板定义提供者

你可以创建一个模板定义提供者类来获取邮件布局模板来更改它的虚拟文件路径.

**\*\*示例: 使用** `/MyTemplates/EmailLayout.tpl` **文件而不是标准模板\*\***

```csharp

using Volo.Abp.DependencyInjection;

using Volo.Abp.Emailing.Templates;

using Volo.Abp.TextTemplating;

namespace MyProject

{

public class MyTemplateDefinitionProvider

: TemplateDefinitionProvider, ITransientDependency

{

public override void Define(ITemplateDefinitionContext context)

{

var emailLayoutTemplate = context.GetOrNull(StandardEmailTemplates.Layout);

emailLayoutTemplate

.WithVirtualFilePath(

"/MyTemplates/EmailLayout.tpl",

isInlineLocalized: true

);

}

}

}

```

你依然应该添加 `/MyTemplates/EmailLayout.tpl` 到虚拟文件系统. 这种方法允许你在任何文件夹中找到模板,而不是在依赖模块定义的文件夹中.

除了模板内容之外你还可以操作模板定义属性, 例如`DisplayName`, `Layout`或`LocalizationSource`.

## 高级功能

本节介绍文本模板系统的一些内部知识和高级用法.

### 模板内容Provider

`TemplateRenderer` 用于渲染模板,这是大多数情况下所需的模板. 但是你可以使用 `ITemplateContentProvider` 获取原始(未渲染的)模板内容.

> `ITemplateRenderer` 内部使用 `ITemplateContentProvider` 获取原始模板内容.

示例:

````csharp

public class TemplateContentDemo : ITransientDependency

{

private readonly ITemplateContentProvider \_templateContentProvider;

public TemplateContentDemo(ITemplateContentProvider templateContentProvider)

{

\_templateContentProvider = templateContentProvider;

}

public async Task RunAsync()

{

var result = await \_templateContentProvider

.GetContentOrNullAsync("Hello");

Console.WriteLine(result);

}

}

````

结果是原始模板内容:

````

Hello {%{{{model.name}}}%} :)

````

\* `GetContentOrNullAsync` 如果没有为请求的模板定义任何内容,则返回 `null`.

\* 它可以获取 `cultureName` 参数,如果模板针对不同的文化具有不同的文件,则可以使用该参数(请参见上面的"多内容本地化"部分).

### 模板内容贡献者

`ITemplateContentProvider` 服务使用 `ITemplateContentContributor` 实现来查找模板内容. 有一个预实现的内容贡献者 `VirtualFileTemplateContentContributor`,它从上面描述的虚拟文件系统中获取模板内容.

你可以实现 `ITemplateContentContributor` 从另一个源读取原始模板内容.

示例:

````csharp

public class MyTemplateContentProvider

: ITemplateContentContributor, ITransientDependency

{

public async Task<string> GetOrNullAsync(TemplateContentContributorContext context)

{

var templateName = context.TemplateDefinition.Name;

//TODO: Try to find content from another source

return null;

}

}

````

如果源无法找到内容, 则返回 `null`, `ITemplateContentProvider` 将回退到下一个贡献者.

### Template Definition Manager

`ITemplateDefinitionManager` 服务可用于获取模板定义(由模板定义提供程序创建).

## 另请参阅

\* 本文开发和引用的[应用程序示例源码](https://github.com/abpframework/abp-samples/tree/master/TextTemplateDemo).

\* [本地化系统](Localization.md).

\* [虚拟文件系统](Virtual-File-System.md).

## GUID 生成

# GUID 生成

GUID是数据库管理系统中使用的常见**\*\*主键类型\*\***, ABP框架更喜欢GUID作为预构建[应用模块](Modules/Index.md)的主要对象. `ICurrentUser.Id` 属性([参见文档](CurrentUser.md))是GUID类型,这意味着ABP框架假定用户ID始终是GUID,

## 为什么偏爱GUID?

GUID有优缺点. 你可以在网上找到许多与此主题相关的文章,因此我们不再赘述,而是列出了最基本的优点:

\* 它可在所有数据库提供程序中**\*\*使用\*\***.

\* 它允许在客户端**\*\*确定主键\*\***,而不需要通过**\*\*数据库往返\*\***来生成Id值. 在向数据库插入新记录时,这可以提高性能并允许我们在与数据库交互之前知道PK.

\* GUID是**\*\*自然唯一的\*\***在以下情况下有一些优势;

\* 你需要与**\*\*外部\*\***系统集成,

\* 你需要**\*\*拆分或合并\*\***不同的表.

\* 你正在创建**\*\*分布式系统\*\***

\* GUID是无法猜测的,因此在某些情况下与自动递增的Id值相比,GUID**\*\*更安全\*\***.

尽管存在一些缺点(只需在Web上搜索),但在设计ABP框架时我们发现这些优点更为重要.

## IGuidGenerator

GUID的最重要问题是**\*\*默认情况下它不是连续的\*\***. 当你将GUID用作主键并将其设置为表的**\*\*聚集索引\*\***(默认设置)时,这会在**\*\*插入时带来严重的性能问题\*\***(因为插入新记录可能需要对现有记录进行重新排序).

所以,**\*\*永远不要为你的实体使用** `Guid.NewGuid()` **创建ID\*\***!.

这个问题的一个好的解决方案是生成**\*\*连续的GUID\*\***,由ABP框架提供的开箱即用的. `IGuidGenerator` 服务创建顺序的GUID(默认由 `SequentialGuidGenerator` 实现). 当需要手动设置[实体](Entities.md)的Id时,请使用 `IGuidGenerator.Create()`.

**\*\*示例: 具有GUID主键的实体并创建该实体\*\***

假设你有一个具有 `Guid` 主键的 `Product` [实体](Entities.md):

````csharp

using System;

using Volo.Abp.Domain.Entities;

namespace AbpDemo

{

public class Product : AggregateRoot<Guid>

{

public string Name { get; set; }

private Product() { /\* This constructor is used by the ORM/database provider \*/ }

public Product(Guid id, string name)

: base(id)

{

Name = name;

}

}

}

````

然后你想要创建一个产品:

````csharp

using System;

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Domain.Repositories;

using Volo.Abp.Guids;

namespace AbpDemo

{

public class MyProductService : ITransientDependency

{

private readonly IRepository<Product, Guid> \_productRepository;

private readonly IGuidGenerator \_guidGenerator;

public MyProductService(

IRepository<Product, Guid> productRepository,

IGuidGenerator guidGenerator)

{

\_productRepository = productRepository;

\_guidGenerator = guidGenerator;

}

public async Task CreateAsync(string productName)

{

var product = new Product(\_guidGenerator.Create(), productName);

await \_productRepository.InsertAsync(product);

}

}

}

````

该服务将 `IGuidGenerator` 注入构造函数中. 如果你的类是[应用服务](Application-Services.md)或派生自其他基类之一,可以直接使用 `GuidGenerator` 基类属性,该属性是预先注入的 `IGuidGenerator` 实例.

## Options

### AbpSequentialGuidGeneratorOptions

`AbpSequentialGuidGeneratorOptions` 是用于配置顺序生成GUID的[选项类](Options.md). 它只有一个属性:

\* `DefaultSequentialGuidType` (`SequentialGuidType` 类型的枚举): 生成GUID值时使用的策略.

数据库提供程序在处理GUID时的行为有所不同,你应根据数据库提供程序进行设置. `SequentialGuidType` 有以下枚举成员:

\* `SequentialAtEnd` (**\*\*default\*\***) 用于[SQL Server](Entity-Framework-Core.md).

\* `SequentialAsString` 用于[MySQL](Entity-Framework-Core-MySQL.md)和[PostgreSQL](Entity-Framework-Core-PostgreSQL.md).

\* `SequentialAsBinary` 用于[Oracle](Entity-Framework-Core-Oracle.md).

在你的[模块](Module-Development-Basics.md)的 `ConfigureServices` 方法配置选项,如下:

````csharp

Configure<AbpSequentialGuidGeneratorOptions>(options =>

{

options.DefaultSequentialGuidType = SequentialGuidType.SequentialAsBinary;

});

````

> EF Core[集成包](https://docs.abp.io/en/abp/latest/Entity-Framework-Core-Other-DBMS)已为相关的DBMS设置相应的值. 如果你正在使用这些集成包,在大多数情况下则无需设置此选项.

## 时钟

# 时钟

使用时间和[时区](https://en.wikipedia.org/wiki/Time\_zone)总是很棘手,尤其是当你需要构建供**\*\*不同时区\*\***的用户使用的**\*\*全局系统\*\***时.

ABP提供了一个基本的基础结构,使其变得容易并在可能的情况下自动进行处理. 本文档涵盖了与时间和时区相关的ABP框架服务和系统.

> 如果你正在创建在单个时区区域运行的本地应用程序,则可能不需要这些系统. 但也建议使用本文中介绍的 `IClock` 服务.

## IClock

`DateTime.Now` 返回带有**\*\*服务器本地日期和时间\*\***的 `DateTime` 对象. `DateTime` 对象**\*\*不存储时区信息\*\***. 因此你无法知道此对象中存储的**\*\*绝对日期和时间\*\***. 你只能做一些**\*\*假设\*\***,例如假设它是在UTC+05时区创建的. 当你此值保存到数据库中并稍后读取,或发送到**\*\*不同时区\*\***的客户端时,事情就变得特别复杂.

解决此问题的一种方法是始终使用 `DateTime.UtcNow` 并将所有 `DateTime` 对象假定为UTC时间. 在这种情况下你可以在需要时将其转换为目标客户端的时区.

`IClock` 在获取当前时间的同时提供了一种抽象,你可以在应用程序中的单个点上控制日期时间的类型(UTC或本地时间).

**\*\*示例: 获取当前时间\*\***

````csharp

using Volo.Abp.DependencyInjection;

using Volo.Abp.Timing;

namespace AbpDemo

{

public class MyService : ITransientDependency

{

private readonly IClock \_clock;

public MyService(IClock clock)

{

\_clock = clock;

}

public void Foo()

{

//Get the current time!

var now = \_clock.Now;

}

}

}

````

\* 当你需要获取当前时间时注入 `IClock` 服务. 常用的服务基类(如ApplicationService)已经注入并且做为基类属性提供,所以你可以直接使用 `Clock`.

\* 使用 `Now` 属性获取当前时间.

> 在大多数情况下 `IClock` 是你需要在应用程序中了解和使用的唯一服务.

### Clock 选项

`AbpClockOptions` 是用于设置时钟种类的[选项](Options.md)类.

**\*\*示例: 使用 UTC Clock\*\***

````csharp

Configure<AbpClockOptions>(options =>

{

options.Kind = DateTimeKind.Utc;

});

````

在你的[模块](Module-Development-Basics.md)的 `ConfigureServices` 方法添加以上内容.

> 默认 `Kind` 是 `Unspecified`,实际上使时钟不存在. 如果要利用Clock系统，要么使用 `Utc` 或 `Local`.

### DateTime 标准化

`IClock` 的其他重要功能是规范化 `DateTime` 对象.

\*\*示例用法 :\*\*

````csharp

DateTime dateTime = ...; //Get from somewhere

var normalizedDateTime = Clock.Normalize(dateTime)

````

`Normalize` 方法的工作原理如下:

\* 如果当前时钟为UTC,并且给定的 `DateTime` 为本地时间,将给定的 `DateTime` 转换为UTC(通过使用 `DateTime.ToUniversalTime()` 方法).

\* 如果当前时钟是本地的,并且给定的 `DateTime` 是UTC,将给定的 `DateTime` 转换为本地时间(通过使用 `DateTime.ToUniversalTime()` 方法).

\* 如果未指定给定的 `DateTime` 的 `Kind`,将给定的 `DateTime` 的 `Kind`(使用 `DateTime.SpecifyKind(...)` 方法)设置为当前时钟的 `Kind`.

当获取的 `DateTime` 不是由 `IClock` 创建且可能与当前Clock类型不兼容的时候,ABP框架会使用 `Normalize` 方法. 例如;

\* ASP.NET Core MVC模型绑定中的 `DateTime` 类型绑定.

\* 通过[Entity Framework Core](Entity-Framework-Core.md)将数据保存到数据库或从数据库读取数据.

\* 在[JSON反序列化](Json.md)上使用 `DateTime` 对象.

#### DisableDateTimeNormalization Attribute

`DisableDateTimeNormalization` attribute可用于禁用所需类或属性的规范化操作.

### 其他 IClock 属性

除了 `Now`, `IClock` 服务还具有以下属性:

\* `Kind`: 返回当前使用的时钟类型(`DateTimeKind.Utc`, `DateTimeKind.Local` 或 `DateTimeKind.Unspecified`)的 `DateTimeKind`.

\* `SupportsMultipleTimezone`: 如果当前时间是UTC,则返回 `true`.

## 时区

本节介绍与管理时区有关的ABP框架基础结构

### 时区设置

ABP框架定义了一个名为 `Abp.Timing.Timezone` 的**\*\*设置\*\***,可用于为应用程序的用户,[租户](Multi-Tenancy.md)或全局设置和获取时区. 默认值为 `UTC`.

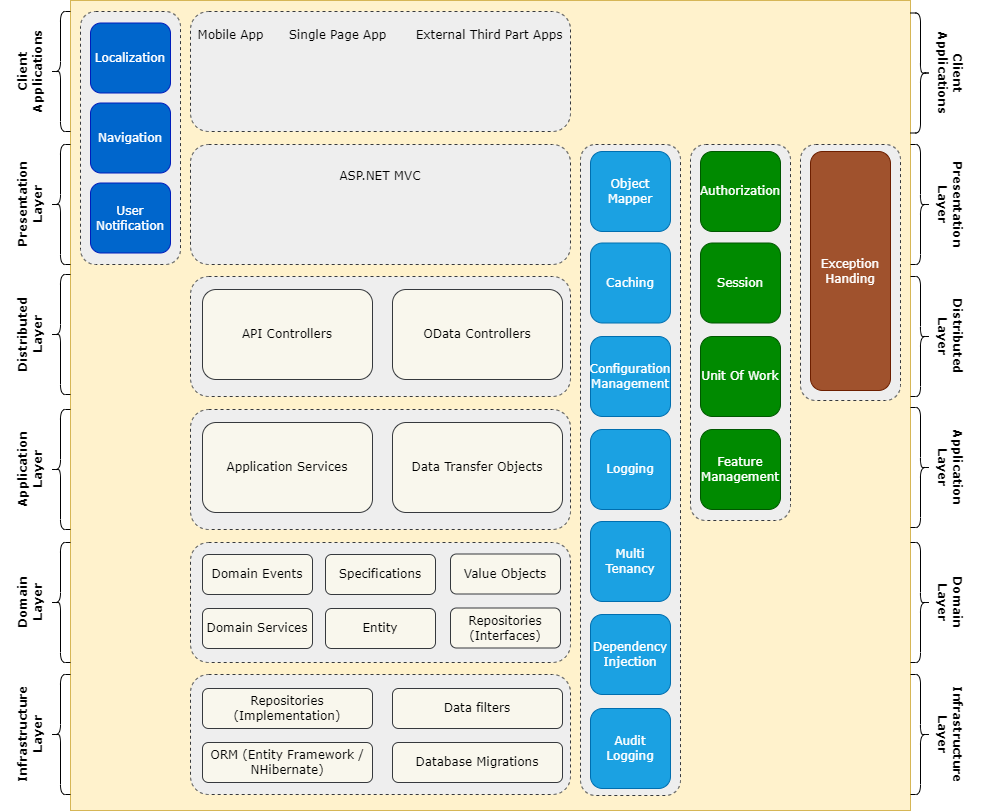
参阅[设置系统]了解更多关于设置系统.

### ITimezoneProvider

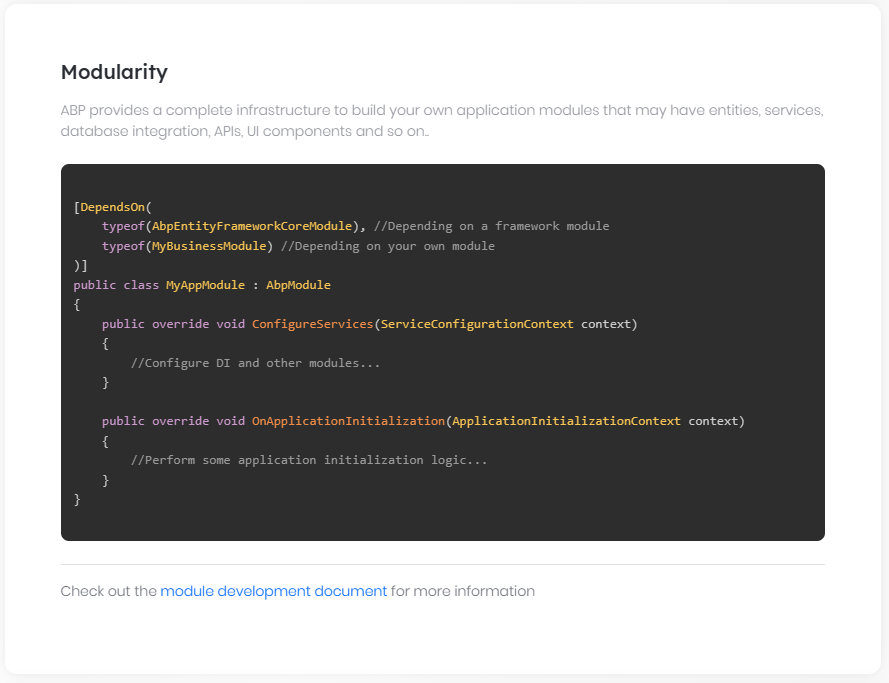
`ITimezoneProvider` 是一个服务,可将[Windows时区ID](https://support.microsoft.com/en-us/help/973627/microsoft-time-zone-index-values)值简单转换为[Iana时区名称](https://www.iana.org/time-zones)值,反之亦然. 它还提供了获取这些时区列表与获取具有给定名称的 `TimeZoneInfo` 的方法.

它已使用[TimeZoneConverter](https://github.com/mj1856/TimeZoneConverter)库实现.

# 架构



## 模块化



基础

## 模块化

### 介绍

ABP本身是一个包含许多nuget包的模块化框架.它还提供了一个完整的基础架构来开发你自己的具有实体, 服务, 数据库集成, API, UI组件等等功能的应用程序模块.

### 模块类

每个模块都应该定义一个模块类.定义模块类的最简单方法是创建一个派生自``AbpModule``的类,如下所示:

````C#

public class BlogModule : AbpModule

{

}

````

#### 配置依赖注入和其他模块

##### ConfigureServices方法

``ConfigureServices``是将你的服务添加到依赖注入系统并配置其他模块的主要方法.例:

````C#

public class BlogModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

//...

}

}

````

你可以按照Microsoft的[文档](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection)中的说明逐个注册依赖项.但ABP有一个**\*\*依照约定的依赖注册系统\*\***,可以自动注册程序集中的所有服务.有关依赖项注入系统的更多信息,请参阅[依赖项注入](Dependency-Injection.md)文档.

你也可以通过这种方式配置其他服务和模块.例:

````C#

public class BlogModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

//为应用程序配置默认的连接字符串

Configure<AbpDbConnectionOptions>(options =>

{

options.ConnectionStrings.Default = "......";

});

}

}

````

有关配置系统的更多信息,请参阅[配置](Configuration.md)文档.

##### 配置服务前和后

``AbpModule``类还定义了``PreConfigureServices``和``PostConfigureServices``方法用来在``ConfigureServices``之前或之后覆盖和编写你的代码.请注意,在这些方法中编写的代码将在所有其他模块的``ConfigureServices``方法之前/之后执行.

#### 应用程序初始化

一旦配置了所有模块的所有服务,应用程序就会通过初始化所有模块来启动.在此阶段,你可以从``IServiceProvider``中获取服务,因为这时它已准备就绪且可用.

##### OnApplicationInitialization方法

你可以在启动应用程序时覆盖``OnApplicationInitialization``方法来执行代码.例:

````C#

public class BlogModule : AbpModule

{

//...

public override void OnApplicationInitialization(ApplicationInitializationContext context)

{

var myService = context.ServiceProvider.GetService<MyService>();

myService.DoSomething();

}

}

````

``OnApplicationInitialization``通常由启动模块用于构建ASP.NET Core应用程序的中间件管道.例:

````C#

[DependsOn(typeof(AbpAspNetCoreMvcModule))]

public class AppModule : AbpModule

{

//...

public override void OnApplicationInitialization(ApplicationInitializationContext context)

{

var app = context.GetApplicationBuilder();

var env = context.GetEnvironment();

if (env.IsDevelopment())

{

app.UseDeveloperExceptionPage();

}

app.UseMvcWithDefaultRoute();

}

}

````

如果模块需要,你还可以执行启动逻辑

##### 应用程序初始化前和后

``AbpModule``类还定义了``OnPreApplicationInitialization``和``OnPostApplicationInitialization``方法用来在``OnApplicationInitialization``之前或之后覆盖和编写你的代码.请注意,在这些方法中编写的代码将在所有其他模块的``OnApplicationInitialization``方法之前/之后执行.

#### 应用程序关闭

最后,如果要在应用程序关闭时执行某些代码,你可以覆盖``OnApplicationShutdown``方法.

### 模块依赖

在模块化应用程序中,一个模块依赖于另一个或几个模块并不罕见.如果一个Abp模块依赖于另一个模块,它必须声明``[DependsOn]``特性,如下所示:

````C#

[DependsOn(typeof(AbpAspNetCoreMvcModule))]

[DependsOn(typeof(AbpAutofacModule))]

public class BlogModule

{

//...

}

````

你可以根据需要使用多个``DependsOn``特性或将多个模块类型传递给单个``DependsOn``特性.

依赖模块可能依赖于另一个模块,但你只需要定义直接依赖项.ABP在启动时会调查应用程序的依赖关系,并以正确的顺序初始化/关闭模块.

## 框架模块 vs 应用程序模块

**\*\*模块分为两种类型.\*\*** 这两种类型并没有任何结构上的区别,只是按功能和用途分类:

- **\*\*框架模块\*\***: 这些是**\*\*框架的核心模块\*\*** 如缓存, 邮件, 主题, 安全, 序列化, 验证, EF Core集成, MongoDB集成... 等. 它们没有应用/业务功能,它们提供了日常开发经常用到的通用基础设施,集成和抽象.

- **\*\*应用程序模块\*\***: 这些模块实现了 **\*\*特定的应用/业务功能\*\*** 像博客, 文档管理, 身份管理, 租户管理... 等等. 它们通常有自己的实体,服务,API和UI组件. 请参阅 [预构建的应用程序模块](Modules/Index.md).

模块插件

# 模块化插件

可以将[模块](Module-Development-Basics.md)加载为插件.这意味着你可能不需要在解决方案中引用模块的程序集,就可以像其它模块一样在启动应用时加载该模块.

## 基本用法

`IServiceCollection.AddApplication<T>()` 扩展方法可以获取配置插件源的选项.

**\*\*示例: 从文件夹加载插件\*\***

````csharp

using Microsoft.AspNetCore.Builder;

using Microsoft.Extensions.DependencyInjection;

using Volo.Abp.Modularity.PlugIns;

namespace MyPlugInDemo.Web

{

public class Startup

{

public void ConfigureServices(IServiceCollection services)

{

services.AddApplication<MyPlugInDemoWebModule>(options =>

{

options.PlugInSources.AddFolder(@"D:\Temp\MyPlugIns");

});

}

public void Configure(IApplicationBuilder app)

{

app.InitializeApplication();

}

}

}

````

\* 这是典型的ASP.NET Core应用程序的`Startup`类.

\* `PlugInSources.AddFolder`从指定的目录中加载程序集(通常为dll).

就这样.ABP将在这个目录中发现这些模块,像其它常规一样配置和初始化它们.

### 插件源

`options.PlugInSources`类实际上是`IPlugInSource`接口的一系列实现并且 `AddFolder`方法仅仅是以下表达式的便捷方法：

````csharp

options.PlugInSources.Add(new FolderPlugInSource(@"D:\Temp\MyPlugIns"));

````

> `AddFolder()`方法仅在给定目录下查找程序集文件,而不在子目录中查找.你可以传递一个`SearchOption.AllDirectories`参数作为第二个参数,来递归地查找它的子目录.

这里有两个内置插件源的示例：

\* `PlugInSources.AddFiles()`方法获取程序集(通常是dll)文件列表.这是使用`FilePlugInSource`类的快捷方式.

\* `PlugInSources.AddTypes()`方法获取模块类类型的列表.如果实用化此方法,则需要自己加载模块的程序集,但是在需要时它提供了灵活性.这是使用`TypePlugInSource`类的快捷方式.

如果需要,你可以创建自己的`IPlugInSource`的接口实现,并像其它方法一样添加到`options.PlugInSources`中.

## 示例：创建一个简单的插件

在一个解决方案中创建一个简单的**\*\*类库项目\*\***

![简单插件库](images/simple-plugin-library.png)

你可以在模块中添加需要使用的ABP框架包.至少,你应该为这个项目添加包`Volo.Abp.Core`：

````

Install-Package Volo.Abp.Core

````

每个[模块](Module-Development-Basics.md)必须声明为一个继承自`AbpModule`的类.这里是一个简单的模块类,用于解析一个服务并在应用启动时对其初始化：

````csharp

using Microsoft.Extensions.DependencyInjection;

using Volo.Abp;

using Volo.Abp.Modularity;

namespace MyPlugIn

{

public class MyPlungInModule : AbpModule

{

public override void OnApplicationInitialization(ApplicationInitializationContext context)

{

var myService = context.ServiceProvider

.GetRequiredService<MyService>();

myService.Initialize();

}

}

}

````

`MyService`可以是注册在[依赖注入](Dependency-Injection.md)系统中的任意类,如下所示:

````csharp

using Microsoft.Extensions.Logging;

using Volo.Abp.DependencyInjection;

namespace MyPlugIn

{

public class MyService : ITransientDependency

{

private readonly ILogger<MyService> \_logger;

public MyService(ILogger<MyService> logger)

{

\_logger = logger;

}

public void Initialize()

{

\_logger.LogInformation("MyService has been initialized");

}

}

}

````

编译这个项目,打开build目录,找到`MyPlugIn.dll`：

![简单dll插件](images/simple-plug-in-dll-file.png)

将`MyPlugIn.dll`复制到到插件目录中(此实例为`D:\Temp\MyPlugIns`).

如果你已经按照上述方式配置了主应用程序(参见“基础用法”部分),那么在应用程序启动时,你可以看到“MyService has been initialized(MyService已经初始化)的日志.

## 示例：创建一个Razor Pages插件

创建内部带视图的插件需要更多的注意.

> 这个示例假设你已经使用应用程序启动模板和MVC / Razor Pages UI[创建了一个新的Web应用程序](https://abp.io/get-started).

在解决方案中创建一个新的**\*\*类库\*\***项目：

![简单razor插件](images/simple-razor-plugin.png)

编辑这个`.csproj`文件内容：

````xml

<Project Sdk="Microsoft.NET.Sdk.Web">

<PropertyGroup>

<TargetFramework>net5.0</TargetFramework>

<OutputType>Library</OutputType>

<IsPackable>true</IsPackable>

</PropertyGroup>

<ItemGroup>

<PackageReference Include="Volo.Abp.AspNetCore.Mvc.UI.Theme.Shared" Version="4.0.1" />

</ItemGroup>

</Project>

````

\* 将`Sdk`修改为`Microsoft.NET.Sdk.Web`.

\* 添加了`OutputType`和`IsPackable`属性.

\* 添加了`Volo.Abp.AspNetCore.Mvc.UI.Theme.Shared`NuGet包.

> 不需要[Volo.Abp.AspNetCore.Mvc.UI.Theme.Shared](https://www.nuget.org/packages/Volo.Abp.AspNetCore.Mvc.UI.Theme.Shared) 包.你可以引用更基础的程序包,例如[Volo.Abp.AspNetCore.Mvc](https://www.nuget.org/packages/Volo.Abp.AspNetCore.Mvc/). 但是,如果需要构建一个UI视图/组件,建议参考[Volo.Abp.AspNetCore.Mvc.UI.Theme.Shared](https://www.nuget.org/packages/Volo.Abp.AspNetCore.Mvc.UI.Theme.Shared)程序包,因为它是最高级的程序包,不依赖于特定[theme](UI/AspNetCore/Theming.md).如果依赖特定主题没有问题,则可以直接引用该主题的程序包,以便能够使用插件中特定于主题的功能.

接下来在插件中创建模块类：

````csharp

using System.IO;

using System.Reflection;

using Microsoft.AspNetCore.Mvc.ApplicationParts;

using Microsoft.Extensions.DependencyInjection;

using Volo.Abp.AspNetCore.Mvc.UI.Theme.Shared;

using Volo.Abp.Modularity;

namespace MyMvcUIPlugIn

{

[DependsOn(typeof(AbpAspNetCoreMvcUiThemeSharedModule))]

public class MyMvcUIPlugInModule : AbpModule

{

public override void PreConfigureServices(ServiceConfigurationContext context)

{

PreConfigure<IMvcBuilder>(mvcBuilder =>

{

// 添加插件程序集

mvcBuilder.PartManager.ApplicationParts.Add(new AssemblyPart(typeof(MyMvcUIPlugInModule).Assembly));

// 添加视图程序集

var viewDllPath = Path.Combine(Path.GetDirectoryName(typeof(MyMvcUIPlugInModule).Assembly.Location), "MyMvcUIPlugIn.Views.dll");

var viewAssembly = new CompiledRazorAssemblyPart(Assembly.LoadFrom(viewDllPath));

mvcBuilder.PartManager.ApplicationParts.Add(viewAssembly);

});

}

}

}

````

\* 由于我们添加了相关的NuGet包,因此取决于`AbpAspNetCoreMvcUiThemeSharedModule`.

\* 添加插件程序集到ASP.NET Core MVC的`PartManager`中.这是ASP.NET Core所必需的.否则,你插件中的控制器将无法正常工作.

\* 添加插件的视图程序集到ASP.NET Core MVC的`PartManager`中.这是ASP.NET Core所必需的.否则,你在插件中的视图将不起作用.

现在,你可以在`Pages`目录下添加一个razor页面,例如`MyPlugInPage.cshtml`：

````html

@page

@model MyMvcUIPlugIn.Pages.MyPlugInPage

<h1>Welcome to my plug-in page</h1>

<p>This page is located inside a plug-in module! :)</p>

````

现在,你可以构建插件项目.它将产生以下输出：

![simple-razor-plug-in-dll-file](images/simple-razor-plug-in-dll-file.png)

将`MyMvcUIPlugIn.dll`和`MyMvcUIPlugIn.Views.dll`复制到到插件目录下(此示例中为`D:\Temp\MyPlugIns`).

如果你已经按照上述方式配置了主应用程序(参见“基础用法”部分),那么在应用程序启动的时候,你应该能够访问`/MyPlugInPage`URL：

![simple-plugin-output](images/simple-plugin-output.png)

## 讨论

在现实世界中,你的插件可能具有一些外部依赖性.另外,你的应用程序可能被设计为支持插件.所有这些都是你自己的系统要求.ABP做的仅仅是在应用程序启动时加载模块.你在这些模块中执行什么操作由你决定.

但是,我们可以为一些常见情况提供一些建议.

### 库依赖

对于包/dll依赖,你可以将相关的dll复制到插件目录下.ABP会自动将所有程序集加载到该目录下,并且你的插件将按预期工作.

> 请参见[Microsoft文档](https://docs.microsoft.com/zh-cn/dotnet/core/tutorials/creating-app-with-plugin-support#plugin-with-library-dependencies).

### 数据库模式

如果你的模块使用关系型数据库和[Entity Framework Core](Entity-Framework-Core.md), 那么它需要在数据库中提供表.有多种不同的方法可确保在应用程序使用插件时创建表.一些例子；

1. 插件可以检查数据库表是否存在,并在应用程序启动时创建表,或者如果插件已更新且需要进行某些架构更改时,则会迁移它们.你可以使用EF Core的迁移API来做到这一点.

2. 你可以改进`DbMigrator`应用程序,用于查找插件的迁移并执行它们.

可能还有其它解决方案.例如,如果你的数据库管理员不允许你在应用程序代码中更改数据库模式,则可能需要手动将SQL文件发送给数据库管理员,以将其应用于数据库.

自定义应用模块

# 自定义现有模块

ABP框架提供的设计旨在支持构建完全[模块化的应用程序](Module-Development-Basics.md)和系统. 它还提供了一些可以在任何类型的应用程序中**\*\*使用\*\***的[预构建应用模块](Modules/Index.md)

例如,你可以在你的应用程序中**\*\*重用\*\***[身份管理模块](Modules/Identity.md)去添加用户,角色和权限管理. [应用程序启动模板](Startup-Templates/Application.md)已经**\*\*预装\*\***了Identity和其他模块.

## 复用应用模块

你有两个选项去复用应用模块:

### 添加包引用

你可以添加相关模块的 **\*\*NuGet\*\*** 和 **\*\*NPM\*\*** 包引用到你的应用程序,并配置模块(根据它的文档)集成到你的应用程序中.

正如前面提到,[应用程序启动模板](Startup-Templates/Application.md)已经**\*\*预装了一些基本模块\*\***,它引用模块的NuGet和NPM包.

这种方法具有以下优点:

\* 你的解决方案会非常**\*\*干净\*\***,只包含你**\*\*自己的应用程序代码\*\***.

\* 你可以**\*\*很简单的\*\***升级模块到最新的可用模板. `abp update` [CLI](CLI.md) 命令会使更新变的更加简单. 通过这种方式, 你可以获得**\*\*最新功能和Bug修复\*\***.

然而有一个缺点:

\* 你可能无法**\*\*自定义\*\***模块,因为模块源码没有在你的解决方案中.

本文档介绍了 **\*\*或者自定义或扩展\*\*** 依赖模块并且无需更改其源码,尽快与更改完整的源码比起是有限的,但仍有一些好的方法可以自定义.

如果你不认为自己会对预构建的模块进行重大更改,那么使用包引用的方法复用模块是推荐的方法.

### 包含源码

如果你想要在预构建的模块上进行**\*\*重大\*\***更改或添加**\*\*主要功能\*\***,但是可用的扩展点不够使用,那么可以考虑直接使用依赖模块的源码.

这种情况下,你通常**\*\*添加模块源码\*\***到你的解决方案中,并将**\*\*包引用替换\*\***为本地项目引用. **\*\*[**ABP CLI**](CLI.md)\*\*** 可以为你自动化这一过程.

#### 分离模块解决方案

你可能不希望将模块源代码**\*\*直接包含在解决方案\*\***中. 每个模块都包含十多个项目文件,添加**\*\*多个模块\*\***会使解决方案变的臃肿可能还会影响**\*\*开发时的加载速度\*\***,另外你可能有不同的开发团队维护不同模块.

无论如何,你都可以为需要的模块创建**\*\*单独的解决方案\*\***,将依赖模块做为解决方案中的项目引用. 比如在[abp仓库](https://github.com/abpframework/abp/),我们就是这样做的.

> 我们看到的一个问题是Visual Studio在这种方式下不能很好的工作(解决方案目录之外对本地项目的引用不能很好地支持). 如果在开发过程中出错(对于外部模块),请在Visual Studio打开应用程序的解决方案后,在命令行运行 `dotnet restore`命令.

#### 发布的自定义模块的包

一个备选方案是将重新打包模块的源代码(NuGet/NPM包),使用包引用. 你可以为公司使用本地私人的Nuget/NPM服务器.

## 模块自定义/扩展途径

如果你决定使用预构建模块的NuGet/NPM包引用方式. 下面的文档详细解释了如何自定义/扩展现有模块的方法:

\* [扩展实体](Customizing-Application-Modules-Extending-Entities.md)

\* [重写服务](Customizing-Application-Modules-Overriding-Services.md)

\* [重写界面](Customizing-Application-Modules-Overriding-User-Interface.md)

### 另请参阅

另外,请参阅以下文档:

\* 参阅 [本地化文档](Localization.md) 学习如何扩展已存在的本地化资源.

\* 参阅 [设置文档](Settings.md) 学习如何更改依赖模块的设置定义.

\* 参阅 [授权文档](Authorization.md) 学习如何更改依赖模块的权限定义.

#### 扩展实体

# 自定义应用模块: 扩展实体

在某些情况下你可能希望为依赖模块中定义的实体添加一些额外的属性(和数据库字段). 本节将介绍一些实现这一目标的不同方法.

## Extra Properties

[Extra properties](Entities.md)是一种存储实体的一些额外数据但不用更改实体的方式. 实体应该实现 `IHasExtraProperties` 接口. 所有预构建模块定义的聚合根实体都实现了 `IHasExtraProperties` 接口,所以你可以在这些实体中存储额外的属性.

示例:

````csharp

//SET AN EXTRA PROPERTY

var user = await \_identityUserRepository.GetAsync(userId);

user.SetProperty("Title", "My custom title value!");

await \_identityUserRepository.UpdateAsync(user);

//GET AN EXTRA PROPERTY

var user = await \_identityUserRepository.GetAsync(userId);

return user.GetProperty<string>("Title");

````

这种方法开箱即用并且非常简单,你可以使用不同的属性名称(如这里的`Title`)在同一时间存储多个属性.

对于EF Core额外的属性被格式化成单个 `JSON` 字符值串存储在数据库中. 对于MongoDB它们做为单独的字段存储.

参阅[实体文档](Entities.md)了解更多关于额外系统.

> 可以基于额外的属性执行**\*\*业务逻辑\*\***. 你可以[重写服务方法](Customizing-Application-Modules-Overriding-Services.md). 然后获取或设置如上所示的值.

## 实体扩展 (EF Core)

如上所述,实体所有的额外属性都作为单个JSON对象存储在数据库表中. 它不适用复杂的场景,特别是在你需要的时候.

\* 使用额外属性创建**\*\*索引\*\***和**\*\*外键\*\***.

\* 使用额外属性编写**\*\*SQL\*\***或**\*\*LINQ\*\***(例如根据属性值搜索).

\* 创建你**\*\*自己的实体\*\***映射到相同的表,但在实体中定义一个额外属性做为 **\*\*常规属性\*\***(参阅 [EF Core迁移文档](Entity-Framework-Core-Migrations.md)了解更多).

为了解决上面的问题,用于EF Core的ABP框架实体扩展系统允许你使用上面定义相同的额外属性API,但将所需的属性存储在单独的数据库表字段中.

假设你想要添加 `SocialSecurityNumber` 到[身份模块](Modules/Identity.md)的 `IdentityUser` 实体. 你可以使用 `ObjectExtensionManager` 类:

````csharp

ObjectExtensionManager.Instance

.MapEfCoreProperty<IdentityUser, string>(

"SocialSecurityNumber",

(entityBuilder, propertyBuilder) =>

{

propertyBuilder.HasMaxLength(32);

}

);

````

\* 你提供了 `IdentityUser` 作为实体名(泛型参数), `string` 做为新属性的类型, `SocialSecurityNumber` 做为属性名(也是数据库表的字段名).

\* 你还需要提供一个使用[EF Core Fluent API](https://docs.microsoft.com/en-us/ef/core/modeling/entity-properties)定义数据库映射属性的操作.

> 必须在使用相关的 `DbContext` 之前执行此代码. 应用程序启动模板定义了一个名为 `YourProjectNameEfCoreEntityExtensionMappings` 的静态类. 你可以在此类中定义扩展确保在正确的时间执行它. 否则你需要自己处理.

定义实体扩展后你需要使用EF Core的[Add-Migration](https://docs.microsoft.com/en-us/ef/core/miscellaneous/cli/powershell#add-migration)和[Update-Database](https://docs.microsoft.com/en-us/ef/core/miscellaneous/cli/powershell#update-database)命令来创建code first迁移类并更新数据库.

然后你可以使用上一部分中定义的相同额外属性系统来操纵实体上的属性.

## 创建新实体映射到同一个数据库表/Collection

另一个方法是**\*\*创建你自己的实体\*\***映射到**\*\*同一个数据库库\*\***(对于MongoDB数据库是collection)

[应用程序启动模板](Startup-Templates/Application.md)的 `AppUser` 已经实现了这种方法. [EF Core迁移文档](Entity-Framework-Core-Migrations.md)描述了在这些情况下如何实现和管理**\*\*EF Core数据库迁移\*\***. 这种方法同样适用于MongoDB,但你不需要处理数据库迁移问题.

## 创建一个拥有自己数据库表/Collection的新实体

映射你的实体到依赖模块的**\*\*已存在的表\*\***有一些缺点;

\* 你需要处理EF Core的**\*\*数据库迁移架构\*\***. 需要特别注意迁移代码,特别是当你需要在实体间添加**\*\*关系\*\***时.

\* 你的应用程序数据库和模块数据库将是 **\*\*同一个物理数据库\*\***. 通常需要时可以将模块数据库分开,但使用相同的表会对其进行限制.

如果你想要使你的实体或模块定义的实体**\*\*低耦合\*\***,那么可以创建自己的数据库表/collection并且将你的实体映射到自己的数据库表.

在这种情况下你需要处理**\*\*同步问题\*\***,尤其是你要**\*\*复制\*\***相关实体的某些属性/字段时,有一些解决方案;

\* 如果你构建的是一个 **\*\*单体\*\*** 应用程序(或者在同一进程管理你的实体和依赖模块的实体),那么你可以使用[本地事件总线](Local-Event-Bus.md)监听实体更改.

\* 如果你构建的是一个 **\*\*分布式\*\*** 系统,模块的实体和你的实体在不同的 进程/服务 管理(创建/更新/删除),那么你可以使用[分布式事件总线](Distributed-Event-Bus.md)订阅实体的更改事件.

在你处理事件时,你可以在自己的数据库中更改自己的实体.

### 订阅本地事件总线

[本地事件总线](Local-Event-Bus.md)系统是发布和订阅同一应用程序中发生的事件的方法.

假设你想要获取 `IdentityUser` 实体的更改信息(创建,更改或删除). 你可以创建一个类实现 `ILocalEventHandler<EntityChangedEventData<IdentityUser>>` 接口.

````csharp

public class MyLocalIdentityUserChangeEventHandler :

ILocalEventHandler<EntityChangedEventData<IdentityUser>>,

ITransientDependency

{

public async Task HandleEventAsync(EntityChangedEventData<IdentityUser> eventData)

{

var userId = eventData.Entity.Id;

var userName = eventData.Entity.UserName;

//...

}

}

````

\* `EntityChangedEventData<T>` 涵盖了给定实体的创建,更新或删除事件. 如果你需要你可以分别订阅创建,更新或删除事件(在同一个类或不同的类中).

\* 这里的代码在**\*\*本地事务之外执行\*\***,因为它监听 `EntityChanged` 事件. 如果当前[工作单元](Unit-Of-Work.md)是事务性的,你可以订阅 `EntityChangingEventData<T>` 事件,它在**\*\*同一本地(进行)事务\*\***中执行事件处理.

> 提醒:这些方法需要在包含处理类的同一进程中更改 `IdentityUser` 实体. 即使在集群环境(同一应用程序的多个实例在不同的服务器进行),它也完美工作.

### 订阅分布式事件总线

[分布式事件总线](Distributed-Event-Bus.md)是在一个应用程序中发布事件,并在相同服务器或不同服务器运行的相同应用程序或不同应用程序中接收事件的方法.

假设你想要获取 `IdentityUser` 实体的创建,更改或删除信息. 你可以像以下一样创建一个类:

````csharp

public class MyDistributedIdentityUserChangeEventHandler :

IDistributedEventHandler<EntityCreatedEto<EntityEto>>,

IDistributedEventHandler<EntityUpdatedEto<EntityEto>>,

IDistributedEventHandler<EntityDeletedEto<EntityEto>>,

ITransientDependency

{

public async Task HandleEventAsync(EntityCreatedEto<EntityEto> eventData)

{

if (eventData.Entity.EntityType == "Volo.Abp.Identity.IdentityUser")

{

var userId = Guid.Parse(eventData.Entity.KeysAsString);

//...handle the "created" event

}

}

public async Task HandleEventAsync(EntityUpdatedEto<EntityEto> eventData)

{

if (eventData.Entity.EntityType == "Volo.Abp.Identity.IdentityUser")

{

var userId = Guid.Parse(eventData.Entity.KeysAsString);

//...handle the "updated" event

}

}

public async Task HandleEventAsync(EntityDeletedEto<EntityEto> eventData)

{

if (eventData.Entity.EntityType == "Volo.Abp.Identity.IdentityUser")

{

var userId = Guid.Parse(eventData.Entity.KeysAsString);

//...handle the "deleted" event

}

}

}

````

\* 它实现了多个 `IDistributedEventHandler` 接口: **\*\*创建\*\***,**\*\*更改\*\***和**\*\*删除\*\***,因为分布式事件总线单独发布事件,没有本地事件总线那样的"Changed"事件.

\* 它订阅了 `EntityEto`, 这是一个通用的事件类,ABP框架针对所有类型的实体**\*\*自动发布\*\***. 这就是为什么它检查**\*\*实体类型\*\***(因为我们没有假设有对 `IdentityUser` 实体有安全的类型引用,所以它是字符串类型的).

预构建应用模块没有定义专门的事件类型(如`UserEto` - "ETO" 意思是 "事件传输对象"). 此功能在路线图上([关注这个issue](https://github.com/abpframework/abp/issues/3033)),一旦完成后,你就可以订阅独立的实体类型:

````csharp

public class MyDistributedIdentityUserCreatedEventHandler :

IDistributedEventHandler<EntityCreatedEto<UserEto>>,

ITransientDependency

{

public async Task HandleEventAsync(EntityCreatedEto<UserEto> eventData)

{

var userId = eventData.Entity.Id;

var userName = eventData.Entity.UserName;

//...handle the "created" event

}

//...

}

````

\* 这个处理程序只会在新用户创建时执行.

> 唯一预定义的专门事件类是 `UserEto`, 你可以订阅 `EntityCreatedEto<UserEto>` 获取用户创建时的通知. 此事件也适用于身份模块.

## 另请参阅

\* [自定义已存在的模块](Customizing-Application-Modules-Guide.md)

#### 重写服务

# 自定义应用模块: 重写服务

你可能想要**\*\*更改\*\***依赖模块的**\*\*行为(业务逻辑)\*\***. 在这种情况下,你可以使用[依赖注入](Dependency-Injection.md)的能力替换服务,控制器甚至页面模型到你自己的实现.

注册到依赖注入的任何类,包括ABP框架的服务都可以被**\*\*替换\*\***.

你可以根据自己的需求使用不同的选项,下面的章节中将介绍这些选项.

> 请注意,某些服务方法可能不是virtual,你可能无法override,我们会通过设计将其virtual,如果你发现任何方法不可以被覆盖,请[创建一个issue](https://github.com/abpframework/abp/issues/new)或者你直接修改后并发送**\*\*pull request\*\***到GitHub.

## 替换接口

如果给定的服务定义了接口,像 `IdentityUserAppService` 类实现了 `IIdentityUserAppService` 接口,你可以为这个接口创建自己的实现并且替换当前的实现. 例如:

````csharp

public class MyIdentityUserAppService : IIdentityUserAppService, ITransientDependency

{

//...

}

````

`MyIdentityUserAppService` 通过命名约定替换了 `IIdentityUserAppService` 的当前实现. 如果你的类名不匹配,你需要手动公开服务接口:

````csharp

[ExposeServices(typeof(IIdentityUserAppService))]

public class TestAppService : IIdentityUserAppService, ITransientDependency

{

//...

}

````

依赖注入系统允许为一个接口注册多个服务. 注入接口时会解析最后一个注入的服务. 显式的替换服务是一个好习惯.

示例:

````csharp

[Dependency(ReplaceServices = true)]

[ExposeServices(typeof(IIdentityUserAppService))]

public class TestAppService : IIdentityUserAppService, ITransientDependency

{

//...

}

````

使用这种方法, `IIdentityUserAppService` 接口将只会有一个实现. 也可以使用以下方法替换服务:

````csharp

context.Services.Replace(

ServiceDescriptor.Transient<IIdentityUserAppService, MyIdentityUserAppService>()

);

````

你可以在[模块](Module-Development-Basics.md)类的 `ConfigureServices` 方法编写替换服务代码.

## 重写一个服务类

大多数情况下,你会仅想改变服务当前实现的一个或几个方法. 重新实现完整的接口变的繁琐,更好的方法是继承原始类并重写方法.

### 示例: 重写服务方法

````csharp

[Dependency(ReplaceServices = true)]

[ExposeServices(typeof(IIdentityUserAppService), typeof(IdentityUserAppService), typeof(MyIdentityUserAppService))]

public class MyIdentityUserAppService : IdentityUserAppService

{

//...

public MyIdentityUserAppService(

IdentityUserManager userManager,

IIdentityUserRepository userRepository,

IGuidGenerator guidGenerator

) : base(

userManager,

userRepository,

guidGenerator)

{

}

public async override Task<IdentityUserDto> CreateAsync(IdentityUserCreateDto input)

{

if (input.PhoneNumber.IsNullOrWhiteSpace())

{

throw new AbpValidationException(

"Phone number is required for new users!",

new List<ValidationResult>

{

new ValidationResult(

"Phone number can not be empty!",

new []{"PhoneNumber"}

)

}

); }

return await base.CreateAsync(input);

}

}

````

示例中**\*\*重写\*\***了 `IdentityUserAppService` [应用程序](Application-Services.md) `CreateAsync` 方法检查手机号码. 然后调用了基类方法继续**\*\*基本业务逻辑\*\***. 通过这种方法你可以在基本业务逻辑**\*\*之前\*\***和**\*\*之后\*\***执行其他业务逻辑.

你也可以完全**\*\*重写\*\***整个业务逻辑去创建用户,而不是调用基类方法.

### 示例: 重写领域服务

````csharp

[Dependency(ReplaceServices = true)]

[ExposeServices(typeof(IdentityUserManager))]

public class MyIdentityUserManager : IdentityUserManager

{

public MyIdentityUserManager(

IdentityUserStore store,

IIdentityRoleRepository roleRepository,

IIdentityUserRepository userRepository,

IOptions<IdentityOptions> optionsAccessor,

IPasswordHasher<IdentityUser> passwordHasher,

IEnumerable<IUserValidator<IdentityUser>> userValidators,

IEnumerable<IPasswordValidator<IdentityUser>> passwordValidators,

ILookupNormalizer keyNormalizer,

IdentityErrorDescriber errors,

IServiceProvider services,

ILogger<IdentityUserManager> logger,

ICancellationTokenProvider cancellationTokenProvider) :

base(store,

roleRepository,

userRepository,

optionsAccessor,

passwordHasher,

userValidators,

passwordValidators,

keyNormalizer,

errors,

services,

logger,

cancellationTokenProvider)

{

}

public async override Task<IdentityResult> CreateAsync(IdentityUser user)

{

if (user.PhoneNumber.IsNullOrWhiteSpace())

{

throw new AbpValidationException(

"Phone number is required for new users!",

new List<ValidationResult>

{

new ValidationResult(

"Phone number can not be empty!",

new []{"PhoneNumber"}

)

}

);

}

return await base.CreateAsync(user);

}

}

````

示例中类继承了 `IdentityUserManager` [领域服务](Domain-Services.md),并且重写了 `CreateAsync` 方法进行了与之前相同的手机号码检查. 结果也是一样的,但是这次我们在领域服务实现了它,假设这是我们系统的**\*\*核心领域逻辑\*\***.

> 这里需要 `[ExposeServices(typeof(IdentityUserManager))]` attribute,因为 `IdentityUserManager` 没有定义接口 (像 `IIdentityUserManager`) ,依赖注入系统并不会按照约定公开继承类的服务(如已实现的接口).

参阅[本地化系统](Localization.md)了解如何自定义错误消息.

### 重写其他服务

控制器,框架服务,视图组件类以及其他类型注册到依赖注入的类都可以像上面的示例那样被重写.

## 扩展数据传输对象

你可以如[扩展实体文档](Customizing-Application-Modules-Extending-Entities.md)所述扩展实体. 并使用上面介绍的重写相关服务**\*\*使用自定义属性\*\*\*\*执行其他业务逻辑\*\***.

应用程序使用的数据传输对象(**\*\*DTO\*\***)同样可扩展. 这样你可以使服务返回其他属性并在UI(或其他客户端)得到其他属性.

### 示例

假设你已经按照[扩展实体文档](Customizing-Application-Modules-Extending-Entities.md)中的说明添加了 `SocialSecurityNumber` 并希望从 `IdentityUserAppService的GetListAsync` 方法获取用户列表时包括此属性.

你可以使用[对象扩展系统](Object-Extensions.md)将属性添加到 `IdentityUserDto`. 在应用程序启动模板带有的 `YourProjectNameDtoExtensions` 类中编写以下代码:

````csharp

ObjectExtensionManager.Instance

.AddOrUpdateProperty<IdentityUserDto, string>(

"SocialSecurityNumber"

);

````

这段代码为 `IdentityUserDto` 类添加了 `string` 类型的 `SocialSecurityNumber` 属性. 现在你可以在RREST API客户端调用 `/api/identity/users` HTTP API(内部使用 `IdentityUserAppService`),你会在 `extraProperties` 部分看到 `SocialSecurityNumber` 值.

````json

{

"totalCount": 1,

"items": [{

"tenantId": null,

"userName": "admin",

"name": "admin",

"surname": null,

"email": "admin@abp.io",

"emailConfirmed": false,

"phoneNumber": null,

"phoneNumberConfirmed": false,

"twoFactorEnabled": false,

"lockoutEnabled": true,

"lockoutEnd": null,

"concurrencyStamp": "b4c371a0ab604de28af472fa79c3b70c",

"isDeleted": false,

"deleterId": null,

"deletionTime": null,

"lastModificationTime": "2020-04-09T21:25:47.0740706",

"lastModifierId": null,

"creationTime": "2020-04-09T21:25:46.8308744",

"creatorId": null,

"id": "8edecb8f-1894-a9b1-833b-39f4725db2a3",

"extraProperties": {

"SocialSecurityNumber": "123456789"

}

}]

}

````

手动添加了 `123456789` 值到数据库中.

所有预构建的模块都在DTO中支持额外属性,你可以对其轻松的配置.

### 定义检查

当为实体[定义](Customizing-Application-Modules-Extending-Entities.md)额外的属性时,由于安全性它不会自动出现在所有相关的DTO中. 额外属性可能包含敏感数据并且你可能不想默认公开给客户端.

因此如果要用于DTO,需要为相应的DTO显式定义相同的属性(如上所述). 如果要允许在用户创建时进行设置还需要为 `IdentityUserCreateDto` 定义.

如果属性并不是安全敏感,这可能会很枯燥. 对象扩展系统允许你忽略检查定义的属性. 参阅示例:

````csharp

ObjectExtensionManager.Instance

.AddOrUpdateProperty<IdentityUser, string>(

"SocialSecurityNumber",

options =>

{

options.MapEfCore(b => b.HasMaxLength(32));

options.CheckPairDefinitionOnMapping = false;

}

);

````

这是定义实体属性的另一种方法( 有关 `ObjectExtensionManager` 更多信息,请参阅[文档](Object-Extensions.md)). 这次我们设置了 `CheckPairDefinitionOnMapping` 为false,在将实体映射到DTO时会跳过定义检查.

如果你不喜欢这种方法,但想简单的向多个对象(DTO)添加单个属性, `AddOrUpdateProperty` 可以使用类型数组添加额外的属性:

````csharp

ObjectExtensionManager.Instance

.AddOrUpdateProperty<string>(

new[]

{

typeof(IdentityUserDto),

typeof(IdentityUserCreateDto),

typeof(IdentityUserUpdateDto)

},

"SocialSecurityNumber"

);

````

### 关于用户界面

该系统允许你向实体和DTO添加额外的属性并执行自定义业务代码,但它与用户界面无关.

参阅 [重写用户界面](Customizing-Application-Modules-Overriding-User-Interface.md) 指南了解关于UI部分.

## 如何找到服务?

[模块文档](Modules/Index.md) 包含了定义的主要服务列表. 另外 你也可以查看[源码](https://github.com/abpframework/abp/tree/dev/modules)找到所有的服务.

#### 重写用户界面

# 重写用户界面

你可以想要重写页面,组件,JavaScript,CSS或你依赖模块的图片文件. 重写UI取决于你使用的UI框架. 选择UI框架以继续:

\* [ASP.NET Core (MVC / Razor Pages)](UI/AspNetCore/Customization-User-Interface.md)

\* [Angular](UI/Angular/Customization-User-Interface.md)

最佳实践

### 介绍

这篇文档描述了想要满足以下规范的**\*\*模块\*\***的**\*\*最佳实践\*\***与**\*\*约定\*\***:

\* 开发应用**\*\*领域驱动设计\*\***模式的最佳实践的模块.

\* 开发 **\*\*DBMS 与 ORM 独立\*\*** 的模块.

\* 开发可用作 **\*\*远程服务 / 微服务\*\*** 的模块, 并可以集成到 **\*\*单体\*\*** 应用程序中.

本指南主要用于 **\*\*应用程序\*\*** 开发.

### 指南

\* 总体

\* [模块架构](Module-Architecture.md)

\* 领域层

\* [实体](Entities.md)

\* [仓储](Repositories.md)

\* [领域服务](Domain-Services.md)

\* 应用程序层

\* [应用程序服务](Application-Services.md)

\* [数据传输对象](Data-Transfer-Objects.md)

\* 数据访问

\* [Entity Framework Core 集成](Entity-Framework-Core-Integration.md)

\* [MongoDB 集成](MongoDB-Integration.md)

#### 总体

### 解决方案结构

\* **\*\*推荐\*\*** 在Visual Sudio中为每个模块创建一个单独的解决方案.

\* **\*\*推荐\*\*** 将解决方案命名为*\*CompanyName.ModuleName\**(对于ABP核心模块来说,它的命名方式是*\*Volo.Abp.ModuleName\**).

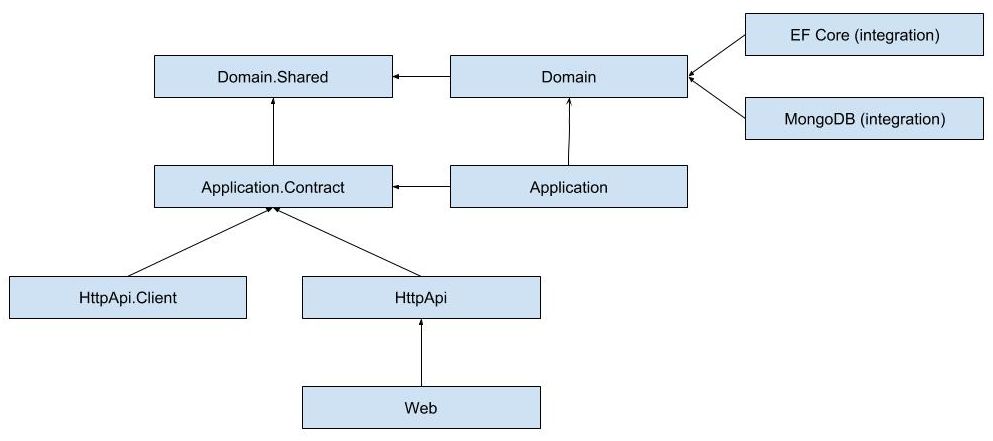
\* **\*\*推荐\*\*** 一个模块做为分层项目开发,因为它有几个包(项目)是相互关联的.

\* 每个包都有自己的模块定义文件,并显式声明所依赖的包/模块的依赖关系.

### 层(layers) & 包(packages)

下面展示了一个分层良好的模块中的包以及它们之间的依赖关系:

![module-layers-and-packages](../images/module-layers-and-packages.jpg)



最终的目地是让应用程序以灵活的方式使用该模块. 示例应用程序:

\* **\*\*A)\*\*** **\*\*单体\*\***应用程序;

\* 添加对**\*\*Web\*\***和**\*\*Application\*\***包的引用.

\* 根据需要添加对**\*\*EF Core\*\*** 或 **\*\*MongoDB\*\*** 包的引用.

\* 效果;

\* 应用程序可以显示**\*\*模块的UI\*\***.

\* 它在**\*\*同一进程\*\***中托管**\*\*应用层\*\***和**\*\*领域层\*\*** (这就是为什么它引用对数据库集成包).

\* 此应用程序还提供了模块的**\*\*HTTP API\*\***(因为它通过Web包引用了HttpApi包).

\* **\*\*B)\*\*** 仅为**\*\*微服务\*\***提供模块的应用程序;

\* 添加对**\*\*HttpApi\*\***和**\*\*Application\*\***包的引用.

\* 根据需要添加对**\*\*EF Core\*\*** 或 **\*\*MongoDB\*\*** 包的引用.

\* 效果;

\* 应用程序**\*\*无法显示模块的UI\*\***, 因为它没有对Web包的引用.

\* 它在**\*\*同一进程\*\***中托管**\*\*应用层\*\***和**\*\*领域层\*\*** (这就是为什么它引用对数据库集成包).

\* 此应用程序提供了模块的**\*\*HTTP API\*\***(它的主要目标).

\* **\*\*C)\*\*** 显示模块**\*\*UI\*\***但是不托管应用层的应用程序(仅将其用作由应用程序A或B托管的远程服务)

\* 添加对**\*\*Web\*\***和**\*\*HttpApi.Client\*\***包的引用.

\* 配置HttpApi.Client包的远程端点.

\* 效果;

\* 应用程序可以显示**\*\*模块的UI\*\***.

\* 它不会在同一进程中托管模块的应用层和领域层. 而是将其用作**\*\*远程服务\*\***.

\* 此应用程序还提供了模块的**\*\*HTTP API\*\***(因为它通过Web包引用了HttpApi包).

\* **\*\*D)\*\*** **\*\*客户端\*\***应用程序 (或微服务) 只使用模块作为远程服务(由应用程序A,B或C托管);

\* 添加对**\*\*HttpApi.Client\*\***包的引用.

\* 配置HttpApi.Client包的远程端点.

\* 效果;

\* 应用程序可以使用模块的所有功能作为**\*\*远程客户端\*\***.

\* 应用程序只是一个客户端,**\*\*无法提供\*\***模块的**\*\*HTTP API\*\***.

\* 应用程序只是一个客户端,**\*\*无法显示\*\***模块的**\*\*UI\*\***.

\* **\*\*E\*\***) 托管模块的HTTP API但只是将所有请求转发给另一个应用程序的代理应用程序 (由应用程序A, B或C托管);

\* 添加对**\*\*HttpApi\*\***和**\*\*HttpApi.Client\*\***包的引用.

\* 配置HttpApi.Client包的远程端点.

\* 效果;

\* 应用程序可以将模块的所有功能用作**\*\*远程客户端\*\***.

\* 应用程序也服务于模块的**\*\*HTTP API\*\***, 但实际上它的工作方式与代理一样,将所有请求(模块)重定向到另一个远程服务器.

下一节将详细地介绍这些包.

#### 领域层

\* **\*\*推荐\*\*** 将领域层划分为两个项目:

\* **\*\*Domain.Shared\*\*** 包(项目) 命名为*\*CompanyName.ModuleName.Domain.Shared\**,包含常量,枚举和其他类型, 它不能包含实体,存储库,域服务或任何其他业务对象. 可以安全地与模块中的所有层使用. 此包也可以与第三方客户端使用.

\* **\*\*Domain\*\*** 包(项目) 命名为*\*CompanyName.ModuleName.Domain\**, 包含实体, 仓储接口,领域服务接口及其实现和其他领域对象.

\* Domain 包依赖于 **\*\*Domain.Shared\*\*** 包.

#### 应用服务层

\* **\*\*推荐\*\*** 将应用服务层划分为两个项目:

\* **\*\*Application.Contracts\*\*** 包(项目) 命名为\*CompanyName.ModuleName.Application.Contracts,包含应用服务接口和相关的数据传输对象(DTO).

\* Application contract 包依赖于 **\*\*Domain.Shared\*\*** 包.

\* **\*\*Application\*\*** 包(项目)命名为*\*CompanyName.ModuleName.Application\**,包含应用服务实现.

\* Application 包依赖于 **\*\*Domain\*\*** 包和 **\*\*Application.Contracts\*\*** 包.

#### 基础设施层

\* **\*\*推荐\*\*** 为每个orm/数据库集成创建一个独立的集成包, 比如Entity Framework Core 和 MongoDB.

\* **\*\*推荐\*\*** 例如, 创建一个抽象Entity Framework Core集成的*\*CompanyName.ModuleName.EntityFrameworkCore\** 包. ORM 集成包依赖于 **\*\*Domain\*\*** 包.

\* **\*\*不推荐\*\*** 依赖于orm/数据库集成包中的其他层.

\* **\*\*推荐\*\*** 为每个主要的库创建一个独立的集成包, 在不影响其他包的情况下可以被另一个库替换.

#### HTTP 层

\* **\*\*推荐\*\*** 创建命名为*\*CompanyName.ModuleName.HttpApi\**的**\*\*HTTP API\*\***包, 为模块开发REST风格的HTTP API.

\* HTTP API 包只依赖于 **\*\*Application.Contracts\*\*** 包. 不要依赖 Application 包.

\* **\*\*推荐\*\*** 为每个应用服务创建一个Controller (通常通过实现其接口). 这些控制器使用应用服务接口来委托操作. 它根据需要配置路由, HTTP方法和其他与Web相关的东西.

\* **\*\*推荐\*\*** 创建一个为HTTP API包提供客户端服务的**\*\*HTTP API Client\*\***包, 它的命名为*\*Companyname.ModuleName.HttpApi.Client\**. 这些客户端服务将应用服务接口实现远程端点的客户端.

\* HTTP API Client 包仅依赖于 **\*\*Application.Contracts\*\*** 包.

\* **\*\*推荐\*\*** 使用ABP框架提供的动态代理HTTP C＃客户端的功能.

#### Web 层

\* **\*\*推荐\*\*** 创建命名为*\*CompanyName.ModuleName.Web\**的 **\*\*Web\*\***包. 包含页面,视图,脚本,样式,图像和其他UI组件.

\* Web 包仅依赖于 **\*\*HttpApi\*\*** 包.

#### 领域层

实体

### 实体

每个聚合根也是一个实体, 所以这些规则对聚合根也是有效的, 除非聚合根的某些规则覆盖了它们.

- **\*\*推荐\*\*** 在 **\*\*领域层\*\*** 中定义实体.

#### 主构造函数

\* **\*\*推荐\*\*** 定义一个 **\*\*主构造函数\*\*** 确保实体在创建时的有效性, 在代码中通过主构造函数创建实体的新实例.

- **\*\*推荐\*\*** 根据需求把主构造函数定义为 `public`,`internal` 或 `protected internal` . 如果它不是public的, 那么应该由领域服务来创建实体.

- **\*\*推荐\*\*** 总是在主构造函数中初始化子集合.

- **\*\*不推荐\*\*** 在主构造函数中生成 `Guid` 键, 应该将其做为参数获取, 在调用时推荐使用 `IGuidGenerator` 生成新的 `Guid` 值做为参数.

#### 无参构造函数

- **\*\*推荐\*\*** 总是定义 `protected` 无参构造函数与ORM兼容.

#### 引用

- **\*\*推荐\*\*** 总是通过 **\*\*id\*\*** **\*\*引用\*\*** 其他聚合根, 不要将导航属性添加到其他聚合根中.

#### 类的其他成员

- **\*\*推荐\*\*** 总是将属性与方法定义为 `virtual` (除了`私有`方法 ). 因为有些ORM和动态代理工具需要.

- **\*\*推荐\*\*** 保持实体在自身边界内始终 **\*\*有效\*\*** 和 **\*\*一致\*\***.

- **\*\*推荐\*\*** 使用 `private`,`protected`,`internal`或`protected internal` setter定义属性, 保护实体的一致性和有效性.

- **\*\*推荐\*\*** 定义 `public`, `internal` 或 `protected internal` (virtual)**\*\*方法\*\***在必要时更改属性值(使用非public setters时).

### 聚合根

#### 主键

\* **\*\*推荐\*\*** 总是使用 **\*\*Id\*\*** 属性做为聚合根主键.

\* **\*\*不推荐\*\*** 在聚合根中使用 **\*\*复合主键\*\***.

\* **\*\*推荐\*\*** 所有的聚合根都使用 **\*\*Guid\*\*** 类型 **\*\*主键\*\***.

#### 基类

\* **\*\*推荐\*\*** 根据需求继承 `AggregateRoot<TKey>` 或以下一个审计类 (`CreationAuditedAggregateRoot<TKey>`, `AuditedAggregateRoot<TKey>` 或 `FullAuditedAggregateRoot<TKey>`).

#### 聚合边界

\* **\*\*推荐\*\*** 聚合**\*\*尽可能小\*\***. 大多数聚合只有原始属性, 不会有子集合. 把这些视为设计决策:

\* 加载和保存聚合的 **\*\*性能\*\*** 与 **\*\*内存\*\*** 成本 (请记住,聚合通常是做为一个单独的单元被加载和保存的). 较大的聚合会消耗更多的CPU和内存.

\* **\*\*一致性\*\*** & **\*\*有效性\*\*** 边界.

### 示例

#### 聚合根

````C#

public class Issue : FullAuditedAggregateRoot<Guid> //使用Guid作为键/标识符

{

public virtual string Title { get; private set; } //使用 SetTitle() 方法set

public virtual string Text { get; set; } //可以直接set,null值也是允许的

public virtual Guid? MilestoneId { get; set; } //引用其他聚合根

public virtual bool IsClosed { get; private set; }

public virtual IssueCloseReason? CloseReason { get; private set; } //一个枚举类型

public virtual Collection<IssueLabel> Labels { get; protected set; } //子集合

protected Issue()

{

/\* 此构造函数是提供给ORM用来从数据库中获取实体.

\* - 无需初始化Labels集合

因为它会被来自数据库的值覆盖.

- It's protected since proxying and deserialization tools

可能不适用于私有构造函数.

\*/

}

//主构造函数

public Issue(

Guid id, //从调用代码中获取Guid值

[NotNull] string title, //表示标题不能为空.

string text = null,

Guid? milestoneId = null) //可选参数

{

Id = id;

Title = Check.NotNullOrWhiteSpace(title, nameof(title)); //验证

Text = text;

MilestoneId = milestoneId;

Labels = new Collection<IssueLabel>(); //总是初始化子集合

}

public virtual void SetTitle([NotNull] string title)

{

Title = Check.NotNullOrWhiteSpace(title, nameof(title)); //验证

}

/\* AddLabel和RemoveLabel方法管理Labels集合

\* 安全的方式(防止两次添加相同的标签) \*/

public virtual void AddLabel(Guid labelId)

{

if (Labels.Any(l => l.LabelId == labelId))

{

return;

}

Labels.Add(new IssueLabel(Id, labelId));

}

public virtual void RemoveLabel(Guid labelId)

{

Labels.RemoveAll(l => l.LabelId == labelId);

}

/\* Close和ReOpen方法可保护一致性

\* IsClosed 与 CloseReason 属性. \*/

public virtual void Close(IssueCloseReason reason)

{

IsClosed = true;

CloseReason = reason;

}

public virtual void ReOpen()

{

IsClosed = false;

CloseReason = null;

}

}

````

#### 实体

````C#

public class IssueLabel : Entity

{

public virtual Guid IssueId { get; private set; }

public virtual Guid LabelId { get; private set; }

protected IssueLabel()

{

}

public IssueLabel(Guid issueId, Guid labelId)

{

IssueId = issueId;

LabelId = labelId;

}

}

````

### 参考文献

\* Effective Aggregate Design by Vaughn Vernon

http://dddcommunity.org/library/vernon\_2011

仓储

### 仓储接口

\* **\*\*推荐\*\*** 在**\*\*领域层\*\***中定义仓储接口.

\* **\*\*推荐\*\*** 为**\*\*每个聚合根\*\***定义仓储接口(如 `IIdentityUserRepository`)并创建相应的实现.

\* **\*\*推荐\*\*** 在应用代码中使用仓储时应该注入仓储接口.

\* **\*\*不推荐\*\*** 在应用代码中使用泛型仓储接口(如 `IRepository<IdentityUser, Guid>`).

\* **\*\*不推荐\*\*** 在应用代码(领域, 应用... 层)中使用 `IQueryable<TEntity>` 特性.

聚合根的示例:

````C#

public class IdentityUser : AggregateRoot<Guid>

{

//...

}

````

定义仓储接口, 如下所示:

````C#

public interface IIdentityUserRepository : IBasicRepository<IdentityUser, Guid>

{

//...

}

````

\* **\*\*不推荐\*\*** 仓储接口继承 `IRepository<TEntity, TKey>` 接口. 因为它继承了 `IQueryable` 而仓储不应该将`IQueryable`暴漏给应用.

\* **\*\*推荐\*\*** 通常仓储接口继承自 `IBasicRepository<TEntity, TKey>` 或更低级别的接口, 如 `IReadOnlyRepository<TEntity, TKey>` (在需要的时候).

\* **\*\*不推荐\*\*** 为实体定义仓储接口,因为它们**\*\*不是聚合根\*\***.

### 仓储方法

\* **\*\*推荐\*\*** 所有的仓储方法定义为 **\*\*异步\*\***.

\* **\*\*推荐\*\*** 为仓储的每个方法添加 **\*\*可选参数\*\*** `cancellationToken` . 例:

````C#

Task<IdentityUser> FindByNormalizedUserNameAsync(

[NotNull] string normalizedUserName,

CancellationToken cancellationToken = default

);

````

\* **\*\*推荐\*\*** 为仓储的每个异步方法创建一个 **\*\*同步扩展\*\*** 方法. 示例:

````C#

public static class IdentityUserRepositoryExtensions

{

public static IdentityUser FindByNormalizedUserName(

this IIdentityUserRepository repository,

[NotNull] string normalizedUserName)

{

return AsyncHelper.RunSync(

() => repository.FindByNormalizedUserNameAsync(normalizedUserName)

);

}

}

````

对于同步方法而言, 这会让它们更方便的调用仓储方法.

\* **\*\*推荐\*\*** 为仓储中返回**\*\*单个实体\*\***的方法添加一个可选参数 `bool includeDetails = true` (默认值为`true`). 示例:

````C#

Task<IdentityUser> FindByNormalizedUserNameAsync(

[NotNull] string normalizedUserName,

bool includeDetails = true,

CancellationToken cancellationToken = default

);

````

该参数由ORM实现, 用来加载实体子集合.

\* **\*\*推荐\*\*** 为仓储中返回**\*\*实体列表\*\***的方法添加一个可选参数 `bool includeDetails = false` (默认值为`false`). 示例:

````C#

Task<List<IdentityUser>> GetListByNormalizedRoleNameAsync(

string normalizedRoleName,

bool includeDetails = false,

CancellationToken cancellationToken = default

);

````

\* **\*\*不推荐\*\*** 创建复合类通过调用仓储单个方法返回组合实体. 比如: *\*UserWithRoles\**, *\*UserWithTokens\**, *\*UserWithRolesAndTokens\**. 相反, 正确的使用 `includeDetails` 选项, 在需要时加载实体所有的详细信息.

\* **\*\*避免\*\*** 为了从仓储中获取实体的部分属性而为实体创建投影类. 比如: 避免通过创建BasicUserView来选择所需的一些属性. 相反可以直接使用聚合根类. 不过这条规则有例外情况:

\* 性能对于用例来说非常重要,而且使用整个聚合根对性能的影响非常大.

### 另外请参阅

\* [Entity Framework Core 集成](Entity-Framework-Core-Integration.md)

\* [MongoDB 集成](MongoDB-Integration.md)

领域服务

> **\*\*This document is not ready yet. Please see the [**Domain Services**](../Domain-Services.md) document.\*\***

#### 应用程序层

应用程序服务

\* **\*\*推荐\*\*** 为每个 **\*\*聚合根\*\*** 创建一个应用服务.

### 应用服务接口

\* **\*\*推荐\*\*** 在 **\*\*application.contracts\*\***层中为每一个应用服务定义一个`接口`.

\* **\*\*推荐\*\*** 继承 `IApplicationService` 接口 .

\* **\*\*推荐\*\*** 接口名称使用`AppService` 后缀 (如: `IProductAppService`).

\* **\*\*推荐\*\*** 为服务创建输入输出DTO(数据传输对象).

\* **\*\*不推荐\*\*** 服务中含有返回实体的方法.

\* **\*\*推荐\*\*** 根据[DTO 最佳实践](Data-Transfer-Objects.md)定义DTO.

#### 输出

\* **\*\*避免\*\*** 为相同或相关实体定义过多的输出DTO. 为实体定义 **\*\*基础\*\*** 和 **\*\*详细\*\*** DTO.

##### 基础DTO

**\*\*推荐\*\*** 为聚合根定义一个**\*\*基础\*\***DTO.

- 直接包含实体中所有的**\*\*原始属性\*\***.

- 例外: 出于**\*\*安全\*\***原因,可以**\*\*排除\*\***某些属性(像 `User.Password`).

- 包含实体中所有**\*\*子集合\*\***, 每个集合项都是一个简单的**\*\*关系DTO\*\***.

示例:

```c#

[Serializable]

public class IssueDto : ExtensibleFullAuditedEntityDto<Guid>

{

public string Title { get; set; }

public string Text { get; set; }

public Guid? MilestoneId { get; set; }

public Collection<IssueLabelDto> Labels { get; set; }

}

[Serializable]

public class IssueLabelDto

{

public Guid IssueId { get; set; }

public Guid LabelId { get; set; }

}

```

##### 详细DTO

**\*\*推荐\*\*** 如果实体持有对其他聚合根的引用,那么应该为其定义**\*\*详细\*\***DTO.

\* 直接包含实体中所有的 **\*\*原始属性\*\***.

- 例外-1: 出于**\*\*安全\*\***原因,可以**\*\*排除\*\***某些属性(像 `User.Password`).

- 例外-2: **\*\*推荐\*\*** 排除引用属性(如上例中的 `MilestoneId`). 为其添加引用属性的详细信息.

\* 为每个引用属性添加其**\*\*基本DTO\*\*** .

\* 包含实体的**\*\*所有子集合\*\***, 集合中的每项都是相关实体的基本DTO.

示例:

````C#

[Serializable]

public class IssueWithDetailsDto : ExtensibleFullAuditedEntityDto<Guid>

{

public string Title { get; set; }

public string Text { get; set; }

public MilestoneDto Milestone { get; set; }

public Collection<LabelDto> Labels { get; set; }

}

[Serializable]

public class MilestoneDto : ExtensibleEntityDto<Guid>

{

public string Name { get; set; }

public bool IsClosed { get; set; }

}

[Serializable]

public class LabelDto : ExtensibleEntityDto<Guid>

{

public string Name { get; set; }

public string Color { get; set; }

}

````

#### 输入

\* **\*\*不推荐\*\*** 在输入DTO中定义未在服务类中使用的属性.

\* **\*\*不推荐\*\*** 在应用服务方法之间共享输入DTO.

\* **\*\*不推荐\*\*** 继承另一个输入DTO类.

\* **\*\*可以\*\*** 继承自抽象基础DTO类, 并以这种方式在不同的DTO之间共享一些属性. 但是在这种情况下需要非常小心, 因为更新基础DTO会影响所有相关的DTO和服务方法. 所以避免这样做是一种好习惯.

#### 方法

\* **\*\*推荐\*\*** 为异步方法使用 **\*\*Async\*\*** 后缀.

\* **\*\*不推荐\*\*** 在方法名中重复实体的名称.

\* 例如: 在 `IProductAppService` 中定义`GetAsync(...)` 而不是 `GetProductAsync(...)` .

##### 获取单一实体

\* **\*\*推荐\*\*** 使用 `GetAsync` 做为**\*\*方法名\*\***.

\* **\*\*推荐\*\*** 使用id做为方法参数.

\* 返回 **\*\*详细DTO\*\***. 示例:

````C#

Task<QuestionWithDetailsDto> GetAsync(Guid id);

````

##### 获取实体集合

\* **\*\*推荐\*\*** 使用 `GetListAsync` 做为**\*\*方法名\*\***.

\* **\*\*推荐\*\*** 如果需要获取单个DTO可以使用参数进行 **\*\*过滤\*\***, **\*\*排序\*\*** 和 **\*\*分页\*\***.

\* **\*\*推荐\*\*** 尽可能让过滤参数可选.

\* **\*\*推荐\*\*** 将排序与分页属性设置为可选, 并且提供默认值.

\* **\*\*推荐\*\*** 限制最大页数大小 (基于性能考虑).

\* **\*\*推荐\*\*** 返回 **\*\*详细DTO\*\***集合. 示例:

````C#

Task<List<QuestionWithDetailsDto>> GetListAsync(QuestionListQueryDto queryDto);

````

##### 创建一个新实体

\* **\*\*推荐\*\*** 使用 `CreateAsync` 做为**\*\*方法名\*\***.

\* **\*\*推荐\*\*** 使用**\*\*专门的输入DTO\*\***来创建实体.

\* **\*\*推荐\*\*** DTO类从 `ExtensibleObject` 类继承(或任何实现 `ExtensibleObject`的类) 以允许在需要时传递额外的属性.

\* **\*\*推荐\*\*** 使用 **\*\*data annotations\*\*** 进行输入验证.

\* 尽可能在**\*\*领域\*\***之间共享常量(通过**\*\*domain shared\*\*** package定义的常量).

\* **\*\*推荐\*\*** 只需要创建实体的**\*\*最少\*\***信息, 但是提供了其他可选属性.

示例**\*\*方法\*\***:

````C#

Task<QuestionWithDetailsDto> CreateAsync(CreateQuestionDto questionDto);

````

输入**\*\*DTO\*\***:

````C#

[Serializable]

public class CreateQuestionDto : ExtensibleObject

{

[Required]

[StringLength(QuestionConsts.MaxTitleLength, MinimumLength = QuestionConsts.MinTitleLength)]

public string Title { get; set; }

[StringLength(QuestionConsts.MaxTextLength)]

public string Text { get; set; } //Optional

public Guid? CategoryId { get; set; } //Optional

}

````

##### 更新已存在的实体

- **\*\*推荐\*\*** 使用 `UpdateAsync` 做为**\*\*方法名\*\***.

- **\*\*推荐\*\*** 使用**\*\*专门的输入DTO\*\***来更新实体.

- **\*\*推荐\*\*** DTO类从 `ExtensibleObject` 类继承(或任何实现 `ExtensibleObject`的类) 以允许在需要时传递额外的属性.

- **\*\*推荐\*\*** 获取实体的id做为分离的原始参数. 不要包含更新DTO.

- **\*\*推荐\*\*** 使用 **\*\*data annotations\*\*** 进行输入验证.

- 尽可能在**\*\*领域\*\***之间共享常量(通过**\*\*domain shared\*\*** package定义的常量).

- **\*\*推荐\*\*** 返回更新实体的**\*\*详细\*\***DTO.

示例:

````C#

Task<QuestionWithDetailsDto> UpdateAsync(Guid id, UpdateQuestionDto updateQuestionDto);

````

##### 删除已存在的实体

- **\*\*推荐\*\*** 使用 `DeleteAsync` 做为**\*\*方法名\*\***.

- **\*\*推荐\*\*** 使用原始参数 id. 示例:

````C#

Task DeleteAsync(Guid id);

````

##### 其他方法

\* **\*\*可以\*\*** 定义其他方法以对实体执行操作. 示例:

````C#

Task<int> VoteAsync(Guid id, VoteType type);

````

此方法为试题投票并返回试题的当前分数.

### 应用服务实现

\* **\*\*推荐\*\*** 开发**\*\*完全独立于web层\*\***的应用层.

\* **\*\*推荐\*\*** 在**\*\*应用层\*\***实现应用服务接口.

\* **\*\*推荐\*\*** 使用命名约定. 如: 为 `IProductAppService` 接口创建 `ProductAppService` 类.

\* **\*\*推荐\*\*** 继承自 `ApplicationService` 基类.

\* **\*\*推荐\*\*** 将所有的公开方法定义为 **\*\*virtual\*\***, 以便开发人员继承和覆盖它们.

\* **\*\*不推荐\*\*** 定义 **\*\*private\*\*** 方法. 应该定义为 **\*\*protected virtual\*\***, 这样开发人员可以继承和覆盖它们.

#### 使用仓储

\* **\*\*推荐\*\*** 使用专门设计的仓储 (如 `IProductRepository`).

\* **\*\*不推荐\*\*** 使用泛型仓储 (如 `IRepository<Product>`).z`

#### 查询数据

\* **\*\*不推荐\*\*** 在应用服务方法中使用linq/sql查询来自数据库的数据. 让仓储负责从数据源执行linq/sql查询.

#### 额外的属性

\* **\*\*推荐\*\*** 使用 `MapExtraPropertiesTo` 扩展方法 ([参阅](../Object-Extensions.md)) 或配置对象映射 (`MapExtraProperties`) 以允许应用开发人员能够扩展对象和服务.

#### 操作/删除 实体

\* **\*\*推荐\*\*** 总是从数据库中获取所有的相关实体以对他们执行操作.

\* **\*\*推荐\*\*** 更新实体后调用存储的Update/UpdateAsync方法.因为并非所有数据库API都支持更改跟踪和自动更新.

#### 处理文件

\* **\*\*不推荐\*\*** 在应用服务中使用任何web组件, 例如`IFormFile`和`Stream`. 如果你想接收一个文件, 可以使用`byte[]`.

\* **\*\*推荐\*\*** 使用`Controller`来处理文件上传, 然后将文件的`byte[]`传递给应用服务的方法。

#### 使用其他应用服务

\* **\*\*不推荐\*\*** 在同一个模块/应用中使用其他应用服务. 相反;

\* 使用领域层执行所需的任务.

\* 提取新类并在应用服务之间共享, 在必要时代码重用. 但要小心不要结合两个用例. 它们在开始时可能看起来相似, 但可能会随时间演变为不同的方向. 请谨慎使用代码共享.

\* **\*\*可以\*\*** 在以下情况下使用其他应用服务;

\* 它们是另一个模块/微服务的一部分.

\* 当前模块仅引用已使用模块的application contracts.

数据传输对象

### 数据传输对象DTO

\* **\*\*推荐\*\*** 在 **\*\*application.contracts\*\*** 层中定义DTO.

\* **\*\*推荐\*\*** 在可能和必要的情况下从预构建的 **\*\*基础DTO类\*\*** 继承 (如 `EntityDto<TKey>`, `CreationAuditedEntityDto<TKey>`, `AuditedEntityDto<TKey>`, `FullAuditedEntityDto<TKey>` 等).

\* **\*\*推荐\*\*** 从**\*\*聚合根\*\***的**\*\*扩展DTO\*\***继承(如 `ExtensibleAuditedEntityDto<TKey>`), 因为聚合根是可扩展的额外的属性使用这种方式映射到DTO.

\* **\*\*推荐\*\*** 定义 **\*\*public getter 和 setter\*\*** 的DTO成员 .

\* **\*\*推荐\*\*** 使用 **\*\*data annotations\*\*** **\*\*验证\*\*** service输入DTO的属性.

\* **\*\*不推荐\*\*** 在DTO中添加任何 **\*\*逻辑\*\***, 在必要的时候可以实现 `IValidatableObject` 接口.

\* **\*\*推荐\*\*** 为所有的DTO标记 **\*\*[Serializable]\*\*** Attribute. 因为它们已经是可序列化的, 开发人员可能会希望进行二进制序列化.

#### 数据访问

Entity Framework Core 集成

## Entity Framework Core 集成最佳实践

> 有关EF Core 集成的基础知识,请参阅[Entity Framework Core 集成文档](../Entity-Framework-Core.md).

- **\*\*推荐\*\*** 为每个模块定义单独的 `DbContext` 接口和类.

- **\*\*不推荐\*\*** 在应用程序开发中使用延迟加载.

- **\*\*不推荐\*\*** 为 `DbContext` 启用延迟加载.

### DbContext Interface

- **\*\*推荐\*\*** 继承自`IEfCoreDbContext` 的 `DbContext` 定义一个相应的 **\*\*interface\*\***.

- **\*\*推荐\*\*** 添加 `ConnectionStringName` **\*\*attribute\*\*** 到 `DbContext` 接口.

- **\*\*推荐\*\*** 将 `DbSet<TEntity>` **\*\*properties\*\*** 添加到 `DbContext` 接口中,注意: 仅适用于聚合根. 例如:

````C#

[ConnectionStringName("AbpIdentity")]

public interface IIdentityDbContext : IEfCoreDbContext

{

DbSet<IdentityUser> Users { get; set; }

DbSet<IdentityRole> Roles { get; set; }

}

````

### DbContext class

\* **\*\*推荐\*\*** `DbContext` 继承自 `AbpDbContext<TDbContext>` 类.

\* **\*\*推荐\*\*** 添加 `ConnectionStringName` **\*\*attribute\*\*** 到 `DbContext` 类.

\* **\*\*推荐\*\*** 实现 `DbContext` 类实现其相应的接口. 例如:

````C#

[ConnectionStringName("AbpIdentity")]

public class IdentityDbContext : AbpDbContext<IdentityDbContext>, IIdentityDbContext

{

public DbSet<IdentityUser> Users { get; set; }

public DbSet<IdentityRole> Roles { get; set; }

public IdentityDbContext(DbContextOptions<IdentityDbContext> options)

: base(options)

{

}

//code omitted for brevity

}

````

### 表前缀与架构

- **\*\*推荐\*\*** 添加静态 **\*\*properties\*\*** `TablePrefix` 与 `Schema` 到 `DbContext` 类. 使用常量为其设置一个默认值. 例如:

````C#

public static string TablePrefix { get; set; } = AbpIdentityConsts.DefaultDbTablePrefix;

public static string Schema { get; set; } = AbpIdentityConsts.DefaultDbSchema;

````

- **\*\*推荐\*\*** 总是使用简短的 `TablePrefix` 值为模块在共享数据库中创建 **\*\*unique table names\*\***. `Abp` 前缀是为ABP Core模块保留的.

- **\*\*推荐\*\*** `Schema` 默认赋值为 `null`.

### Model Mapping

- **\*\*推荐\*\*** 重写 `DbContext` 的 `OnModelCreating` 方法显式 **\*\*配置所有实体\*\***. 例如:

````C#

protected override void OnModelCreating(ModelBuilder builder)

{

base.OnModelCreating(builder);

builder.ConfigureIdentity(options =>

{

options.TablePrefix = TablePrefix;

options.Schema = Schema;

});

}

````

- **\*\*不推荐\*\*** 直接在 `OnModelCreating` 方法中配置model, 而是为 `ModelBuilder` 定义一个 **\*\*扩展方法\*\***. 使用Configure*\*ModuleName\**作为方法名称. 例如:

````C#

public static class IdentityDbContextModelBuilderExtensions

{

public static void ConfigureIdentity(

[NotNull] this ModelBuilder builder,

Action<IdentityModelBuilderConfigurationOptions> optionsAction = null)

{

Check.NotNull(builder, nameof(builder));

var options = new IdentityModelBuilderConfigurationOptions();

optionsAction?.Invoke(options);

builder.Entity<IdentityUser>(b =>

{

b.ToTable(options.TablePrefix + "Users", options.Schema);

b.ConfigureByConvention();

//code omitted for brevity

});

builder.Entity<IdentityUserClaim>(b =>

{

b.ToTable(options.TablePrefix + "UserClaims", options.Schema);

b.ConfigureByConvention();

//code omitted for brevity

});

//code omitted for brevity

}

}

````

\* **\*\*推荐\*\*** 为每个Enttiy映射调用 `b.ConfigureByConvention();`(如上所示).

\* **\*\*推荐\*\*** 通过继承 `AbpModelBuilderConfigurationOptions` 来创建 **\*\*configuration Options\*\*** 类. 例如:

````C#

public class IdentityModelBuilderConfigurationOptions : AbpModelBuilderConfigurationOptions

{

public IdentityModelBuilderConfigurationOptions()

: base(AbpIdentityConsts.DefaultDbTablePrefix, AbpIdentityConsts.DefaultDbSchema)

{

}

}

````

### 仓储实现

- **\*\*推荐\*\*** 从 `EfCoreRepository<TDbContext,TEntity,TKey>` 类 **\*\*继承\*\*** 仓储并实现相应的仓储接口. 例如:

````C#

public class EfCoreIdentityUserRepository

: EfCoreRepository<IIdentityDbContext, IdentityUser, Guid>, IIdentityUserRepository

{

public EfCoreIdentityUserRepository(

IDbContextProvider<IIdentityDbContext> dbContextProvider)

: base(dbContextProvider)

{

}

}

````

\* **\*\*推荐\*\*** 使用 `DbContext` 接口而不是类来作为泛型参数.

\* **\*\*推荐\*\*** 使用 `GetCancellationToken` 帮助方法将 `cancellationToken` 传递给EF Core. 例如:

````C#

public virtual async Task<IdentityUser> FindByNormalizedUserNameAsync(

string normalizedUserName,

bool includeDetails = true,

CancellationToken cancellationToken = default)

{

return await DbSet

.IncludeDetails(includeDetails)

.FirstOrDefaultAsync(

u => u.NormalizedUserName == normalizedUserName,

GetCancellationToken(cancellationToken)

);

}

````

如果调用者代码中未提供取消令牌,则 `GetCancellationToken` 会从`ICancellationTokenProvider.Token` 获取取消令牌.

- **\*\*推荐\*\*** 为具有 **\*\*子集合\*\*** 的聚合根创建 `IQueryable<TEntity>` 返回类型的 `IncludeDetails` **\*\*扩展方法\*\***. 例如:

````C#

public static IQueryable<IdentityUser> IncludeDetails(

this IQueryable<IdentityUser> queryable,

bool include = true)

{

if (!include)

{

return queryable;

}

return queryable

.Include(x => x.Roles)

.Include(x => x.Logins)

.Include(x => x.Claims)

.Include(x => x.Tokens);

}

````

\* **\*\*推荐\*\*** 推荐在仓储其他方法中使用 `IncludeDetails` 扩展方法, 就像上面的示例代码一样(参阅 FindByNormalizedUserNameAsync).

- **\*\*推荐\*\*** 覆盖具有 **\*\*子集合\*\*** 的聚合根仓储中的 `WithDetails` 方法. 例如:

````C#

public override IQueryable<IdentityUser> WithDetails()

{

return GetQueryable().IncludeDetails(); // Uses the extension method defined above

}

````

### Module Class

- **\*\*推荐\*\*** 为Entity Framework Core集成包定义一个Module类.

- **\*\*推荐\*\*** 使用 `AddAbpDbContext<TDbContext>` 方法将 `DbContext` 添加到 `IServiceCollection`.

- **\*\*推荐\*\*** 将已实现的仓储添加到 `AddAbpDbContext<TDbContext>` 方法的options中. 例如:

````C#

[DependsOn(

typeof(AbpIdentityDomainModule),

typeof(AbpEntityFrameworkCoreModule)

)]

public class AbpIdentityEntityFrameworkCoreModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

context.Services.AddAbpDbContext<IdentityDbContext>(options =>

{

options.AddRepository<IdentityUser, EfCoreIdentityUserRepository>();

options.AddRepository<IdentityRole, EfCoreIdentityRoleRepository>();

});

}

}

````

MongoDB 集成

## MongoDB 集成

\* **\*\*推荐\*\*** 为每个模块定义一个独立的 `MongoDbContext` 接口与实现类.

### MongoDbContext 接口

- **\*\*推荐\*\*** 定义 `MongoDbContext` **\*\*接口\*\*** 时继承自 `IAbpMongoDbContext`.

- **\*\*推荐\*\*** 添加 `ConnectionStringName` **\*\*attribute\*\*** 到 `MongoDbContext` 接口.

- **\*\*推荐\*\*** 只把聚合根做为 `IMongoCollection<TEntity>` **\*\*properties\*\*** 添加到 `MongoDbContext` 接口. 示例:

````C#

[ConnectionStringName("AbpIdentity")]

public interface IAbpIdentityMongoDbContext : IAbpMongoDbContext

{

IMongoCollection<IdentityUser> Users { get; }

IMongoCollection<IdentityRole> Roles { get; }

}

````

### MongoDbContext 类

- **\*\*推荐\*\*** `MongoDbContext` 继承自 `AbpMongoDbContext` 类.

- **\*\*推荐\*\*** 添加 `ConnectionStringName` **\*\*attribute\*\*** 到 `MongoDbContext` 类.

- **\*\*推荐\*\*** `MongoDbContext` 类实现相对应的**\*\*接口\*\***. 示例:

```c#

[ConnectionStringName("AbpIdentity")]

public class AbpIdentityMongoDbContext : AbpMongoDbContext, IAbpIdentityMongoDbContext

{

public IMongoCollection<IdentityUser> Users => Collection<IdentityUser>();

public IMongoCollection<IdentityRole> Roles => Collection<IdentityRole>();

//code omitted for brevity

}

```

### Collection 前缀

- **\*\*推荐\*\*** 添加静态 `CollectionPrefix` **\*\*property\*\*** 到 `DbContext` 类中并使用常量为其设置默认值. 示例:

```c#

public static string CollectionPrefix { get; set; } = AbpIdentityConsts.DefaultDbTablePrefix;

```

在此示例中使用与EF Core集成表前缀相同的常量.

- **\*\*推荐\*\*** 总是使用简短的 `CollectionPrefix` 值为模块在共享数据库中创建 **\*\*unique collection names\*\***. `Abp` collection前缀是为ABP Core模块保留的.

### Collection 映射

- **\*\*推荐\*\*** 通过重写 `MongoDbContext` 的 `CreateModel` 方法 **\*\*配置所有的聚合根\*\*** . 示例:

```c#

protected override void CreateModel(IMongoModelBuilder modelBuilder)

{

base.CreateModel(modelBuilder);

modelBuilder.ConfigureIdentity(options =>

{

options.CollectionPrefix = CollectionPrefix;

});

}

```

- **\*\*不推荐\*\*** 直接在 `CreateModel` 方法中配置model,而是为 `IMongoModelBuilder` 定义一个 **\*\*扩展方法\*\***. 使用Configure*\*ModuleName\**作为方法名称. 示例:

```c#

public static class AbpIdentityMongoDbContextExtensions

{

public static void ConfigureIdentity(

this IMongoModelBuilder builder,

Action<IdentityMongoModelBuilderConfigurationOptions> optionsAction = null)

{

Check.NotNull(builder, nameof(builder));

var options = new IdentityMongoModelBuilderConfigurationOptions();

optionsAction?.Invoke(options);

builder.Entity<IdentityUser>(b =>

{

b.CollectionName = options.CollectionPrefix + "Users";

});

builder.Entity<IdentityRole>(b =>

{

b.CollectionName = options.CollectionPrefix + "Roles";

});

}

}

```

- **\*\*推荐\*\*** 通过继承 `AbpMongoModelBuilderConfigurationOptions` 来创建 **\*\*configuration Options\*\*** 类. 示例:

```c#

public class IdentityMongoModelBuilderConfigurationOptions

: AbpMongoModelBuilderConfigurationOptions

{

public IdentityMongoModelBuilderConfigurationOptions()

: base(AbpIdentityConsts.DefaultDbTablePrefix)

{

}

}

```

\* **\*\*推荐\*\*** 创建一个静态方法, 显示地为所有的实体配置 `BsonClassMap`. 示例:

````C#

public static class AbpIdentityBsonClassMap

{

private static readonly OneTimeRunner OneTimeRunner = new OneTimeRunner();

public static void Configure()

{

OneTimeRunner.Run(() =>

{

BsonClassMap.RegisterClassMap<IdentityUser>(map =>

{

map.AutoMap();

map.ConfigureExtraProperties();

});

BsonClassMap.RegisterClassMap<IdentityRole>(map =>

{

map.AutoMap();

});

});

}

}

````

`BsonClassMap` 适用于静态方法. 所以只需要在应用程序配置一次实体. `OneTimeRunner` 以线程安全的方式运行, 并且在应用程序生命周期中只运行一次. 上面代码中的映射确保单元测试可以正确运行. 此代码将由下面的**\*\*模块类\*\***调用.

### 仓储实现

- **\*\*推荐\*\*** 仓储 **\*\*继承自\*\*** `MongoDbRepository<TMongoDbContext, TEntity, TKey>` 类并且实现其相应的接口. 示例:

```c#

public class MongoIdentityUserRepository

: MongoDbRepository<IAbpIdentityMongoDbContext, IdentityUser, Guid>,

IIdentityUserRepository

{

public MongoIdentityUserRepository(

IMongoDbContextProvider<IAbpIdentityMongoDbContext> dbContextProvider)

: base(dbContextProvider)

{

}

}

```

- **\*\*推荐\*\*** 使用 `GetCancellationToken` 帮助方法将 `cancellationToken` 传递给MongoDB驱动程序. 示例:

```c#

public async Task<IdentityUser> FindByNormalizedUserNameAsync(

string normalizedUserName,

bool includeDetails = true,

CancellationToken cancellationToken = default)

{

return await GetMongoQueryable()

.FirstOrDefaultAsync(

u => u.NormalizedUserName == normalizedUserName,

GetCancellationToken(cancellationToken)

);

}

```

如果调用者代码中未提供取消令牌, 则 `GetCancellationToken` 会从`ICancellationTokenProvider.Token` 获取取消令牌

`GetCancellationToken`.

\* **\*\*推荐\*\*** 忽略仓储实现中的 `includeDetails` 参数, 因为MongoDB在默认情况下将聚合根作为一个整体(包括子集合)加载.

\* **\*\*推荐\*\*** 使用 `GetMongoQueryable()` 方法获取 `IQueryable<TEntity>` 以尽可能执行查询use the `GetMongoQueryable()` method to obtain an `IQueryable<TEntity>` to perform queries wherever possible. 因为;

\* `GetMongoQueryable()` 方法在内部使用 `ApplyDataFilters` 方法根据当前的过滤器 (如 软删除与多租户)过滤数据.

\* 使用`IQueryable<TEntity>`让代码与EF Core仓储实现类似, 易于使用.

\* **\*\*推荐\*\*** 如果无法使用 `GetMongoQueryable()` 方法, 则应自行实现数据过滤.

### 模块类

- **\*\*推荐\*\*** 为MongoDB集成包定义一个模块类.

- **\*\*推荐\*\*** 使用 `AddMongoDbContext<TMongoDbContext>` 方法将 `MongoDbContext` 添加到 `IServiceCollection`.

- **\*\*推荐\*\*** 将已实现的仓储添加到 `AddMongoDbContext<TMongoDbContext>` 方法options中. 示例:

```c#

[DependsOn(

typeof(AbpIdentityDomainModule),

typeof(AbpUsersMongoDbModule)

)]

public class AbpIdentityMongoDbModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

AbpIdentityBsonClassMap.Configure();

context.Services.AddMongoDbContext<AbpIdentityMongoDbContext>(options =>

{

options.AddRepository<IdentityUser, MongoIdentityUserRepository>();

options.AddRepository<IdentityRole, MongoIdentityRoleRepository>();

});

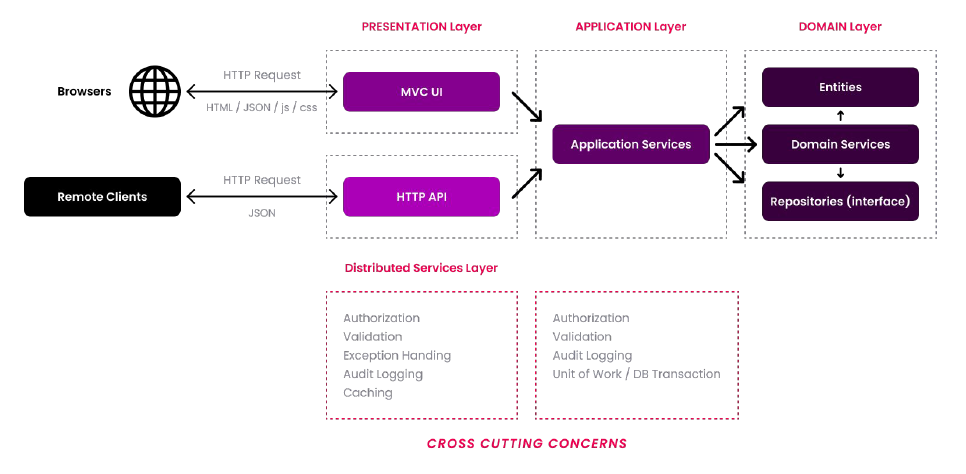
}

}

```

需要注意的是, 模块类还调用上面定义的静态 `BsonClassMap` 配置方法.

## 领域驱动设计



概述



# 领域驱动设计

## 什么是DDD?

ABP框架提供了**\*\*基础设施\*\***使基于**\*\*领域驱动设计\*\***的开发更易实现. DDD在[维基百科中的定义](https://zh.wikipedia.org/wiki/%E5%9F%9F%E9%A9%B1%E5%8A%A8%E5%BC%80%E5%8F%91)如下:

> **\*\*领域驱动设计(DDD)\*\*** 是一种通过将实现连接到持续进化的模型来满足复杂需求的软件开发方法. 领域驱动设计的前提是:

>

> - 把项目的主要重点放在核心领域和领域逻辑上

> - 把复杂的设计放在领域模型上

> - 发起技术专家和领域专家之间的创造性协作,以迭代方式完善解决特定领域问题的概念模型

### 分层

ABP框架遵循DDD原则和模式去实现分层应用程序模型,该模型由四个基本层组成:

- **\*\*表示层\*\***: 为用户提供接口. 使用*\*应用层\**实现与用户交互.

- **\*\*应用层\*\***: 表示层与领域层的中介,编排业务对象执行特定的应用程序任务. 使用应用程序逻辑实现用例.

- **\*\*领域层\*\***: 包含业务对象以及业务规则. 是应用程序的核心.

- **\*\*基础设施层\*\***: 提供通用的技术功能,支持更高的层,主要使用第三方类库.

## 内容

\* **\*\*领域层\*\***

\* [实体与聚合根](Entities.md)

\* 值对象

\* [仓储](Repositories.md)

\* 领域服务

\* 规约

\* **\*\*应用层\*\***

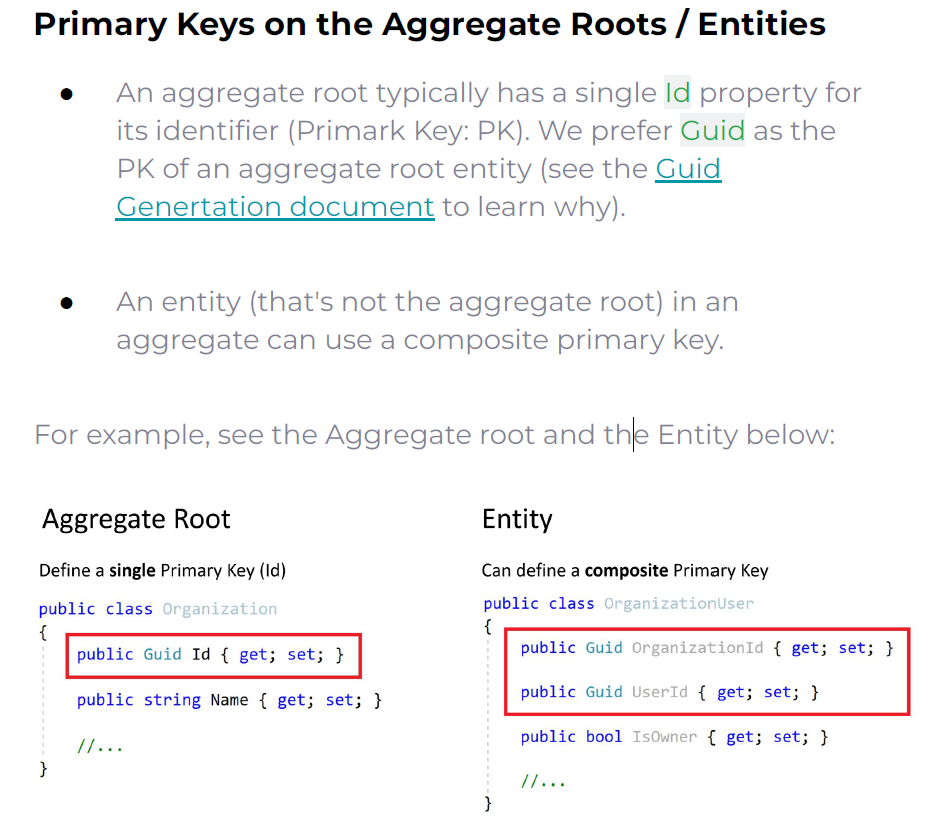
\* [应用服务](Application-Services.md)

\* [数据传输对象(DTO)](Data-Transfer-Objects.md)

\* 工作单元

领域层

#### 实体&聚合根



# 实体

实体是DDD(Domain Driven Design)中核心概念.Eric Evans是这样描述实体的 "一个没有从其属性,而是通过连续性和身份的线索来定义的对象"

实体通常映射到关系型数据库的表中.

## 实体类

实体都继承自`Entity<TKey>`类,如下所示:

```C#

public class Book : Entity<Guid>

{

public string Name { get; set; }

public float Price { get; set; }

}

```

> 如果你不想继承基类`Entity<TKey>`,也可以直接实现`IEntity<TKey>`接口

`Entity<TKey>`类只是用给定的主 **\*\*键类型\*\*** 定义了一个`Id`属性,在上面的示例中是`Guid`类型.可以是其他类型如`string`, `int`, `long`或其他你需要的类型.

### Guid主键的实体

如果你的实体Id类型为 `Guid`,有一些好的实践可以实现:

\* 创建一个构造函数,获取ID作为参数传递给基类.

\* 如果没有为GUID Id赋值,**\*\*ABP框架会在保存时设置它\*\***,但是在将实体保存到数据库之前最好在实体上有一个有效的Id.

\* 如果使用带参数的构造函数创建实体,那么还要创建一个 `private` 或 `protected` 构造函数. 当数据库提供程序从数据库读取你的实体时(反序列化时)将使用它.

\* 不要使用 `Guid.NewGuid()` 来设置Id! 在创建实体的代码中**\*\*使用[**`IGuidGenerator`服务**](Guid-Generation.md)\*\*** 传递Id参数. `IGuidGenerator`经过优化可以产生连续的GUID.这对于关系数据库中的聚集索引非常重要.

示例实体:

````csharp

public class Book : Entity<Guid>

{

public string Name { get; set; }

public float Price { get; set; }

protected Book()

{

}

public Book(Guid id)

: base(id)

{

}

}

````

在[应用服务](Application-Services.md)中使用示例:

````csharp

public class BookAppService : ApplicationService, IBookAppService

{

private readonly IRepository<Book> \_bookRepository;

public BookAppService(IRepository<Book> bookRepository)

{

\_bookRepository = bookRepository;

}

public async Task CreateAsync(CreateBookDto input)

{

await \_bookRepository.InsertAsync(

new Book(GuidGenerator.Create())

{

Name = input.Name,

Price = input.Price

}

);

}

}

````

\* `BookAppService` 注入图书实体的默认[仓储](Repositories.md),使用`InsertAsync`方法插入 `Book` 到数据库中.

\* `GuidGenerator`类型是 `IGuidGenerator`,它是在`ApplicationService`基类中定义的属性. ABP将这样常用属性预注入,所以不需要手动[注入](Dependency-Injection.md).

\* 如果你想遵循DDD最佳实践,请参阅下面的*\*聚合示例\**部分.

### 具有复合键的实体

有些实体可能需要 **\*\*复合键\*\*** .在这种情况下,可以从非泛型`Entity`类派生实体.如:

````C#

public class UserRole : Entity

{

public Guid UserId { get; set; }

public Guid RoleId { get; set; }

public DateTime CreationTime { get; set; }

public UserRole()

{

}

public override object[] GetKeys()

{

return new object[] { UserId, RoleId };

}

}

````

上面的例子中,复合键由`UserId`和`RoleId`组成.在关系数据库中,它是相关表的复合主键. 具有复合键的实体应当实现上面代码中所示的`GetKeys()`方法.

> 你还需要在**\*\*对象关系映射\*\***(ORM)中配置实体的键. 参阅[Entity Framework Core](Entity-Framework-Core.md)集成文档查看示例.

> 需要注意,复合主键实体不可以使用 `IRepository<TEntity, TKey>` 接口,因为它需要一个唯一的Id属性. 但你可以使用 `IRepository<TEntity>`.更多信息请参见[仓储](Repositories.md)的文档.

### EntityEquals

`Entity.EntityEquals(...)` 方法用于检查两个实体对象是否相等.

示例:

```csharp

Book book1 = ...

Book book2 = ...

if (book1.EntityEquals(book2)) //Check equality

{

...

}

```

## 聚合根

"*\*聚合是域驱动设计中的一种模式.DDD的聚合是一组可以作为一个单元处理的域对象.例如,订单及订单系列的商品,这些是独立的对象,但将订单(连同订单系列的商品)视为一个聚合通常是很有用的\**"( [查看详细介绍](http://martinfowler.com/bliki/DDD\_Aggregate.html))

`AggregateRoot<TKey>`类继承自`Entity<TKey>`类,所以默认有`Id`这个属性

> 值得注意的是 ABP 会默认为聚合根创建仓储,当然,ABP也可以为所有的实体创建仓储,详情参见[仓储](Repositories.md).

ABP不强制你使用聚合根,实际上你可以使用上面定义的`Entity`类,当然,如果你想实现[领域驱动设计](Domain-Driven-Design.md)并且创建聚合根,这里有一些最佳实践仅供参考:

\* 聚合根需要维护自身的完整性,所有的实体也是这样.但是聚合根也要维护子实体的完整性.所以,聚合根必须一直有效.

\* 使用Id引用聚合根,而不使用导航属性

\* 聚合根被视为一个单元.它是作为一个单元检索和更新的.它通常被认为是一个交易边界.

\* 不单独修改聚合根中的子实体

如果你想在应用程序中实现DDD,请参阅[实体设计最佳实践指南](Best-Practices/Entities.md).

### 聚合根例子

这是一个具有子实体集合的聚合根例子:

````C#

public class Order : AggregateRoot<Guid>

{

public virtual string ReferenceNo { get; protected set; }

public virtual int TotalItemCount { get; protected set; }

public virtual DateTime CreationTime { get; protected set; }

public virtual List<OrderLine> OrderLines { get; protected set; }

protected Order()

{

}

public Order(Guid id, string referenceNo)

{

Check.NotNull(referenceNo, nameof(referenceNo));

Id = id;

ReferenceNo = referenceNo;

OrderLines = new List<OrderLine>();

}

public void AddProduct(Guid productId, int count)

{

if (count <= 0)

{

throw new ArgumentException(

"You can not add zero or negative count of products!",

nameof(count)

);

}

var existingLine = OrderLines.FirstOrDefault(ol => ol.ProductId == productId);

if (existingLine == null)

{

OrderLines.Add(new OrderLine(this.Id, productId, count));

}

else

{

existingLine.ChangeCount(existingLine.Count + count);

}

TotalItemCount += count;

}

}

public class OrderLine : Entity

{

public virtual Guid OrderId { get; protected set; }

public virtual Guid ProductId { get; protected set; }

public virtual int Count { get; protected set; }

protected OrderLine()

{

}

internal OrderLine(Guid orderId, Guid productId, int count)

{

OrderId = orderId;

ProductId = productId;

Count = count;

}

internal void ChangeCount(int newCount)

{

Count = newCount;

}

public override object[] GetKeys()

{

return new Object[] {OrderId, ProductId};

}

}

````

> 如果你不想你的聚合根继承`AggregateRoot<TKey>`类,你可以直接实现`IAggregateRoot<TKey>`接口

`Order`是一个具有`Guid`类型`Id`属性的 **\*\*聚合根\*\***.它有一个`OrderLine`实体集合.`OrderLine`是一个具有组合键(`OrderId`和 ` ProductId`)的实体.

虽然这个示例可能无法实现聚合根的所有最佳实践,但它仍然遵循良好的实践:

\* `Order`有一个公共的构造函数,它需要 **\*\*minimal requirements\*\*** 来构造一个"订单"实例.因此,在没有`Id`和`referenceNo`的时候是无法创建订单的.**\*\*protected/private\*\***的构造函数只有从数据库读取对象时 **\*\*反序列化\*\*** 才需要.

\* `OrderLine`的构造函数是internal的,所以它只能由领域层来创建.在`Order.AddProduct`这个方法的内部被使用.

\* `Order.AddProduct`实现了业务规则将商品添加到订单中

\* 所有属性都有`protected`的set.这是为了防止实体在实体外部任意改变.因此,在没有向订单中添加新产品的情况下设置 `TotalItemCount`将是危险的.它的值由`AddProduct`方法维护.

ABP框架不强制你应用任何DDD规则或模式.但是,当你准备应用的DDD规则或模式时候,ABP会让这变的可能而且更简单.文档同样遵循这个原则.

### 带有组合键的聚合根

虽然这种聚合根并不常见(也不建议使用),但实际上可以按照与上面提到的跟实体相同的方式定义复合键.在这种情况下,要使用非泛型的`AggregateRoot`基类.

### BasicAggregateRoot类

`AggregateRoot` 类实现了 `IHasExtraProperties` 和 `IHasConcurrencyStamp` 接口,这为派生类带来了两个属性. `IHasExtraProperties` 使实体可扩展(请参见下面的 *\*额外的属性\**部分) 和 `IHasConcurrencyStamp` 添加了由ABP框架管理的 `ConcurrencyStamp` 属性实现[乐观并发](https://docs.microsoft.com/zh-cn/ef/core/saving/concurrency). 在大多数情况下,这些是聚合根需要的功能.

但是,如果你不需要这些功能,你的聚合根可以继承 `BasicAggregateRoot<TKey>`(或`BasicAggregateRoot`).

## 基类和接口的审计属性

有一些属性,像`CreationTime`,`CreatorId`,`LastModificationTime`...在所有应用中都很常见. ABP框架提供了一些接口和基类来**\*\*标准化\*\***这些属性,并**\*\*自动设置它们的值\*\***.

### 审计接口

有很多的审计接口,你可以实现一个你需要的那个.

> 虽然可以手动实现这些接口,但是可以使用下一节中定义的**\*\*基类\*\***简化代码.

\* `IHasCreationTime` 定义了以下属性:

\* `CreationTime`

\* `IMayHaveCreator` 定义了以下属性:

\* `CreatorId`

\* `ICreationAuditedObject` 继承 `IHasCreationTime` 和 `IMayHaveCreator`, 所以它定义了以下属性:

\* `CreationTime`

\* `CreatorId`

\* `IHasModificationTime` 定义了以下属性:

\* `LastModificationTime`

\* `IModificationAuditedObject` 扩展 `IHasModificationTime` 并添加了 `LastModifierId` 属性. 所以它定义了以下属性:

\* `LastModificationTime`

\* `LastModifierId`

\* `IAuditedObject` 扩展 `ICreationAuditedObject` 和 `IModificationAuditedObject`, 所以它定义了以下属性:

\* `CreationTime`

\* `CreatorId`

\* `LastModificationTime`

\* `LastModifierId`

\* `ISoftDelete` (参阅 [数据过滤文档](Data-Filtering.md)) 定义了以下属性:

\* `IsDeleted`

\* `IHasDeletionTime` 扩展 `ISoftDelete` 并添加了 `DeletionTime` 属性. 所以它定义了以下属性:

\* `IsDeleted`

\* `DeletionTime`

\* `IDeletionAuditedObject` 扩展 `IHasDeletionTime` 并添加了 `DeleterId` 属性. 所以它定义了以下属性:

\* `IsDeleted`

\* `DeletionTime`

\* `DeleterId`

\* `IFullAuditedObject` 继承 `IAuditedObject` 和 `IDeletionAuditedObject`, 所以它定义了以下属性:

\* `CreationTime`

\* `CreatorId`

\* `LastModificationTime`

\* `LastModifierId`

\* `IsDeleted`

\* `DeletionTime`

\* `DeleterId`

当你实现了任意接口,或者从下一节定义的类派生,ABP框架就会尽可能地自动管理这些属性.

> 实现 `ISoftDelete` , `IDeletionAuditedObject` 或 `IFullAuditedObject` 让你的实体**\*\*软删除\*\***. 参阅[数据过滤文档](Data-Filtering.md),了解软删除模式.

### 审计基类

虽然可以手动实现以上定义的任何接口,但建议从这里定义的基类继承:

\* `CreationAuditedEntity<TKey>` 和 `CreationAuditedAggregateRoot<TKey>` 实现了 `ICreationAuditedObject` 接口.

\* `AuditedEntity<TKey>` 和 `AuditedAggregateRoot<TKey>` 实现了 `IAuditedObject` 接口.

\* `FullAuditedEntity<TKey>` and `FullAuditedAggregateRoot<TKey>` 实现了 `IFullAuditedObject` 接口.

所有这些基类都有非泛型版本,可以使用 `AuditedEntity` 和 `FullAuditedAggregateRoot` 来支持复合主键;

所有这些基类也有 `... WithUser`,像 `FullAuditedAggregateRootWithUser<TUser>` 和 `FullAuditedAggregateRootWithUser<TKey, TUser>`. 这样就可以将导航属性添加到你的用户实体. 但在聚合根之间添加导航属性不是一个好做法,所以这种用法是不建议的(除非你使用EF Core之类的ORM可以很好地支持这种情况,并且你真的需要它. 请记住这种方法不适用于NoSQL数据库(如MongoDB),你必须真正实现聚合模式）.

## 额外的属性

ABP定义了 `IHasExtraProperties` 接口,可以由实体实现,以便能够动态地设置和获取的实体属性. `AggregateRoot` 基类已经实现了 `IHasExtraProperties` 接口. 如果你从这个类(或者上面定义的一个相关审计类)派生,那么你可以直接使用API​.

### GetProperty 和 SetProperty 扩展方法

这些扩展方法是获取和设置实体数据的推荐方法. 例:

````csharp

public class ExtraPropertiesDemoService : ITransientDependency

{

private readonly IIdentityUserRepository \_identityUserRepository;

public ExtraPropertiesDemoService(IIdentityUserRepository identityUserRepository)

{

\_identityUserRepository = identityUserRepository;

}

public async Task SetTitle(Guid userId, string title)

{

var user = await \_identityUserRepository.GetAsync(userId);

//SET A PROPERTY

user.SetProperty("Title", title);

await \_identityUserRepository.UpdateAsync(user);

}

public async Task<string> GetTitle(Guid userId)

{

var user = await \_identityUserRepository.GetAsync(userId);

//GET A PROPERTY

return user.GetProperty<string>("Title");

}

}

````

\* 属性的**\*\*值是object\*\***,可以是任何类型的对象(string,int,bool...等).

\* 如果给定的属性未设置值, `GetProperty` 方法会返回 `null`.

\* 你可以使用不同的**\*\*属性名称\*\***(如这里的`Title`)同时存储多个属性.

最好为属性名**\*\*定义一个常量\*\***防止拼写错误. 最佳方式是定义**\*\*扩展方法\*\***来利用智能感知. 例:

````csharp

public static class IdentityUserExtensions

{

private const string TitlePropertyName = "Title";

public static void SetTitle(this IdentityUser user, string title)

{

user.SetProperty(TitlePropertyName, title);

}

public static string GetTitle(this IdentityUser user)

{

return user.GetProperty<string>(TitlePropertyName);

}

}

````

然后你可以直接使用 `IdentityUser` 对象的 `user.SetTitle("...")` 和 `user.GetTitle()`.

### HasProperty 和 RemoveProperty 扩展方法

\* `HasProperty` 用于检查对象是否设置了属性.

\* `RemoveProperty` 用于从对象中删除属性. 你可以使用它来替代设置 `null` 值.

### 它是如何实现的?

`IHasExtraProperties` 要求实现类定义一个名称为 `ExtraProperties` 的`Dictionary<string, object>` 属性.

所以,如果你需要你可以直接使用 `ExtraProperties` 属性来使用字典API,但是推荐使用 `SetProperty` 和 `GetProperty` 方法,因为它们会检查 `null` 值.

#### 它是如何存储的?

存储字典的方式取决于你使用的数据库提供程序.

\* 对于 [Entity Framework Core](Entity-Framework-Core.md),这是两种类型的配置;

\* 默认它以 `JSON` 字符串形式存储在 `ExtraProperties` 字段中. 序列化到 `JSON` 和反序列化到 `JSON` 由ABP使用EF Core的[值转换](https://docs.microsoft.com/zh-cn/ef/core/modeling/value-conversions)系统自动完成.

\* 如果需要,你可以使用 `ObjectExtensionManager` 为所需的额外属性定义一个单独的数据库字段. 未使用 `ObjectExtensionManager` 配置的属性继续使用单个 `JSON` 字段. 当你使用预构建的[应用模块](Modules/Index.md)并且想要[扩展模块的实体](Customizing-Application-Modules-Extending-Entities.md). 参阅[EF Core迁移文档](Entity-Framework-Core.md)了解如何使用 `ObjectExtensionManager`.

\* 对于 [MongoDB](MongoDB.md), 它以 **\*\*常规字段\*\*** 存储, 因为 MongoDB 天生支持这种 [额外](https://mongodb.github.io/mongo-csharp-driver/1.11/serialization/#supporting-extra-elements) 系统.

### 讨论额外的属性

如果你使用**\*\*可重复使用的模块\*\***,其中定义了一个实体,你想使用简单的方式get/set此实体相关的一些数据,那么额外的属性系统是非常有用的.

你通常 **\*\*不需要\*\*** 为自己的实体使用这个系统,是因为它有以下缺点:

\* 它不是**\*\*完全类型安全的\*\***,因为它使用字符串用作属性名称.

\* 这些属性**\*\*不容易[**自动映射**](Object-To-Object-Mapping.md)到其他对象\*\***.

\* 它**\*\*不会\*\***为EF Core在数据库表中**\*\*创建字段\*\***,因此在数据库中针对这个字段创建索引或搜索/排序并不容易.

### 额外属性背后的实体

`IHasExtraProperties` 不限于与实体一起使用. 你可以为任何类型的类实现这个接口,使用 `GetProperty`,`SetProperty` 和其他相关方法.

## 另请参阅

\* [实体设计最佳实践指南](Best-Practices/Entities.md)

#### 值对象

## 值对象

> 一个对象,表示领域的描述方面,没有概念上的身份被称为 值对象.

>

> (Eric Evans)

属性相同但`Id`不同的两个[实体](https://docs.abp.io/zh-Hans/abp/latest/Entities) 被视为不同的实体.但是,值对象没有`Id`

## 值对象的类

值对象是一个抽象类,可以继承它来创建值对象类

**\*\*示例: An Address class\*\***

```csharp

public class Address : ValueObject

{

public Guid CityId { get; private set; }

public string Street { get; private set; }

public int Number { get; private set; }

private Address()

{

}

public Address(

Guid cityId,

string street,

int number)

{

CityId = cityId;

Street = street;

Number = number;

}

protected override IEnumerable<object> GetAtomicValues()

{

yield return Street;

yield return CityId;

yield return Number;

}

}

```

- 值对象类必须实现 `GetAtomicValues()`方法来返回原始值

### ValueEquals

`ValueObject.ValueEquals(...)` 用于检测两个值是否相等

**\*\*示例: Check if two addresses are equals\*\***

```csharp

Address address1 = ...

Address address2 = ...

if (address1.ValueEquals(address2)) //Check equality

{

...

}

```

## 最佳实践

以下是使用值对象时的一些最佳实践:

- 如果没有充分的理由将值对象设计为可变的,则将其设计为**\*\*不可变\*\***（如上面的地址）.

- 构成一个值对象的属性应该形成一个概念整体.例如：CityId,Street和Number不应是个人实体的单独属性.这也使Person实体更简单.

## 另请参阅

- [实体](Entities.md)

#### 仓储

# 仓储

"*\*在领域层和数据映射层之间进行中介,使用类似集合的接口来操作领域对象.\**" (Martin Fowler).

实际上,仓储用于领域对象在数据库(参阅[实体](Entities.md))中的操作, 通常每个 **\*\*聚合根\*\*** 或不同的实体创建对应的仓储.

## 通用(泛型)仓储

ABP为每个聚合根或实体提供了 **\*\*默认的通用(泛型)仓储\*\*** . 你可以在服务中[注入](Dependency-Injection.md) `IRepository<TEntity, TKey>` 使用标准的**\*\*CRUD\*\***操作.

> 数据库提供程序层应正确配置为能够使用默认的通用存储库. 如果你已经使用启动模板创建了项目,则这些配置 **\*\*已经完成\*\***了. 如果不是,请参考数据库提供程序文档([EF Core](Entity-Framework-Core.md) / [MongoDB](MongoDB.md))进行配置.

**\*\*默认通用仓储用法示例:\*\***

````C#

using System;

using System.Threading.Tasks;

using Volo.Abp.Application.Services;

using Volo.Abp.Domain.Repositories;

namespace Demo

{

public class PersonAppService : ApplicationService

{

private readonly IRepository<Person, Guid> \_personRepository;

public PersonAppService(IRepository<Person, Guid> personRepository)

{

\_personRepository = personRepository;

}

public async Task CreateAsync(CreatePersonDto input)

{

var person = new Person(input.Name);

await \_personRepository.InsertAsync(person);

}

public async Task<int> GetCountAsync(string filter)

{

return await \_personRepository.CountAsync(p => p.Name.Contains(filter));

}

}

}

````

在这个例子中;

\* `PersonAppService` 在它的构造函数中注入了 `IRepository<Person, Guid>` 。

\* `CreateAsync` 方法使用了 `InsertAsync` 创建并保存新的实体。

\* `GetCountAsync` 方法用来从数据库中获取符合指定条件的的人员的数量。

### 标准仓储方法

通用仓储提供了一些开箱即用的标准CRUD功能:

\* `GetAsync`: 根据指定的`Id`或断言(lambda表达式)返回实体。

\* 将在指定的实体不存在时，抛出异常 `EntityNotFoundException`

\* 如果指定的条件存在多个实体时，抛出异常 `InvalidOperationException`

\* `FindAsync`: 根据指定的`Id`或断言(lambda表达式)返回实体。

\* 如果指定的实体不存在时，返回 `null` 。

\* 如果指定的条件存在多个实体时，抛出异常 `InvalidOperationException`

\* `InsertAsync`: 在数据库里插入一个新的实体。

\* `UpdateAsync`: 在数据库里更新一个已经存在的实体。

\* `DeleteAsync`: 从数据库里删除指定的实体。

\* 这个方法还有一个重载根据指定的断言(lambda表达式)来删除满足条件的多个实体。

\* `GetListAsync`: 返回数据库里的所有实体。

\* `GetPagedListAsync`: 返回一个指定长度的实体列表。 他拥有 `skipCount`, `maxResultCount` and `sorting` 参数.

\* `GetCountAsync`: 获取数据库里所有实体的数量

这些方法还有还一些重载。

\* 提供 `UpdateAsync` 和 `DeleteAsync` 方法根据实体对象或者id来更新或者删除实体。

\* 提供 `DeleteAsync` 方法用来删除符合指定条件的多个实体。

### 在实体中使用LINQ

仓储提供了一个`GetQueryableAsync`方法来获取一个`IQueryable<TEntity>`对象。你可以通过这个对象来对实体执行LINQ查询以操作数据库。

**\*\*示例: 在仓储中使用LINQ表达式\*\***

````csharp

using System;

using System.Linq;

using System.Collections.Generic;

using System.Threading.Tasks;

using Volo.Abp.Application.Services;

using Volo.Abp.Domain.Repositories;

namespace Demo

{

public class PersonAppService : ApplicationService

{

private readonly IRepository<Person, Guid> \_personRepository;

public PersonAppService(IRepository<Person, Guid> personRepository)

{

\_personRepository = personRepository;

}

public async Task<List<PersonDto>> GetListAsync(string filter)

{

// 获取 IQueryable<Person>

IQueryable<Person> queryable = await \_personRepository.GetQueryableAsync();

// 创建一个查询

var query = from person in queryable

where person.Name == filter

orderby person.Name

select person;

// 执行查询

var people = query.ToList();

// 转DTO并返回给客户端

return people.Select(p => new PersonDto {Name = p.Name}).ToList();

}

}

}

````

你也可以使用LINQ扩展方法:

````csharp

public async Task<List<PersonDto>> GetListAsync(string filter)

{

// 获取 IQueryable<Person>

IQueryable<Person> queryable = await \_personRepository.GetQueryableAsync();

// 创建一个查询

var people = queryable

.Where(p => p.Name.Contains(filter))

.OrderBy(p => p.Name)

.ToList();

// 转DTO并返回给客户端

return people.Select(p => new PersonDto {Name = p.Name}).ToList();

}

````

你可以使用仓储返回的`IQueryable` 配合标准LINQ方法自由查询。

> 在这个例子中使用了 `ToList()` 方法, 但是**\*\*强烈建议使用异步方法\*\***来执行数据库查询，比如在这个例子中，可以使用 `ToListAsync()`

>

> 查看 **\*\*IQueryable & 异步操作\*\*** 小节来学习如何做到这一点。

### 批量操作

下面这些方法可以用来对数据库执行批量操作;

\* `InsertManyAsync`

\* `UpdateManyAsync`

\* `DeleteManyAsync`

这些方法可以操作多个实体，如果底层数据库提供程序支持，则可以进行批量操作。

> 使用`UpdateManyAsync`和`DeleteManyAsSync`方法时，乐观锁可能会失效。

### 软 / 硬 删除

如果一个实体是**\*\*软删除\*\***实体（即实现了`ISoftDelete`接口），则仓储的`DeleteSync`方法不会删除该实体，而是在数据库中标记为“已删除”。数据过滤器系统确保不会从数据库中正常检索软删除的实体。

如果您的实体是软删除实体，如果您需要物理删除这个实体，您可以使用`HardDeleteAsync`方法强制删除。

> 阅读 [数据过滤](Data-Filtering.md) 文档以了解更多关于软删除。

### 确保实体存在

`EnsureExistsAsync`扩展方法通过实体id或实体查询表达式来确保实体存在，如果其不存在，它将抛出`EntityNotFoundException`异常。

## 其他通用仓储类型

`IRepository<TEntity, TKey>` 接口扩展了标准 `IQueryable<TEntity>` 你可以使用标准LINQ方法自由查询。这对于大多数应用程序都很好。但是，某些ORM提供程序或数据库系统可能不支持`IQueryable`接口。如果您想使用这样的提供者，就不能依赖`IQueryable`。

### 基础仓储

ABP提供了 `IBasicRepository<TEntity, TPrimaryKey>` 和 `IBasicRepository<TEntity>` 接口来支持这样的场景. 你可以扩展这些接口（并可选择性地从`BasicRepositoryBase`派生）为你的实体创建自定义存储库.

依赖于 `IBasicRepository` 而不是依赖 `IRepository` 有一个优点, 即使它们不支持 `IQueryable` 也可以使用所有的数据源。

但主要的供应商, 像 Entity Framework, NHibernate 或 MongoDb 已经支持了 `IQueryable`.

因此, 使用 `IRepository` 是典型应用程序的 **\*\*建议方法\*\***. 但是可重用的模块开发人员可能会考虑使用 `IBasicRepository` 来支持广泛的数据源.

### 只读仓储

对于想要使用只读仓储的开发者,我们提供了`IReadOnlyRepository<TEntity, TKey>` 与 `IReadOnlyBasicRepository<Tentity, TKey>`接口。

### 无主键的通用(泛型)仓储

如果你的实体没有id主键 (例如, 它可能具有复合主键) 那么你不能使用上面定义的 `IRepository<TEntity, TKey>`, 在这种情况下你可以仅使用实体(类型)注入 `IRepository<TEntity>`。

> `IRepository<TEntity>` 有一些缺失的方法, 通常与实体的 `Id` 属性一起使用. 由于实体在这种情况下没有 `Id` 属性, 因此这些方法不可用. 比如 `Get` 方法通过id获取具有指定id的实体. 不过, 你仍然可以使用`IQueryable<TEntity>`的功能通过标准LINQ方法查询实体。

### 自定义仓储

对于大多数情况, 默认通用仓储就足够了。但是, 你可能会需要为实体创建自定义仓储类。

#### 自定义仓储示例

ABP不会强制你实现任何接口或从存储库的任何基类继承。它可以只是一个简单的POCO类。 但是建议继承现有的仓储接口和类，获得开箱即用的标准方法使你的工作更轻松。

#### 自定义仓储接口

首先在领域层定义一个仓储接口:

```c#

public interface IPersonRepository : IRepository<Person, Guid>

{

Task<Person> FindByNameAsync(string name);

}

```

此接口扩展了 `IRepository<Person, Guid>` 以使用已有的通用仓储功能。

#### 自定义仓储实现

自定义存储库依赖于你使用的数据访问工具。 在此示例中, 我们将使用Entity Framework Core:

````C#

public class PersonRepository : EfCoreRepository<MyDbContext, Person, Guid>, IPersonRepository

{

public PersonRepository(IDbContextProvider<TestAppDbContext> dbContextProvider)

: base(dbContextProvider)

{

}

public async Task<Person> FindByNameAsync(string name)

{

var dbContext = await GetDbContextAsync();

return await dbContext.Set<Person>()

.Where(p => p.Name == name)

.FirstOrDefaultAsync();

}

}

````

你可以直接使用数据库访问提供程序 (本例中是 `dbContext` ) 来执行操作.

> 请参阅[EF Core](Entity-Framework-Core.md)或[MongoDb](MongoDB.md)了解如何自定义仓储.

## IQueryable & 异步操作

`IRepository`提供`GetQueryableAsync()`来获取`IQueryable`，这意味着您可以**\*\*直接在其上使用LINQ扩展方法\*\***，如上面的“\*在实体中使用LINQ”部分所示。

**\*\*示例: 使用** `Where(...)` **和** `ToList()` **扩展方法\*\***

````csharp

var queryable = await \_personRepository.GetQueryableAsync();

var people = queryable

.Where(p => p.Name.Contains(nameFilter))

.ToList();

````

`.ToList`, `Count()`... 是在 `System.Linq` 命名空间下定义的扩展方法. ([参阅所有方法](https://docs.microsoft.com/en-us/dotnet/api/system.linq.queryable)).

你通常想要使用 `.ToListAsync()`, `.CountAsync()`.... 来编写**\*\*真正的异步代码\*\***.

但在你使用标准的[应用程序启动模板](Startup-Templates/Application.md)时会发现无法在应用层或领域层使用这些异步扩展方法,因为:

\* 这里异步方法**\*\*不是标准LINQ方法\*\***,它们定义在[Microsoft.EntityFrameworkCore](https://www.nuget.org/packages/Microsoft.EntityFrameworkCore)Nuget包中.

\* 标准模板应用层与领域层**\*\*不引用\*\***EF Core 包以实现数据库提供程序独立.

根据你的需求和开发模式,你可以根据以下选项使用异步方法.

> 强烈建议使用异步方法! 在执行数据库查询时不要使用同步LINQ方法,以便能够开发可伸缩的应用程序.

### 选项-1: 引用EF Core

**\*\*最简单的方法\*\***是在你想要使用异步方法的项目直接引用EF Core包.

> 添加[Volo.Abp.EntityFrameworkCore](https://www.nuget.org/packages/Volo.Abp.EntityFrameworkCore)NuGet包到你的项目间接引用EF Core包. 这可以确保你的应用程序其余部分兼容正确版本的EF Core.

当你添加NuGet包后,你可以使用全功能的EF Core扩展方法.

**\*\*示例: 直接使用** `ToListAsync()`**\*\***

````csharp

var queryable = await \_personRepository.GetQueryableAsync();

var people = queryable

.Where(p => p.Name.Contains(nameFilter))

.ToListAsync();

````

当以下情况时，这个方法是推荐的：

\* 如果你正在开发一个应用程序并且**\*\*不打算在将来\*\*** 更新FE Core，或者如果以后真的需要更改，你也能**\*\*容忍\*\***它。我们认为，如果您正在开发最终应用程序，这是合理的。

#### MongoDB

如果使用的是MongoDB,则需要将[Volo.Abp.MongoDB] NuGet包添加到项目中. 但在这种情况下你也不能直接使用异步LINQ扩展(例如`ToListAsync`),因为MongoDB不提供 `IQueryable<T>`的异步扩展方法,而是提供 `IMongoQueryable<T>`. 你需要先将查询强制转换为 `IMongoQueryable<T>` 才能使用异步扩展方法.

**\*\*示例: 转换Cast** `IQueryable<T>` **为** `IMongoQueryable<T>` **并且使用** `ToListAsync()`**\*\***

````csharp

var queryable = await \_personRepository.GetQueryableAsync();

var people = ((IMongoQueryable<Person>) queryable

.Where(p => p.Name.Contains(nameFilter)))

.ToListAsync();

````

### 选项-2: 使用IRepository异步扩展方法

ABP框架为仓储提供异步扩展方法，与异步LINQ扩展方法类似。

\*\*示例: 在仓储中使用 `CountAsync` 和 `FirstOrDefaultAsync` 方法 \*\*

````csharp

var countAll = await \_personRepository

.CountAsync();

var count = await \_personRepository

.CountAsync(x => x.Name.StartsWith("A"));

var book1984 = await \_bookRepository

.FirstOrDefaultAsync(x => x.Name == "John");

````

支持这些标准的LINQ方法: *\*AllAsync, AnyAsync, AverageAsync, ContainsAsync, CountAsync, FirstAsync, FirstOrDefaultAsync, LastAsync, LastOrDefaultAsync, LongCountAsync, MaxAsync, MinAsync, SingleAsync, SingleOrDefaultAsync, SumAsync, ToArrayAsync, ToListAsync\**.

这种方法仍有**\*\*局限性\*\***。您需要直接在存储库对象上调用扩展方法。例如，以下用法**\*\*不受支持\*\***：

```csharp

var queryable = await \_bookRepository.GetQueryableAsync();

var count = await queryable.Where(x => x.Name.Contains("A")).CountAsync();

```

这是因为本例中的`CountAsync()`方法是在`IQueryable`接口上调用的，而不是在存储库对象上调用的。请参见此类情况的其他选项。

建议 **\*\*尽可能使用此方法\*\***.

### 选项-3: IAsyncQueryableExecuter

`IAsyncQueryableExecuter` 是一个用于异步执行 `IQueryable<T>` 对象的服务,**\*\*不依赖于实际的数据库提供程序\*\***.

**\*\*示例: 注入并使用** `IAsyncQueryableExecuter.ToListAsync()` **方法\*\***

````csharp

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Volo.Abp.Application.Dtos;

using Volo.Abp.Application.Services;

using Volo.Abp.Domain.Repositories;

using Volo.Abp.Linq;

namespace AbpDemo

{

public class ProductAppService : ApplicationService, IProductAppService

{

private readonly IRepository<Product, Guid> \_productRepository;

private readonly IAsyncQueryableExecuter \_asyncExecuter;

public ProductAppService(

IRepository<Product, Guid> productRepository,

IAsyncQueryableExecuter asyncExecuter)

{

\_productRepository = productRepository;

\_asyncExecuter = asyncExecuter;

}

public async Task<ListResultDto<ProductDto>> GetListAsync(string name)

{

//Obtain the IQueryable<T>

var queryable = await \_productRepository.GetQueryableAsync();

//Create the query

var query = queryable

.Where(p => p.Name.Contains(name))

.OrderBy(p => p.Name);

//Run the query asynchronously

List<Product> products = await \_asyncExecuter.ToListAsync(query);

//...

}

}

}

````

> `ApplicationService` 和 `DomainService` 基类已经预属性注入了 `AsyncExecuter` 属性,所以你可直接使用.

ABP框架使用实际数据库提供程序的API异步执行查询。虽然这不是执行查询的常见方式，但它是使用异步API而不依赖于数据库提供者的最佳方式。

当以下情况时，这个方法是推荐的：

\* 如果您想开发应用程序代码**\*\*而不依赖\*\***数据库提供程序。

\* 如果你正在构建一个没有数据库提供程序集成包的**\*\*可重用库\*\***,但是在某些情况下需要执行 `IQueryable<T>`对象.

例如,ABP框架在 `CrudAppService` 基类中(参阅[应用程序](Application-Services.md)文档)使用 `IAsyncQueryableExecuter`.

### 选项-4: 自定义仓储方法

你始终可以创建自定义仓储方法并使用特定数据库提供程序的API,比如这里的异步扩展方法. 有关自定义存储库的更多信息,请参阅[EF Core](Entity-Framework-Core.md)或[MongoDb](MongoDB.md)文档.

当以下情况时，这个方法是推荐的：

\* 如果你想**\*\*完全隔离\*\***你的领域和应用层和数据库提供程序.

\* 如果你开发可**\*\*重用的[**应用模块**](Modules/Index.md)\*\***,并且不想强制使用特定的数据库提供程序,这应该作为一种[最佳实践](Best-Practices/Index.md).

#### 领域服务

# 领域服务

## 介绍

在 [领域驱动设计](Domain-Driven-Design.md) (DDD) 解决方案中,核心业务逻辑通常在聚合 ([实体](Entities.md)) 和领域服务中实现. 在以下情况下特别需要创建领域服务

\* 你实现了依赖于某些服务（如存储库或其他外部服务）的核心域逻辑.

\* 你需要实现的逻辑与多个聚合/实体相关,因此它不适合任何聚合.

## ABP 领域服务基础设施

领域服务是简单的无状态类. 虽然你不必从任何服务或接口派生,但 ABP 框架提供了一些有用的基类和约定.

### DomainService 和 IDomainService

从 `DomainService` 基类派生领域服务或直接实现 `IDomainService` 接口.

**\*\*示例: 创建从** `DomainService` **基类派生的领域服务.\*\***

````csharp

using Volo.Abp.Domain.Services;

namespace MyProject.Issues

{

public class IssueManager : DomainService

{

}

}

````

当你这样做时:

\* ABP 框架自动将类注册为瞬态生命周期到依赖注入系统.

\* 你可以直接使用一些常用服务作为基础属性,而无需手动注入 (例如 [ILogger](Logging.md) and [IGuidGenerator](Guid-Generation.md)).

> 建议使用 `Manager` 或 `Service` 后缀命名领域服务. 我们通常使用如上面示例中的 `Manager` 后缀.

**\*\*示例: 实现将问题分配给用户的领域逻辑\*\***

````csharp

public class IssueManager : DomainService

{

private readonly IRepository<Issue, Guid> \_issueRepository;

public IssueManager(IRepository<Issue, Guid> issueRepository)

{

\_issueRepository = issueRepository;

}

public async Task AssignAsync(Issue issue, AppUser user)

{

var currentIssueCount = await \_issueRepository

.CountAsync(i => i.AssignedUserId == user.Id);

//Implementing a core business validation

if (currentIssueCount >= 3)

{

throw new IssueAssignmentException(user.UserName);

}

issue.AssignedUserId = user.Id;

}

}

````

问题是定义如下所示的 [聚合根](Entities.md):

````csharp

public class Issue : AggregateRoot<Guid>

{

public Guid? AssignedUserId { get; internal set; }

//...

}

````

\* 使用 `internal` 的 set 确保外层调用者不能直接在调用 set ,并强制始终使用 `IssueManager` 为 `User` 分配 `Issue`.

### 使用领域服务

领域服务通常用于 [应用程序服务](Application-Services.md).

**\*\*示例: 使用** `IssueManager` **将问题分配给用户\*\***

````csharp

using System;

using System.Threading.Tasks;

using MyProject.Users;

using Volo.Abp.Application.Services;

using Volo.Abp.Domain.Repositories;

namespace MyProject.Issues

{

public class IssueAppService : ApplicationService, IIssueAppService

{

private readonly IssueManager \_issueManager;

private readonly IRepository<AppUser, Guid> \_userRepository;

private readonly IRepository<Issue, Guid> \_issueRepository;

public IssueAppService(

IssueManager issueManager,

IRepository<AppUser, Guid> userRepository,

IRepository<Issue, Guid> issueRepository)

{

\_issueManager = issueManager;

\_userRepository = userRepository;

\_issueRepository = issueRepository;

}

public async Task AssignAsync(Guid id, Guid userId)

{

var issue = await \_issueRepository.GetAsync(id);

var user = await \_userRepository.GetAsync(userId);

await \_issueManager.AssignAsync(issue, user);

await \_issueRepository.UpdateAsync(issue);

}

}

}

````

由于 `IssueAppService` 在应用层, 它不能直接将问题分配给用户.因此,它使用 `IssueManager`.

## 应用程序服务与领域服务

虽然 [应用服务](Application-Services.md) 和领域服务都实现了业务规则,但存在根本的逻辑和形式差异:

\* 应用程序服务实现应用程序的 **\*\*用例\*\*** (典型 Web 应用程序中的用户交互), 而领域服务实现 **\*\*核心的、用例独立的领域逻辑\*\***.

\* 应用程序服务获取/返回 [数据传输对象](Data-Transfer-Objects.md), 领域服务方法通常获取和返回 **\*\*领域对象\*\*** ([实体](Entities.md), [值对象](Value-Objects.md)).

\* 领域服务通常由应用程序服务或其他领域服务使用,而应用程序服务由表示层或客户端应用程序使用.

## 生命周期

领域服务的生命周期是 [瞬态](https://docs.abp.io/en/abp/latest/Dependency-Injection) 的,它们会自动注册到依赖注入服务.

#### 规约

## 规约

规约模式用于为实体和其他业务对象定义 **\*\*命名、可复用、可组合和可测试的过滤器\*\*** .

> 规约是领域层的一部分.

## 安装

> 这个包 **\*\*已经安装\*\*** 在启动模板中.所以,大多数时候你不需要手动去安装.

添加 [Volo.Abp.Specifications](https://abp.io/package-detail/Volo.Abp.Specifications) 包到你的项目. 如果当前文件夹是你的项目的根目录(`.csproj`)时,你可以在命令行终端中使用 [ABP CLI](CLI.md) *\*add package\** 命令:

````bash

abp add-package Volo.Abp.Specifications

````

## 定义规约

假设你定义了如下的顾客实体:

````csharp

using System;

using Volo.Abp.Domain.Entities;

namespace MyProject

{

public class Customer : AggregateRoot<Guid>

{

public string Name { get; set; }

public byte Age { get; set; }

public long Balance { get; set; }

public string Location { get; set; }

}

}

````

你可以创建一个由 `Specification<Customer>` 派生的新规约类.

**\*\*例如:规定选择一个18岁以上的顾客\*\***

````csharp

using System;

using System.Linq.Expressions;

using Volo.Abp.Specifications;

namespace MyProject

{

public class Age18PlusCustomerSpecification : Specification<Customer>

{

public override Expression<Func<Customer, bool>> ToExpression()

{

return c => c.Age >= 18;

}

}

}

````

你只需通过定义一个lambda[表达式](https://docs.microsoft.com/zh-cn/dotnet/csharp/language-reference/operators/lambda-expressions)来定义规约.

> 你也可以直接实现`ISpecification<T>`接口,但是基类`Specification<T>`做了大量简化.

## 使用规约

这里有两种常见的规约用例.

### IsSatisfiedBy

`IsSatisfiedBy` 方法可以用于检查单个对象是否满足规约.

**\*\*例如:如果顾客不满足年龄规定,则抛出异常\*\***

````csharp

using System;

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

namespace MyProject

{

public class CustomerService : ITransientDependency

{

public async Task BuyAlcohol(Customer customer)

{

if (!new Age18PlusCustomerSpecification().IsSatisfiedBy(customer))

{

throw new Exception(

"这位顾客不满足年龄规定!"

);

}

//TODO...

}

}

}

````

### ToExpression & Repositories

`ToExpression()` 方法可用于将规约转化为表达式.通过这种方式,你可以使用规约在**\*\*数据库查询时过滤实体\*\***.

````csharp

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Domain.Repositories;

using Volo.Abp.Domain.Services;

namespace MyProject

{

public class CustomerManager : DomainService, ITransientDependency

{

private readonly IRepository<Customer, Guid> \_customerRepository;

public CustomerManager(IRepository<Customer, Guid> customerRepository)

{

\_customerRepository = customerRepository;

}

public async Task<List<Customer>> GetCustomersCanBuyAlcohol()

{

var queryable = await \_customerRepository.GetQueryableAsync();

var query = queryable.Where(

new Age18PlusCustomerSpecification().ToExpression()

);

return await AsyncExecuter.ToListAsync(query);

}

}

}

````

> 规约被正确地转换为SQL/数据库查询语句,并且在DBMS端高效执行.虽然它与规约无关,但如果你想了解有关 `AsyncExecuter` 的更多信息,请参阅[仓储](Repositories.md)文档.

实际上,没有必要使用 `ToExpression()` 方法,因为规约会自动转换为表达式.这也会起作用:

````csharp

var queryable = await \_customerRepository.GetQueryableAsync();

var query = queryable.Where(

new Age18PlusCustomerSpecification()

);

````

## 编写规约

规约有一个强大的功能是,它们可以与`And`、`Or`、`Not`以及`AndNot`扩展方法组合使用.

假设你有另一个规约,定义如下:

```csharp

using System;

using System.Linq.Expressions;

using Volo.Abp.Specifications;

namespace MyProject

{

public class PremiumCustomerSpecification : Specification<Customer>

{

public override Expression<Func<Customer, bool>> ToExpression()

{

return (customer) => (customer.Balance >= 100000);

}

}

}

```

你可以将 `PremiumCustomerSpecification` 和 `Age18PlusCustomerSpecification` 结合起来,查询优质成人顾客的数量,如下所示:

````csharp

using System;

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Domain.Repositories;

using Volo.Abp.Domain.Services;

using Volo.Abp.Specifications;

namespace MyProject

{

public class CustomerManager : DomainService, ITransientDependency

{

private readonly IRepository<Customer, Guid> \_customerRepository;

public CustomerManager(IRepository<Customer, Guid> customerRepository)

{

\_customerRepository = customerRepository;

}

public async Task<int> GetAdultPremiumCustomerCountAsync()

{

return await \_customerRepository.CountAsync(

new Age18PlusCustomerSpecification()

.And(new PremiumCustomerSpecification()).ToExpression()

);

}

}

}

````

如果你想让这个组合成为一个可复用的规约,你可以创建这样一个组合的规约类,它派生自`AndSpecification`:

````csharp

using Volo.Abp.Specifications;

namespace MyProject

{

public class AdultPremiumCustomerSpecification : AndSpecification<Customer>

{

public AdultPremiumCustomerSpecification()

: base(new Age18PlusCustomerSpecification(),

new PremiumCustomerSpecification())

{

}

}

}

````

现在,你就可以向下面一样重新编写 `GetAdultPremiumCustomerCountAsync` 方法:

````csharp

public async Task<int> GetAdultPremiumCustomerCountAsync()

{

return await \_customerRepository.CountAsync(

new AdultPremiumCustomerSpecification()

);

}

````

> 你可以从这些例子中看到规约的强大之处.如果你之后想要更改 `PremiumCustomerSpecification` ,比如将余额从 `100.000` 修改为 `200.000` ,所有查询语句和合并的规约都将受到本次更改的影响.这是减少代码重复的好方法！

## 讨论

虽然规约模式通常与C#的lambda表达式相比较,算是一种更老的方式.一些开发人员可能认为不再需要它,我们可以直接将表达式传入到仓储或领域服务中,如下所示:

````csharp

var count = await \_customerRepository.CountAsync(c => c.Balance > 100000 && c.Age => 18);

````

自从ABP的[仓储](Repositories.md)支持表达式,这是一个完全有效的用法.你不必在应用程序中定义或使用任何规约,可以直接使用表达式.

所以,规约的意义是什么?为什么或者应该在什么时候考虑去使用它?

### 何时使用?

使用规约的一些好处:

- **\*\*可复用\*\***:假设你在代码库的许多地方都需要用到优质顾客过滤器.如果使用表达式而不创建规约,那么如果以后更改“优质顾客”的定义会发生什么?假设你想将最低余额从100000美元更改为250000美元,并添加另一个条件,成为顾客超过3年.如果使用了规约,只需修改一个类.如果在任何其他地方重复（复制/粘贴）相同的表达式,则需要更改所有的表达式.

- **\*\*可组合\*\***:可以组合多个规约来创建新规约.这是另一种可复用性.

- **\*\*命名\*\***:`PremiumCustomerSpecification` 更好地解释了为什么使用规约,而不是复杂的表达式.因此,如果在你的业务中使用了一个有意义的表达式,请考虑使用规约.

- **\*\*可测试\*\***:规约是一个单独（且易于）测试的对象.

### 什么时侯不要使用?

- **\*\*没有业务含义的表达式\*\***:不要对与业务无关的表达式和操作使用规约.

- **\*\*报表\*\***:如果只是创建报表,不要创建规约,而是直接使用 `IQueryable` 和LINQ表达式.你甚至可以使用普通SQL、视图或其他工具生成报表.DDD不关心报表,因此从性能角度来看,查询底层数据存储的方式可能很重要.

应用服务层

#### 应用服务

## 应用服务

应用服务实现应用程序的**\*\*用例\*\***, 将**\*\*领域层逻辑公开给表示层\*\***.

从表示层(可选)调用应用服务,**\*\*DTO ([**数据传对象**](Data-Transfer-Objects.md))\*\*** 作为参数. 返回(可选)DTO给表示层.

## 示例

### 图书实体

假设你有一个`Book`实体(聚合根), 如下所示:

````csharp

public class Book : AggregateRoot<Guid>

{

public const int MaxNameLength = 128;

public virtual string Name { get; protected set; }

public virtual BookType Type { get; set; }

public virtual float? Price { get; set; }

protected Book()

{

}

public Book(Guid id, [NotNull] string name, BookType type, float? price = 0)

{

Id = id;

Name = CheckName(name);

Type = type;

Price = price;

}

public virtual void ChangeName([NotNull] string name)

{

Name = CheckName(name);

}

private static string CheckName(string name)

{

if (string.IsNullOrWhiteSpace(name))

{

throw new ArgumentException($"name can not be empty or white space!");

}

if (name.Length > MaxNameLength)

{

throw new ArgumentException($"name can not be longer than {MaxNameLength} chars!");

}

return name;

}

}

````

\* `Book`实体中定义`MaxNameLength`限制`Name`属性的最大长度.

\* `Book`构造函数与`ChangeName`确保`Name`属性值的有效性. 请注意, `Name`的setter不是`public`.

ABP不会强制开发者这样设计实体, 可以将所有的属性设置Public set/get. 由你来决定是否全面实施DDD.

### IBookAppService接口

在ABP中应用程序服务应该实现`IApplicationService接口`. 推荐每个应用程序服务创建一个接口:

````csharp

public interface IBookAppService : IApplicationService

{

Task CreateAsync(CreateBookDto input);

}

````

我们将实现Create方法作为示例. CreateBookDto定义如下:

```csharp

public class CreateBookDto

{

[Required]

[StringLength(Book.MaxNameLength)]

public string Name { get; set; }

public BookType Type { get; set; }

public float? Price { get; set; }

}

```

有关DTO更多的教程,请参见[数据传输对象文档](Entities.md)

### BookAppService(实现)

````csharp

public class BookAppService : ApplicationService, IBookAppService

{

private readonly IRepository<Book, Guid> \_bookRepository;

public BookAppService(IRepository<Book, Guid> bookRepository)

{

\_bookRepository = bookRepository;

}

public async Task CreateAsync(CreateBookDto input)

{

var book = new Book(

GuidGenerator.Create(),

input.Name,

input.Type,

input.Price

);

await \_bookRepository.InsertAsync(book);

}

}

````

\* `BookAppService`继承了基类`ApplicationService`· 这不是必需的, 但是`ApplicationService`提供了应用服务常见的需求(比如本示例服务中使用的`GuidGenerator`). 如果不继承它, 我们需要在服务中手动注入`IGuidGenerator`(参见[Guid生成](Guid-Generation.md)文档)

\* `BookAppService`按照预期实现了`IBookAppService`

\* `BookAppService` 注入了 `IRepository<Book, Guid>`(请参见[仓储](Repositories.md))在CreateAsync方法内部使用仓储将新实体插入数据库.

\* `CreateAsync`使用`Book`实体的构造函数从给定的Input值创建新的`Book`对象

### 数据传输对象

应用服务使用并返回DTO而不是实体. ABP不会强制执行此规则. 但是将实体暴露给表示层(或远程客户端)存在重大问题, 所以不建议返回实体.

有关更多信息, 请参见[DTO文档](Entities.md).

### 对象到对象映射

`CreateBook`方法使用参数`CreateBookDto`对象手动创建`Book`实体. 因为`Book`实体的构造函数强制执行(我们是这样设计的).

但是在很多情况下使用**\*\*自动对象映射\*\***从相似对象设置对象的属性更加方便实用. ABP提供了一个[对象到对象映射](Object-To-Object-Mapping.md)基础设施,使其变得更加容易.

让我们创建另一种获取`Book`的方法. 首先,在`IBookAppService`接口中定义方法:

````csharp

public interface IBookAppService : IApplicationService

{

Task CreateAsync(CreateBookDto input);

Task<BookDto> GetAsync(Guid id); //New method

}

````

`BookDto`是一个简单的[DTO](Data-Transfer-Objects.md)类, 定义如下:

````csharp

public class BookDto

{

public Guid Id { get; set; }

public string Name { get; set; }

public BookType Type { get; set; }

public float? Price { get; set; }

}

````

我们创建一个Automapper的[Profile](https://docs.automapper.org/en/stable/Configuration.html#profile-instances)类. 例如:

```csharp

public class MyProfile : Profile

{

public MyProfile()

{

CreateMap<Book, BookDto>();

}

}

```

然后使用`AbpAutoMapperOptions`注册配置文件:

````csharp

[DependsOn(typeof(AbpAutoMapperModule))]

public class MyModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpAutoMapperOptions>(options =>

{

//Add all mappings defined in the assembly of the MyModule class

options.AddMaps<MyModule>();

});

}

}

````

`AddMaps` 注册给定类的程序集中所有的配置类,通常使用模块类. 它还会注册 [attribute 映射](https://docs.automapper.org/en/stable/Attribute-mapping.html). 更多信息请参考[对象到对象映射](Object-To-Object-Mapping.md)文档

然后你可以实现`GetAsync`方法. 如下所示:

````csharp

public async Task<BookDto> GetAsync(Guid id)

{

var book = await \_bookRepository.GetAsync(id);

return book.MapTo<BookDto>();

}

````

`MapTo`扩展方法通过复制具有相同命名的所有属性将`Book`对象转换为`BookDto`对象.

`MapTo`的另一种替代方法是使用`IObjectMapper`服务:

````csharp

public async Task<BookDto> GetAsync(Guid id)

{

var book = await \_bookRepository.GetAsync(id);

return ObjectMapper.Map<Book, BookDto>(book);

}

````

虽然第二种语法编写起来有点困难,但是如果你编写单元测试,它会更好地工作.

有关更多信息,请参阅[对象到对象映射](Object-To-Object-Mapping)文档.

### 验证

自动验证应用服务方法的输入(如ASP.NET Core 控制器的actions). 你可以使用标准数据注释属性或自定义验证方法来执行验证. ABP还确保输入不为空.

请参阅[验证](Validation.md)文档了解更多信息.

### 授权

可以对应用程序服务方法使用声明性和命令式授权.

请参阅[授权](Authorization.md)文档了解更多信息.

### CRUD应用服务

如果需要创建具有Create,Update,Delete和Get方法的简单**\*\*CRUD应用服务\*\***,则可以使用ABP的基类轻松构建服务. 你可以继承CrudAppService.

示例:

创建继承`ICrudAppService`接口的`IBookAppService`接口.

````csharp

public interface IBookAppService :

ICrudAppService< //Defines CRUD methods

BookDto, //Used to show books

Guid, //Primary key of the book entity

PagedAndSortedResultRequestDto, //Used for paging/sorting on getting a list of books

CreateUpdateBookDto, //Used to create a new book

CreateUpdateBookDto> //Used to update a book

{

}

````

`ICrudAppService` 有泛型参数来获取实体的主键类型和CRUD操作的DTO类型(它不获取实体类型,因为实体类型未向客户端公开使用此接口).

> 为应用程序服务创建一个接口是最佳做法,但是ABP框架并不强制你这么做,你可以跳过接口部分.

`ICrudAppService`声明以下方法:

````csharp

public interface ICrudAppService<

TEntityDto,

in TKey,

in TGetListInput,

in TCreateInput,

in TUpdateInput>

: IApplicationService

where TEntityDto : IEntityDto<TKey>

{

Task<TEntityDto> GetAsync(TKey id);

Task<PagedResultDto<TEntityDto>> GetListAsync(TGetListInput input);

Task<TEntityDto> CreateAsync(TCreateInput input);

Task<TEntityDto> UpdateAsync(TKey id, TUpdateInput input);

Task DeleteAsync(TKey id);

}

````

示例中使用的DTO类是`BookDto`和`CreateUpdateBookDto`:

````csharp

public class BookDto : AuditedEntityDto<Guid>

{

public string Name { get; set; }

public BookType Type { get; set; }

public float Price { get; set; }

}

public class CreateUpdateBookDto

{

[Required]

[StringLength(128)]

public string Name { get; set; }

[Required]

public BookType Type { get; set; } = BookType.Undefined;

[Required]

public float Price { get; set; }

}

````

DTO类的[Profile](https://docs.automapper.org/en/stable/Configuration.html#profile-instances)类.

```csharp

public class MyProfile : Profile

{

public MyProfile()

{

CreateMap<Book, BookDto>();

CreateMap<CreateUpdateBookDto, Book>();

}

}

```

\* `CreateUpdateBookDto`由创建和更新操作共享,但你也可以使用单独的DTO类.

最后`BookAppService`实现非常简单:

````csharp

public class BookAppService :

CrudAppService<Book, BookDto, Guid, PagedAndSortedResultRequestDto,

CreateUpdateBookDto, CreateUpdateBookDto>,

IBookAppService

{

public BookAppService(IRepository<Book, Guid> repository)

: base(repository)

{

}

}

````

`CrudAppService`实现了`ICrudAppService`接口中声明的所有方法. 然后,你可以添加自己的自定义方法或重写和自定义实现.

> `CrudAppService` 有不同数量泛型参数的版本,你可以选择适合的使用.

### AbstractKeyCrudAppService

`CrudAppService` 要求你的实体拥有一个Id属性做为主键. 如果你使用的是复合主键,那么你无法使用它.

`AbstractKeyCrudAppService` 实现了相同的 `ICrudAppService` 接口,但它没有假设你的主键.

#### 示例

假设你有实体 `District`,它的`CityId` 和 `Name` 做为复合主键,使用 `AbstractKeyCrudAppService` 时需要你自己实现 `DeleteByIdAsync` 和 `GetEntityByIdAsync` 方法:

````csharp

public class DistrictAppService

: AbstractKeyCrudAppService<District, DistrictDto, DistrictKey>

{

public DistrictAppService(IRepository<District> repository)

: base(repository)

{

}

protected async override Task DeleteByIdAsync(DistrictKey id)

{

await Repository.DeleteAsync(d => d.CityId == id.CityId && d.Name == id.Name);

}

protected async override Task<District> GetEntityByIdAsync(DistrictKey id)

{

return await AsyncQueryableExecuter.FirstOrDefaultAsync(

Repository.Where(d => d.CityId == id.CityId && d.Name == id.Name)

);

}

}

````

这个实现需要你创建一个类做为复合键:

````csharp

public class DistrictKey

{

public Guid CityId { get; set; }

public string Name { get; set; }

}

````

### 生命周期

应用服务的生命周期是[transient](Dependency-Injection)的,它们会自动注册到依赖注入系统.

#### 数据传输对象(DTO)

# 数据传输对象

## 介绍

**\*\*数据传输对象\*\***(DTO)用于在**\*\*应用层\*\***和**\*\*表示层\*\***或其他类型的客户端之间传输数据.

通常用**\*\*DTO\*\***作为参数在表示层(可选)调用[应用服务](Application-Services.md). 它使用领域对象执行某些**\*\*特定的业务逻辑\*\***,并(可选)将DTO返回到表示层.因此表示层与领域层完全**\*\*隔离\*\***.

### DTO的需求

> 如果你感觉你已经知道并确认使用DTO的好处,你可以**\*\*跳过这一节\*\***.

首先为每个应用程序服务方法创建DTO类可能被看作是一项冗长而耗时的工作. 但是如果正确使用它们,它们可以保存在应用程序. 为什么和如何>

#### 领域层的抽象

DTO提供了一种从表示层**\*\*抽象领域对象\*\***的有效方法. 实际上你的**\*\*层\*\***被正确地分开了. 如果希望完全更改表示层,可以继续使用现有的应用程序层和领域层. 或者你可以重写领域层完全更改数据库架构,实体和O/RM框架,而无需更改表示层. 当然前提是应用程序服务的契约(方法签名和dto)保持不变.

#### 数据隐藏

假设你有一个具有属性Id,名称,电子邮件地址和密码的 `User` 实体. 如果 `UserAppService` 的 `GetAllUsers()` 方法返回 `List<User>`,任何人都可以访问你所有用户的密码,即使你没有在屏幕上显示它. 这不仅关乎安全,还关乎数据隐藏. 应用程序服务应该只返回表示层(或客户端)所需要的内容,不多也不少.

#### 序列化和延迟加载问题

当你将数据(一个对象)返回到表示层时,它很可能是序列化的. 例如在返回JSON的REST API中,你的对象将被序列化为JSON并发送给客户端. 在这方面将实体返回到表示层可能会有问题,尤其是在使用关系数据库和像Entity Framework Core这样的ORM提供者时.

在真实的应用程序中你的实体可以相互引用. `User` 实体可以引用它的角色. 如果你想序列化用户,它的角色也必须是序列化的. `Role` 类可以有 `List <Permission>`,而 `Permission` 类可以有一个对 `PermissionGroup` 类的引用,依此类推...想象一下所有这些对象都被立即序列化了. 你可能会意外地序列化整个数据库! 同样,如果你的对象具有循环引用,则它们可能根本**\*\*不会\*\***序列化成功.

有什么解决方案? 将属性标记为 `NonSerialized` 吗? 不,你不知道什么时候应该序列化什么时候应该序列化. 一个应用程序服务方法可能需要它,而另一个则不需要. 在这种情况下返回安全,可序列化且经过特殊设计的DTO是一个不错的选择.

几乎所有的O/RM框架都支持延迟加载. 此功能可在需要时从数据库加载实体. 假设 `User` 类具有对 `Role` 类的引用. 当你从数据库中获取用户时,`Role` 属性(或集合)不会被立即填充. 首次读取 `Role` 属性时,它是从数据库加载的. 因此如果将这样的实体返回到表示层,它将通过执行额外的查询从数据库中检索额外的实体. 如果序列化工具读取实体,它会递归读取所有属性,并且可以再次检索整个数据库(如果实体之间存在关系).

如果在表示层中使用实体,可能会出现更多问题.**\*\*最好不要在表示层中引用领域/业务层程序集\*\***.

如果你确定使用DTO,我们可以继续讨论ABP框架提供的关于dto的建议.

> ABP并不强迫你使用DTO,但是**\*\*强烈建议将DTO作为最佳实践\*\***.

## 标准接口和基类

DTO是一个没有依赖性的简单类,你可以用任何方式进行设计. 但是ABP引入了一些**\*\*接口\*\***来确定命名**\*\*标准属性\*\***和**\*\*基类\*\***的**\*\*约定\*\***,以免在声明**\*\*公共属性\*\***时**\*\*重复工作\*\***.

**\*\*它们都不是必需的\*\***,但是使用它们可以**\*\*简化和标准化\*\***应用程序代码.

### 实体相关DTO

通常你需要创建与你的实体相对应的DTO,从而生成与实体类似的类. ABP框架在创建DTO时提供了一些基类来简化.

#### EntityDto

`IEntityDto<TKey>` 是一个只定义 `Id` 属性的简单接口. 你可以实现它或从 `EntityDto<TKey>` 继承.

**\*\*Example:\*\***

````csharp

using System;

using Volo.Abp.Application.Dtos;

namespace AbpDemo

{

public class ProductDto : EntityDto<Guid>

{

public string Name { get; set; }

//...

}

}

````

#### 审计DTO

如果你的实体继承自被审计的实体类(或实现审计接口)可以使用以下基类来创建DTO:

\* `CreationAuditedEntityDto`

\* `CreationAuditedEntityWithUserDto`

\* `AuditedEntityDto`

\* `AuditedEntityWithUserDto`

\* `FullAuditedEntityDto`

\* `FullAuditedEntityWithUserDto`

#### 可扩展的DTO

如果你想为你的DTO使用[对象扩展系统](Object-Extensions.md),你可以使用或继承以下DTO类:

\* `ExtensibleObject` 实现 `IHasExtraProperties` (其它类继承这个类).

\* `ExtensibleEntityDto`

\* `ExtensibleCreationAuditedEntityDto`

\* `ExtensibleCreationAuditedEntityWithUserDto`

\* `ExtensibleAuditedEntityDto`

\* `ExtensibleAuditedEntityWithUserDto`

\* `ExtensibleFullAuditedEntityDto`

\* `ExtensibleFullAuditedEntityWithUserDto`

### 列表结果

通常将DTO列表返回给客户端. `IListResult<T>` 接口和 `ListResultDto<T>` 类用于使其成为标准.

`IListResult<T>` 接口的定义:

````csharp

public interface IListResult<T>

{

IReadOnlyList<T> Items { get; set; }

}

````

**\*\*示例: 返回产品列表\*\***

````csharp

using System;

using System.Collections.Generic;

using System.Threading.Tasks;

using Volo.Abp.Application.Dtos;

using Volo.Abp.Application.Services;

using Volo.Abp.Domain.Repositories;

namespace AbpDemo

{

public class ProductAppService : ApplicationService, IProductAppService

{

private readonly IRepository<Product, Guid> \_productRepository;

public ProductAppService(IRepository<Product, Guid> productRepository)

{

\_productRepository = productRepository;

}

public async Task<ListResultDto<ProductDto>> GetListAsync()

{

//Get entities from the repository

List<Product> products = await \_productRepository.GetListAsync();

//Map entities to DTOs

List<ProductDto> productDtos =

ObjectMapper.Map<List<Product>, List<ProductDto>>(products);

//Return the result

return new ListResultDto<ProductDto>(productDtos);

}

}

}

````

你可以简单地返回 `productDtos` 对象(并更改方法的返回类型), 这也没有错. 返回一个 `ListResultDto` 会使`List<ProductDto>` 做为 `Item` 属性包装到另一个对象中. 这具有一个优点:以后可以在不破坏远程客户端的情况下(当它们作为JSON结果获得值时)在返回值中添加更多属性. 在开发可重用的应用程序模块时特别建议使用这种方式.

### 分页 & 排序列表结果

从服务器请求分页列表并将分页列表返回给客户端是更常见的情况. ABP定义了一些接口和类来对其进行标准化:

#### 输入 (请求) 类型

以下接口和类用于标准化客户端发送的输入.

\* `ILimitedResultRequest`: 定义 `MaxResultCount`(`int`) 属性从服务器请求指定数量的结果.

\* `IPagedResultRequest`: 继承自 `ILimitedResultRequest` (所以它具有 `MaxResultCount` 属性)并且定义了 `SkipCount` (`int`)用于请求服务器的分页结果时跳过计数.

\* `ISortedResultRequest`: 定义 `Sorting` (`string`)属性以请求服务器的排序结果. 排序值可以是“名称”，"*\*Name\**", "*\*Name DESC\**", "*\*Name ASC, Age DESC\**"... 等.

\* `IPagedAndSortedResultRequest` 继承自 `IPagedResultRequest` 和 `ISortedResultRequest`,所以它有上述所有属性.

建议你继承以下基类DTO类之一,而不是手动实现接口:

\* `LimitedResultRequestDto` 实现了 `ILimitedResultRequest`.

\* `PagedResultRequestDto` 实现了 `IPagedResultRequest` (和继承自 `LimitedResultRequestDto`).

\* `PagedAndSortedResultRequestDto` 实现了 `IPagedAndSortedResultRequest` (和继承自 `PagedResultRequestDto`).

##### 最大返回数量

`LimitedResultRequestDto`(和其它固有的)通过以下规则限制和验证 `MaxResultCount`;

\* 如果客户端未设置 `MaxResultCount`,则假定为**\*\*10\*\***(默认页面大小). 可以通过设置 `LimitedResultRequestDto.DefaultMaxResultCoun` t静态属性来更改此值.

\* 如果客户端发送的 `MaxResultCount` 大于\*1,000\*\*,则会产生**\*\*验证错误\*\***. 保护服务器免受滥用服务很重要. 如果需要可以通过设置 `LimitedResultRequestDto.MaxMaxResultCount` 静态属性来更改此值.

建议在应用程序启动时设置静态属性,因为它们是静态的(全局).

#### 输出 (响应) 类型

以下接口和类用于标准化发送给客户端的输出.

\* `IHasTotalCount` 定义 `TotalCount`(`long`)属性以在分页的情况下返回记录的总数.

\* `IPagedResult<T>` 集成自 `IListResult<T>` 和 `IHasTotalCount`, 所以它有 `Items` 和 `TotalCount` 属性.

建议你继承以下基类DTO类之一,而不是手动实现接口:

\* `PagedResultDto<T>` 继承自 `ListResultDto<T>` 和实现了 `IPagedResult<T>`.

**\*\*示例: 从服务器请求分页和排序的结果并返回分页列表\*\***

````csharp

using System;

using System.Collections.Generic;

using System.Linq;

using System.Linq.Dynamic.Core;

using System.Threading.Tasks;

using Microsoft.EntityFrameworkCore;

using Volo.Abp.Application.Dtos;

using Volo.Abp.Application.Services;

using Volo.Abp.Domain.Repositories;

namespace AbpDemo

{

public class ProductAppService : ApplicationService, IProductAppService

{

private readonly IRepository<Product, Guid> \_productRepository;

public ProductAppService(IRepository<Product, Guid> productRepository)

{

\_productRepository = productRepository;

}

public async Task<PagedResultDto<ProductDto>> GetListAsync(

PagedAndSortedResultRequestDto input)

{

//Create the query

var query = \_productRepository

.OrderBy(input.Sorting);

//Get total count from the repository

var totalCount = await query.CountAsync();

//Get entities from the repository

List<Product> products = await query

.Skip(input.SkipCount)

.Take(input.MaxResultCount)

.ToListAsync();

//Map entities to DTOs

List<ProductDto> productDtos =

ObjectMapper.Map<List<Product>, List<ProductDto>>(products);

//Return the result

return new PagedResultDto<ProductDto>(totalCount, productDtos);

}

}

}

````

ABP框架还定义了一种 `PageBy` 扩展方法(与`IPagedResultRequest`兼容),可用于代替 `Skip` + `Take`调用:

````csharp

var query = \_productRepository

.OrderBy(input.Sorting)

.PageBy(input);

````

> 注意我们将`Volo.Abp.EntityFrameworkCore`包添加到项目中以使用 `ToListAsync` 和 `CountAsync` 方法,因为它们不包含在标准LINQ中,而是由Entity Framework Core定义.

如果你不了解示例代码,另请参阅[仓储文档](Repositories.md).

## 相关话题

### 验证

[应用服务](Application-Services.md)方法,控制器操作,页面模型输入...的输入会自动验证. 你可以使用标准数据注释属性或自定义验证方法来执行验证.

参阅[验证文档](Validation.md)了解更多.

### 对象到对象的映射

创建与实体相关的DTO时通常需要映射这些对象. ABP提供了一个对象到对象的映射系统简化映射过程. 请参阅以下文档:

\* [对象到对象映射文档](Object-To-Object-Mapping.md)介绍了这些功能.

\* [应用服务文档](Application-Services.md)提供了完整的示例.

## 最佳实践

你可以自由设计DTO类,然而这里有一些你可能想要遵循的最佳实践和建议.

### 共同原则

\* DTO应该是**\*\*可序列化的\*\***,因为它们通常是序列化和反序列化的(JSON或其他格式). 如果你有另一个带参数的构造函数,建议使用空(无参数)的公共构造函数.

\* 除某些[验证](Validation.md)代码外,DTO**\*\*不应包含任何业务逻辑\*\***.

\* DTO不要继承实体,也**\*\*不要引用实体\*\***. [应用程序启动模板](Startup-Templates/Application.md)已经通过分隔项目来阻止它.

\* 如果你使用自动[对象到对象](Object-To-Object-Mapping.md)映射库,如AutoMapper,请启用**\*\*映射配置验证\*\***以防止潜在的错误.

### 输入DTO原则

\* 只定义用例**\*\*所需的属性\*\***. 不要包含不用于用例的属性,这样做会使开发人员感到困惑.

\* **\*\*不要在\*\***不同的应用程序服务方法之间重用输入DTO. 因为不同的用例将需要和使用DTO的不同属性,从而导致某些属性在某些情况下没有使用,这使得理解和使用服务更加困难,并在将来导致潜在的错误.

### 输出DTO原则

\* 如果在所有情况下填充**\*\*所有属性\*\***,就可以**\*\*重用输出DTO\*\***.

#### 工作单元

# 工作单元

ABP框架的工作单元(UOW)实现提供了对应用程序中的**\*\*数据库连接和事务范围\*\***的抽象和控制.

一旦一个新的UOW启动,它将创建一个**\*\*环境作用域\*\***,当前作用域中执行的**\*\*所有数据库操作\*\***都将参与该作用域并将其视为单个事务边界. 操作一起**\*\*提交\*\***(成功时)或**\*\*回滚\*\***(异常时).

ABP的UOW系统是;

\* **\*\*按约定工作\*\***, 所以大部分情况下你不需要处理UOW.

\* **\*\*数据库提供者独立\*\***.

\* **\*\*Web独立\*\***, 这意味着你可以在Web应用程序/服务之外的任何类型的应用程序中创建工作单元作用域.

## 约定

以下方法类型被认为是一个工作单元:

\* ASP.NET Core MVC **\*\*Controller Actions\*\***.

\* ASP.NET Core Razor **\*\*Page Handlers\*\***.

\* **\*\*应用程序\*\*** 方法.

\* **\*\*仓储方法\*\***.

UOW自动针对这些方法开始,除非\*\*周围已经有一个(环境)\*\*UOW在运行.示例;

\* 如果你调用一个[仓储](Repositories.md)方法,但还没有启动UOW,它将自动**\*\*启动一个新的事务UOW\*\***,其中包括在仓储方法中完成的所有操作,如果仓储方法没有抛出任何异常,则**\*\*提交事务\*\***. 仓储方法根本不知道UOW或事务. 它只在一个常规的数据库对象上工作(例如用于[EF Core](Entity-Framework-Core.md)的`DbContext`),而UOW由ABP框架处理.

\* 如果调用[应用服务](Application-Services.md)方法,则相同的UOW系统将按上述说明工作. 如果应用服务方法使用某些仓储,这些仓储**\*\*不会开始新的UOW\*\***,而是**\*\*参与由ABP框架为应用程序服务方法启动的当前工作单元中\*\***.

\* ASP.NET Core控制器操作也是如此. 如果操作以控制器action开始,**\*\*UOW范围是控制器action的方法主体\*\***.

所有这些都是由ABP框架自动处理的.

### 数据库事务行为

虽然上一节解释了UOW是数据库事务,但实际上UOW不必是事务性的. 默认情况下;

\* **\*\*HTTP GET\*\***请求不会启动事务性UOW. 它们仍然启动UOW,但**\*\*不创建数据库事务\*\***.

\* 如果底层数据库提供程序支持数据库事务,那么所有其他HTTP请求类型都使用数据库事务启动UOW.

这是因为HTTP GET请求不会(也不应该)在数据库中进行任何更改. 你可以使用下面解释的选项来更改此行为.

## 默认选项

`AbpUnitOfWorkDefaultOptions` 用于配置工作单元系统的默认选项.在你的[模块](Module-Development-Basics.md)的 `ConfigureServices` 方法中配置选项.

**\*\*示例: 完全禁用数据库事务\*\***

````csharp

Configure<AbpUnitOfWorkDefaultOptions>(options =>

{

options.TransactionBehavior = UnitOfWorkTransactionBehavior.Disabled;

});

````

### 选项属性

\* `TransactionBehavior` (`enum`: `UnitOfWorkTransactionBehavior`). 配置事务行为的全局点. 默认值为 `Auto` ,按照上面"\*数据库事务行为"\*一节的说明工作. 你可以使用此选项启用(甚至对于HTTP GET请求)或禁用事务.

\* `TimeOut` (`int?`): 用于设置UOW的超时值. **\*\*默认值是** `null`**\*\*** 并使用基础数据库提供程序的默认值.

\* `IsolationLevel` (`IsolationLevel?`): 如果UOW是事务性的用于设置数据库事务的[隔离级别](https://docs.microsoft.com/en-us/dotnet/api/system.data.isolationlevel).

## 控制工作单元

在某些情况下你可能希望更改常规事务作用域,创建内部作用域或精细控制事务行为. 下面几节将介绍这些可能性.

### IUnitOfWorkEnabled 接口

这是为不是按照上面解释的约定作为工作单元的类(或类的层次结构)启用UOW的一种简单方法.

**\*\*示例: 为任意服务实现** `IUnitOfWorkEnabled`**\*\***

````csharp

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Uow;

namespace AbpDemo

{

public class MyService : ITransientDependency, IUnitOfWorkEnabled

{

public virtual async Task FooAsync()

{

//this is a method with a UOW scope

}

}

}

````

然后 `MyService`(和它的派生类)方法都将是UOW.

但是为了使它工作,**\*\*有些规则应该被遵守\*\***;

\* 如果你不是通过接口(如`IMyService`)注入服务,则服务的方法必须是 `virtual` 的(否则[动态代理/拦截](Dynamic-Proxying-Interceptors.md)系统将无法工作).

\* 仅异步方法(返回`Task`或`Task<T>`的方法)被拦截. 因此同步方法无法启动UOW.

> 注意,如果 `FooAsync` 在UOW作用域内被调用,那么它已经参与了UOW,不需要 `IUnitOfWorkEnabled` 或其他配置.

### UnitOfWorkAttribute

`UnitOfWork` attribute提供了更多的可能性,比如启用或禁用UOW和控制事务行为.

`UnitOfWork` attribute可以用于**\*\*类\*\***或**\*\*方法\*\***级别.

**\*\*示例: 为类的特定方法启用UOW\*\***

````csharp

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Uow;

namespace AbpDemo

{

public class MyService : ITransientDependency

{

[UnitOfWork]

public virtual async Task FooAsync()

{

//this is a method with a UOW scope

}

public virtual async Task BarAsync()

{

//this is a method without UOW

}

}

}

````

**\*\*示例: 为类的所有方法启用UOW\*\***

````csharp

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Uow;

namespace AbpDemo

{

[UnitOfWork]

public class MyService : ITransientDependency

{

public virtual async Task FooAsync()

{

//this is a method with a UOW scope

}

public virtual async Task BarAsync()

{

//this is a method with a UOW scope

}

}

}

````

**\*\*同样的规则\*\***也适用于此:

\* 如果你不是通过接口(如`IMyService`)注入服务,则服务的方法必须是 `virtual` 的(否则[动态代理/拦截](Dynamic-Proxying-Interceptors.md)系统将无法工作).

\* 仅异步方法(返回`Task`或`Task<T>`的方法)被拦截. 因此同步方法无法启动UOW.

#### UnitOfWorkAttribute 属性

\* `IsTransactional` (`bool?`): 用于设置UOW是否是事务性的. **\*\*默认值为** `null`**\*\***. 如果你让它为 `null`,它会通过约定和配置自动确定.

\* `TimeOut` (`int?`): 用于设置UOW的超时值. \*\*默认值为 `null`\*\*并回退到默认配置值.

\* `IsolationLevel` (`IsolationLevel?`): 如果UOW是事务的,用于设置数据库事务的[隔离级别](https://docs.microsoft.com/en-us/dotnet/api/system.data.isolationlevel). 如果未设置,则使用默认值.

\* `IsDisabled` (`bool`): 用于禁用当前方法/类的UOW.

> 如果在环境UOW作用域内调用方法,将忽略 `UnitOfWork` 属性,并且该方法参与周围的事务.

**\*\*示例: 为控制器action禁用UOW\*\***

````csharp

using System.Threading.Tasks;

using Volo.Abp.AspNetCore.Mvc;

using Volo.Abp.Uow;

namespace AbpDemo.Web

{

public class MyController : AbpController

{

[UnitOfWork(IsDisabled = true)]

public virtual async Task FooAsync()

{

//...

}

}

}

````

## IUnitOfWorkManager

`IUnitOfWorkManager` 是用于控制工作单元系统的主要服务. 下面的部分解释了如何使用此服务(大多数时候你并不需要).

### 开始新的工作单元

`IUnitOfWorkManager.Begin` 方法用于创建一个新的UOW作用域.

**\*\*示例: 创建一个新的非事务性UOW作用域\*\***

````csharp

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Uow;

namespace AbpDemo

{

public class MyService : ITransientDependency

{

private readonly IUnitOfWorkManager \_unitOfWorkManager;

public MyService(IUnitOfWorkManager unitOfWorkManager)

{

\_unitOfWorkManager = unitOfWorkManager;

}

public virtual async Task FooAsync()

{

using (var uow = \_unitOfWorkManager.Begin(

requiresNew: true, isTransactional: false

))

{

//...

await uow.CompleteAsync();

}

}

}

}

````

`Begin` 方法有以下可选参数:

\* `requiresNew` (`bool`): 设置为 `true` 可忽略周围的工作单元,并使用提供的选项启动新的UOW. **\*\*默认值为**`false`**. 如果为**`false`**,并且周围有UOW,则** `Begin` **方法实际上不会开始新的UOW,而是以静默方式参与现有的UOW\*\***.

\* `isTransactional` (`bool`). 默认为 `false`.

\* `isolationLevel` (`IsolationLevel?`): 如果UOW是事务的,用于设置数据库事务的[隔离级别](https://docs.microsoft.com/en-us/dotnet/api/system.data.isolationlevel). 如果未设置,则使用默认值.

\* `TimeOut` (`int?`): 用于设置UOW的超时值. \*\*默认值为 `null`\*\*并回退到默认配置值.

### 当前工作单元

如上所述UOW是环境的. 如果需要访问当前的工作单元,可以使用 `IUnitOfWorkManager.Current` 属性.

**\*\*示例: 获取当前UOW\*\***

````csharp

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Uow;

namespace AbpDemo

{

public class MyProductService : ITransientDependency

{

private readonly IUnitOfWorkManager \_unitOfWorkManager;

public MyProductService(IUnitOfWorkManager unitOfWorkManager)

{

\_unitOfWorkManager = unitOfWorkManager;

}

public async Task FooAsync()

{

var uow = \_unitOfWorkManager.Current;

//...

}

}

}

````

`Current` 属性返回一个 `IUnitOfWork` 对象.

> 如果没有周围的工作单元,则**\*\*当前工作单元可以为**`null`**\*\***. 如上所述,如果你的类是常规的UOW类,你将其手动设置为UOW或在UOW作用域内调用它,那么该值就不会为 `null`.

#### SaveChangesAsync

`IUnitOfWork.SaveChangesAsync()` 方法将到目前为止的所有更改保存到数据库中. 如果你正在使用EF Core,它的行为完全相同. 如果当前UOW是事务性的,即使已保存的更改也可以在错误时回滚(对于支持的数据库提供程序).

**\*\*示例: 插入实体后保存更改以获取其自动增量ID\*\***

````csharp

using System.Threading.Tasks;

using Volo.Abp.Application.Services;

using Volo.Abp.Domain.Repositories;

namespace AbpDemo

{

public class CategoryAppService : ApplicationService, ICategoryAppService

{

private readonly IRepository<Category, int> \_categoryRepository;

public CategoryAppService(IRepository<Category, int> categoryRepository)

{

\_categoryRepository = categoryRepository;

}

public async Task<int> CreateAsync(string name)

{

var category = new Category {Name = name};

await \_categoryRepository.InsertAsync(category);

//Saving changes to be able to get the auto increment id

await UnitOfWorkManager.Current.SaveChangesAsync();

return category.Id;

}

}

}

````

示例的 `Category` [实体](Entities.md)使用自动递增的 `int` 主键. 自动增量PK需要将实体保存到数据库中来获得新实体的ID.

示例是从基类 `ApplicationService` 派生的[应用服务](Application-Services.md), `IUnitOfWorkManager` 服务已经作为 `UnitOfWorkManager` 属性注入,所以无需手动注入.

获取当前UOW非常常见,所以还有一个 `UnitOfWorkManager.Current` 的快捷属性 `CurrentUnitOfWork`. 所以可以对上面的例子进行以下更改:

````csharp

await CurrentUnitOfWork.SaveChangesAsync();

````

##### SaveChanges() 的替代方法

由于经常需要在插入,更新或删除实体后保存更改,相应的[仓储](Repositories.md)方法有一个可选的 `autoSave` 参数. 可以将上面的 `CreateAsync` 方法按如下重写:

````csharp

public async Task<int> CreateAsync(string name)

{

var category = new Category {Name = name};

await \_categoryRepository.InsertAsync(category, autoSave: true);

return category.Id;

}

````

如果你的目的只是在创建/更新/删除实体后保存更改,建议你使用 `autoSave` 选项,而不是手动使用 `CurrentUnitOfWork.SaveChangesAsync()`.

> **\*\*Note-1\*\***: 当工作单元结束而没有任何错误时,所有更改都会自动保存. 所以除非确实需要,否则不要调用 `SaveChangesAsync()` 和设置 `autoSave` 为 `true`.

>

> **\*\*Note-2\*\***: 如果你使用 `Guid` 作为主键,则无需插入时保存来获取生成的id,因为 `Guid` 主键是在应用程序中设置的,创建新实体后立即可用.

#### IUnitOfWork 其他属性/方法

\* `OnCompleted` 方法获得一个回调动作,当工作单元成功完成时调用(在这里你可以确保所有更改都保存了).

\* `Failed` 和 `Disposed` 事件可以用于UOW失败和被销毁的通知.

\* `Complete` 和 `Rollback` 方法用于完成(提交)或回滚当前 UOW, 通常ABP框架在内部使用,如果你使用 `IUnitOfWorkManager.Begin` 方法手动启动事务,那么你可以手动使用这些方法.

\* `Options` 可用于获取启动UOW时使用的选项.

\* `Items` 字典可用于在同一工作单元内存储和获取任意对象,可以实现自定义逻辑.

## ASP.NET Core 集成

工作单元系统已完全集成到ASP.NET Core. 它为UOW系统定义了动作过滤器和页面过滤器. 当你使用ASP.NET Core MVC控制器或Razor页面时,它可以正常工作.

> 使用ASP.NET Core时,通常你不需要做任何操作配置UOW.

### 工作单元中间件

`AbpUnitOfWorkMiddleware` 是可以在ASP.NET Core请求管道中启用UOW的中间件. 如果你需要扩大UOW范围以涵盖其他一些中间件,可以这样做.

**\*\*示例:\*\***

````csharp

app.UseUnitOfWork();

app.UseConfiguredEndpoints();

````

指南: 实现DDD

# 实现领域驱动设计

## 总述

本文是实现领域驱动设计(DDD)的**\*\*实用指南\*\***.虽然在实现中依赖了ABP框架,但是本文中的概念,理论和设计模式同样适用于其它类型的项目,不仅限于.Net项目.

### 目标

本文的目标是:

\* **\*\*介绍并解释\*\***DDD的架构,概念,原理及构建.

\* **\*\*解释\*\***ABP框架的分层架构及解决方案结构.

\* 通过**\*\*案例\*\***,介绍实现DDD的一些**\*\*规则\*\***及最佳实践.

\* 展示**\*\*ABP框架\*\***为DDD的实现提供了哪些基础设施.

\* 最后,基于软件开发**\*\*最佳实践\*\***和我们的经验提供**\*\*建议\*\***来创建一个**\*\*可维护的代码库\*\***.

### 简单的代码

> **\*\*踢足球\*\***非常**\*\*简单\*\***,但是**\*\*踢简单的足球\*\***却**\*\*非常难\*\***.&mdash; <cite>约翰·克鲁伊夫(Johan Cruyff)</cite>

在编码的世界中,引用此名言:

> **\*\*写代码\*\***非常**\*\*简单\*\***,但是**\*\*写简单的代码\*\***却**\*\*非常难\*\*** &mdash; <cite>???</cite>

在本文中,我们将介绍一些容易实现的规则.

随着**\*\*应用程序的变化\*\***,有时候,为了节省开发时间会**\*\*违反一些本应遵守的规则\*\***,使得代码变得**\*\*复杂\*\***且难以维护.短期来看确实节省了开发时间,但是后期可能需要花费更多的时间为之前的偷懒而**\*\*买单\*\***.**\*\*无法对原有的代码进行维护\*\***,导致大量的逻辑都需要进行**\*\*重写\*\***.

如果你**\*\*遵循规则并按最佳实践的方式\*\***进行编码,那么你的代码将易于维护,你的业务逻辑将**\*\*更快的满足\*\***需求的变化.

## 什么是领域驱动设计?

领域驱动设计(DDD)是一种将实现与**\*\*持续进化\*\***的模型连接在一起来满足**\*\*复杂\*\***需求的软件开发方法.

DDD适用于**\*\*复杂领域\*\***或**\*\*较大规模\*\***的系统,而不是简单的CRUD程序.它着重于**\*\*核心领域逻辑\*\***,而不是基础架构.这样有助于构建一个**\*\*灵活\*\***,模块化,**\*\*可维护\*\***的代码库.

### OOP & SOLID

实现DDD高度依赖面向对象编程思想(OOP)和[SOLID](https://zh.wikipedia.org/wiki/SOLID\_(%E9%9D%A2%E5%90%91%E5%AF%B9%E8%B1%A1%E8%AE%BE%E8%AE%A1))原则.事实上,DDD已经**\*\*实现\*\***并**\*\*延伸\*\***了这些原则,因此,**\*\*深入了解\*\***OOP和SOLID对实施DDD十分有利.

### DDD分层与整洁架构

基于DDD的架构分为四个基础层

![domain-driven-design-layers](images/domain-driven-design-layers.png)

**\*\*业务逻辑\*\***分为两层,分别为 *\*领域(Domain)\** 层和 *\*应用(Application)\** 层,它们包含不同类型的业务逻辑.

\* **\*\*领域层\*\***:只实现领域业务逻辑,与用例无关.

\* **\*\*应用层\*\***:基于领域层来实现满足用例的业务逻辑.用例可以看作是用户界面(UI)或外部应用程序的交互.

\* **\*\*展现层\*\***:包含应用程序的UI元素.

\* **\*\*基础设施层\*\***:通过对第三方库的集成或抽象,来满足其它层的非核心业务逻辑的实现.

同样的分层架构也可以如下图所示:被称为 **\*\*整洁架构\*\***, 又或者称为 **\*\*洋葱架构\*\***:

![domain-driven-design-clean-architecture](images/domain-driven-design-clean-architecture.png)

在整洁架构中,**\*\*每层只依赖内部的层\*\***,独立的层在圆圈的最中心,也就是领域层.

### 核心构建组成

DDD的关注点在**\*\*领域层\*\***和**\*\*应用层\*\***上,而展现层和基础设施层则视为*\*细节\**(这个词原文太抽象,自己体会吧),业务层不应依赖它们.

这并不意味着展现层和基础设施层不重要.它们非常重要,但*\*UI框架\** 和 *\*数据库提供程序\** 需要你自己定义规则和总结最佳实践.这些不在DDD的讨论范围中.

本节将介绍领域层和应用层的基本构建组件.

#### 领域层构建组成

\* **\*\*实体(Entity)\*\***: [实体](Entities.md)是种领域对象,它有自己的属性(状态,数据)和执行业务逻辑的方法.实体由唯一标识符(Id)表示,不同ID的两个实体被视为不同的实体.

\* **\*\*值对象(Value Object)\*\***: [值对象](Value-Objects.md)是另外一种类型的领域对象,使用值对象的属性来判断两个值对象是否相同,而非使用ID判断.如果两个值对象的属性值全部相同就被视为同一对象.值对象通常是不可变的,大多数情况下它比实体简单.

\* **\*\*聚合(Aggregate) 和 聚合根(Aggregate Root)\*\***: [聚合](Entities.md)是由**\*\*聚合根\*\***包裹在一起的一组对象(实体和值对象).聚合根是一种具有特定职责的实体.

\* **\*\*仓储(Repository)\*\*** (接口): [仓储](Repositories.md)是被领域层或应用层调用的数据库持久化接口.它隐藏了DBMS的复杂性,领域层中只定义仓储接口,而非实现.

\* **\*\*领域服务(Domain Service)\*\***: [领域服务](Domain-Services.md)是一种无状态的服务,它依赖多个聚合(实体)或外部服务来实现该领域的核心业务逻辑.

\* **\*\*规约(Specification)\*\***: [规约](Specifications.md)是一种**\*\*强命名\*\***,**\*\*可重用\*\***,**\*\*可组合\*\***,**\*\*可测试\*\***的实体过滤器.

\* **\*\*领域事件(Domain Event)\*\***: [领域事件](Event-Bus.md)是当领域某个事件发生时,通知其它领域服务的方式,为了解耦领域服务间的依赖.

#### 应用层构建组成

\* **\*\*应用服务(Application Service)\*\***: [应用服务](Application-Services.md)是为实现用例的无状态服务.展现层调用应用服务获取DTO.应用服务调用多个领域服务实现用例.用例通常被视为一个工作单元.

\* **\*\*数据传输对象(DTO)\*\***: [DTO](Data-Transfer-Objects.md)是一个不含业务逻辑的简单对象,用于应用服务层与展现层间的数据传输.

\* **\*\*工作单元(UOW)\*\***: [工作单元](Unit-Of-Work.md)是事务的原子操作.UOW内所有操作,当成功时全部提交,失败时全部回滚.

## 实现领域驱动:重点

### .NET解决方案分层

下图是使用[ABP的启动模板](Startup-Templates/Application.md)创建的解决方案:

![domain-driven-design-vs-solution](images/domain-driven-design-vs-solution.png)

解决方案名为 `IssueTracking` ,它包含多个项目.该解决方案出于**\*\*DDD原则\*\***及**\*\*开发\*\***和**\*\*部署\*\***的实践来进行分层.后面会在各小节中介绍解决方案中的项目.

> 在使用启动模板时,如果选择了其它类型的UI或*\*数据库提供程序\**,解决方案的结构会略有不同.但是领域层和应用层是一样的,这才是DDD的重点.如果你想了解有关解决方案的更多信息,请参见[启动模板](Startup-Templates/Application.md).

#### 领域层

领域层分为两个项目:

\* `IssueTracking.Domain`是**\*\*领域层中必需\*\***的,它包含之前介绍的**\*\*构建组成\*\***(实体,值对象,领域服务,规约,仓储接口等).

\* `IssueTracking.Domain.Shared`是领域层中**\*\*很薄的项目\*\***,它只包含领域层与其它层共享的数据类型的定义.例如,枚举,常量等.

#### 应用层

应用层也被分为了两个项目:

\* `IssueTracking.Application.Contracts`包含**\*\*接口\*\***的定义及接口依赖的**\*\*DTO\*\***,此项目可以被展现层或其它客户端应用程序引用.

\* `IssueTracking.Application`是**\*\*应用层中必需\*\***的,它实现了`IssueTracking.Application.Contracts`项目中定义的接口.

#### 展现层

\* `IssueTracking.Web`是一个ASP.NET Core MVC / Razor Pages应用程序.它是提供UI元素及API服务的可执行程序.

> ABP框架还支持其它类型的UI框架,包括[Angular](UI/Angular/Quick-Start.md)和[Blazor](UI/Blazor/Overall.md).当选择*\*Angular\**或*\*Blazor\**时,解决方案不会有`IssueTracking.Web`项目.但是会添加`IssueTracking.HttpApi.Host`在解决方案中,此项目会提供HTTP API供UI调用.

#### 远程服务层

\* `IssueTracking.HttpApi`包含了HTTP API的定义.它通常包含*\*MVC Controller\** 和 *\*Model\**(如果有).因此,你可以在此项目中提供HTTP API.

> 大多数情况下,通过使用*\*API Controller\** 包装应用服务,供客户端远程调用.ABP框架的[API自发现系统](API/Auto-API-Controllers.md)可以自动将应用服务公开为API,因此,通常不需要在此项目中再创建Controller.出于需要手动添加额外Controller的情况,也包含在模板解决方案中.

\* `IssueTracking.HttpApi.Client`当C#客户端应用程序需要调用`IssueTracking.HttpApi`的API时,这个项目非常有用.客户端程序仅需引用此项目就可以通过依赖注入方式,远程调用应用服务.它是通过ABP框架的[动态C#客户端API代理系统](API/Dynamic-CSharp-API-Clients.md)来实现的.

> 在解决方案文件夹`test`下,有一个名为`IssueTracking.HttpApi.Client.ConsoleTestApp`的控制台程序.它演示了如何使用`IssueTracking.HttpApi.Client`项目来远程调用应用程序公开的API.因为只是演示,你可以删除此项目,再或者你认为`IssueTracking.HttpApi`不需要,同样可以删除.

#### 基础设施层

你可能只创建一个基础设施项目来完成所有抽象类的定义及外部类的集成,又或者为不同的依赖创建多个不同的项目.

我们建议采用一种平衡的方法:为主要的依赖的库(例如 Entity Framework Core)创建一个独立的项目,为其它的依赖库创建一个公共的基础设施项目.

ABP的启动解决方案中包含两个用于集成Entity Framework Core的项目:

\* `IssueTracking.EntityFrameworkCore`是必需的,因为需要集成*\*EF Core\**.应用程序的`数据库上下文(DbContext)`,数据库对象映射,仓储接口的实现,以及其它与*\*EF Core\**相关的内容都位于此项目中.

\* `IssueTracking.EntityFrameworkCore.DbMigrations`是管理Code First方式数据库迁移记录的特殊项目.此项目定义了一个独立的`DbContext`来追踪迁移记录.只有当添加一个新的数据库迁移记录或添加一个新的[应用模块](Modules/Index.md)时,才会使用此项目,否则,其它情况无需修改此项目内容.

> 你可能想知道为什么会有两个EF Core项目,主要是因为[模块化](Module-Development-Basics.md).每个应用模块都有自己独立的`DbContext`,你的应用程序也有自己`DbContext`.`DbMigrations`项目包含**\*\*合并\*\***所有模块迁移记录的**\*\*单个迁移路径\*\***.虽然大多数情况下你无需过多了解,但也可以查看[EF Core迁移](Entity-Framework-Core-Migrations.md)了解更多信息.

#### 其它项目

另外还有一个项目`IssueTracking.DbMigrator`,它是一个简单的控制台程序,用来执行数据库迁移,包括**\*\*初始化\*\***数据库及创建**\*\*种子数据\*\***.这是一个非常实用的应用程序,你可以在开发环境或生产环境中使用它.

### 项目间的依赖关系

下图展示了解决方案中项目间的依赖关系(有些项目比较简单就未展示):

![domain-driven-design-project-relations](images/domain-driven-design-project-relations.png)

之前已介绍了这些项目.现在,我们来解释依赖的原因:

\* `Domain.Shared` 所有项目直接或间接依赖此项目.此项目中的所有类型都可以被其它项目所引用.

\* `Domain` 仅依赖`Domain.Shared`项目,因为`Domain.Shared`本就属于领域层的一部分.例如,`Domain.Shared`项目中的枚举类型 `IssueType` 被`Domain`项目中的`Issue`实体所引用.

\* `Application.Contracts` 依赖`Domain.Shared`项目,可以在DTO中重用`Domain.Shared`中的类型.例如,`Domain.Shared`项目中的枚举类型 `IssueType` 同样被`Contracts`项目中的`CreateIssueDto`DTO所引用.

\* `Application` 依赖`Application.Contracts`项目,因为此项目需要实现应用服务的接口及接口使用的DTO.另外也依赖`Domain`项目,因为应用服务的实现必须依赖领域层中的对象.

\* `EntityFrameworkCore` 依赖`Domain`项目,因为此项目需要将领域对象(实体或值对象)映射到数据库的表,另外还需要实现`Domain`项目中的仓储接口.

\* `HttpApi` 依赖`Application.Contracts`项目,因为Controllers需要注入应用服务.

\* `HttpApi.Client` 依赖`Application.Contracts`项目,因为此项目需要使用应用服务.

\* `Web` 依赖`HttpApi`项目,因为此项目对外提供HTTP APIs.另外Pages或Components 需要使用应用服务,所以还间接依赖了`Application.Contracts`项目

#### 虚线依赖

你在上图中会发现用虚线表示了另外两个依赖.`Web`项目依赖了 `Application` and `EntityFrameworkCore`,理论上`Web`不应该依赖这两个项目,但实际上依赖了.原因如下:

`Web`是最终的运行程序,是负责托管Web的宿主,它在运行时需要**\*\*应用服务和仓储的实现类\*\***.

这种依赖关系的设计,可能会让你有机会在展现层直接使用到EF Core的对象,**\*\*应该严格禁止这样的做法\*\***.如果想在解决方案分层上规避这种问题,有下面两种方式,相对复杂一些:

\* 将`Web`项目类型改为razor类库,并创建一个新项目,比如`Web.Host`,`Web.Host`依赖`Web`,`Application`,`EntityFrameworkCore`三个项目,并作为Web宿主程序运行.注意,不要写任何与UI相关的代码,只是作为**\*\*宿主运行\*\***.

\* 在`Web`项目中移除对`Application`和`EntityFrameworkCore`的引用,`Web`在启动时,再动态加载程序集`IssueTracking.Application.dll`和`IssueTracking.EntityFrameworkCore.dll`.可以使用ABP框架的[插件模块](PlugIn-Modules.md)来动态加载程序集.

### DDD模式的应用程序执行顺序

下图展示了基于DDD模式下的Web应用程序执行顺序:

![](images/domain-driven-design-web-request-flow.png)

\* 通常由UI(用例)发起一个HTTP请求到服务器.

\* 由展现层(或分布式服务层)中的一个*\*MVC Controller\**或*\*Razor Page Handler\**处理请求,在这个阶段可以执行一些AOP逻辑([授权](Authorization.md),[验证](Validation.md),[异常处理](Exception-Handling.md)等),*\*MVC Controller\**或*\*Razor Page Handler\**调用注入的应用服务接口,并返回其调用后的结果(DTO)..

\* 应用服务使用领域层的对象(实体,仓储接口,领域服务等)来实现UI(用例)交互.此阶段同样可以执行一些AOP逻辑(授权,验证等).应用服务中的每个方法应该是一个[工作单元](Unit-Of-Work.md),代表它是一次原子性操作.

跨域问题大多数由**\*\*ABP框架自动实现\*\***,通常不需要为此额外编码.

### 通用原则

在详细介绍之前,我们先来看一些DDD的总体原则.

#### 数据库提供程序 / ORM 独立原则

领域层和应用层应该与*\*数据库提供程序 / ORM\**无关.领域层和应用层仅依赖仓储接口,并且仓储接口不依赖特定的ORM对象.

原因如下:

1. 未来领域层或应用层的基础设施会发生改变,例如,需要支持另外一种数据库类型,因此需要保持**\*\*领域层或应用层的基础设施是独立的\*\***.

2. 将基础设施的实现隐藏在仓储中,使得领域层或应用层更**\*\*专注于业务逻辑代码\*\***.

3. 可以通过模拟仓储接口,使得自动化测试更为方便.

> 关于此原则, `EntityFrameworkCore`项目只被启动程序项目所引用,解决方案中其它项目均未引用.

##### 关于数据库独立原则的讨论

**\*\*原因1\*\***会非常影响你**\*\*领域对象的建模\*\***(特别是实体间的关系)及**\*\*应用程序的代码\*\***.假如,开始选择了关系型数据库,并使用了[Entity Framework Core](Entity-Framework-Core.md),后面尝试切换到[MongoDB](MongoDB.md),那么 **\*\*EF Core 中一些非常有用的特性\*\***你就不能使用了,例如:

\* 无法使用[变更追踪](https://docs.microsoft.com/zh-cn/ef/core/querying/tracking) ,因为*\*MongoDB provider\**没有提供此功能,因此,你始终需要显式的更新已变更的实体.

\* 无法在不同的聚合间使用[导航属性](https://docs.microsoft.com/zh-cn/ef/core/modeling/relationships),因为文档型数据库是不支持的.有关更多信息,请参见"规则:聚合间仅通过Id关联".

如果你认为这些功能对你很**\*\*重要\*\***,并且你永远都不会**\*\*离开\*\*** *\*EF Core\**,那么我们认为你可以忽略这一原则.假如你在设计实体关系时使用了*\*EF Core\**,你甚至可以在应用层引用*\*EF Core Nuget\**包,并直接使用异步的LINQ扩展方法,例如 `ToListAsync()`(有关更多信息,请参见[仓储](Repositories.md)文档中的*\*IQueryable\**和*\*Async Operations\**).

但是我们仍然建议采用仓储模式来隐藏基础设施中实现过程.

#### 展现层技术无关原则

展现层技术(UI框架)是现代应用程序中最多变的部分之一.**\*\*领域层和应用层\*\***应该对展现层所采用的技术或框架**\*\*一无所知\*\***.使用ABP启动模板就非常容易实现此原则.

在某些情况下,你可能需要在应用层和展现层中写重复的逻辑,例如,参数验证和授权检查.展现层检查出于**\*\*用户体验\*\***,应用层或领域层检查出于**\*\*数据安全性\*\***和**\*\*数据完整性\*\***.

#### 关注状态的变化,而不是报表/查询

DDD关注领域对象的**\*\*变化和相互作用\*\***,如何创建或修改一个具有数据**\*\*完整性,有效性\*\***,符合**\*\*业务规则\*\***的实体对象.

DDD忽略**\*\*领域对象的数据展示\*\***,这并不意味着它们并不重要,如果应用程序没有精美的看板和报表,谁会愿意用呢?但是报表是另外一个讨论话题,你可以通过使用SQL Server报表功能或ElasticSearch来提供数据展示,又或者使用优化后的SQL查询语句,创建数据库索引或存储过程.唯一的原则是不要将这些内容混入领域的业务逻辑中.

## 实现领域驱动:构建组成

这是本指南的重要部分,我们将通过示例介绍并解释一些**\*\*明确的规则\*\***,在实现领域驱动设计时,你可以遵循这些规则并将其应用于解决方案中.

### 领域

示例中会使用一些概念,这些概念在Github中被使用,例如, `Issue`(问题), `Repository`(仓库), `Label`(标签) 和`User`(用户).下图中展示了一些聚合,聚合根,实体,值对象以及它们之间的关系:

![domain driven design example schema](images/domain-driven-design-example-domain-schema.png)

**\*\*问题聚合\*\*** 由`Issue` 聚合根,及其包含的 `Comment` 和`IssueLabel` 集合组成.我们将重要讨论 `Issue` 聚合根:

![domain-driven-design-issue-aggregate-diagram](images/domain-driven-design-issue-aggregate-diagram.png)

### Aggregates

如前所述, [聚合](Entities.md)是由**\*\*聚合根\*\***包裹在一起的一组对象(实体和值对象).本节将介绍于聚合的有关原理和规则.

> 后面的文档中,我们将使用 *\*实体\** 替代 *\*聚合根\** 或 *\*子集合实体\** ,除非我们明确指明使用 *\*聚合根\** 或 *\*子集合实体\** .

#### 聚合 / 聚合根 原则

##### 业务规则

实体负责实现与其自身属性相关的业务规则.同时*\*聚合根实体\**还负责它们的子集合实体.

聚合应该通过领域规则和约束来保证自身的**\*\*完整性\*\***和**\*\*有效性\*\***.这意味着,实体与DTO是不同的,实体应该比DTO多了些**\*\*实现业务逻辑的方法\*\***.我们应该尽可能地在实体上实现业务规则.

##### 独立单元

应该在一个独立的单元中完成**\*\*一个聚合的获取及保存\*\***,包括自身属性及其子集合.假如我们需要在`Issue`中添加一个新的`Comment`,步骤如下:

\* 从数据库中获取一个 `Issue` 对象,包括所有子集合(`Comments`和`IssueLabels`).

\* 使用 `Issue` 实体上的添加新`Comment`的方法,例如 `Issue.AddComment(...);`.

\* 在数据库的单次操作中完成整个 `Issue`对象(包括子集合)的保存.

对于在关系型数据库上用过 **\*\*EF Core\*\*** 的开发人员会认为,获取`Issue`的同时加载子集合没有必要并且还影响性能.为什么不使用SQL语句`Insert`来直接插入记录呢?

这样做的原因是我们需要执行业务规则来保证数据的一致性和完整性.假如有一个业务规则:"用户不能对已锁定的问题进行评论".那如何在不查询数据库的情况下,获取问题是否已被锁定?所以,只有关联的对象都被加载了的时候,我们才可以执行业务规则.

另外,使用**\*\*MongoDB\*\***的开发人员就认为此原则很好理解.在MongoDB中,聚合对象(包含子集合)会被保存到一个`collection`中.因而,无需任何其它配置,就可以实现查询一个聚合,同时包含所有子对象.

ABP框架有助于你实现这一原则

**\*\*例子: 问题追加评论\*\***

````csharp

public class IssueAppService : ApplicationService, IIssueAppService

{

private readonly IRepository<Issue, Guid> \_issueRepository;

public IssueAppService(IRepository<Issue, Guid> issueRepository)

{

\_issueRepository = issueRepository;

}

[Authorize]

public async Task CreateCommentAsync(CreateCommentDto input)

{

var issue = await \_issueRepository.GetAsync(input.IssueId);

issue.AddComment(CurrentUser.GetId(), input.Text);

await \_issueRepository.UpdateAsync(issue);

}

}

````

通过`\_issueRepository.GetAsync`方法来获取`Issue`对象时,默认就已经加载了所有子集合.对于MongoDB很简单,EF Core 则需要额外配置,一旦配置,ABP仓储类会自动处理.`\_issueRepository.GetAsync`方法还有个可选参数`includeDetails`,可以传`false`,手动禁止加载子集合.

> 如何配置及替代方案,请参考[EF Core document](Entity-Framework-Core.md)的*\*加载关联实体\** 章节.

`Issue.AddComment`接收两个参数,分别是`userId`和`text`,再执行自己的业务规则,最终将评论添加到`Issue`的评论集合中.

最后,我们使用`\_issueRepository.UpdateAsync`方法,将对象保存到数据库中.

> EF Core 具有**\*\*变更追踪\*\***的功能,因此,不需要调用`\_issueRepository.UpdateAsync`方法.ABP的工作单元会在方法结束时,自动执行`DbContext.SaveChanges()`的.如果使用MongoDB则需要显式手动调用.

>

> 因此,当需要额外编写仓储层的实现,应该在实体变化时始终调用 `UpdateAsync` 方法.

##### 事务边界

通常认为一个聚合就是一个事务边界.如果用例只涉及单个聚合,那么读取及修改就是一个操作单元.对聚合内所有对象的修改都将作为原子操作一起保存,无需显式创建数据库事务.

但是,实际上,可能需要在一个用例中更改**\*\*多个聚合对象的实例\*\***,并且还要求创建事务来保证**\*\*原子更新\*\***和**\*\*数据一致性\*\***.因此,ABP框架提供了为每个用例(应用服务中的方法),可以创建显式事务的功能.有关更多信息,请参见文档[工作单元](Unit-Of-Work.md).

##### 序列化

一个聚合(包含聚合根及子集合)可以被序列化或反序列化.例如,MongoDB在保存对象到数据库时,会将聚合序列化为JSON文件,读取时再进行反序列化.

> 使用关系型数据库+ORM时,这个原则不是必须的,但是,这是领域驱动设计的重要实践.

以下规则遵循序列化原则

#### 聚合/聚合根规则及最佳实践

以下规则是遵循上述原则.

##### 聚合间只通过ID相互引用

聚合应该只引用其它聚合的ID,也就是说,不允许定义导航属性关联至其它聚合.

\* 该规则遵循了可序列化原则.

\* 该规则还可以避免不同聚合彼此间的相互操作以及业务逻辑的暴露.

下图中,可以看到两个聚合根,`GitRepository` 和`Issue` :

![domain-driven-design-reference-by-id-sample](images/domain-driven-design-reference-by-id-sample.png)

\* `GitRepository` 不应该包含 `Issue`的集合,因为`Issue`属于不同的聚合.

\* `Issue` 不应该包含导航属性至 `GitRepository` .因为 `GitRepository`属于不同的聚合.

\* `Issue` 可以有 `RepositoryId` 的引用.

因此,若要获取`Issue`关联的 `GitRepository`对象,需要使用`Issue`的`RepositoryId`在数据库中进行一次查询.

###### 对于EF Core和关系型数据库

MongoDB中不适合使用导航属性或集合的,原因是:当前源聚合对象会被序列化为JSON,其中会保存导航目标聚合的副本.

在使用EF Core在关系型数据库上进行操作时,开发者可能认为此规则没必要.但是我们认为这是一条非常重要的规则,有助于降**\*\*低领域的复杂性\*\***减少风险.我们强烈建议遵守此规则.如果你确定要忽略此规则,请参见上面的"关于数据库独立原理的讨论"章节.

##### 保持聚合尽量的小

保持聚合简单而小巧是一个比较好的做法.因为聚合的读取与保存是一个整体,当处理较大对象时会出现性能问题,如下所示:

![domain-driven-design-aggregate-keep-small](images/domain-driven-design-aggregate-keep-small.png)

角色聚合包含`UserRole`值对象集合,方便查询该角色下有哪些用户.注意,`UserRole`不是聚合,并且也遵守了*\*聚合间只通过ID相互引用\**的规则.但是在现实场景中,一个角色可能被给成千上万个用户,当从数据库中加载一个角色时,会关联加载数千个用户对象,这里会有 严重的性能问题.

反过来看,`User`也可以有`Roles` 集合,现实中一个用户不会具有太多的角色,因此采用`User`关联`Roles`这种方式比较合适.

当使用**\*\*非关系型数据库时\*\***,`User`和`Role` 同时都有关联子集合,会出现另外一个问题.相同的记录会被重复记录在不同的集合中,并且难以保证数据一致性(需要添加记录到`User.Roles`和`Role.Users`中)

因此,请根据以下注意事项来确定聚合的边界:

\* 同时被使用的对象.

\* 查询(读取/保存)性能和内存消耗.

\* 数据完整性,有效性,一致性.

现实情况:

\* 大多数聚合根**\*\*没有子集合\*\***.

\* 子集合的数量控制在**\*\*100-150个\*\***.如果集合数量超过150个,考虑将子对象改成聚合根.

##### 聚合根 / 实体的主键

\* 聚合根通常具有唯一的标识符ID (主键: PK).我们建议使用 `Guid`作为聚合根的主键类型. (原因请参见[Guid生成文档](Guid-Generation.md)).

\* 聚合中的实体(非聚合根)可以使用联合主键.

如图所示:

![domain-driven-design-entity-primary-keys](images/domain-driven-design-entity-primary-keys.png)

\* `Organization`有一个`Guid`的标识符 (`Id`).

\* `OrganizationUser`是一个子集合,使用 `OrganizationId` 和`UserId`作为联合主键.

并不是所有的子集合的主键都是联合主键,有些情况下,可以使用单独的`Id`作为主键.

> 联合主键实际上是关系型数据库中的概念,因为子集合对象有与之对应的数据库表,而表也要有主键.但是在非关系型数据库中,无需为子集合实体定义主键,因为它们本身就已属于一个聚合根.

##### 聚合根 / 实体的构造函数

构造函数是实体生命周期的开始被执行.以下是构造函数的编写建议:

\* 将实体的**\*\*必填属性\*\***作为构造函数参数,这样可以创建一个**\*\*有效(符合规则)的实体\*\***.另外,将非必填属性作为构造函数的可选参数.

\* 参数必须**\*\*检查有效性\*\***.

\* 所有**\*\*子集合\*\***对象必须被初始化.

**\*\*示例:** `Issue` **(聚合根) 构造函数\*\***

````csharp

using System;

using System.Collections.Generic;

using System.Collections.ObjectModel;

using Volo.Abp;

using Volo.Abp.Domain.Entities;

namespace IssueTracking.Issues

{

public class Issue : AggregateRoot<Guid>

{

public Guid RepositoryId { get; set; }

public string Title { get; set; }

public string Text { get; set; }

public Guid? AssignedUserId { get; set; }

public bool IsClosed { get; set; }

public IssueCloseReason? CloseReason { get; set; } //enum

public ICollection<IssueLabel> Labels { get; set; }

public Issue(

Guid id,

Guid repositoryId,

string title,

string text = null,

Guid? assignedUserId = null

) : base(id)

{

RepositoryId = repositoryId;

Title = Check.NotNullOrWhiteSpace(title, nameof(title));

Text = text;

AssignedUserId = assignedUserId;

Labels = new Collection<IssueLabel>();

}

private Issue() { /\* for deserialization & ORMs \*/ }

}

}

````

\* `Issue` 通过构造函数的参数,对必填属性进行赋值,而从创建一个有效的实体对象.

\* 构造函数对参数进行**\*\*验证\*\***(`Check.NotNullOrWhiteSpace(...)`必填项为空,抛出异常).

\* 初始化子集合.在实例化`Issue`对象后,访问 `Labels` ,不会出现空指针异常.

\* 构造函数还将 `id`传递给父类,不要在构造函数内生成 `Guid`(参阅 [Guid生成](Guid-Generation.md)).

\* 为ORM保留**\*\*私有的无参构造函数\*\***.防止编写代码时,意外使用了无参构造函数.

> 参见 [实体](Entities.md) 文档,了解更多使用ABP框架创建实体的信息.

##### 实体属性访问器和方法

上面的示例中,我们在构造函数中对 `Title` 进行了非空检查.但是开发人员可以再次对`Title`进行赋值.

如果我们使用**\*\*公开的属性\*\***,则无法在实体控制数据的**\*\*有效性\*\***和**\*\*完整性\*\***.建议:

\* 如果某个属性具有业务逻辑,则将该属性的**\*\*setter\*\***改为私有.

\* 定义公开的方法来修改属性.

**\*\*示例:提供方法修改属性\*\***

````csharp

using System;

using Volo.Abp;

using Volo.Abp.Domain.Entities;

namespace IssueTracking.Issues

{

public class Issue : AggregateRoot<Guid>

{

public Guid RepositoryId { get; private set; } //Never changes

public string Title { get; private set; } //Needs validation

public string Text { get; set; } //No validation

public Guid? AssignedUserId { get; set; } //No validation

public bool IsClosed { get; private set; } //Should change with CloseReason

public IssueCloseReason? CloseReason { get; private set; } //Should change with IsClosed

//...

public void SetTitle(string title)

{

Title = Check.NotNullOrWhiteSpace(title, nameof(title));

}

public void Close(IssueCloseReason reason)

{

IsClosed = true;

CloseReason = reason;

}

public void ReOpen()

{

IsClosed = false;

CloseReason = null;

}

}

}

````

\* `RepositoryId` setter是私有的.创建后无法变更,业务规则不允许将已有的问题移到其它仓库.

\* `Title` setter是私有的.它的修改时需要加以验证.

\* `Text` 和 `AssignedUserId` setter 是公开的.因为业务规则允许它们为空或任意值.我们认为没必要将它们改为私有的,如果将来业务发生变化,再将setter改为私有,并提供公开的方法进行修改.另外实体属于领域层,不会直接暴露属性给应用层(或其它层),目前将其公开不是什么大问题.

\* `IsClosed` 和 `IssueCloseReason` 它们是一组属性.定义 `Close` 和 `ReOpen` 方法来同时对这两个属性进行赋值.这样可以防止问题被无故关闭.

##### 业务逻辑与实体异常

对实体进行验证,或执行业务逻辑时,通常需要抛出异常:

\* 领域中定义的**\*\*特定的异常\*\***.

\* 实体方法中**\*\*抛出的异常\*\***.

**\*\*示例\*\***

````csharp

public class Issue : AggregateRoot<Guid>

{

//...

public bool IsLocked { get; private set; }

public bool IsClosed { get; private set; }

public IssueCloseReason? CloseReason { get; private set; }

public void Close(IssueCloseReason reason)

{

IsClosed = true;

CloseReason = reason;

}

public void ReOpen()

{

if (IsLocked)

{

throw new IssueStateException(

"Can not open a locked issue! Unlock it first."

);

}

IsClosed = false;

CloseReason = null;

}

public void Lock()

{

if (!IsClosed)

{

throw new IssueStateException(

"Can not open a locked issue! Unlock it first."

);

}

IsLocked = true;

}

public void Unlock()

{

IsLocked = false;

}

}

````

这里有两个业务规则:

\* 已锁定的问题无法重新开启.

\* 无法锁定已开打的问题.

当违反业务规则时,`Issue` 会抛出 `IssueStateException` 异常:

````csharp

using System;

namespace IssueTracking.Issues

{

public class IssueStateException : Exception

{

public IssueStateException(string message)

: base(message)

{

}

}

}

````

抛出异常会引发两个问题:

1. 当异常发生时,**\*\*用户\*\***应该看到异常(错误)信息吗?如果需要看到,异常消息如何实现本地化? 实体中无法注入[本地化](Localization.md)的 `IStringLocalizer` 接口.

2. 对于Web应用或HTTP API,应向客户端返回什么**\*\*HTTP状态代码\*\***.

ABP框架的 [异常处理](Exception-Handling.md) 可以解决上述问题.

**\*\*示例:使用异常编码\*\***

````csharp

using Volo.Abp;

namespace IssueTracking.Issues

{

public class IssueStateException : BusinessException

{

public IssueStateException(string code)

: base(code)

{

}

}

}

````

\* `IssueStateException` 继承至 `BusinessException` .对于`BusinessException`的派生类,ABP框架默认返回的HTTP状态码是403 (默认是服务器内部错误 状态码 500)

\* 将`code` 作为Key,在本地化资源中查找对应的文字.

现在,我们修改 `ReOpen` 方法:

````csharp

public void ReOpen()

{

if (IsLocked)

{

throw new IssueStateException("IssueTracking:CanNotOpenLockedIssue");

}

IsClosed = false;

CloseReason = null;

}

````

> 使用常量而不是魔法字符串.

在本地化资源文件中添加对应的记录:

````json

"IssueTracking:CanNotOpenLockedIssue": "Can not open a locked issue! Unlock it first."

````

\* 异常发生时,ABP将自动使用本地化消息(基于当前语言).

\* 异常编码 (`IssueTracking:CanNotOpenLockedIssue` )会被发送到客户端.同样可以以编程方式处理此异常.

> 你可以无需定义 `IssueStateException`,直接抛出`BusinessException`异常.详细信息,参见[异常处理文档](Exception-Handling.md)

##### 实体中业务逻辑依赖外部服务时

仅依赖实体本身的属性执行的业务规非常简单.但是有时候,复杂的业务逻辑会**\*\*查询数据库\*\***或使用[依赖注入](Dependency-Injection.md)中的其它服务,这该怎么办?注意:**\*\*实体是无法注入服务的\*\***.

实现这种业务逻辑有两种方式:

\* 将依赖的服务以**\*\*方法的参数\*\***,传递到实体的业务逻辑方法中.

\* 定义一个**\*\*领域服务\*\***.

领域服务我们后面再说.我们先看看在实体类中如何实现:

**\*\*示例:业务规则: 不允许将3个以上未解决的问题关联到一个用户\*\***

````csharp

public class Issue : AggregateRoot<Guid>

{

//...

public Guid? AssignedUserId { get; private set; }

public async Task AssignToAsync(AppUser user, IUserIssueService userIssueService)

{

var openIssueCount = await userIssueService.GetOpenIssueCountAsync(user.Id);

if (openIssueCount >= 3)

{

throw new BusinessException("IssueTracking:ConcurrentOpenIssueLimit");

}

AssignedUserId = user.Id;

}

public void CleanAssignment()

{

AssignedUserId = null;

}

}

````

\* `AssignedUserId` 私有的属性setter.此属性只能通过`AssignToAsync` 和`CleanAssignment` 方法来修改.

\* `AssignToAsync` 通过 `user.Id` 属性获取一个 `AppUser` 实体.

\* `IUserIssueService` 是获取用户未解决问题的服务.

\* `AssignToAsync` 当不满足业务规则时抛出异常.

\* 最后,符合规则,就对属性`AssignedUserId` 进行赋值.

这样就解决了将问题关联到用户时,需要调用外部服务的问题,但是它也存在几个问题:

\* 实体**\*\*依赖了外部服务\*\***,实体变得**\*\*复杂\*\***.

\* 实体调用变的复杂.在调用`AssignToAsync` 方法时还需要传递 `IUserIssueService` 服务.

实现这种业务逻辑另外一种方案,使用领域服务,后面将详细说明.

### 仓储

[仓储](Repositories.md)是一个类集合的接口,它通常被领域层或应用层调用,负责访问持久化系统(数据库),读取写入业务对象(聚合).

仓储的原则:

\* 在**\*\*领域层\*\***中定义仓储接口,因为仓储会被领域层或应用层调用,在**\*\*基础设施层中实现\*\***(*\*EntityFrameworkCore\** 项目).

\* 仓储中**\*\*不要写任何业务逻辑\*\***.

\* 仓储接口不依赖 **\*\*数据库提供程序 / ORM\*\***.例如,不要在仓储中返回 `DbSet` 类型,因为 `DbSet`是EF Core中的对象.

\* **\*\*仅为聚合根定义仓储\*\***,非聚合根对象不要提供仓储,因为子集合可以通过聚合根来进行持久化.

#### 仓储中不要写任何业务逻辑

我们经常不小心把业务逻辑编写到了仓储层.

**\*\*示例:从仓储中获取非活动的问题\*\***

````csharp

using System;

using System.Collections.Generic;

using System.Threading.Tasks;

using Volo.Abp.Domain.Repositories;

namespace IssueTracking.Issues

{

public interface IIssueRepository : IRepository<Issue, Guid>

{

Task<List<Issue>> GetInActiveIssuesAsync();

}

}

````

`IIssueRepository`继承至`IRepository<...>`接口,并新增一个新接口`GetInActiveIssuesAsync`.此仓储为聚合根`Issue`提供查询的实现.

````csharp

public class Issue : AggregateRoot<Guid>, IHasCreationTime

{

public bool IsClosed { get; private set; }

public Guid? AssignedUserId { get; private set; }

public DateTime CreationTime { get; private set; }

public DateTime? LastCommentTime { get; private set; }

//...

}

````

(上面的属性仅为了演示此示例)

原则要求仓储不包含业务逻辑,上面的示例"**\*\*什么是非活动的问题?\*\***"这个属于业务规则吗?

````csharp

using System;

using System.Collections.Generic;

using System.Linq;

using System.Threading.Tasks;

using IssueTracking.EntityFrameworkCore;

using Microsoft.EntityFrameworkCore;

using Volo.Abp.Domain.Repositories.EntityFrameworkCore;

using Volo.Abp.EntityFrameworkCore;

namespace IssueTracking.Issues

{

public class EfCoreIssueRepository :

EfCoreRepository<IssueTrackingDbContext, Issue, Guid>,

IIssueRepository

{

public EfCoreIssueRepository(

IDbContextProvider<IssueTrackingDbContext> dbContextProvider)

: base(dbContextProvider)

{

}

public async Task<List<Issue>> GetInActiveIssuesAsync()

{

var daysAgo30 = DateTime.Now.Subtract(TimeSpan.FromDays(30));

return await DbSet.Where(i =>

//Open

!i.IsClosed &&

//Assigned to nobody

i.AssignedUserId == null &&

//Created 30+ days ago

i.CreationTime < daysAgo30 &&

//No comment or the last comment was 30+ days ago

(i.LastCommentTime == null || i.LastCommentTime < daysAgo30)

).ToListAsync();

}

}

}

````

(使用EF Core来实现. 如何使用EF Core实现仓储,请参见[EF Core集成文档](Entity-Framework-Core.md) )

来看一下`GetInActiveIssuesAsync`的实现,可以看到定义了一个**\*\*非活动问题的业务规则\*\***:

- 是*\*open\**的(非*\*IsClosed\** )

- 没有关联到任何人

- 创建时间大于30天

- 最近30天没有评论

这个业务逻辑就被实现在了仓储内部,当我们需要重用这个业务规则时就会出现问题.

例如:我们需要再实体`Issue`上添加一个方法来判断是否非活动`bool IsInActive()`,以方便我们在`Issue`实例上获取.

代码如下:

````csharp

public class Issue : AggregateRoot<Guid>, IHasCreationTime

{

public bool IsClosed { get; private set; }

public Guid? AssignedUserId { get; private set; }

public DateTime CreationTime { get; private set; }

public DateTime? LastCommentTime { get; private set; }

//...

public bool IsInActive()

{

var daysAgo30 = DateTime.Now.Subtract(TimeSpan.FromDays(30));

return

//Open

!IsClosed &&

//Assigned to nobody

AssignedUserId == null &&

//Created 30+ days ago

CreationTime < daysAgo30 &&

//No comment or the last comment was 30+ days ago

(LastCommentTime == null || LastCommentTime < daysAgo30);

}

}

````

我们需要拷贝代码来实现,如果将来业务规则发送变化,我们就必须修改这两处的代码,这样做非常危险.

这里有一个很好的解决方案,就是使用*\*规约模式\**.

### 规约模式

[规约](Specifications.md)是一种**\*\*强命名\*\***,**\*\*可重用\*\***,**\*\*可组合\*\***,**\*\*可测试\*\***的实体过滤器.

ABP框架提供了基础设施来轻松定义规约类,你可以在代码中方便使用.我们来将非活动问题使用规约方式实现:

````csharp

using System;

using System.Linq.Expressions;

using Volo.Abp.Specifications;

namespace IssueTracking.Issues

{

public class InActiveIssueSpecification : Specification<Issue>

{

public override Expression<Func<Issue, bool>> ToExpression()

{

var daysAgo30 = DateTime.Now.Subtract(TimeSpan.FromDays(30));

return i =>

//Open

!i.IsClosed &&

//Assigned to nobody

i.AssignedUserId == null &&

//Created 30+ days ago

i.CreationTime < daysAgo30 &&

//No comment or the last comment was 30+ days ago

(i.LastCommentTime == null || i.LastCommentTime < daysAgo30);

}

}

}

````

基类`Specification<T>`通过表达式简化了创建规约的过程,仅需要将仓储中的表达式迁移至规约中.

现在,我们可以在`Issue` 和 `EfCoreIssueRepository`中重用规约`InActiveIssueSpecification`了.

#### 在实体内使用规约

`Specification` 类提供了一个`IsSatisfiedBy`方法,在实例对象上应用规约检查,判断是否满足规约的要求.代码如下:

````csharp

public class Issue : AggregateRoot<Guid>, IHasCreationTime

{

public bool IsClosed { get; private set; }

public Guid? AssignedUserId { get; private set; }

public DateTime CreationTime { get; private set; }

public DateTime? LastCommentTime { get; private set; }

//...

public bool IsInActive()

{

return new InActiveIssueSpecification().IsSatisfiedBy(this);

}

}

````

实例化一个新的规约`InActiveIssueSpecification`实例,并通过`IsSatisfiedBy` 方法进行规约检查.

#### 在仓储内使用规约

首先,我们先修改一下仓储接口:

````csharp

public interface IIssueRepository : IRepository<Issue, Guid>

{

Task<List<Issue>> GetIssuesAsync(ISpecification<Issue> spec);

}

````

先将`GetInActiveIssuesAsync` 方法改名为`GetIssuesAsync` ,因为我们改为使用规约方式,现在就无需为不同的查询条件创建不同的接口方法(例如:`GetAssignedIssues(...)`,`GetLockedIssues(...)`)

再修改下仓储实现:

````csharp

public class EfCoreIssueRepository :

EfCoreRepository<IssueTrackingDbContext, Issue, Guid>,

IIssueRepository

{

public EfCoreIssueRepository(

IDbContextProvider<IssueTrackingDbContext> dbContextProvider)

: base(dbContextProvider)

{

}

public async Task<List<Issue>> GetIssuesAsync(ISpecification<Issue> spec)

{

return await DbSet

.Where(spec.ToExpression())

.ToListAsync();

}

}

````

由于`ToExpression()`方法返回一个表达式,因此可以直接将其传递给`Where`方法来过滤实体.

我们可以在调用`GetIssuesAsync`方法时,传递任何规约的实例.

````csharp

public class IssueAppService : ApplicationService, IIssueAppService

{

private readonly IIssueRepository \_issueRepository;

public IssueAppService(IIssueRepository issueRepository)

{

\_issueRepository = issueRepository;

}

public async Task DoItAsync()

{

var issues = await \_issueRepository.GetIssuesAsync(

new InActiveIssueSpecification()

);

}

}

````

##### 默认仓储使用规约

实际上,我们不必非要创建一个自定义仓储来使用规约方式,泛型仓储`IRepository`同样可以使用规约,因为`IRepository`已扩展了`IQueryable`对象,因此可以在泛型仓储上使用,代码如下:

````csharp

public class IssueAppService : ApplicationService, IIssueAppService

{

private readonly IRepository<Issue, Guid> \_issueRepository;

public IssueAppService(IRepository<Issue, Guid> issueRepository)

{

\_issueRepository = issueRepository;

}

public async Task DoItAsync()

{

var issues = AsyncExecuter.ToListAsync(

\_issueRepository.Where(new InActiveIssueSpecification())

);

}

}

````

`AsyncExecuter`是ABP框架提供的一个异步LINQ扩展方法(与`ToListAsync`类似),这个方法不依赖依赖EF Core,请参见[仓储文档](Repositories.md).

#### 组合规约

规约强大的能力就是可组合.假设我们还有一个业务规则:`Issue` 仅在里程碑中才返回`true`:

````csharp

public class MilestoneSpecification : Specification<Issue>

{

public Guid MilestoneId { get; }

public MilestoneSpecification(Guid milestoneId)

{

MilestoneId = milestoneId;

}

public override Expression<Func<Issue, bool>> ToExpression()

{

return i => i.MilestoneId == MilestoneId;

}

}

````

此规约与`InActiveIssueSpecification`的区别,它是有参数的.我们可以组合两种规约来实现获取指定里程碑下的非活动问题列表

````csharp

public class IssueAppService : ApplicationService, IIssueAppService

{

private readonly IRepository<Issue, Guid> \_issueRepository;

public IssueAppService(IRepository<Issue, Guid> issueRepository)

{

\_issueRepository = issueRepository;

}

public async Task DoItAsync(Guid milestoneId)

{

var issues = AsyncExecuter.ToListAsync(

\_issueRepository

.Where(

new InActiveIssueSpecification()

.And(new MilestoneSpecification(milestoneId))

.ToExpression()

)

);

}

}

````

上面的示例使用了`And`扩展方法来组合规约.还有更多的组合方法,如:`Or(...)`和`AndNot(...)`.

> 有关ABP框架提供的规约更多信息,请参见[规约文档](Specifications.md).

### 领域服务

领域服务主要用来实现本领域的逻辑:

\* 依赖**\*\*服务和仓储\*\***.

\* 需要使用多个聚合.

领域服务和领域对象一起使用.领域服务可以获取并返回**\*\*实体\*\***,**\*\*值对象\*\***等,它们不返回**\*\*DTO\*\***.DTO属于应用层的一部分.

**\*\*示例:用户关联一个问题\*\***

需要在`Issue`实体中实现问题的关联:

````csharp

public class Issue : AggregateRoot<Guid>

{

//...

public Guid? AssignedUserId { get; private set; }

public async Task AssignToAsync(AppUser user, IUserIssueService userIssueService)

{

var openIssueCount = await userIssueService.GetOpenIssueCountAsync(user.Id);

if (openIssueCount >= 3)

{

throw new BusinessException("IssueTracking:ConcurrentOpenIssueLimit");

}

AssignedUserId = user.Id;

}

public void CleanAssignment()

{

AssignedUserId = null;

}

}

````

现在我们把逻辑迁移到领域服务中实现.

首先,修改一下 `Issue` 类:

````csharp

public class Issue : AggregateRoot<Guid>

{

//...

public Guid? AssignedUserId { get; internal set; }

}

````

\* 删除关联的相关方法.

\* 修改属性 `AssignedUserId` 的 setter 为 `internal`,以允许领域服务可以修改.

下一步是创建一个名为`IssueManager`的领域服务,此领域服务的`AssignToAsync`方法负责将问题关联至指定的用户.

````csharp

public class IssueManager : DomainService

{

private readonly IRepository<Issue, Guid> \_issueRepository;

public IssueManager(IRepository<Issue, Guid> issueRepository)

{

\_issueRepository = issueRepository;

}

public async Task AssignToAsync(Issue issue, AppUser user)

{

var openIssueCount = await \_issueRepository.CountAsync(

i => i.AssignedUserId == user.Id && !i.IsClosed

);

if (openIssueCount >= 3)

{

throw new BusinessException("IssueTracking:ConcurrentOpenIssueLimit");

}

issue.AssignedUserId = user.Id;

}

}

````

`IssueManager`可以注入其它服务,来查询指定用户已经关联的未解决问题数量.

> 我们建议使用 `Manager` 后缀来命名领域服务.

这种设计的唯一缺陷是可以在类`Issue`外部修改`Issue.AssignedUserId`属性.但是它的访问级别是`internal`而非`public`,在`IssueTracking.Domain`项目内部才能被修改,我们认为这样是合理的:

\* 开发人员清楚领域层的开发规则,他们会使用`IssueManager`来执行业务逻辑.

\* 应用层开发人员只能使用`IssueManager`,因此他们无法直接修改实体属性.

尽管两种方式有各自的优势,但我们更喜欢创建领域服务并注入其它服务来执行业务逻辑这种方式.

### 应用服务

应用服务是实现**\*\*用例\*\***的无状态服务.应用服务通常**\*\*获取并返回DTO\*\***.应用服务被展现层所使用,应用服务**\*\*调用领域对象\*\***(实体,仓储等)来实现用例.

应用服务的通用原则:

\* 实现特定用例的**\*\*应用程序逻辑\*\***,不要在应用服务内实现核心领域的逻辑.

\* 应用服务的方法**\*\*不要返回实体\*\***.始终只返回DTO.

**\*\*示例:用户关联一个问题\*\***

````csharp

using System;

using System.Threading.Tasks;

using IssueTracking.Users;

using Microsoft.AspNetCore.Authorization;

using Volo.Abp.Application.Services;

using Volo.Abp.Domain.Repositories;

namespace IssueTracking.Issues

{

public class IssueAppService : ApplicationService, IIssueAppService

{

private readonly IssueManager \_issueManager;

private readonly IRepository<Issue, Guid> \_issueRepository;

private readonly IRepository<AppUser, Guid> \_userRepository;

public IssueAppService(

IssueManager issueManager,

IRepository<Issue, Guid> issueRepository,

IRepository<AppUser, Guid> userRepository)

{

\_issueManager = issueManager;

\_issueRepository = issueRepository;

\_userRepository = userRepository;

}

[Authorize]

public async Task AssignAsync(IssueAssignDto input)

{

var issue = await \_issueRepository.GetAsync(input.IssueId);

var user = await \_userRepository.GetAsync(input.UserId);

await \_issueManager.AssignToAsync(issue, user);

await \_issueRepository.UpdateAsync(issue);

}

}

}

````

应用服务的方法通常包含三个步骤:

1. 从数据库获取用例所需的领域对象.

2. 使用领域对象(领域服务,实体等)执行业务逻辑.

3. 将实体的变更持久化至数据库.

> 如果使用的是EF Core,第三步不是必须的,因为EF Core有追踪实体变化的功能.如果要利用此功能,请参阅上面的"关于数据库独立原则的讨论"章节.

`IssueAssignDto` 是本示例中一个简单的DTO对象:

````csharp

using System;

namespace IssueTracking.Issues

{

public class IssueAssignDto

{

public Guid IssueId { get; set; }

public Guid UserId { get; set; }

}

}

````

### 数据传输对象

[DTO](Data-Transfer-Objects.md)是应用层与展现层间传输数据的简单对象.应用服务方法获取并返回Dto.

#### DTO通用原则和最佳实践

\* DTO应该是**\*\*可被序列化\*\***的.因为大所数情况下,DTO是通过网络传输的,因此它应该具有**\*\*无参的构造函数\*\***.

\* 不应该包含任何**\*\*业务逻辑\*\***.

\* **\*\*切勿\*\***继承或引用**\*\*实体\*\***.

**\*\*输入DTO\*\***(应用服务方法的参数)与 **\*\*输出DTO\*\*** (应用服务方法的返回对象)具有不同的作用,因此,它们应该区别对待.

#### 输入DTO 最佳实践

##### 不要在输入DTO中定义不使用的属性

**\*\*仅\*\***在输入DTO中定义用例**\*\*所需要的属性\*\***！否则,会造成调用应用服务的客户端产生困惑.

这个规则好像没什么必要,因为没人会在方法参数(输入DTO)中添加无用的属性.但是,有时候,特别是在重用DTO时,输入DTO会包含无用的属性.

##### 不要重用输入DTO

**\*\*为每个用例\*\***(应用服务的方法)单独定义一个**\*\*专属的输入DTO\*\***.否则,在一些情况下,会添加一些不被使用的属性,这样就违反上面的规则:不要在输入DTO中定义不使用的属性.

在两个用例中重用相同的DTO似乎很有吸引力,因为它们的属性是一模一样的.现阶段它们是一样的,但是随着业务变化,可能它们会产生差异,届时你可能还是需要进行拆分.**\*\*和用例间的耦合相比,代码的复制可能是更好的做法\*\***.

重用输入DTO的另外一种方式是**\*\*继承\*\***DTO,这同样会产生上面描述的问题.

**\*\*示例:用户应用服务\*\***

````csharp

public interface IUserAppService : IApplicationService

{

Task CreateAsync(UserDto input);

Task UpdateAsync(UserDto input);

Task ChangePasswordAsync(UserDto input);

}

````

`UserDto`作为`IUserAppService`所有方法的输入DTO,代码如下:

````csharp

public class UserDto

{

public Guid Id { get; set; }

public string UserName { get; set; }

public string Email { get; set; }

public string Password { get; set; }

public DateTime CreationTime { get; set; }

}

````

对于上面的示例:

\* `Id` 属性在 *\*Create\** 方法中,没有被使用,因为`Id`由服务器生成.

\* `Password` 属性在 *\*Update\** 方法中,没有被使用.因为有修改密码的单独方法.

\* `CreationTime` 属性未被使用,因为不允许客户端发送创建时间属性,这个应该由服务器生成.

较好的做法应该这样:

````csharp

public interface IUserAppService : IApplicationService

{

Task CreateAsync(UserCreationDto input);

Task UpdateAsync(UserUpdateDto input);

Task ChangePasswordAsync(UserChangePasswordDto input);

}

````

下面是输入DTO的定义:

````csharp

public class UserCreationDto

{

public string UserName { get; set; }

public string Email { get; set; }

public string Password { get; set; }

}

public class UserUpdateDto

{

public Guid Id { get; set; }

public string UserName { get; set; }

public string Email { get; set; }

}

public class UserChangePasswordDto

{

public Guid Id { get; set; }

public string Password { get; set; }

}

````

虽然编写了更多的代码,但是这样可维护性更高.

**\*\*例外情况:\*\*** 该规则有一些例外的情况,例如,你想开发两个方法,它们共用相同的输入DTO(通过继承或重用),有一个报表页面有多个过滤条件,多个应用服务使用相同的输入参数返回不同的结果(如,大屏展示数据,Excel报表,csv报表).这种情况下,你是需要修改一个参数,多个应用服务都应该一起被修改.

##### 输入DTO中验证逻辑

- 仅在DTO内执行**\*\*简单验证\*\***.使用数据注解验证属性或通过`IValidatableObject` 方式.

- **\*\*不要执行领域验证\*\***.例如,不要在DTO中检查用户名是否唯一的验证.

**\*\*示例:使用注解方式\*\***

````csharp

using System.ComponentModel.DataAnnotations;

namespace IssueTracking.Users

{

public class UserCreationDto

{

[Required]

[StringLength(UserConsts.MaxUserNameLength)]

public string UserName { get; set; }

[Required]

[EmailAddress]

[StringLength(UserConsts.MaxEmailLength)]

public string Email { get; set; }

[Required]

[StringLength(UserConsts.MaxEmailLength,

MinimumLength = UserConsts.MinPasswordLength)]

public string Password { get; set; }

}

}

````

当输入无效时,ABP框架会自动验证输入DTO,抛出`AbpValidationException`异常,并向客户返回`400`的HTTP状态码.

> 一些开发人员认为最好将验证规则和DTO分离.我们认为声明性(数据注解)方式是比较实用的,不会引起任何设计问题.如果你喜欢其它方式,ABP还支持[FluentValidation继承](FluentValidation.md).有关所有验证的详细文档,请参见[验证文档](Validation.md).

#### 输出DTO最佳实践

\* 保持**\*\*数量较少\*\***的输出DTO,尽可能**\*\*重用输出DTO\*\***(例外:不要将输入DTO作为输出DTO).

\* 输出DTO可以包含比用例需要的属性**\*\*更多\*\***的属性.

\* 针对 **\*\*Create\*\*** 和 **\*\*Update\*\*** 方法,返回实体的DTO.

以上建议的原因是:

\* 使客户端代码易于开发和扩展:

\* 客户端处理**\*\*相似但不相同\*\***的DTO是有问题的.

\* 将来UI或客户端通常会使用到DTO上的**\*\*其它属性\*\***.返回实体的所有属性,可以在无需修改服务端代码的情况下,只修改客户端代码.

\* 在开放API给**\*\*第三方客户端\*\***时,避免不同需求的返回不同的DTO.

\* 使服务器端代码易于开发和扩展:

\* 你需要**\*\*维护\*\***的类的数量较少.

\* 你可以重用Entity->DTO**\*\*对象映射\*\***的代码.

\* 不同的方法返回相同的类型,可以使得在**\*\*添加新方法\*\***时变的简单明了.

**\*\*示例:不同的方法返回不同的DTO\*\***

````csharp

public interface IUserAppService : IApplicationService

{

UserDto Get(Guid id);

List<UserNameAndEmailDto> GetUserNameAndEmail(Guid id);

List<string> GetRoles(Guid id);

List<UserListDto> GetList();

UserCreateResultDto Create(UserCreationDto input);

UserUpdateResultDto Update(UserUpdateDto input);

}

````

> 这里我们没有使用异步方式,是为了示例更清晰,你实际代码中应该使用异步方式)

上面的示例代码中,每个方法都返回了不同的DTO类型,这样处理,会导致查询数据,映射对象都会有很多重复的代码.

应用服务`IUserAppService` 可以简化成如下代码:

````csharp

public interface IUserAppService : IApplicationService

{

UserDto Get(Guid id);

List<UserDto> GetList();

UserDto Create(UserCreationDto input);

UserDto Update(UserUpdateDto input);

}

````

只需使用一个DTO对象

````csharp

public class UserDto

{

public Guid Id { get; set; }

public string UserName { get; set; }

public string Email { get; set; }

public DateTime CreationTime { get; set; }

public List<string> Roles { get; set; }

}

````

\* 删除 `GetUserNameAndEmail`和`GetRoles` 方法,因为,返回的DTO中已经包含了对应的信息.

\* `GetList`方法的返回的泛型类型与`Get`方法的返回类型一致.

\* `Create` 与 `Update`的返回类型都是 `UserDto`.

如上所述,使用相同的DTO有很多优点.例如,我们在UI上使用**\*\*表格\*\***展现用户集合,再用户数据更新后,我们可以获取到返回对象,并对**\*\*表格数据源进行更新\*\***.因此,我们无需再次调用`GetList`来获取全部数据.这就是我们为什么建议`Create` 与 `Update`方法都返回相同`UserDto`的原因.

##### 讨论

输出DTO的建议并不适用于所有情况.出于**\*\*性能\*\***原因,我们可以忽略这些建议,尤其是在返回**\*\*大量数据\*\***,为UI定制,**\*\*并发量较高\*\***时.

在这些情况下,你可以定制仅包含**\*\*必要信息的DTO\*\***.上面的建议只适用于额外多些属性并**\*\*不会损失太多性能\*\***,并关注代码**\*\*可维护\*\***的应用系统.

#### 对象映射到对象

当两个对象具有相同或相似的属性,自动将[对象映射到对象](Object-To-Object-Mapping.md)是一种将值从一个对象复制到另外一个对象非常有用的方法.

DTO和实体通常具有相同或相似的属性,你经常需要从一个实体创建一个DTO对象.相较于手动映射,基于[AutoMapper](http://automapper.org/)的ABP[对象映射系统](Object-To-Object-Mapping.md),更加方便简单.

\* **\*\*仅\*\***在**\*\*实体=>输出DTO\*\***的时候使用自动映射.

\* 不要在**\*\*输入DTO=>Entity\*\***的时候使用自动映射.

因为以下原因,你不应该在输入DTO=>Entity的时候使用自动映射:

1. 实体类通常具有一个**\*\*构造函数\*\***,该构造函数带有参数,确保创建有效的对象,而自动对象映射通常需要一个无参构造函数.

2. 大多数实体中的属性setter是**\*\*私有的\*\***,你只能调用实体上的方法来修改属性.

3. 另外,需要进行对用户或客户端的**\*\*输入参数进行验证\*\***,而不是盲目映射到实体属性上.

尽管其中一些问题可以额外配置映射来解决(如,AutoMapper允许自定义映射规则),但这样会使业务逻辑被**\*\*耦合到基础设施代码中\*\***.我们认为业务代码应该明确,清晰且易于理解.

有关此部分的建议,请参加下面的"*\*实体创建\**"部分

## 用例

本节将演示一些用例,并讨论替代方案

### 实体创建

实体或聚合根的创建,是实体生命周期的开始."*\*聚合/聚合根规则及最佳实践\**"章节中建议为Entity类定义**\*\*一个主构造函数\*\***,以确保创建一个**\*\*有效的实体\*\***.因此,需要创建该实体对象实例时,都应该**\*\*使用该构造函数\*\***.

`Issue` 聚合根的代码如下:

````csharp

public class Issue : AggregateRoot<Guid>

{

public Guid RepositoryId { get; private set; }

public string Title { get; private set; }

public string Text { get; set; }

public Guid? AssignedUserId { get; internal set; }

public Issue(

Guid id,

Guid repositoryId,

string title,

string text = null

) : base(id)

{

RepositoryId = repositoryId;

Title = Check.NotNullOrWhiteSpace(title, nameof(title));

Text = text; //Allow empty/null

}

private Issue() { /\* Empty constructor is for ORMs \*/ }

public void SetTitle(string title)

{

Title = Check.NotNullOrWhiteSpace(title, nameof(title));

}

//...

}

````

\* 通过其非空参数的构造函数创建有效的实体.

\* 如需修改 `Title` 属性,必须通过`SetTitle` 方法,来确保被设置值的有效性.

\* 如需将此问题关联至用户,则需要使用`IssueManager`(关联前需要执行一些业务逻辑,相关逻辑参见上面的"*\*领域服务\**"部分)

\* `Text` 属性setter是公开的,因为它可以为null,并且本示例中也没有验证规则,它在构造函数中也是可选的.

创建问题的应用服务代码:

````csharp

public class IssueAppService : ApplicationService, IIssueAppService

{

private readonly IssueManager \_issueManager;

private readonly IRepository<Issue, Guid> \_issueRepository;

private readonly IRepository<AppUser, Guid> \_userRepository;

public IssueAppService(

IssueManager issueManager,

IRepository<Issue, Guid> issueRepository,

IRepository<AppUser, Guid> userRepository)

{

\_issueManager = issueManager;

\_issueRepository = issueRepository;

\_userRepository = userRepository;

}

public async Task<IssueDto> CreateAsync(IssueCreationDto input)

{

// Create a valid entity

var issue = new Issue(

GuidGenerator.Create(),

input.RepositoryId,

input.Title,

input.Text

);

// Apply additional domain actions

if (input.AssignedUserId.HasValue)

{

var user = await \_userRepository.GetAsync(input.AssignedUserId.Value);

await \_issueManager.AssignToAsync(issue, user);

}

// Save

await \_issueRepository.InsertAsync(issue);

// Return a DTO represents the new Issue

return ObjectMapper.Map<Issue, IssueDto>(issue);

}

}

````

`CreateAsync` 方法;

\* 使用 `Issue` **\*\*构造函数\*\*** 创建一个有效的问题.`Id` 属性通过[IGuidGenerator](Guid-Generation.md)服务生成.此处没有使用对象自动映射.

\* 如果需要将**\*\*问题关联至用户\*\***,则通过 `IssueManager`来执行关联逻辑.

\* **\*\*保存\*\*** 实体至数据库.

\* 最后,使用 `IObjectMapper` 将`Issue`实体**\*\*映射\*\***为 `IssueDto` 并返回.

#### 在创建实体时执行领域规则

`Issue`除了在构造函数中进行了一些简单验证外,示例中没其它业务验证.在有些情况下,在创建实体时会有一些其它业务规则.

假如,已经存在一个完全相同的问题,那么就不要再创建问题.这个规则应该在哪里执行?在**\*\*应用服务中执行是不对的\*\***,因为它是**\*\*核心业务(领域)的规则\*\***,应该将此规则在领域服务中执行.在这种情况下,我们应该在`IssueManager`中执行此规则,因此应该强制应用服务调用领域服务`IssueManager`来新建`Issue`.

首先修改 `Issue` 构造函数的访问级别为 `internal`:

````csharp

public class Issue : AggregateRoot<Guid>

{

//...

internal Issue(

Guid id,

Guid repositoryId,

string title,

string text = null

) : base(id)

{

RepositoryId = repositoryId;

Title = Check.NotNullOrWhiteSpace(title, nameof(title));

Text = text; //Allow empty/null

}

//...

}

````

这样可以防止,应用服务直接使用`Issue` 的构造函数去创建`Issue` 实例,必须使用 `IssueManager`来创建.然后我们再添加一个`CreateAsync`方法:

````csharp

using System;

using System.Threading.Tasks;

using Volo.Abp;

using Volo.Abp.Domain.Repositories;

using Volo.Abp.Domain.Services;

namespace IssueTracking.Issues

{

public class IssueManager : DomainService

{

private readonly IRepository<Issue, Guid> \_issueRepository;

public IssueManager(IRepository<Issue, Guid> issueRepository)

{

\_issueRepository = issueRepository;

}

public async Task<Issue> CreateAsync(

Guid repositoryId,

string title,

string text = null)

{

if (await \_issueRepository.AnyAsync(i => i.Title == title))

{

throw new BusinessException("IssueTracking:IssueWithSameTitleExists");

}

return new Issue(

GuidGenerator.Create(),

repositoryId,

title,

text

);

}

}

}

````

\* `CreateAsync` 方法会检查标题是否已经存在,当有相同标题的问题时,会抛出业务异常.

\* 如果标题没有重复的,则创建并返回一个新的 `Issue`对象.

再修改`IssueAppService` 的代码,来调用 `IssueManager`的 `CreateAsync` 方法:

````csharp

public class IssueAppService : ApplicationService, IIssueAppService

{

private readonly IssueManager \_issueManager;

private readonly IRepository<Issue, Guid> \_issueRepository;

private readonly IRepository<AppUser, Guid> \_userRepository;

public IssueAppService(

IssueManager issueManager,

IRepository<Issue, Guid> issueRepository,

IRepository<AppUser, Guid> userRepository)

{

\_issueManager = issueManager;

\_issueRepository = issueRepository;

\_userRepository = userRepository;

}

public async Task<IssueDto> CreateAsync(IssueCreationDto input)

{

// Create a valid entity using the IssueManager

var issue = await \_issueManager.CreateAsync(

input.RepositoryId,

input.Title,

input.Text

);

// Apply additional domain actions

if (input.AssignedUserId.HasValue)

{

var user = await \_userRepository.GetAsync(input.AssignedUserId.Value);

await \_issueManager.AssignToAsync(issue, user);

}

// Save

await \_issueRepository.InsertAsync(issue);

// Return a DTO represents the new Issue

return ObjectMapper.Map<Issue, IssueDto>(issue);

}

}

// \*\*\* IssueCreationDto class \*\*\*

public class IssueCreationDto

{

public Guid RepositoryId { get; set; }

[Required]

public string Title { get; set; }

public Guid? AssignedUserId { get; set; }

public string Text { get; set; }

}

````

##### 讨论:为什么`IssueManager`中没有执行`Issue`的保存?

你可能会问"**\*\*为什么**`IssueManager`**中没有执行**`Issue`**的保存?\*\***".我们认为这是应用服务的职责.

因为,应用服务可能在保存`Issue`对象之前,需要对其它对象进行修改.如果领域服务执行了保存,那么*\*保存\**操作就是重复的.

\* 会触发两次数据库交互,这会导致性能损失.

\* 需要额外添加显式的事务来包含这两个操作,才能保证数据一致性.

\* 如果因为业务规则取消了实体的创建,则应该在数据库事务中回滚事务,取消所有操作.

假如在`IssueManager.CreateAsync`中先保存一次数据,那么数据会先执行一次*\*Insert\**操作,后面关联用户的逻辑执行后,又会再执行一次*\*Update\**操作.

如果不在`IssueManager.CreateAsync`中保存数据,那么,新建`Issue`和关联用户,只会执行一次*\*Insert\**操作.

##### 讨论:为什么没有在应用服务中执行标题是否重复的检查?

简单地说"因为它是**\*\*核心领域逻辑\*\***,应该在领域层实现".这又带来一个新问题,"**\*\*如何确定\*\***是领域层逻辑,还是应用层逻辑"?(这个我们后面再详细讨论)

对于此示例,可以用一个简单的问题来判断到底是领域逻辑还是应用逻辑:"如果还有另外一种创建`Issue`的方式(用例),我们是否还需要执行?如果需要执行,就属于领域层逻辑,不需要执行就是应用层逻辑".你可能认为为什么还有别的用例来创建`Issue`呢?

\* 应用程序的**\*\*最终用户\*\***可能会在UI上创建`Issue`.

\* 系统内部人员,可以在**\*\*后台管理\*\***端采用另外一种方式创建`Issue`(这种情况下,可能使用不同的业务规则).

\* 对**\*\*第三方客户端\*\***开放的API,它们的规则又有所不同.

\* 还有**\*\*后台作业系统\*\***会执行某些操作时创建`Issue`,这样,它是在没有任何用户交互情况下创建`Issue`.

\* 还有可能是UI上某个按钮,可以将某些内容(例如,讨论)转为`Issue`.

我们还可以举更多例子.所有这些都应该通过**\*\*不同的应用服务方法来实现\*\***(请参见下面的"*\*多个应用服务层\**"部分),但是它们**\*\*始终遵循\*\***以下的规则:

新的问题标题不能与任何已有的问题标题相同.这就是为什么说的"*\*标题是否重复的检查\**"属于核心领域逻辑的原因,这个逻辑应该在领域层,而**\*\*不应该\*\***在应用层的所有方法中**\*\*去重复\*\***定义.

### 修改实体

创建实体后,将根据用例对实体进行修改,直到将其从系统中删除.可以有不同的用例直接或间接的修改实体.

在本节中,我们将讨论一种典型的修改操作,该操作会修改`Issue`的多个属性.

从*\*Update\** DTO开始:

````csharp

public class UpdateIssueDto

{

[Required]

public string Title { get; set; }

public string Text { get; set; }

public Guid? AssignedUserId { get; set; }

}

````

对比`IssueCreationDto`,可以发现,缺少了`RepositoryId`属性,因为我们不允许跨仓库移动`Issue`.仅`Title`属性是必填的.

`IssueAppService`中*\*Update\**的实现如下::

````csharp

public class IssueAppService : ApplicationService, IIssueAppService

{

private readonly IssueManager \_issueManager;

private readonly IRepository<Issue, Guid> \_issueRepository;

private readonly IRepository<AppUser, Guid> \_userRepository;

public IssueAppService(

IssueManager issueManager,

IRepository<Issue, Guid> issueRepository,

IRepository<AppUser, Guid> userRepository)

{

\_issueManager = issueManager;

\_issueRepository = issueRepository;

\_userRepository = userRepository;

}

public async Task<IssueDto> UpdateAsync(Guid id, UpdateIssueDto input)

{

// Get entity from database

var issue = await \_issueRepository.GetAsync(id);

// Change Title

await \_issueManager.ChangeTitleAsync(issue, input.Title);

// Change Assigned User

if (input.AssignedUserId.HasValue)

{

var user = await \_userRepository.GetAsync(input.AssignedUserId.Value);

await \_issueManager.AssignToAsync(issue, user);

}

// Change Text (no business rule, all values accepted)

issue.Text = input.Text;

// Update entity in the database

await \_issueRepository.UpdateAsync(issue);

// Return a DTO represents the new Issue

return ObjectMapper.Map<Issue, IssueDto>(issue);

}

}

````

\* `UpdateAsync` 方法参数 `id`被作为独立参数,放置在`UpdateIssueDto`之外.这是一项设计决策,当你将此应用服务[自动导出](API/Auto-API-Controllers.md)为HTTP API时,API端点时帮助ABP正确定义HTTP路由,这与DDD无关.

\* 首先从数据库中**\*\*获取\*\*** `Issue` 实体.

\* 通过 `IssueManager`的 `ChangeTitleAsync`方法修改标题,而非直接通过 `Issue.SetTitle(...)`直接修改.因为我们需要像创建时那样,**\*\*执行标题的重复检查逻辑\*\***.这需要对`Issue`类和`IssueManager`类进行一些调整(将在下面说明).

\* 通过 `IssueManager`的 `AssignToAsync` 方法来**\*\*关联用户\*\***.

\* 直接设置 `Issue.Text`属性,因为它本身没有任何业务逻辑需要执行.如果以后需要可以再进行重构.

\* **\*\*保存修改\*\***至数据库.同样,保存修改后的实体属于应用服务的职责,它可以协调业务对象和事务.如果在`IssueManager`内部的 `ChangeTitleAsync` 和 `AssignToAsync` 方法中进行保存,则会导致两次数据库操作(请参见上面的*\*讨论:为什么`IssueManager`中没有执行`Issue`的保存?\**)

\* 最后,使用 `IObjectMapper` 将`Issue`实体**\*\*映射\*\***为 `IssueDto` 并返回.

如前所述,我们需要对`Issue`类和`IssueManager`类进行一些调整:

首先,修改 `SetTitle`方法的访问级别为internal:

````csharp

internal void SetTitle(string title)

{

Title = Check.NotNullOrWhiteSpace(title, nameof(title));

}

````

再在`IssueManager`中添加一个新方法来修改标题:

````csharp

public async Task ChangeTitleAsync(Issue issue, string title)

{

if (issue.Title == title)

{

return;

}

if (await \_issueRepository.AnyAsync(i => i.Title == title))

{

throw new BusinessException("IssueTracking:IssueWithSameTitleExists");

}

issue.SetTitle(title);

}

````

## 领域逻辑和应用逻辑

如前所述,领域驱动设计中的*\*业务逻辑\**分为两部分(各层):领域逻辑和应用逻辑

![domain-driven-design-domain-vs-application-logic](images/domain-driven-design-domain-vs-application-logic.png)

领域逻辑是系统的*\*核心领域规则\**组成,而应用逻辑则满足特定的*\*用例\**.

虽然定义很明确,但是实施起来却并不容易.你可能无法确定哪些代码应该属于领域层,哪些代码应该属于应用层,本节会尝试解释差异.

### 多应用层

当你的系统很大时,DDD有助于**\*\*处理复杂问题\*\***.尤其是,**\*\*单个领域\*\***需要多个**\*\*应用程序运行\*\***,那么**\*\*领域逻辑与应用逻辑分离\*\***就变的非常重要.

假设你正在构建一个具有多个应用程序的系统:

\* 一个**\*\*公开的应用网站\*\***,使用ASP.NET Core MVC构建,展示商品给来访者.这样的网站不需要身份验证即可查看商品.来访者只有执行了某些操作(例如,将商品添加到购物车)后,才需要登录网站.

\* 一个**\*\*后台管理系统\*\***,UI使用Angular,通过REST API请求数据.内部员工使用这个系统来维护数据(例如,编辑商品说明).

\* 一个**\*\*移动端应用程序\*\***,它比公开的网站UI上更加简洁.它通过REST API或其它技术(例如,TCP sockets)请求数据.

![domain-driven-design-multiple-applications](images/domain-driven-design-multiple-applications.png)

每个应用程序都有不同的**\*\*需求\*\***,不同的**\*\*用例\*\***(应用服务方法),不同的DTO,不同的**\*\*验证\*\***和**\*\*授权\*\***规则等.

将所有这些逻辑都集中到一个应用层中,会使你的服务包含太多的`if`条件分支及**\*\*复杂的业务逻辑\*\***,从而使你的代码开发,**\*\*维护\*\***,测试,引发各种问题.

如果你在一个领域中有多个应用程序

- 为每种应用程序或客户端创建独立的应用层,并在这些单独层中执行特定于应用业务逻辑.

- 使用共享的核心领域逻辑

为了实现这样的设计,首先我们需要区分领域逻辑和应用逻辑.

为了更清楚的实现,你可以为不同的应用类型创建不同的项目(`.csproj`):

\* `IssueTracker.Admin.Application` 和 `IssueTracker.Admin.Application.Contracts` 为后台管理系统提供服务.

\* `IssueTracker.Public.Application` 和 `IssueTracker.Public.Application.Contracts` 为公开网站提供服务.

\* `IssueTracker.Mobile.Application` 和 `IssueTracker.Mobile.Application.Contracts` 为移动端应用提供服务.

### 示例

本节包含一些应用服务及领域服务的示例,讨论业务逻辑应该放置在哪一层

**\*\*示例:在领域服务中创建**`Organization`**\*\***

````csharp

public class OrganizationManager : DomainService

{

private readonly IRepository<Organization> \_organizationRepository;

private readonly ICurrentUser \_currentUser;

private readonly IAuthorizationService \_authorizationService;

private readonly IEmailSender \_emailSender;

public OrganizationManager(

IRepository<Organization> organizationRepository,

ICurrentUser currentUser,

IAuthorizationService authorizationService,

IEmailSender emailSender)

{

\_organizationRepository = organizationRepository;

\_currentUser = currentUser;

\_authorizationService = authorizationService;

\_emailSender = emailSender;

}

public async Task<Organization> CreateAsync(string name)

{

if (await \_organizationRepository.AnyAsync(x => x.Name == name))

{

throw new BusinessException("IssueTracking:DuplicateOrganizationName");

}

await \_authorizationService.CheckAsync("OrganizationCreationPermission");

Logger.LogDebug($"Creating organization {name} by {\_currentUser.UserName}");

var organization = new Organization();

await \_emailSender.SendAsync(

"systemadmin@issuetracking.com",

"New Organization",

"A new organization created with name: " + name

);

return organization;

}

}

````

我们来逐个检查`CreateAsync`方法中的代码,讨论是否应该在领域服务中

\* **\*\*正确\*\***:首先检查有**\*\*无重复的组织名称\*\***,并抛出异常.这与核心领域规则有关,因为我们绝对不允许重复的名称.

\* **\*\*错误\*\***:领域服务不应该执行**\*\*授权检查\*\***,[授权](Authorization.md)应该在应用层处理.

\* **\*\*错误\*\***:它记录了日志,包括[当前用户](CurrentUser.md)的`UserName`.领域服务不应该依赖当前用户,即便系统中没有用户,领域服务也应可用.当前用户应该是与展现层或应用层有关的概念.

\* **\*\*错误\*\***:它发送了有关新组织被创建的[邮件](Emailing.md),我们认为这也是特定用例的业务逻辑,你可能像在不同的用例中创建不同的邮件,又或者某些情况无需发送邮件.

**\*\*示例:应用服务中创建**`Organization`**\*\***

````csharp

public class OrganizationAppService : ApplicationService

{

private readonly OrganizationManager \_organizationManager;

private readonly IPaymentService \_paymentService;

private readonly IEmailSender \_emailSender;

public OrganizationAppService(

OrganizationManager organizationManager,

IPaymentService paymentService,

IEmailSender emailSender)

{

\_organizationManager = organizationManager;

\_paymentService = paymentService;

\_emailSender = emailSender;

}

[UnitOfWork]

[Authorize("OrganizationCreationPermission")]

public async Task<Organization> CreateAsync(CreateOrganizationDto input)

{

await \_paymentService.ChargeAsync(

CurrentUser.Id,

GetOrganizationPrice()

);

var organization = await \_organizationManager.CreateAsync(input.Name);

await \_organizationManager.InsertAsync(organization);

await \_emailSender.SendAsync(

"systemadmin@issuetracking.com",

"New Organization",

"A new organization created with name: " + input.Name

);

return organization; // !!!

}

private double GetOrganizationPrice()

{

return 42; //Gets from somewhere else...

}

}

````

我们来逐个检查`CreateAsync`方法中的代码,讨论是否应该在应用服务中

\* **\*\*正确\*\***:应用服务的方法应该是一个工作单元(事务).ABP的[工作单元](Unit-Of-Work.md)系统可以使得此工作自动进行(甚至无需`[UnitOfWork]`注解).

\* **\*\*正确\*\***: [授权](Authorization.md)应该在应用层处理.这里通过使用`[Authorize]`来完成.

\* **\*\*正确\*\***:调用付款(基础设施服务)为此操作收取费用(创建组织是我们业务中的付费服务).

\* **\*\*正确\*\***:应用服务负责将变更的数据保存到数据库.

\* **\*\*正确\*\***:我们可以将[邮件](Emailing.md)作为通知发送给管理员.

\* **\*\*错误\*\***:请勿从应用服务中返回实体,应该返回DTO.

**\*\*讨论:为什么不将支付逻辑移到领域服务中?\*\***

你可能想知道为什么付款逻辑代码不在`OrganizationManager`中.付款是非常**\*\*重要的事情\*\***,我们不能**\*\*遗漏任何一次付款\*\***.

它确实非常重要,但是,它不能放到领域服务中.我们可能还有**\*\*其它用例\*\***来创建组织但不收取任何费用.例如:

\* 管理员可以在后台管理系统创建新组织,而无需支付任何费用.

\* 后台作业系统导入,集成,同步组织而无需支付费用.

如你所见,**\*\*付款不是创建有效组织的必要操作\*\***.它是特定的应用服务逻辑.

**\*\*示例:CRUD操作\*\***

````csharp

public class IssueAppService

{

private readonly IssueManager \_issueManager;

public IssueAppService(IssueManager issueManager)

{

\_issueManager = issueManager;

}

public async Task<IssueDto> GetAsync(Guid id)

{

return await \_issueManager.GetAsync(id);

}

public async Task CreateAsync(IssueCreationDto input)

{

await \_issueManager.CreateAsync(input);

}

public async Task UpdateAsync(UpdateIssueDto input)

{

await \_issueManager.UpdateAsync(input);

}

public async Task DeleteAsync(Guid id)

{

await \_issueManager.DeleteAsync(id);

}

}

````

该应用服务本身**\*\*不执行任何操作\*\***,并将所有**\*\*操作转发给\*\*** *\*领域服务\**.它甚至将DTO传递给`IssueManager`

\* 如果没有**\*\*任何业务逻辑\*\***,只有简单的**\*\*CRUD\*\***操作,**\*\*请勿\*\***创建领域服务.

\* **\*\*切勿\*\***将**\*\*DTO\*\***传递给领域服务,或从领域服务返回**\*\*DTO\*\***.

可以在应用服务中直接注入仓储,实现查询,创建,更新及删除操作.除非在这些操作过程中需要执行某些领域逻辑,在这种情况下,请创建领域服务.

> 不要创建"将来可能需要"这种CRUD领域服务方法([YAGNI](https://en.wikipedia.org/wiki/You\_aren%27t\_gonna\_need\_it)),在需要时重构它并重构现有代码. 由于应用层优雅地抽象了领域层,因此重构过程不会影响UI层和其他客户端.

## 相关书籍

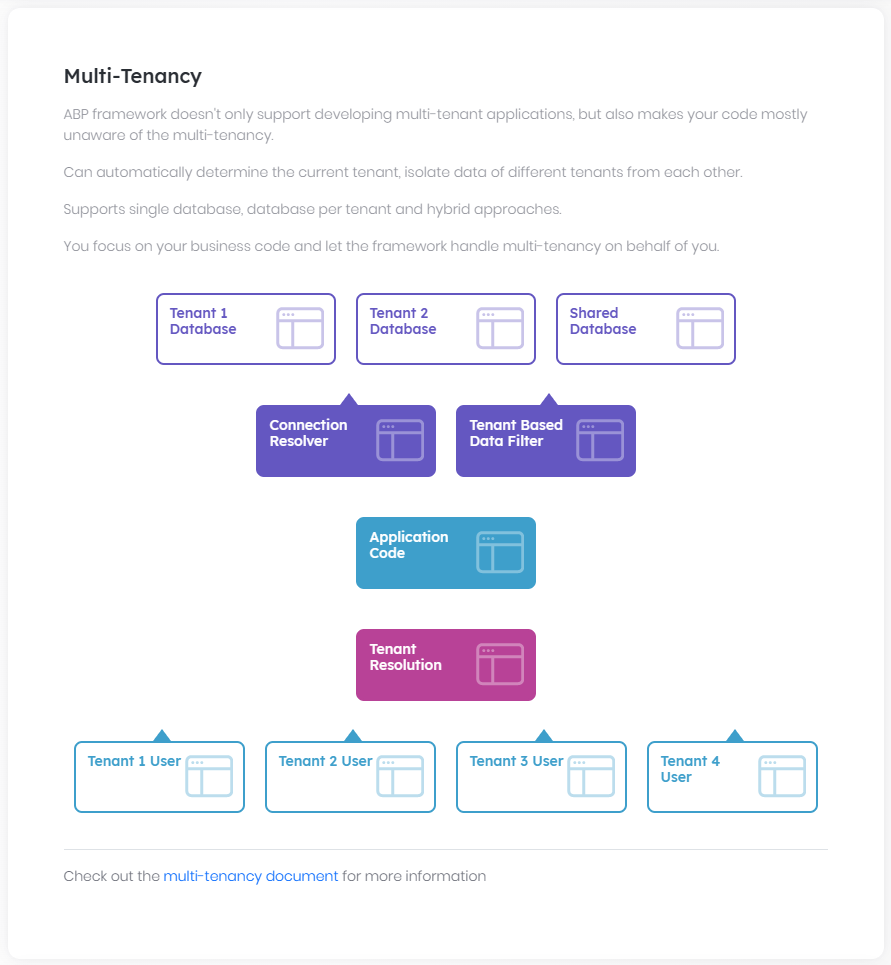
如果你对领域驱动设计和构建大型系统有兴趣,建议将以下书籍作为参考书籍:

\* "*\*Domain Driven Design\**" by Eric Evans

\* "*\*Implementing Domain Driven Design\**" by Vaughn Vernon

\* "*\*Clean Architecture\**" by Robert C. Martin

## 多租户



## 多租户

ABP的多租户模块提供了创建多租户应用程序的基本功能.

维基百科中是这样[定义](https://en.wikipedia.org/wiki/Multitenancy)多租户的:

> 软件多租户技术指的是一种软件架构,这种架构可以使用软件的单实例运行并为多个租户提供服务.租户是通过软件实例的特定权限共享通用访问的一组用户.使用多租户架构,软件应用为每个租户提供实例的专用共享,包括实例的数据、配置、用户管理、租户的私有功能和非功能属性.多租户与多实例架构形成对比,将软件实例的行为根据不同的租户分割开来.

### Volo.Abp.MultiTenancy

Volo.Abp.MultiTenancy"multi-tenancy ready",使用包管理器控制台(PMC)将它安装到你的项目中:

````

Install-Package Volo.Abp.MultiTenancy

````

> 这个包默认安装在了快速启动模板中.所以,大多数情况下,你不需要手动安装它.

然后你可以添加 **\*\*AbpMultiTenancyModule\*\*** 依赖到你的模块:

````C#

using Volo.Abp.Modularity;

using Volo.Abp.MultiTenancy;

namespace MyCompany.MyProject

{

[DependsOn(typeof(AbpMultiTenancyModule))]

public class MyModule : AbpModule

{

//...

}

}

````

> 随着"Multi-tenancy ready"的概念,我们打算开发我们的代码和多租户方法兼容.然后它可以被用于多租户和非多租户的程序中,这取决于最终程序的需求.

### AbpMultiTenancyOptions: 处理不活跃或不存在的租户

`MultiTenancyMiddlewareErrorPageBuilder` 或 `AbpMultiTenancyOptions` 用于 处理不活跃或不存在的租户.

默认情况下会响应错误页面, 你可以根据自己的需要更改它, 比如: 只输出错误日志并继续ASP NET Core的请求管道

```csharp

Configure<AbpMultiTenancyOptions>(options =>

{

options.MultiTenancyMiddlewareErrorPageBuilder = async (context, exception) =>

{

// Handle the exception.

};

});

```

#### 定义实体

你可以在你的实体中实现 **\*\*IMultiTenant\*\*** 接口来实现多租户,例如:

````C#

using System;

using Volo.Abp.Domain.Entities;

using Volo.Abp.MultiTenancy;

namespace MyCompany.MyProject

{

public class Product : AggregateRoot, IMultiTenant

{

public Guid? TenantId { get; set; } //IMultiTenant 定义了 TenantId 属性

public string Name { get; set; }

public float Price { get; set; }

}

}

````

实现IMultiTenant接口,需要在实体中定义一个 **\*\*TenantId\*\*** 的属性(查看更多有关[实体](Entities.md)的文档)

#### 获取当前租户的Id

你的代码中可能需要获取当前租户的Id(先不管它具体是怎么取得的).对于这种情况你可以[注入](Dependency-Injection.md)并使用 **\*\*ICurrentTenant\*\*** 接口.例如:

````C#

using Volo.Abp.DependencyInjection;

using Volo.Abp.MultiTenancy;

namespace MyCompany.MyProject

{

public class MyService : ITransientDependency

{

private readonly ICurrentTenant \_currentTenant;

public MyService(ICurrentTenant currentTenant)

{

\_currentTenant = currentTenant;

}

public void DoIt()

{

var tenantId = \_currentTenant.Id;

//在你的代码中使用tenantId

}

}

}

````

#### 改变当前租户

TODO: ...

#### 确定当前租户

多租户的应用程序运行的时候首先要做的就是确定当前租户.

Volo.Abp.MultiTenancy只提供了用于确定当前租户的抽象(称为租户解析器),但是并没有现成的实现.

**\*\*Volo.Abp.AspNetCore.MultiTenancy\*\***已经实现了从当前Web请求(从子域名,请求头,cookie,路由...等)中确定当前租户.本文后面会介绍Volo.Abp.AspNetCore.MultiTenancy.

##### 自定义租户解析器

你可以像下面这样,在你模块的ConfigureServices方法中将自定义解析器并添加到 **\*\*AbpTenantResolveOptions\*\***中:

````C#

using Microsoft.Extensions.DependencyInjection;

using Volo.Abp.Modularity;

using Volo.Abp.MultiTenancy;

namespace MyCompany.MyProject

{

[DependsOn(typeof(AbpMultiTenancyModule))]

public class MyModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpTenantResolveOptions>(options =>

{

options.TenantResolvers.Add(new MyCustomTenantResolveContributor());

});

//...

}

}

}

````

`MyCustomTenantResolveContributor`必须像下面这样实现**\*\*ITenantResolveContributor\*\***接口:

````C#

using System.Threading.Tasks;

using Volo.Abp.MultiTenancy;

namespace MyCompany.MyProject

{

public class MyCustomTenantResolveContributor : ITenantResolveContributor

{

public override Task ResolveAsync(ITenantResolveContext context)

{

context.TenantIdOrName = ... //从其他地方获取租户id或租户名字...

}

}

}

````

如果能确定租户id或租户名字可以在租户解析器中设置 **\*\*TenantIdOrName\*\***.如果不能确定,那就空着让下一个解析器来确定它.

#### 租户存储

Volo.Abp.MultiTenancy中定义了 **\*\*ITenantStore\*\*** 从框架中抽象数据源.你可以实现ITenantStore,让它跟任何存储你租户的数据源(例如关系型数据库)一起工作.

##### 配置数据存储

有一个内置的(默认的)租户存储,叫ConfigurationTenantStore.它可以被用于存储租户,通过标准的[配置系统](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration/)(使用[Microsoft.Extensions.Configuration](https://www.nuget.org/packages/Microsoft.Extensions.Configuration)).因此,你可以通过硬编码或者在appsettings.json文件中定义租户.

###### 例子:硬编码定义租户

````C#

using System;

using Microsoft.Extensions.DependencyInjection;

using Volo.Abp.Data;

using Volo.Abp.Modularity;

using Volo.Abp.MultiTenancy;

namespace MyCompany.MyProject

{

[DependsOn(typeof(AbpMultiTenancyModule))]

public class MyModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpDefaultTenantStoreOptions>(options =>

{

options.Tenants = new[]

{

new TenantConfiguration(

Guid.Parse("446a5211-3d72-4339-9adc-845151f8ada0"), //Id

"tenant1" //Name

),

new TenantConfiguration(

Guid.Parse("25388015-ef1c-4355-9c18-f6b6ddbaf89d"), //Id

"tenant2" //Name

)

{

//tenant2 有单独的数据库连接字符串

ConnectionStrings =

{

{ConnectionStrings.DefaultConnectionStringName, "..."}

}

}

};

});

}

}

}

````

###### 例子:appsettings.json定义租户

首先从appsetting.json文件中创建你的配置.

````C#

using System.IO;

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.DependencyInjection;

using Volo.Abp.Modularity;

using Volo.Abp.MultiTenancy;

namespace MyCompany.MyProject

{

[DependsOn(typeof(AbpMultiTenancyModule))]

public class MyModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

var configuration = BuildConfiguration();

Configure<AbpDefaultTenantStoreOptions>(configuration);

}

private static IConfigurationRoot BuildConfiguration()

{

return new ConfigurationBuilder()

.SetBasePath(Directory.GetCurrentDirectory())

.AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)

.Build();

}

}

}

````

然后在appsettings.json中添加 "**\*\*Tenants\*\***" 节点:

````json

"Tenants": [

{

"Id": "446a5211-3d72-4339-9adc-845151f8ada0",

"Name": "tenant1"

},

{

"Id": "25388015-ef1c-4355-9c18-f6b6ddbaf89d",

"Name": "tenant2",

"ConnectionStrings": {

"Default": "...write tenant2's db connection string here..."

}

}

]

````

##### Volo.Abp... Package (TODO)

TODO: This package implements ITenantStore using a real database...

#### 租户信息

ITenantStore跟 **\*\*TenantConfiguration\*\***类一起工作,并且包含了几个租户属性:

\* **\*\*Id\*\***:租户的唯一Id.

\* **\*\*Name\*\***: 租户的唯一名称.

\* **\*\*ConnectionStrings\*\***:如果这个租户有专门的数据库来存储数据.它可以提供数据库的字符串(它可以具有默认的连接字符串和每个模块的连接字符串).

多租户应用程序可能需要其他租户属性,但这些属性是框架与多个租户一起使用的最低要求.

#### 代码中改变租户

TODO...

### Volo.Abp.AspNetCore.MultiTenancy

Volo.Abp.AspNetCore.MultiTenancy将多租户整合到了ASP.NET Core的程序中.在PMC中使用下面的代码将它安装到项目中.

````

Install-Package Volo.Abp.AspNetCore.MultiTenancy

````

然后添加 **\*\*AbpAspNetCoreMultiTenancyModule\*\*** 依赖到你的模块:

````C#

using Volo.Abp.Modularity;

using Volo.Abp.AspNetCore.MultiTenancy;

namespace MyCompany.MyProject

{

[DependsOn(typeof(AbpAspNetCoreMultiTenancyModule))]

public class MyModule : AbpModule

{

//...

}

}

````

#### 多租户中间件

Volo.Abp.AspNetCore.MultiTenancy包含了多租户中间件...

````C#

app.UseMultiTenancy();

````

TODO:...

#### 从Web请求中确定当前租户

Volo.Abp.AspNetCore.MultiTenancy 添加了下面这些租户解析器,从当前Web请求(按优先级排序)中确定当前租户.

\* **\*\*CurrentUserTenantResolveContributor\*\***: 如果当前用户已登录,从当前用户的声明中获取租户Id. **\*\*出于安全考虑,应该始终将其做为第一个Contributor\*\***.

\* **\*\*QueryStringTenantResolveContributor\*\***: 尝试从query string参数中获取当前租户,默认参数名为"\_\_tenant".

\* **\*\*RouteTenantResolveContributor\*\***:尝试从当前路由中获取(URL路径),默认是变量名是"\_\_tenant".所以,如果你的路由中定义了这个变量,就可以从路由中确定当前租户.

\* **\*\*HeaderTenantResolveContributor\*\***: 尝试从HTTP header中获取当前租户,默认的header名称是"\_\_tenant".

\* **\*\*CookieTenantResolveContributor\*\***: 尝试从当前cookie中获取当前租户.默认的Cookie名称是"\_\_tenant".

> 如果你使用nginx作为反向代理服务器,请注意如果`TenantKey`包含下划线或其他特殊字符可能存在问题, 请参考:

http://nginx.org/en/docs/http/ngx\_http\_core\_module.html#ignore\_invalid\_headers

http://nginx.org/en/docs/http/ngx\_http\_core\_module.html#underscores\_in\_headers

可以使用AbpAspNetCoreMultiTenancyOptions修改默认的参数名"\_\_tenant".例如:

````C#

services.Configure<AbpAspNetCoreMultiTenancyOptions>(options =>

{

options.TenantKey = "MyTenantKey";

});

````

##### 域名租户解析器

实际项目中,大多数情况下你想通过子域名(如mytenant1.mydomain.com)或全域名(如mytenant.com)中确定当前租户.如果是这样,你可以配置AbpTenantResolveOptions添加一个域名租户解析器.

###### 例子:添加子域名解析器

````C#

using Microsoft.Extensions.DependencyInjection;

using Volo.Abp.AspNetCore.MultiTenancy;

using Volo.Abp.Modularity;

using Volo.Abp.MultiTenancy;

namespace MyCompany.MyProject

{

[DependsOn(typeof(AbpAspNetCoreMultiTenancyModule))]

public class MyModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpTenantResolveOptions>(options =>

{

//子域名格式: {0}.mydomain.com (作为第二优先级解析器添加, 位于CurrentUserTenantResolveContributor之后)

options.TenantResolvers.Insert(1, new DomainTenantResolveContributor("{0}.mydomain.com"));

});

//...

}

}

}

````

{0}是用来确定当前租户唯一名称的占位符.

你可以使用下面的方法,代替``options.TenantResolvers.Insert(1, new DomainTenantResolveContributor("{0}.mydomain.com"));``:

````C#

options.AddDomainTenantResolver("{0}.mydomain.com");

````

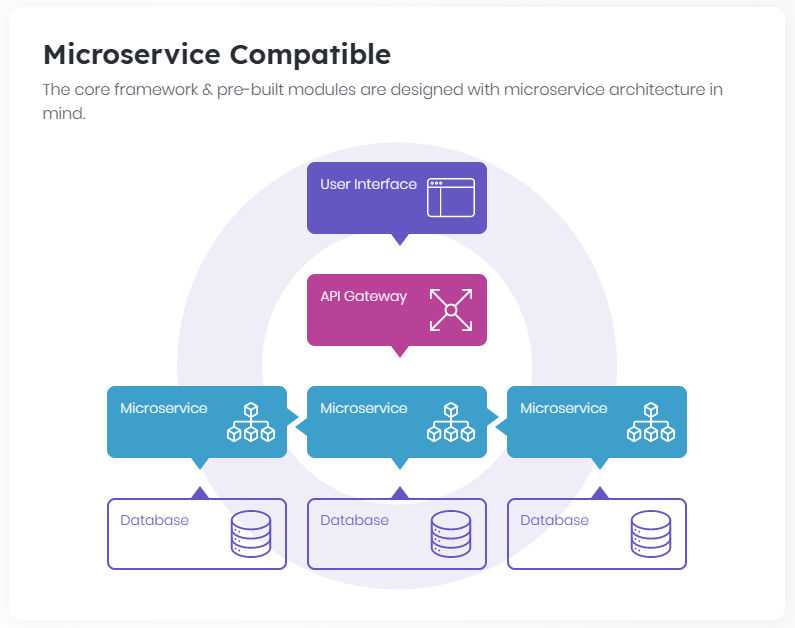
###### 例子:添加全域名解析器

````C#

options.AddDomainTenantResolver("{0}.com");

````

## 微服务架构



# 微服务架构

*\*"作为****\*\*面向服务架构\*\*****(SOA)的一个变体,微服务是一种将应用程序分解成****\*\*松散耦合服务\*\*****的新型架构风格. 通过****\*\*细粒度\*\*****的服务和****\*\*轻量级\*\*****的协议,微服务提供了更多的****\*\*模块化\*\*****,使应用程序更容易理解,开发,测试,并且更容易抵抗架构侵蚀. 它使小型团队能够****\*\*开发,部署和扩展\*\*****各自的服务,实现开发的****\*\*并行化\*\*****.它还允许通过****\*\*连续重构\*\*****形成单个服务的架构. 基于微服务架构可以实现****\*\*持续交付和部署\*\*****."\**

— [维基百科](https://zh.wikipedia.org/wiki/Microservices)

## 介绍

ABP框架的主要目标之一就是提供**\*\*便捷的基础设施来创建微服务解决方案\*\***. 我们做了以下工作:

\* 提供[模块系统](Module-Development-Basics.md),允许将应用程序拆分为模块,其中每个模块可以拥有自己的数据库,实体,服务,API,UI组件/页面....等.

\* 提供[架构模型](Best-Practices/Module-Architecture.md)来开发模块,与微服务开发和部署兼容.

\* 提供[最佳实践指南](Best-Practices/Index.md)制定模块开发标准.

\* 提供基础设施来实现微服务中的[领域驱动设计](Domain-Driven-Design.md).

\* 提供从应用程序服务[自动生成REST风格的API](API/Auto-API-Controllers.md)的服务.

\* 提供[自动创建C＃API客户端](API/Dynamic-CSharp-API-Clients.md)服务,以便从其他服务/应用程序使用你服务.

\* 提供[分布式事件总线](Event-Bus.md)用于服务通信.

\* 提供更多其他服务,使日常开发更加简便.

## 在新应用程序中使用微服务

开始一个新解决方案建议**\*\*始终从单体开始\*\***, 保持模块化,在单体成为问题时将其拆分为微服务.这使初期进度会很快,特别是如果你的团队人数不多,并且不想处理微服务架构带来的各种挑战.

然而开发一个良好的模块化应用程序不是那么简单,因为很难像微服务那样**\*\*保持模块之间的隔离\*\*** (参阅 [Stefan Tilkov的文章](https://martinfowler.com/articles/dont-start-monolith.html)). 微服务架构会自然的让你开发隔离的服务,但是在模块化的单体应用程序中,模块很容易彼此紧密耦合并设计出**\*\*弱模块边界\*\***和API约定.

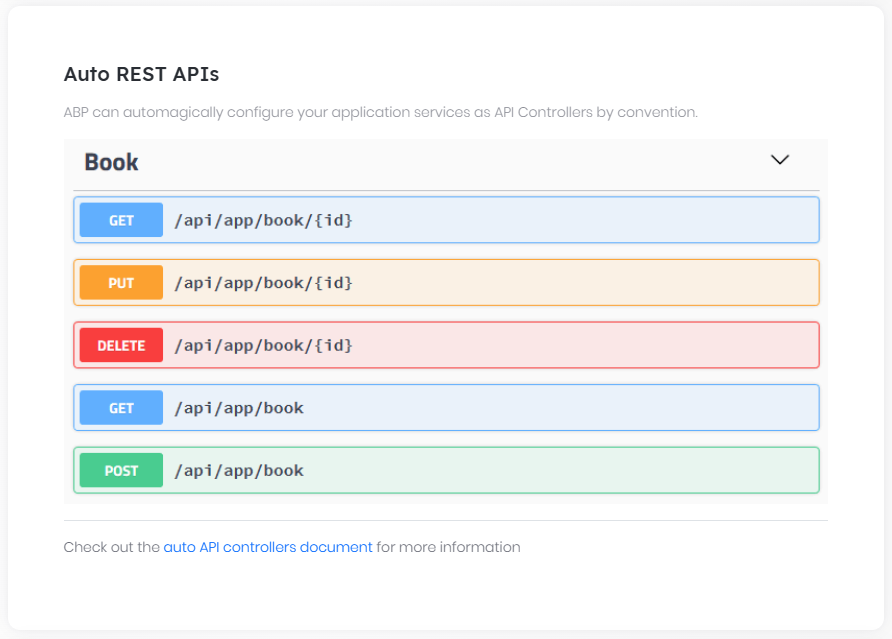
ABP可以帮助你,它提供了与**\*\*与微服务兼容的严格模块架构\*\*** 在这个架构中你的模块被分割成多个层/项目,在自己的VS解决方案中进行开发,该解决方案完全独立于其它模块. 这种方式开发的模块是一种天然的微服务,但是它可以很容易的插入到单体应用程序中. 请参阅**\*\*微服务优先的模块设计\*\***的[模块开发最佳实践指南](Best-Practices/Index.md). 所有[标准的ABP模块](https://github.com/abpframework/abp/tree/master/modules)都是基于本指南开发的. 因此你可以将这些模块嵌入到单体解决方案中使用它们,也可以单独部署通过远程API调用. 它们可以共享一个数据库,也可以通过简单配置使用自己的数据库.

## 微服务解决方案示例

[微服务解决方案示例](Samples/Microservice-Demo.md)演示了基于ABP框架的完整的微服务的解决方案.

# API

## 自动API控制器



# 自动API控制器

创建[应用程序服务](Application-Services.md)后, 通常需要创建API控制器以将此服务公开为HTTP(REST)API端点. 典型的API控制器除了将方法调用重定向到应用程序服务并使用[HttpGet],[HttpPost],[Route]等属性配置REST API之外什么都不做.

ABP可以按照惯例 **\*\*自动\*\*** 将你的应用程序服务配置为API控制器. 大多数时候你不关心它的详细配置,但它可以完全被自定义.

## 配置

基本配置很简单. 只需配置`AbpAspNetCoreMvcOptions`并使用`ConventionalControllers.Create`方法,如下所示:

````csharp

[DependsOn(BookStoreApplicationModule)]

public class BookStoreWebModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpAspNetCoreMvcOptions>(options =>

{

options

.ConventionalControllers

.Create(typeof(BookStoreApplicationModule).Assembly);

});

}

}

````

此示例代码配置包含类`BookStoreApplicationModule`的程序集中的所有应用程序服务.下图显示了[Swagger UI](https://swagger.io/tools/swagger-ui/)上的API内容.

![bookstore-apis](../images/bookstore-apis.png)

### 例子

一些示例方法名称和按约定生成的相应路由:

| 服务方法名称 | HTTP Method | 路由 |

| ----------------------------------------------------- | ----------- | -------------------------- |

| GetAsync(Guid id) | GET | /api/app/book/{id} |

| GetListAsync() | GET | /api/app/book |

| CreateAsync(CreateBookDto input) | POST | /api/app/book |

| UpdateAsync(Guid id, UpdateBookDto input) | PUT | /api/app/book/{id} |

| DeleteAsync(Guid id) | DELETE | /api/app/book/{id} |

| GetEditorsAsync(Guid id) | GET | /api/app/book/{id}/editors |

| CreateEditorAsync(Guid id, BookEditorCreateDto input) | POST | /api/app/book/{id}/editor |

### HTTP Method

ABP在确定服务方法的HTTP Method时使用命名约定:

- **\*\*Get\*\***: 如果方法名称以`GetList`,`GetAll`或`Get`开头.

- **\*\*Put\*\***: 如果方法名称以`Put`或`Update`开头.

- **\*\*Delete\*\***: 如果方法名称以`Delete`或`Remove`开头.

- **\*\*Post\*\***: 如果方法名称以`Create`,`Add`,`Insert`或`Post`开头.

- **\*\*Patch\*\***: 如果方法名称以`Patch`开头.

- 其他情况, **\*\*Post\*\*** 为 **\*\*默认方式\*\***.

如果需要为特定方法自定义HTTP Method, 则可以使用标准ASP.NET Core的属性([HttpPost], [HttpGet], [HttpPut]... 等等.). 这需要添加[Microsoft.AspNetCore.Mvc.Core](https://www.nuget.org/packages/Microsoft.AspNetCore.Mvc.Core)的Nuget包.

### 路由

路由根据一些惯例生成:

\* 它始终以 **\*\*/api\*\***开头.

\* 接着是**\*\*路由路径\*\***. 默认值为"**\*\*/app\*\***", 可以进行如下配置:

````csharp

Configure<AbpAspNetCoreMvcOptions>(options =>

{

options.ConventionalControllers

.Create(typeof(BookStoreApplicationModule).Assembly, opts =>

{

opts.RootPath = "volosoft/book-store";

});

});

````

然后获得一本书的路由将是'**\*\*/api/volosoft/book-store/book/{id}\*\***'. 此示例使用两级根路径,但通常使用单个级别的深度.

\* 接着 **\*\*标准化控制器/服务名称\*\***. 会删除`AppService`,`ApplicationService`和`Service`的后缀并将其转换为 **\*\*camelCase\*\***. 如果你的应用程序服务类名称为`BookAppService`.那么它将变为`/book`.

\* 如果要自定义命名, 则设置`UrlControllerNameNormalizer`选项. 它是一个委托允许你自定义每个控制器/服务的名称.

\* 如果该方法具有 '**\*\*id\*\***'参数, 则会在路由中添加'**\*\*/{id}\*\***'.

\* 如有必要,它会添加操作名称. 操作名称从服务上的方法名称获取并标准化;

\* 删除'**\*\*Async\*\***'后缀. 如果方法名称为'GetPhonesAsync',则变为`GetPhones`.

\* 删除**\*\*HTTP method前缀\*\***. 基于的HTTP method删除`GetList`,`GetAll`,`Get`,`Put`,`Update`,`Delete`,`Remove`,`Create`,`Add`,`Insert`,`Post`和`Patch`前缀, 因此`GetPhones`变为`Phones`, 因为`Get`前缀和GET请求重复.

\* 将结果转换为**\*\*camelCase\*\***.

\* 如果生成的操作名称为**\*\*空\*\***,则它不会添加到路径中.否则它会被添加到路由中(例如'/phones').对于`GetAllAsync`方法名称,它将为空,因为`GetPhonesAsync`方法名称将为`phone`.

\* 可以通过设置`UrlActionNameNormalizer`选项来自定义.It's an action delegate that is called for every method.

\* 如果有另一个带有'Id'后缀的参数,那么它也会作为最终路线段添加到路线中(例如'/phoneId').

## 服务选择

创建的HTTP API控制器并不是应用服务所独有的功能.

### IRemoteService 接口

如果一个类实现了`IRemoteService`接口, 那么它会被自动选择为API控制器. 由于应用程序服务本身实现了`IRemoteService`接口, 因此它自然就成为API控制器.

### RemoteService Attribute

`RemoteService`可用于将实现`IRemoteService`接口的类标记为远程服务或禁用它. 例如:

````csharp

[RemoteService(IsEnabled = false)] //or simply [RemoteService(false)]

public class PersonAppService : ApplicationService

{

}

````

### TypePredicate 选项

你可以通过提供`TypePedicate`选项进一步过滤类以成为API控制器:

````csharp

services.Configure<AbpAspNetCoreMvcOptions>(options =>

{

options.ConventionalControllers

.Create(typeof(BookStoreApplicationModule).Assembly, opts =>

{

opts.TypePredicate = type => { return true; };

});

});

````

如果你不想将此类型公开为API控制器, 则可以在类型检查时返回`false`.

## API Explorer

API Explorer是可以由客户端获取API结构的服务. Swagger使用它为endpoint创建文档和test UI.

默认情况下, HTTP API控制器会自动启用API Explorer, 可以使用`RemoteService`按类或方法的级别控制它. 例如:

````csharp

[RemoteService(IsMetadataEnabled = false)]

public class PersonAppService : ApplicationService

{

}

````

禁用`IsMetadataEnabled`从而从API Explorer中隐藏此服务, 并且无法被发现. 但是它仍然可以被知道确切API路径/路由的客户端使用.

## 动态C# API客户端

# 动态 C# API 客户端

ABP可以自动创建C# API 客户端代理来调用远程HTTP服务(REST APIS).通过这种方式,你不需要通过 `HttpClient` 或者其他低级的HTTP功能调用远程服务并获取数据.

## 服务接口

你的service或controller需要实现一个在服务端和客户端共享的接口.因此,首先需要在一个共享的类库项目中定义一个服务接口.例如:

````csharp

public interface IBookAppService : IApplicationService

{

Task<List<BookDto>> GetListAsync();

}

````

为了能自动被发现,你的接口需要实现`IRemoteService`接口.由于`IApplicationService`继承自`IRemoteService`接口.所以`IBookAppService`完全满足这个条件.

在你的服务中实现这个类,你可以使用[auto API controller system](Auto-API-Controllers.md)将你的服务暴漏为一个REST API 端点.

## 客户端代理生成

首先,将[Volo.Abp.Http.Client](https://www.nuget.org/packages/Volo.Abp.Http.Client) nuget包添加到你的客户端项目中:

````

Install-Package Volo.Abp.Http.Client

````

然后给你的模块添加`AbpHttpClientModule`依赖:

````csharp

[DependsOn(typeof(AbpHttpClientModule))] //添加依赖

public class MyClientAppModule : AbpModule

{

}

````

现在,已经可以创建客户端代理了.例如:

````csharp

[DependsOn(

typeof(AbpHttpClientModule), //用来创建客户端代理

typeof(BookStoreApplicationContractsModule) //包含应用服务接口

)]

public class MyClientAppModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

//创建动态客户端代理

context.Services.AddHttpClientProxies(

typeof(BookStoreApplicationContractsModule).Assembly

);

}

}

````

`AddHttpClientproxies`方法获得一个程序集,找到这个程序集中所有的服务接口,创建并注册代理类.

### Endpoint配置

`appsettings.json`文件中的`RemoteServices`节点被用来设置默认的服务地址.下面是最简单的配置:

````

{

"RemoteServices": {

"Default": {

"BaseUrl": "http://localhost:53929/"

}

}

}

````

查看下面的"AbpRemoteServiceOptions"章节获取更多详细配置.

## 使用

可以很直接地使用.只需要在你的客户端程序中注入服务接口:

````csharp

public class MyService : ITransientDependency

{

private readonly IBookAppService \_bookService;

public MyService(IBookAppService bookService)

{

\_bookService = bookService;

}

public async Task DoIt()

{

var books = await \_bookService.GetListAsync();

foreach (var book in books)

{

Console.WriteLine($"[BOOK {book.Id}] Name={book.Name}");

}

}

}

````

本例注入了上面定义的`IBookAppService`服务接口.当客户端调用服务方法的时候动态客户端代理就会创建一个HTTP调用.

### IHttpClientProxy接口

你可以像上面那样注入`IBookAppService`来使用客户端代理,也可以注入`IHttpClientProxy<IBookAppService>`获取更多明确的用法.这种情况下你可以使用`IHttpClientProxy<T>`接口的`Service`属性.

## 配置

### AbpRemoteServiceOptions

默认情况下`AbpRemoteServiceOptions`从`appsettings.json`获取.或者,你可以使用`Configure`方法来设置或重写它.如:

````csharp

public override void ConfigureServices(ServiceConfigurationContext context)

{

context.Services.Configure<AbpRemoteServiceOptions>(options =>

{

options.RemoteServices.Default =

new RemoteServiceConfiguration("http://localhost:53929/");

});

//...

}

````

### 多个远程服务端点

上面的例子已经配置了"Default"远程服务端点.你可能需要为不同的服务创建不同的端点.(就像在微服务方法中一样,每个微服务具有不同的端点).在这种情况下,你可以在你的配置文件中添加其他的端点:

````json

{

"RemoteServices": {

"Default": {

"BaseUrl": "http://localhost:53929/"

},

"BookStore": {

"BaseUrl": "http://localhost:48392/"

}

}

}

````

`AddHttpClientProxies`方法有一个可选的参数来定义远程服务的名字:

````csharp

context.Services.AddHttpClientProxies(

typeof(BookStoreApplicationContractsModule).Assembly,

remoteServiceConfigurationName: "BookStore"

);

````

`remoteServiceConfigurationName`参数会匹配通过`AbpRemoteServiceOptions`配置的服务端点.如果`BookStore`端点没有定义就会使用默认的`Default`端点.

### 作为默认服务

当你为`IBookAppService`创建了一个服务代理,你可以直接注入`IBookAppService`来使用代理客户端(像上面章节中将的那样).你可以传递`asDefaultService:false`到`AddHttpClientProxies`方法来禁用此功能.

````csharp

context.Services.AddHttpClientProxies(

typeof(BookStoreApplicationContractsModule).Assembly,

asDefaultServices: false

);

````

如果你的程序中已经有一个服务的实现并且你不想用你的客户端代理重写或替换其他的实现,就需要使用`asDefaultServices:false`

> 如果你禁用了`asDefaultService`,你只能使用`IHttpClientProxy<T>`接口去使用客户端代理.(参见上面的相关章节).

## ABP端点

应用程序配置

# 应用程序配置端点

ABP框架提供了一个预构建的标准端点,其中包含一些有关应用程序/服务的有用信息. 这里是此端点的一些基本信息的列表:

\* [本地化](../Localization.md)值, 支持应用程序的当前语言.

\* 当前用户可用和已授予的[策略](../Authorization.md)(权限).

\* 当前用户的[设置](../Settings.md)值.

\* 关于[当前用户](../CurrentUser.md)的信息 (如 id 和用户名).

\* 关于当前[租户](../Multi-Tenancy.md)的信息 (如 id 和名称).

\* 当前用户的[时区](../Timing.md)信息和应用程序的[时钟](../Timing.md)类型.

## HTTP API

如果您导航到基于ABP框架的web应用程序或HTTP服务的 `/api/abp/application-configuration` URL, 你可以得到JSON对象形式配置. 该端点对于创建应用程序的客户端很有用.

## Script

对于ASP.NET Core MVC(剃刀页)应用程序,同样的配置值在JavaScript端也可用. `/Abp/ApplicationConfigurationScript` 是基于上述HTTP API自动生成的脚本的URL.

参阅 [JavaScript API文档](../UI/AspNetCore/JavaScript-API/Index.md) 了解关于ASP.NET Core UI.

其他UI类型提供相关平台的本地服务. 例如查看[Angular UI本地化文档](../UI/Angular/Localization.md)来学习如何使用这个端点公开的本地化值.

## API版本控制

# 自动API控制器

创建[应用程序服务](Application-Services.md)后, 通常需要创建API控制器以将此服务公开为HTTP(REST)API端点. 典型的API控制器除了将方法调用重定向到应用程序服务并使用[HttpGet],[HttpPost],[Route]等属性配置REST API之外什么都不做.

ABP可以按照惯例 **\*\*自动\*\*** 将你的应用程序服务配置为API控制器. 大多数时候你不关心它的详细配置,但它可以完全被自定义.

## 配置

基本配置很简单. 只需配置`AbpAspNetCoreMvcOptions`并使用`ConventionalControllers.Create`方法,如下所示:

````csharp

[DependsOn(BookStoreApplicationModule)]

public class BookStoreWebModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpAspNetCoreMvcOptions>(options =>

{

options

.ConventionalControllers

.Create(typeof(BookStoreApplicationModule).Assembly);

});

}

}

````

此示例代码配置包含类`BookStoreApplicationModule`的程序集中的所有应用程序服务.下图显示了[Swagger UI](https://swagger.io/tools/swagger-ui/)上的API内容.

![bookstore-apis](../images/bookstore-apis.png)

### 例子

一些示例方法名称和按约定生成的相应路由:

| 服务方法名称 | HTTP Method | 路由 |

| ----------------------------------------------------- | ----------- | -------------------------- |

| GetAsync(Guid id) | GET | /api/app/book/{id} |

| GetListAsync() | GET | /api/app/book |

| CreateAsync(CreateBookDto input) | POST | /api/app/book |

| UpdateAsync(Guid id, UpdateBookDto input) | PUT | /api/app/book/{id} |

| DeleteAsync(Guid id) | DELETE | /api/app/book/{id} |

| GetEditorsAsync(Guid id) | GET | /api/app/book/{id}/editors |

| CreateEditorAsync(Guid id, BookEditorCreateDto input) | POST | /api/app/book/{id}/editor |

### HTTP Method

ABP在确定服务方法的HTTP Method时使用命名约定:

- **\*\*Get\*\***: 如果方法名称以`GetList`,`GetAll`或`Get`开头.

- **\*\*Put\*\***: 如果方法名称以`Put`或`Update`开头.

- **\*\*Delete\*\***: 如果方法名称以`Delete`或`Remove`开头.

- **\*\*Post\*\***: 如果方法名称以`Create`,`Add`,`Insert`或`Post`开头.

- **\*\*Patch\*\***: 如果方法名称以`Patch`开头.

- 其他情况, **\*\*Post\*\*** 为 **\*\*默认方式\*\***.

如果需要为特定方法自定义HTTP Method, 则可以使用标准ASP.NET Core的属性([HttpPost], [HttpGet], [HttpPut]... 等等.). 这需要添加[Microsoft.AspNetCore.Mvc.Core](https://www.nuget.org/packages/Microsoft.AspNetCore.Mvc.Core)的Nuget包.

### 路由

路由根据一些惯例生成:

\* 它始终以 **\*\*/api\*\***开头.

\* 接着是**\*\*路由路径\*\***. 默认值为"**\*\*/app\*\***", 可以进行如下配置:

````csharp

Configure<AbpAspNetCoreMvcOptions>(options =>

{

options.ConventionalControllers

.Create(typeof(BookStoreApplicationModule).Assembly, opts =>

{

opts.RootPath = "volosoft/book-store";

});

});

````

然后获得一本书的路由将是'**\*\*/api/volosoft/book-store/book/{id}\*\***'. 此示例使用两级根路径,但通常使用单个级别的深度.

\* 接着 **\*\*标准化控制器/服务名称\*\***. 会删除`AppService`,`ApplicationService`和`Service`的后缀并将其转换为 **\*\*camelCase\*\***. 如果你的应用程序服务类名称为`BookAppService`.那么它将变为`/book`.

\* 如果要自定义命名, 则设置`UrlControllerNameNormalizer`选项. 它是一个委托允许你自定义每个控制器/服务的名称.

\* 如果该方法具有 '**\*\*id\*\***'参数, 则会在路由中添加'**\*\*/{id}\*\***'.

\* 如有必要,它会添加操作名称. 操作名称从服务上的方法名称获取并标准化;

\* 删除'**\*\*Async\*\***'后缀. 如果方法名称为'GetPhonesAsync',则变为`GetPhones`.

\* 删除**\*\*HTTP method前缀\*\***. 基于的HTTP method删除`GetList`,`GetAll`,`Get`,`Put`,`Update`,`Delete`,`Remove`,`Create`,`Add`,`Insert`,`Post`和`Patch`前缀, 因此`GetPhones`变为`Phones`, 因为`Get`前缀和GET请求重复.

\* 将结果转换为**\*\*camelCase\*\***.

\* 如果生成的操作名称为**\*\*空\*\***,则它不会添加到路径中.否则它会被添加到路由中(例如'/phones').对于`GetAllAsync`方法名称,它将为空,因为`GetPhonesAsync`方法名称将为`phone`.

\* 可以通过设置`UrlActionNameNormalizer`选项来自定义.It's an action delegate that is called for every method.

\* 如果有另一个带有'Id'后缀的参数,那么它也会作为最终路线段添加到路线中(例如'/phoneId').

## 服务选择

创建的HTTP API控制器并不是应用服务所独有的功能.

### IRemoteService 接口

如果一个类实现了`IRemoteService`接口, 那么它会被自动选择为API控制器. 由于应用程序服务本身实现了`IRemoteService`接口, 因此它自然就成为API控制器.

### RemoteService Attribute

`RemoteService`可用于将实现`IRemoteService`接口的类标记为远程服务或禁用它. 例如:

````csharp

[RemoteService(IsEnabled = false)] //or simply [RemoteService(false)]

public class PersonAppService : ApplicationService

{

}

````

### TypePredicate 选项

你可以通过提供`TypePedicate`选项进一步过滤类以成为API控制器:

````csharp

services.Configure<AbpAspNetCoreMvcOptions>(options =>

{

options.ConventionalControllers

.Create(typeof(BookStoreApplicationModule).Assembly, opts =>

{

opts.TypePredicate = type => { return true; };

});

});

````

如果你不想将此类型公开为API控制器, 则可以在类型检查时返回`false`.

## API Explorer

API Explorer是可以由客户端获取API结构的服务. Swagger使用它为endpoint创建文档和test UI.

默认情况下, HTTP API控制器会自动启用API Explorer, 可以使用`RemoteService`按类或方法的级别控制它. 例如:

````csharp

[RemoteService(IsMetadataEnabled = false)]

public class PersonAppService : ApplicationService

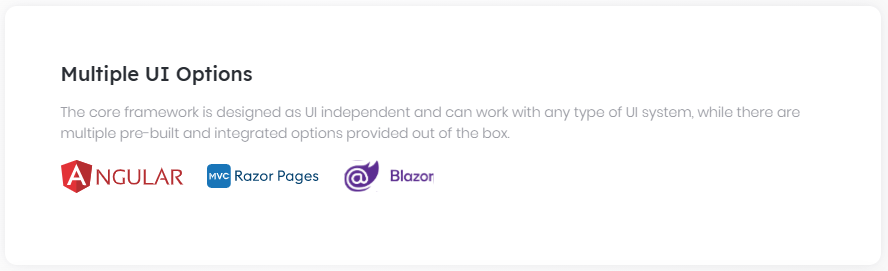
{

}

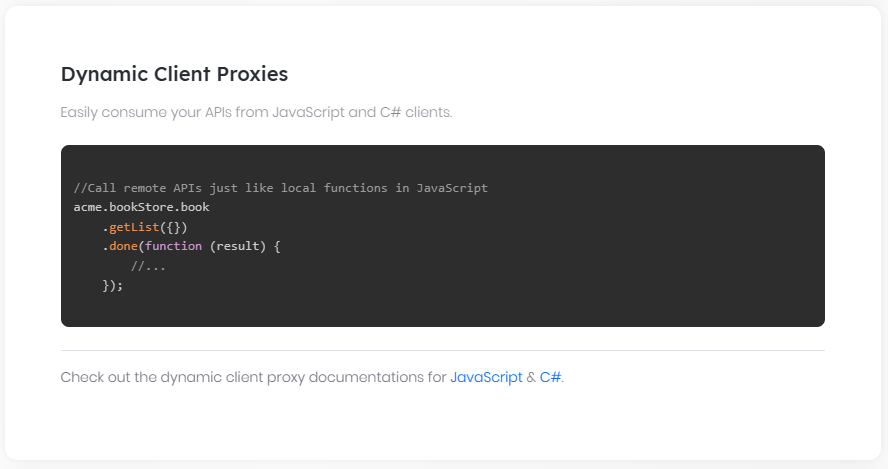
````

禁用`IsMetadataEnabled`从而从API Explorer中隐藏此服务, 并且无法被发现. 但是它仍然可以被知道确切API路径/路由的客户端使用.

# 用户界面



## ASP.NET Core



客户端包管理

## ASP.NET Core MVC 客户端包管理

ABP框架可以与任何类型的客户端包管理系统一起使用. 甚至你可以决定不使用包管理系统并手动管理依赖项.

但是, ABP框架最适用于**\*\*NPM/Yarn\*\***. 默认情况下,内置模块配置为与NPM/Yarn一起使用.

最后, 我们建议[\*\*Yarn\*\*](https://classic.yarnpkg.com/)而不是NPM,因为它更快,更稳定并且与NPM兼容.

### @ABP NPM Packages

ABP是一个模块化平台. 每个开发人员都可以创建模块, 模块应该在**\*\*兼容\*\***和**\*\*稳定\*\***状态下协同工作.

一个挑战是依赖NPM包的**\*\*版本\*\***. 如果两个不同的模块使用相同的JavaScript库但其不同(并且可能不兼容)的版本会怎样.

为了解决版本问题, 我们创建了一套**\*\*标准包\*\***, 这取决于一些常见的第三方库. 一些示例包是[@abp/jquery](https://www.npmjs.com/package/@abp/jquery), [@ abp/bootstrap](https://www.npmjs.com/package/@abp/bootstrap)和[@abp/font-awesome](https://www.npmjs.com/package/@abp/font-awesome). 你可以从[Github存储库](https://github.com/volosoft/abp/tree/master/npm/packs)中查看**\*\*列表\*\***.

**\*\*标准包\*\***的好处是:

\* 它取决于包装的**\*\*标准版本\*\***. 取决于此包是**\*\*安全\*\***,因为所有模块都依赖于相同的版本.

\* 它包含将库资源(js,css,img...文件)从**\*\*node\_modules\*\***文件夹复制到**\*\*wwwroot/libs\*\***文件夹. 有关更多信息, 请参阅 *\*映射库资源\** 部分.

依赖标准包装很容易. 只需像往常一样将它添加到**\*\*package.json\*\***文件中. 例如:

````

{

...

"dependencies": {

"@abp/bootstrap": "^1.0.0"

}

}

````

建议依赖于标准软件包, 而不是直接依赖于第三方软件包.

#### 安装包

依赖于NPM包后, 你应该做的就是从命令行运行**\*\*yarn\*\***命令来安装所有包及其依赖项:

````

yarn

````

虽然你可以使用`npm install`,但如前所述,建议使用[Yarn](https://classic.yarnpkg.com/).

#### 贡献包

如果你需要不在标准软件包中的第三方NPM软件包,你可以在Github[repository](https://github.com/volosoft/abp)上创建Pull请求. 接受遵循这些规则的拉取请求:

\* 对于NPM上的`package-name`, 包名称应该命名为`@abp/package-name`(例如:`bootstrap`包的`@abp/bootstrap`).

\* 它应该是**\*\*最新的稳定\*\***版本的包.

\* 它应该只依赖于**\*\*单个\*\***第三方包. 它可以依赖于多个`@abp/\*`包.

\* 包应包含一个`abp.resourcemapping.js`文件格式,如*\*映射库资源\**部分中所定义. 此文件应仅映射所依赖包的资源.

\* 你还需要为你创建的包创建[bundle贡献者](Bundling-Minification.md).

有关示例, 请参阅当前标准包.

### 映射库资源

使用NPM包和NPM/Yarn工具是客户端库的事实标准. NPM/Yarn工具在Web项目的根文件夹中创建一个**\*\*node\_modules\*\***文件夹.

下一个挑战是将所需的资源(js,css,img ...文件)从`node\_modules`复制到**\*\*wwwroot\*\***文件夹内的文件夹中,以使其可供客户端/浏览器访问.

ABP的 `install-libs` 命令**\*\*将资源\*\***从**\*\*node\_modules\*\***复制到**\*\*wwwroot/libs\*\***文件夹. 每个**\*\*标准包\*\***(参见\*@ABP NPM Packages\*部分)定义了自己文件的映射. 因此, 大多数情况你只配置依赖项.

**\*\*启动模板\*\***已经配置为开箱即用的所有这些. 本节将介绍配置选项.

#### 资源映射定义文件

模块应该定义一个名为`abp.resourcemapping.js`的JavaScript文件,其格式如下例所示:

````js

module.exports = {

aliases: {

"@node\_modules": "./node\_modules",

"@libs": "./wwwroot/libs"

},

clean: [

"@libs"

],

mappings: {

}

}

````

\* **\*\*aliases\*\***部分定义了可在映射路径中使用的标准别名(占位符). **\*\*@node\_modules\*\***和 **\*\*@libs\*\***是必需的(通过标准包), 你可以定义自己的别名以减少重复.

\* **\*\*clean\*\***部分是在复制文件之前要清理的文件夹列表.

\* **\*\*mappings\*\***部分是要复制的文件/文件夹的映射列表.此示例不会复制任何资源本身,但取决于标准包.

示例映射配置如下所示:

````js

mappings: {

"@node\_modules/bootstrap/dist/css/bootstrap.css": "@libs/bootstrap/css/",

"@node\_modules/bootstrap/dist/js/bootstrap.bundle.js": "@libs/bootstrap/js/"

}

````

#### install-libs 命令

正确配置`abp.resourcemapping.js`文件后, 可以从命令行运行ABP CLI命令:

````bash

abp install-libs

````

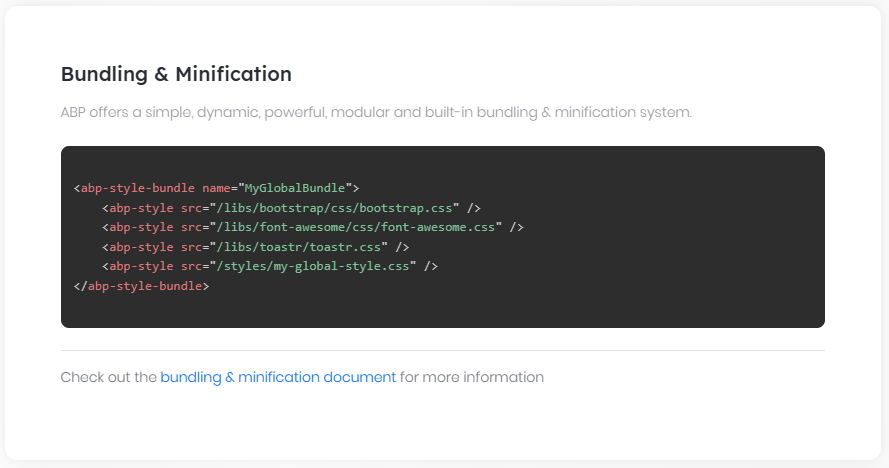
当你运行这个命令时,所有包都会将自己的资源复制到**\*\*wwwroot/libs\*\***文件夹中. 只有在**\*\*package.json\*\***文件中对依赖项进行更改时, 才需要运行`abp install-libs`.

#### 参见

\* [捆绑 & 压缩](Bundling-Minification.md)

\* [主题](Theming.md)

捆绑&压缩



## ASP.NET Core MVC 捆绑 & 压缩

有许多方法可以捆绑&压缩客户端资源(JavaScript和CSS文件). 最常见的方式是:

\* 使用Visual Studio[捆绑&压缩](https://marketplace.visualstudio.com/items?itemName=MadsKristensen.BundlerMinifier)扩展或者其它的[NuGet相关包](https://www.nuget.org/packages/BuildBundlerMinifier/).

\* 使用[Gulp](https://gulpjs.com/)/[Grunt](https://gruntjs.com/)及其插件.

ABP内置了简单,动态,强大,模块化的方式.

### Volo.Abp.AspNetCore.Mvc.UI.Bundling 包

> 默认情况下已在启动模板安装此软件包. 大多数情况下,你不需要手动安装它.

将`Volo.Abp.AspNetCore.Mvc.UI.Bundling` nuget包安装到你的项目中:

````

install-package Volo.Abp.AspNetCore.Mvc.UI.Bundling

````

然后将`AbpAspNetCoreMvcUiBundlingModule`依赖项添加到你的模块上:

````C#

using Volo.Abp.Modularity;

using Volo.Abp.AspNetCore.Mvc.UI.Bundling;

namespace MyCompany.MyProject

{

[DependsOn(typeof(AbpAspNetCoreMvcUiBundlingModule))]

public class MyWebModule : AbpModule

{

//...

}

}

````

### Razor Bundling Tag Helpers

创建bundle的最简单方法是使用`abp-script-bundle`或`abp-style-bundle` tag helpers. 例如:

````html

<abp-style-bundle name="MyGlobalBundle">

<abp-style src="/libs/bootstrap/css/bootstrap.css" />

<abp-style src="/libs/font-awesome/css/font-awesome.css" />

<abp-style src="/libs/toastr/toastr.css" />

<abp-style src="/styles/my-global-style.css" />

</abp-style-bundle>

````

`abp-style-bundle`定义了一个带有**\*\*唯一名称\*\***的样式包:`MyGlobalBundle`. 使用方法很容易理解. 让我们看看它是如何*\*工作的\**:

\* 当首次请求时,ABP从提供的文件中 **\*\*(延迟)lazy\*\*** 创建. 后续将从 **\*\*缓存\*\*** 中返回内容. 这意味着如果你有条件地将文件添加到包中,它只执行一次, 并且条件的任何更改都不会影响下一个请求的包.

\* 在`development`环境中ABP会将包文件**\*\*单独\*\***添加到页面中, 其他环境(`staging`,`production`...)会自动捆绑和压缩.

\* 捆绑文件可以是**\*\*物理\*\***文件或[\*\*虚拟/嵌入\*\*](../../Virtual-File-System.md)的文件.

\* ABP自动将 **\*\*版本查询字符串(version query string)\*\*** 添加到捆绑文件的URL中,以防止浏览器缓存. 如:?\_v=67872834243042(从文件的上次更改日期生成). 即使捆绑文件单独添加到页面(在`development`环境中), 版本控制仍然有效.

#### 导入 Bundling Tag Helpers

> 默认情况下已在启动模板导入. 大多数情况下,你不需要手动安装它.

要使用`bundle tag helpers`, 你需要将其添加到`\_ViewImports.cshtml`文件或页面中:

````

@addTagHelper \*, Volo.Abp.AspNetCore.Mvc.UI.Bundling

````

#### 未命名的 Bundles

对于razor bundle tag helpers, `name`是**\*\*可选\*\***. 如果没有定义一个名字,它将根据使用的捆绑文件名自动**\*\*计算生成\*\***(they are **\*\*concatenated\*\*** and **\*\*hashed\*\***) 例:

````html

<abp-style-bundle>

<abp-style src="/libs/bootstrap/css/bootstrap.css" />

<abp-style src="/libs/font-awesome/css/font-awesome.css" />

<abp-style src="/libs/toastr/toastr.css" />

@if (ViewBag.IncludeCustomStyles != false)

{

<abp-style src="/styles/my-global-style.css" />

}

</abp-style-bundle>

````

这将潜在地创建**\*\*两个不同的bundles\*\***(一个包括`my-global-style.css`而另一个则不包括).

**\*\*未命名的\*\*** bundles优点:

\* 可以**\*\*有条件地将项目\*\***添加到捆绑包中. 但这可能会导致基于条件的捆绑的存在多种变化.

**\*\*命名\*\*** bundles优点:

\* 其他模块可以通过其名称为捆绑包做出贡献(参见下面的部分).

#### 单个文件

如果你只需要在页面中添加一个文件, 你可以使用`abp-script`或`abp-style`而不需要包含在`abp-script-bundle`或`abp-style-bundle`中. 例:

````xml

<abp-script src="/scripts/my-script.js" />

````

对于上面的示例,包名称将是 *\*scripts.my-scripts\**("/"替换为"."). 所有捆绑功能也可以按预期应用于单个文件.

### Bundling 选项

如果你需要在 **\*\*多个页面中使用相同的包\*\*** 或想要使用更多 **\*\*强大功能\*\***, 你可以在[模块](../../Module-Development-Basics.md)类中进行**\*\*配置\*\***.

#### 创建一个新的捆绑包

用法示例:

````C#

[DependsOn(typeof(AbpAspNetCoreMvcUiBundlingModule))]

public class MyWebModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpBundlingOptions>(options =>

{

options

.ScriptBundles

.Add("MyGlobalBundle", bundle => {

bundle.AddFiles(

"/libs/jquery/jquery.js",

"/libs/bootstrap/js/bootstrap.js",

"/libs/toastr/toastr.min.js",

"/scripts/my-global-scripts.js"

);

});

});

}

}

````

> 你可以在脚本和样式包中使用相同的名称(*\*MyGlobalBundle\**), 因为它们被添加到不同的集合(`ScriptBundles`和`StyleBundles`).

在定义bundle之后, 可以使用上面定义的相同tag helpers将其包括在页面中. 例如:

````html

<abp-script-bundle name="MyGlobalBundle" />

````

这次tag helper定义中没有定义文件, 因为捆绑文件是由代码定义的.

#### 配置现有的 Bundle

ABP也支持[模块化](../../Module-Development-Basics.md)捆绑. 模块可以修改由依赖模块创建的捆绑包.

例如:

````C#

[DependsOn(typeof(MyWebModule))]

public class MyWebExtensionModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpBundlingOptions>(options =>

{

options

.ScriptBundles

.Configure("MyGlobalBundle", bundle => {

bundle.AddFiles(

"/scripts/my-extension-script.js"

);

});

});

}

}

````

你也可以使用 `ConfigureAll` 方法配置所有现有的捆绑包:

````C#

[DependsOn(typeof(MyWebModule))]

public class MyWebExtensionModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<AbpBundlingOptions>(options =>

{

options

.ScriptBundles

.ConfigureAll(bundle => {

bundle.AddFiles(

"/scripts/my-extension-script.js"

);

});

});

}

}

````

### Bundle 贡献者

将文件添加到现有bundle似乎很有用. 如果你需要**\*\*替换\*\***bundle中的文件或者你想**\*\*有条件地\*\***添加文件怎么办? 定义bundle贡献者可为此类情况提供额外的功能.

一个bundle的贡献者使用自定义版本bootstrap.css替换示例:

````C#

public class MyExtensionGlobalStyleContributor : BundleContributor

{

public override void ConfigureBundle(BundleConfigurationContext context)

{

context.Files.ReplaceOne(

"/libs/bootstrap/css/bootstrap.css",

"/styles/extensions/bootstrap-customized.css"

);

}

}

````

然后你可以按照下面的代码使用这个贡献者:

````C#

services.Configure<AbpBundlingOptions>(options =>

{

options

.ScriptBundles

.Configure("MyGlobalBundle", bundle => {

bundle.AddContributors(typeof(MyExtensionGlobalStyleContributor));

});

});

````

贡献者也可以在bundle tag helpers中使用.

例如:

````xml

<abp-style-bundle>

<abp-style type="@typeof(BootstrapStyleContributor)" />

<abp-style src="/libs/font-awesome/css/font-awesome.css" />

<abp-style src="/libs/toastr/toastr.css" />

</abp-style-bundle>

````

`abp-style`和`abp-script`标签可以使用`type`属性(而不是`src`属性), 如本示例所示. 添加bundle贡献者时, 其依赖关系也会自动添加到bundle中.

#### 贡献者依赖关系

bundle贡献者可以与其他贡献者具有一个或多个依赖关系.

例如:

````C#

[DependsOn(typeof(MyDependedBundleContributor))] //Define the dependency

public class MyExtensionStyleBundleContributor : BundleContributor

{

//...

}

````

添加bundle贡献者时,其依赖关系将 **\*\*自动并递归\*\*** 添加. **\*\*依赖顺序\*\*** 通过阻止 **\*\*重复\*\*** 添加的依赖关系. 即使它们处于分离的bundle中,也会阻止重复. ABP在页面中组织所有bundle并消除重复.

创建贡献者和定义依赖关系是一种跨不同模块组织bundle创建的方法.

#### 贡献者扩展

在某些高级应用场景中, 当用到一个bundle贡献者时,你可能想做一些额外的配置. 贡献者扩展可以和被扩展的贡献者无缝衔接.

下面的示例为 prism.js 脚本库添加一些样式:

````csharp

public class MyPrismjsStyleExtension : BundleContributor

{

public override void ConfigureBundle(BundleConfigurationContext context)

{

context.Files.AddIfNotContains("/libs/prismjs/plugins/toolbar/prism-toolbar.css");

}

}

````

然后你可以配置 `AbpBundleContributorOptions` 去扩展已存在的 `PrismjsStyleBundleContributor`.

````csharp

Configure<AbpBundleContributorOptions>(options =>

{

options

.Extensions<PrismjsStyleBundleContributor>()

.Add<MyPrismjsStyleExtension>();

});

````

任何时候当 `PrismjsStyleBundleContributor` 被添加到bundle中时, `MyPrismjsStyleExtension` 也会被自动添加.

#### 访问 IServiceProvider

虽然很少需要它, 但是`BundleConfigurationContext`有一个`ServiceProvider`属性, 你可以在`ConfigureBundle`方法中解析服务依赖.

#### 标准包装贡献者

将特定的NPM包资源(js,css文件)添加到包中对于该包非常简单. 例如, 你总是为bootstrap NPM包添加`bootstrap.css`文件.

所有[标准NPM包](Client-Side-Package-Management.md)都有内置的贡献者. 例如,如果你的贡献者依赖于bootstrap,你可以声明它,而不是自己添加bootstrap.css.

````C#

[DependsOn(typeof(BootstrapStyleContributor))] //Define the bootstrap style dependency

public class MyExtensionStyleBundleContributor : BundleContributor

{

//...

}

````

使用标准包的内置贡献者:

\* 防止你输入**\*\*无效的资源路径\*\***.

\* 如果资源 **\*\*路径发生变化\*\*** (依赖贡献者将处理它),则防止更改你的贡献者.

\* 防止多个模块添加**\*\*重复文件\*\***.

\* 以递归方式管理依赖项(如果需要,添加依赖项的依赖项).

##### Volo.Abp.AspNetCore.Mvc.UI.Packages 包

> 默认情况下已在启动模板安装此软件包. 大多数情况下,你不需要手动安装它.

标准包贡献者在`Volo.Abp.AspNetCore.Mvc.UI.Packages` NuGet包中定义.

将它安装到你的项目中:

````

install-package Volo.Abp.AspNetCore.Mvc.UI.Packages

````

然后将`AbpAspNetCoreMvcUiPackagesModule`模块依赖项添加到你的模块中;

````C#

using Volo.Abp.Modularity;

using Volo.Abp.AspNetCore.Mvc.UI.Bundling;

namespace MyCompany.MyProject

{

[DependsOn(typeof(AbpAspNetCoreMvcUiPackagesModule))]

public class MyWebModule : AbpModule

{

//...

}

}

````

#### Bundle 继承

在某些特定情况下, 可能需要从其他bundle创建一个 **\*\*新\*\*** bundle **\*\*继承\*\***, 从bundle继承(递归)会继承该bundle的所有文件/贡献者. 然后派生的bundle可以添加或修改文件/贡献者**\*\*而无需修改\*\***原始bundle.

例如:

````c#

services.Configure<AbpBundlingOptions>(options =>

{

options

.StyleBundles

.Add("MyTheme.MyGlobalBundle", bundle => {

bundle

.AddBaseBundles("MyGlobalBundle") //Can add multiple

.AddFiles(

"/styles/mytheme-global-styles.css"

);

});

});

````

### 主题

主题使用标准包贡献者将库资源添加到页面布局. 主题还可以定义一些标准/全局包, 因此任何模块都可以为这些标准/全局包做出贡献. 有关更多信息, 请参阅[主题文档](Theming.md).

### 最佳实践 & 建议

建议为应用程序定义多个包, 每个包用于不同的目的.

\* **\*\*全局包\*\***: 应用程序中的每个页面都包含全局样式/脚本包. 主题已经定义了全局样式和脚本包. 你的模块可以为他们做出贡献.

\* **\*\*布局包\*\***: 这是针对单个布局的特定包. 仅包含在所有页面之间共享的资源使用布局. 使用bundling tag helpers创建捆绑包是一种很好的做法.

\* **\*\*模块包\*\***: 用于单个模块页面之间的共享资源.

\* **\*\*页面包\*\***: 为每个页面创建的特定包. 使用bundling tag helpers创建捆绑包作为最佳实践.

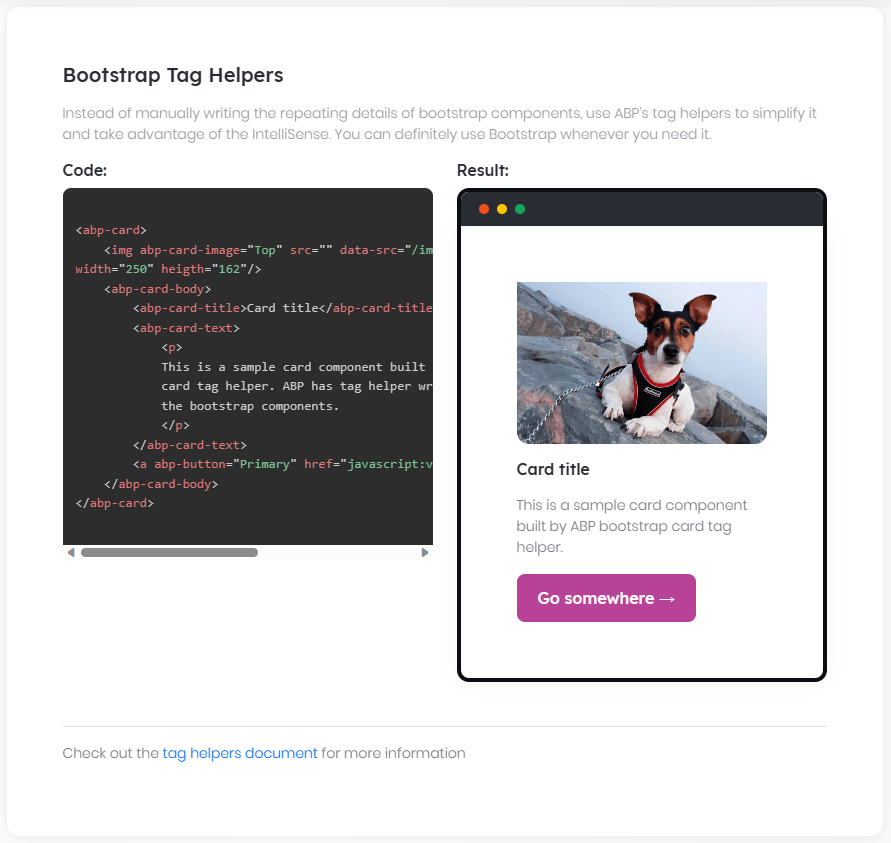
在性能,网络带宽使用和捆绑包的数量之间建立平衡.

### 参见

\* [客户端包管理](Client-Side-Package-Management.md)

\* [主题](Theming.md)

Tag Helpers



# ABP Tag Helpers

ABP框架定义了一组**\*\*标签助手组件\*\***. 简化开发ASP.NET Core (MVC / Razor Pages) 应用程序界面.

## bootstrap 组件包装

大多数标签助手是[Bootstrap](https://getbootstrap.com/) (v5+)的包装. 编写bootstrap代码并不是那么简单,其中包含太多的重复HTML标签并且也没有类型安全. ABP标签助手使其 **\*\*简单\*\*** 并且 **\*\*类型安全\*\***.

我们的目标并不是100%的包装bootstrap组件. 仍然可以编写 **\*\*原生bootstrap代码\*\*** (实际上标签助手生成的也是原生的bootstrap代码), 但我们建议尽量使用标签助手.

ABP框架还向标准bootstrap组件添加了一些**\*\*实用的功能\*\***.

这里是ABP框架包装的组件列表:

\* [Alerts](Alerts.md)

\* [Buttons](Buttons.md)

\* [Cards](Cards.md)

\* [Collapse](Collapse.md)

\* [Dropdowns](Dropdowns.md)

\* [Grids](Grids.md)

\* [List Groups](List-Groups.md)

\* [Modals](Modals.md)

\* [Paginator](Paginator.md)

\* [Popovers](Popovers.md)

\* [Progress Bars](Progress-Bars.md)

\* [Tabs](Tabs.md)

\* [Tooltips](Tooltips.md)

\* ...

> 在为所有的标签助手完成文档之前,你可以访问 https://bootstrap-taghelpers.abp.io/ 查看在线示例.

## 表单元素

参阅 [demo](https://bootstrap-taghelpers.abp.io/Components/FormElements).

## 动态表单

参阅 [demo](https://bootstrap-taghelpers.abp.io/Components/DynamicForms).

#### 表单元素

# 表单元素

## 简介

Abp提供表单输入标签助手以便更轻松地构建表单。

## 演示

请查看[表单元素演示页面](https://bootstrap-taghelpers.abp.io/Components/FormElements)。

## abp-input

`abp-input`标签为给定的C#属性创建一个Bootstrap表单输入。它在后台使用了[Asp.Net Core Input Tag Helper](https://docs.microsoft.com/tr-tr/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-3.1#the-input-tag-helper)，所以Asp.Net Core的每个input标签助手的数据注释属性对于`abp-input`也有效。

用法：

````xml

<abp-input asp-for="@Model.MyModel.Name"/>

<abp-input asp-for="@Model.MyModel.Description"/>

<abp-input asp-for="@Model.MyModel.Password"/>

<abp-input asp-for="@Model.MyModel.IsActive"/>

````

Model:

````csharp

public class FormElementsModel : PageModel

{

public SampleModel MyModel { get; set; }

public void OnGet()

{

MyModel = new SampleModel();

}

public class SampleModel

{

[Required]

[Placeholder("Enter your name...")]

[InputInfoText("What is your name?")]

public string Name { get; set; }

[Required]

[FormControlSize(AbpFormControlSize.Large)]

public string SurName { get; set; }

[TextArea(Rows = 4)]

public string Description { get; set; }

[Required]

[DataType(DataType.Password)]

public string Password { get; set; }

public bool IsActive { get; set; }

}

}

````

### 特性(Attributes)

你可以在你的c#属性上设置一些特性，或者直接在html标签上设置。如果您要在[abp-dynamic-form](Dynamic-forms.md)中使用此属性，则只能通过属性特性设置这些属性。

#### 属性特性(Property Attributes)

- `[TextArea()]`: 将输入转换为文本区域。

\* `[Placeholder()]`: 为输入设置占位符。您可以直接使用本地化键。

\* `[InputInfoText()]`: 为输入设置小型信息文本。您可以直接使用本地化键。

\* `[FormControlSize()]`: 设置表单控件包装器元素的大小。可用的值为

- `AbpFormControlSize.Default`

- `AbpFormControlSize.Small`

- `AbpFormControlSize.Medium`

- `AbpFormControlSize.Large`

\* `[DisabledInput]` : 输入被禁用。

\* `[ReadOnlyInput]`: 输入为只读。

#### 标签属性(Tag Attributes)

\* `info`: 为输入设置小型信息文本。您可以直接使用本地化键。

\* `auto-focus`: 如果为true，则浏览器会自动聚焦在该元素上。

\* `size`: 设置表单控件包装器元素的大小。可用的值为

- `AbpFormControlSize.Default`

- `AbpFormControlSize.Small`

- `AbpFormControlSize.Medium`

- `AbpFormControlSize.Large`

\* `disabled`: 输入被禁用。

\* `readonly`: 输入为只读。

\* `label`: 为输入设置标签。

\* `display-required-symbol`: 如果输入为必填项，则向标签添加必需符号（\*）。默认为`True`。

[Asp.Net Core Input Tag Helper](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-3.1#the-input-tag-helper)的`asp-format`、`name`和`value`属性也对`abp-input`标签助手有效。

### 标签和本地化

您可以通过不同的方式设置输入的标签：

- 您可以使用`Label`属性直接设置标签。 但是，它不会自动本地化您的本地化键。 因此，请使用`label = "@L [“ {LocalizationKey}”] .Value"`。

- 您可以使用Asp.Net Core的`[Display(name = "{LocalizationKey}")]`属性设置标签。

- 您可以让 **\*\*abp\*\*** 查找属性的本地化键。 如果未设置`label`或`[DisplayName]`属性，则会尝试查找“DisplayName：{PropertyName}”或“{PropertyName}”本地化键。

## abp-select

`abp-select` 标签为给定的 C# 属性创建了一个 Bootstrap 表单选择器。它在后台使用 [Asp.Net Core 选择标签助手](https://docs.microsoft.com/tr-tr/aspnet/core/mvc/views/working-with-forms?view=aspnetcore-3.1#the-select-tag-helper)，因此 `Asp.Net Core` 的 `select` 标签助手的每个数据注释属性也适用于 `abp-select`。

`abp-select` 标签需要一个 `Microsoft.AspNetCore.Mvc.Rendering.SelectListItem` 的列表来工作。它可以通过标签上的 `asp-items` 属性或 C# 属性上的 `[SelectItems()]` 属性来提供（如果您使用的是 [abp-dynamic-form](Dynamic-forms.md)，则只能使用 C# 属性的方式。）

`abp-select` 支持多重选择。

`abp-select` 自动为 **\*\*枚举（Enum）\*\*** 属性创建选择列表。不需要额外的数据。如果属性是可空的，则会在自动生成的列表顶部添加一个空键和值。

使用：

````xml

<abp-select asp-for="@Model.MyModel.City" asp-items="@Model.CityList"/>

<abp-select asp-for="@Model.MyModel.AnotherCity"/>

<abp-select asp-for="@Model.MyModel.MultipleCities" asp-items="@Model.CityList"/>

<abp-select asp-for="@Model.MyModel.MyCarType"/>

<abp-select asp-for="@Model.MyModel.MyNullableCarType"/>

````

Model:

````csharp

public class FormElementsModel : PageModel

{

public SampleModel MyModel { get; set; }

public List<SelectListItem> CityList { get; set; }

public void OnGet()

{

MyModel = new SampleModel();

CityList = new List<SelectListItem>

{

new SelectListItem { Value = "NY", Text = "New York"},

new SelectListItem { Value = "LDN", Text = "London"},

new SelectListItem { Value = "IST", Text = "Istanbul"},

new SelectListItem { Value = "MOS", Text = "Moscow"}

};

}

public class SampleModel

{

public string City { get; set; }

[SelectItems(nameof(CityList))]

public string AnotherCity { get; set; }

public List<string> MultipleCities { get; set; }

public CarType MyCarType { get; set; }

public CarType? MyNullableCarType { get; set; }

}

public enum CarType

{

Sedan,

Hatchback,

StationWagon,

Coupe

}

}

````

### 属性

您可以在 C# 属性上或直接在 HTML 标签上设置一些属性。如果您将在 [abp-dynamic-form](Dynamic-forms.md) 中使用此属性，则只能通过属性属性设置这些属性。

#### 属性特性(Property Attributes)

\* `[SelectItems()]`: 设置选择数据。参数应为数据列表的名称。（见上面的示例）

- `[InputInfoText()]`：设置输入的小信息文本。您可以直接使用本地化键。

- `[FormControlSize()]`：设置表单控件包装元素的大小。可用值为

- `AbpFormControlSize.Default`

- `AbpFormControlSize.Small`

- `AbpFormControlSize.Medium`

- `AbpFormControlSize.Large`

#### 标签属性(Tag Attributes)

- `asp-items`: 设置选择数据。这应该是一个 `SelectListItem` 列表。

- `info`: 为输入设置一个小的信息文本。您可以直接使用本地化键。

- `size`: 设置表单控件包装元素的大小。可用值为

- `AbpFormControlSize.Default`

- `AbpFormControlSize.Small`

- `AbpFormControlSize.Medium`

- `AbpFormControlSize.Large`

- `label`: 为输入设置标签。

- `display-required-symbol`: 如果输入是必需的，则向标签添加必需符号 (\*)。默认为 `True`。

- `floating-label`: 设置输入的标签是否应该是浮动的。默认为 `False`。

### 标签和本地化

您可以以不同的方式设置输入的标签：

- 您可以使用 `Label` 属性并直接设置标签。但是它不会自动本地化您的本地化键。因此，请将其用作 `label="@L["{LocalizationKey}"].Value"`。

- 您可以使用 Asp.Net Core 的 `[Display(name="{LocalizationKey}")]` 属性进行设置。

- 您可以让 **\*\*abp\*\*** 查找属性的本地化键。它将尝试查找“DisplayName:{PropertyName}”或“{PropertyName}”本地化键。

对于**\*\*枚举\*\***属性，`abp-select`会设置下拉框的本地化值。它会查找"{EnumTypeName}.{EnumPropertyName}"或"{EnumPropertyName}"的本地化键。例如，在上面的示例中，当它本地化下拉框的值时，它将使用"CarType.StationWagon"或"StationWagon"键。

## abp-radio

`abp-radio`标签为给定的C#属性创建一个Bootstrap表单单选框组。使用方式与`abp-select`标签非常相似。

Usage:

````xml

<abp-radio asp-for="@Model.MyModel.CityRadio" asp-items="@Model.CityList" inline="true"/>

<abp-radio asp-for="@Model.MyModel.CityRadio2"/>

````

Model:

````csharp

public class FormElementsModel : PageModel

{

public SampleModel MyModel { get; set; }

public List<SelectListItem> CityList { get; set; } = new List<SelectListItem>

{

new SelectListItem { Value = "NY", Text = "New York"},

new SelectListItem { Value = "LDN", Text = "London"},

new SelectListItem { Value = "IST", Text = "Istanbul"},

new SelectListItem { Value = "MOS", Text = "Moscow"}

};

public void OnGet()

{

MyModel = new SampleModel();

MyModel.CityRadio = "IST";

MyModel.CityRadio2 = "MOS";

}

public class SampleModel

{

public string CityRadio { get; set; }

[SelectItems(nameof(CityList))]

public string CityRadio2 { get; set; }

}

}

````

### 特性(attributes)

您可以在C#属性或直接在HTML标签上设置一些属性。如果您将在[abp-dynamic-form](Dynamic-forms.md)中使用此属性，则只能通过属性属性设置这些属性。

#### 属性特性(Property Attributes)

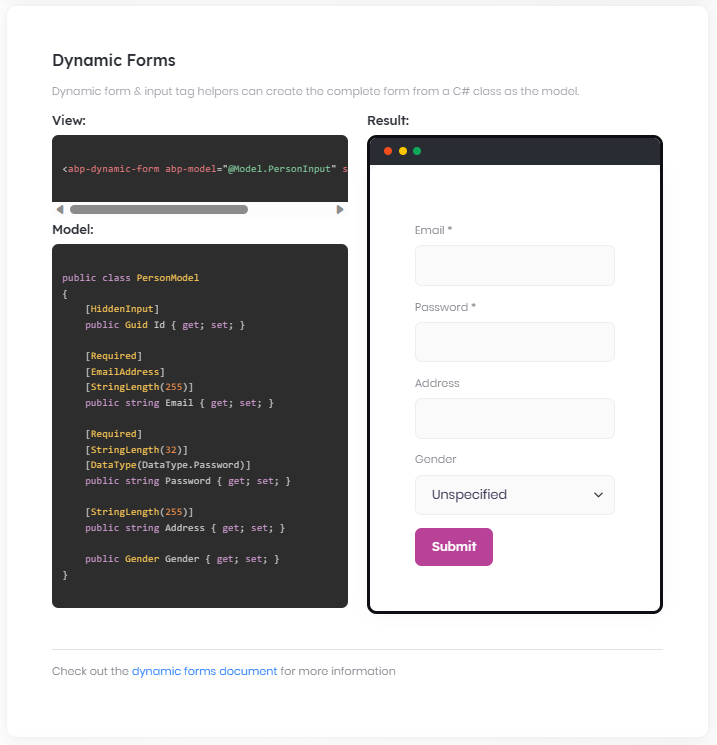
- `[SelectItems()]`:设置选择数据。参数应为数据列表的名称。 （见上面的示例）

#### 标签特性(Tag Attributes)

- `asp-items`: 设置选择数据。这应该是一个SelectListItem列表。

- `Inline`: 如果为true，则单选按钮将在单行中，相互紧挨着。如果为false，则它们将在彼此下面。

#### 动态表单



# 动态表单

`提示:` 在开始阅读本文档之前,请确保你已经看过并理解了[abp表单元素](Form-elements.md)文档.

## 介绍

`abp-dynamic-form` 为给定c#模型创建bootstrap表单.

基本用法:

````xml

<abp-dynamic-form abp-model="@Model.MyDetailedModel"/>

````

Model:

````csharp

public class DynamicFormsModel : PageModel

{

[BindProperty]

public DetailedModel MyDetailedModel { get; set; }

public List<SelectListItem> CountryList { get; set; } = new List<SelectListItem>

{

new SelectListItem { Value = "CA", Text = "Canada"},

new SelectListItem { Value = "US", Text = "USA"},

new SelectListItem { Value = "UK", Text = "United Kingdom"},

new SelectListItem { Value = "RU", Text = "Russia"}

};

public void OnGet()

{

MyDetailedModel = new DetailedModel

{

Name = "",

Description = "Lorem ipsum dolor sit amet.",

IsActive = true,

Age = 65,

Day = DateTime.Now,

MyCarType = CarType.Coupe,

YourCarType = CarType.Sedan,

Country = "RU",

NeighborCountries = new List<string>() { "UK", "CA" }

};

}

public class DetailedModel

{

[Required]

[Placeholder("Enter your name...")]

[Display(Name = "Name")]

public string Name { get; set; }

[TextArea(Rows = 4)]

[Display(Name = "Description")]

[InputInfoText("Describe Yourself")]

public string Description { get; set; }

[Required]

[DataType(DataType.Password)]

[Display(Name = "Password")]

public string Password { get; set; }

[Display(Name = "Is Active")]

public bool IsActive { get; set; }

[Required]

[Display(Name = "Age")]

public int Age { get; set; }

[Required]

[Display(Name = "My Car Type")]

public CarType MyCarType { get; set; }

[Required]

[AbpRadioButton(Inline = true)]

[Display(Name = "Your Car Type")]

public CarType YourCarType { get; set; }

[DataType(DataType.Date)]

[Display(Name = "Day")]

public DateTime Day { get; set; }

[SelectItems(nameof(CountryList))]

[Display(Name = "Country")]

public string Country { get; set; }

[SelectItems(nameof(CountryList))]

[Display(Name = "Neighbor Countries")]

public List<string> NeighborCountries { get; set; }

}

public enum CarType

{

Sedan,

Hatchback,

StationWagon,

Coupe

}

}

````

## Demo

参阅 [动态表单demo页面](https://bootstrap-taghelpers.abp.io/Components/Dropdowns)查看示例.

## Attributes

### abp-model

为动态表单设置c#模型,模型的属性以表单形式转化为输入.

### column-size

此处使用 `col-sm` 来设置大小。当设置该属性是会同时添加 `col-12` .

### submit-button

可以为 `True` 或 `False`.

如果为 `True`,则会在表单底部生成一个提交按钮.

默认值是 `False`.

### required-symbols

可以为 `True` 或 `False`.

如果为 `True`,则必需的输入将带有一个符号(\*),表示它们是必需的.

默认值是 `True`.

## 表单内容布局

默认情况下,“`abp-dynamic-form` 会清除内部html并将inputs放入自身. 如果要向动态表单添加其他内容或将inputs放置到某些特定区域,可以使用`<abp-form-content />`标签. 这个标签将被表单内容替换, 而 `abp-dynamic-form` 标签的内部html的其余部分将保持不变.

用法:

````xml

<abp-dynamic-form abp-model="@Model.MyExampleModel">

<div>

Some content....

</div>

<div class="input-area">

<abp-form-content />

</div>

<div>

Some more content....

</div>

</abp-dynamic-form>

````

## 输入排序

`abp-dynamic-form` 通过 `DisplayOrder` attribute对属性进行排序,然后按模型类中的属性顺序进行排序.

默认每个属性的 `DisplayOrder` attribute值是10000.

参见以下示例:

````csharp

public class OrderExampleModel

{

[DisplayOrder(10004)]

public string Name{ get; set; }

[DisplayOrder(10005)]

public string Surname{ get; set; }

//Default 10000

public string EmailAddress { get; set; }

[DisplayOrder(10003)]

public string PhoneNumber { get; set; }

[DisplayOrder(9999)]

public string City { get; set; }

}

````

在这个示例中,inputs字段顺序为: `City` > `EmailAddress` > `PhoneNumber` > `Name` > `Surname`.

## 忽略属性

默认情况下, `abp-dynamic-form` 会为模型类中的每个属性生成输入. 如果要忽略属性请使用 `DynamicFormIgnore` attribute.

参见以下示例:

````csharp

public class MyModel

{

public string Name { get; set; }

[DynamicFormIgnore]

public string Surname { get; set; }

}

````

在这个示例中,不会为 `Surname` 属性生成输入.

## 指示文本框,单选按钮组和组合框

如果你已经阅读了[表单元素文档](Form-elements.md),你会注意到在c#模型上 `abp-radio` 和 `abp-select` 标签非常相. 我们必须使用 `[AbpRadioButton()]` attribute来告诉 `abp-dynamic-form` 你的哪些属性是单选按钮组,哪些属性是组合框.

参见以下示例:

````xml

<abp-dynamic-form abp-model="@Model.MyDetailedModel"/>

````

Model:

````csharp

public class DynamicFormsModel : PageModel

{

[BindProperty]

public DetailedModel MyDetailedModel { get; set; }

public List<SelectListItem> CountryList { get; set; } = new List<SelectListItem>

{

new SelectListItem { Value = "CA", Text = "Canada"},

new SelectListItem { Value = "US", Text = "USA"},

new SelectListItem { Value = "UK", Text = "United Kingdom"},

new SelectListItem { Value = "RU", Text = "Russia"}

};

public void OnGet()

{

MyDetailedModel = new DetailedModel

{

ComboCarType = CarType.Coupe,

RadioCarType = CarType.Sedan,

ComboCountry = "RU",

RadioCountry = "UK"

};

}

public class DetailedModel

{

public CarType ComboCarType { get; set; }

[AbpRadioButton(Inline = true)]

public CarType RadioCarType { get; set; }

[SelectItems(nameof(CountryList))]

public string ComboCountry { get; set; }

[AbpRadioButton()]

[SelectItems(nameof(CountryList))]

public string RadioCountry { get; set; }

}

public enum CarType

{

Sedan,

Hatchback,

StationWagon,

Coupe

}

}

````

正如你上面的例子中看到:

\* 如果在**\*\*Enum\*\***属性上使用 `[AbpRadioButton()]`,它将是一个单选按钮组. 否则它是组合框.

\* 如果在属性上使用 `[SelectItems()]` 和 `[AbpRadioButton()]`,那么它将是一个单选按钮组.

\* 如果只在属性上使用 `[SelectItems()]`,它将是一个组合框.

\* 如果一个属性没有使用这些属性,它将是一个文本框.

## 本地化

`abp-dynamic-form` 会处理本地化.

默认情况下, 它将尝试查找 "DisplayName:{PropertyName}" 或 "{PropertyName}" 定位本地化键,并将定位值设置为label.

你可以使用Asp.Net Core的 `[Display()]` attribute自行设置. 可以在此属性中使用本地化密钥. 请参阅以下示例:

````csharp

[Display(Name = "Name")]

public string Name { get; set; }

````

仪表板和小部件(Widget)系统

# 小部件

ABP为创建**\*\*可重用的部件\*\***提供了模型和基础设施. 部件系统是[ASP.NET Core ViewComponents](https://docs.microsoft.com/en-us/aspnet/core/mvc/views/view-components)的扩展. 在你有以下需求时,小部件会非常有用;

\* 在可复用的 **\*\*[**模块**](../../Module-Development-Basics.md)\*\*** 中定义部件.

\* 在部件中引用 **\*\*scripts & styles\*\*** 脚本.

\* 使用部件创建 **\*\*仪表盘\*\***.

\* 支持 **\*\*[**授权**](../../Authorization.md)\*\*** 与 **\*\*[**捆绑`bundling`**](Bundling-Minification.md)\*\*** 的部件

## 基本部件定义

### 创建一个视图组件

第一部,创建一个新的ASP.NET Core View Component:

![widget-basic-files](../../images/widget-basic-files.png)

**\*\*MySimpleWidgetViewComponent.cs\*\***:

````csharp

using Microsoft.AspNetCore.Mvc;

using Volo.Abp.AspNetCore.Mvc;

namespace DashboardDemo.Web.Pages.Components.MySimpleWidget

{

public class MySimpleWidgetViewComponent : AbpViewComponent

{

public IViewComponentResult Invoke()

{

return View();

}

}

}

````

继承 `AbpViewComponent` 不是必需的. 你也可以继承ASP.NET Core的 `ViewComponent`. `AbpViewComponent` 只是定义了一些基本的实用属性.

**\*\*Default.cshtml\*\***:

```xml

<div class="my-simple-widget">

<h2>My Simple Widget</h2>

<p>This is a simple widget!</p>

</div>

```

### 定义部件

添加 `Widget` attribute 到 `MySimpleWidgetViewComponent` 类,将此视图组件标记为部件:

````csharp

using Microsoft.AspNetCore.Mvc;

using Volo.Abp.AspNetCore.Mvc;

using Volo.Abp.AspNetCore.Mvc.UI.Widgets;

namespace DashboardDemo.Web.Pages.Components.MySimpleWidget

{

[Widget]

public class MySimpleWidgetViewComponent : AbpViewComponent

{

public IViewComponentResult Invoke()

{

return View();

}

}

}

````

## 渲染部件

渲染部件的用法是ASP.NET Core的标准用法. 在razor view/page中使用 `Component.InvokeAsync` 方法, 就像渲染一个View Component一样. 例如:

````xml

@await Component.InvokeAsync("MySimpleWidget")

@await Component.InvokeAsync(typeof(MySimpleWidgetViewComponent))

````

第一行代码使用名称渲染了部件,第二行代码使用type渲染了View Comonent.

## 部件名称

默认下名称是根据View Conponent组件的名称计算的, 比如你的视图组件名是 `MySimpleWidgetViewComponent`, 那么部件的名称就是 `MySimpleWidget` (删除`ViewComponent`后缀). 这与ASP.NET Core的默认视图组件名称的方式一样.

想要自定义组件名称,只需要使用ASP.NET Core的 `ViewComponent` attribute:

```csharp

using Microsoft.AspNetCore.Mvc;

using Volo.Abp.AspNetCore.Mvc;

using Volo.Abp.AspNetCore.Mvc.UI.Widgets;

namespace DashboardDemo.Web.Pages.Components.MySimpleWidget

{

[Widget]

[ViewComponent(Name = "MyCustomNamedWidget")]

public class MySimpleWidgetViewComponent : AbpViewComponent

{

public IViewComponentResult Invoke()

{

return View("~/Pages/Components/MySimpleWidget/Default.cshtml");

}

}

}

```

ABP会通过自定义的名称去处理部件.

> 如果视图组件名与视图组件的文件夹名称不匹配,那么需要像本例中那样去手动编写视图路径.

### 显示名称

你还可以定义对于使用者友好的本地化显示名称. 需要时在UI中使用显示名称. 显示名称是可选的,在 `Widget` attribute 的`DisplayName`属性中定义:

````csharp

using DashboardDemo.Localization;

using Microsoft.AspNetCore.Mvc;

using Volo.Abp.AspNetCore.Mvc;

using Volo.Abp.AspNetCore.Mvc.UI.Widgets;

namespace DashboardDemo.Web.Pages.Components.MySimpleWidget

{

[Widget(

DisplayName = "MySimpleWidgetDisplayName", //Localization key

DisplayNameResource = typeof(DashboardDemoResource) //localization resource

)]

public class MySimpleWidgetViewComponent : AbpViewComponent

{

public IViewComponentResult Invoke()

{

return View();

}

}

}

````

参阅 [本地化文档](../../Localization.md) 学习关于本地化资源的更多内容.

## 引用 Style & Script

当部件含有样式和scirpt文件时,会存在一些挑战;

\* 使用部件的页面应该将 **\*\*script & styles\*\*** 文件引用到页面中.

\* 页面还需要解析部件的 `依赖库/文件`.

将资源与部件正确的关联在一起时,ABP会解决这些问题. 使用正确的方法,就不用担心部件的依赖关系.

### 定义一个简单的文件路径

下面的示例中部件添加了样式和scirpt文件:

````csharp

using Microsoft.AspNetCore.Mvc;

using Volo.Abp.AspNetCore.Mvc;

using Volo.Abp.AspNetCore.Mvc.UI.Widgets;

namespace DashboardDemo.Web.Pages.Components.MySimpleWidget

{

[Widget(

StyleFiles = new[] { "/Pages/Components/MySimpleWidget/Default.css" },

ScriptFiles = new[] { "/Pages/Components/MySimpleWidget/Default.js" }

)]

public class MySimpleWidgetViewComponent : AbpViewComponent

{

public IViewComponentResult Invoke()

{

return View();

}

}

}

````

ABP会考虑到这些依赖关系, 在view/page中使用正确的方法添加部件 . 样式和script可以是物理文件也可以是虚拟文件. 它于[虚拟文件系统](../../Virtual-File-System.md)完全集成].

### 定义 Bundle

页面中使用的组件的所有资源都做为捆绑包添加(如果没有其他配置,会在生产中合并和压缩). 除了简单的添加文件,你还可以充分的利用捆绑功能.

下面的示例与上面的代码相同,但是在添加文件时文件路径替换成了 `BundleContributor`:

````csharp

using System.Collections.Generic;

using Microsoft.AspNetCore.Mvc;

using Volo.Abp.AspNetCore.Mvc;

using Volo.Abp.AspNetCore.Mvc.UI.Bundling;

using Volo.Abp.AspNetCore.Mvc.UI.Widgets;

namespace DashboardDemo.Web.Pages.Components.MySimpleWidget

{

[Widget(

StyleTypes = new []{ typeof(MySimpleWidgetStyleBundleContributor) },

ScriptTypes = new[]{ typeof(MySimpleWidgetScriptBundleContributor) }

)]

public class MySimpleWidgetViewComponent : AbpViewComponent

{

public IViewComponentResult Invoke()

{

return View();

}

}

public class MySimpleWidgetStyleBundleContributor : BundleContributor

{

public override void ConfigureBundle(BundleConfigurationContext context)

{

context.Files

.AddIfNotContains("/Pages/Components/MySimpleWidget/Default.css");

}

}

public class MySimpleWidgetScriptBundleContributor : BundleContributor

{

public override void ConfigureBundle(BundleConfigurationContext context)

{

context.Files

.AddIfNotContains("/Pages/Components/MySimpleWidget/Default.js");

}

}

}

````

捆绑系统非常强大,如果你的部件使用了JavaScript库来呈现图表, 你可以将它声明为依赖项, 如果之前未添加JavaScript库. 则会自动添加到页面中. 使用这种方式让页面使用部件时不用关心依赖项.

参阅 [捆包&压缩 文档](Bundling-Minification.md) 了解更多内容.

## 授权

某些组件可能只对通过身份验证或授权的用户可用,这时可以使用 `Widget` attribute 的以下属性:

\* `RequiresAuthentication` (`bool`): 设置为true,只有通过身份验证的用户(登录用户)可用.

\* `RequiredPolicies` (`List<string>`): 授权用户的策略名称列表. 有关策略的详细信息请参阅[授权文档](../../Authorization.md).

示例:

````csharp

using Microsoft.AspNetCore.Mvc;

using Volo.Abp.AspNetCore.Mvc;

using Volo.Abp.AspNetCore.Mvc.UI.Widgets;

namespace DashboardDemo.Web.Pages.Components.MySimpleWidget

{

[Widget(RequiredPolicies = new[] { "MyPolicyName" })]

public class MySimpleWidgetViewComponent : AbpViewComponent

{

public IViewComponentResult Invoke()

{

return View();

}

}

}

````

## 部件选项

`AbpWidgetOptions` 是 `Widget` attribute 替代, 你可以使用它去配置部件:

```csharp

Configure<AbpWidgetOptions>(options =>

{

options.Widgets.Add<MySimpleWidgetViewComponent>();

});

```

将上面的代码写到[模块](../../Module-Development-Basics.md)的 `ConfigureServices` 方法中. `AbpWidgetOptions` 可以完成 `Widget` attribute 的所有功能. 比如为组件添加样式:

````csharp

Configure<AbpWidgetOptions>(options =>

{

options.Widgets

.Add<MySimpleWidgetViewComponent>()

.WithStyles("/Pages/Components/MySimpleWidget/Default.css");

});

````

> 提示: `AbpWidgetOptions` 还可以更改现有的部件配置. 如果要修改应用程序使用的模块内的组件配置,这会很有用. 使用 `options.Widgets.Find` 获取现有的 `WidgetDefinition`.

自定义/扩展UI

# ASP.NET Core (MVC / Razor Pages) 用户界面自定义指南

本文档解释了如何重写ASP.NET Core MVC / Razor Page 应用程序依赖[应用模块](../../Modules/Index.md)的用户界面.

## 重写页面

本节介绍了[Razor 页面](https://docs.microsoft.com/zh-cn/aspnet/core/razor-pages/)开发,它是ASP.NET Core推荐的服务端渲染用户页面的方法. 预构建的模块通常使用Razor页面替代经典的MVC方式(下一节也介绍MVC模式).

你通过有三种重写页面的需求:

\* 仅**\*\*重写页面模型\*\***(C#)端执行其他逻辑,不更改UI.

\* 仅**\*\*重写Razor页面\*\***(.cshtml文件),不更改逻辑.

\* **\*\*完全重写\*\*** 页面.

### 重写页面模型 (C#)

````csharp

using System.Threading.Tasks;

using Microsoft.AspNetCore.Mvc;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Identity;

using Volo.Abp.Identity.Web.Pages.Identity.Users;

namespace Acme.BookStore.Web.Pages.Identity.Users

{

[Dependency(ReplaceServices = true)]

[ExposeServices(typeof(EditModalModel))]

public class MyEditModalModel : EditModalModel

{

public MyEditModalModel(

IIdentityUserAppService identityUserAppService,

IIdentityRoleAppService identityRoleAppService

) : base(

identityUserAppService,

identityRoleAppService)

{

}

public async override Task<IActionResult> OnPostAsync()

{

//TODO: Additional logic

await base.OnPostAsync();

//TODO: Additional logic

}

}

}

````

\* 这个类继承并替换 `EditModalModel` ,重写了 `OnPostAsync` 方法在基类代码的前后执行附加逻辑

\* 它使用 `ExposeServices` 和 `Dependency` attributes去替换这个类.

### 重写Razor页面 (.CSHTML)

同一路径下创建相同的`.cshtml`文件可以实现重写功能(razor page, razor view, view component... 等.)

#### 示例

这个示例重写了[账户模块](../../Modules/Account.md)定义的**\*\*登录页面\*\***UI

账户模块在 `Pages/Account` 文件夹下定义了 `Login.cshtml` 文件. 所以你可以在同一路径下创建文件覆盖它:

![overriding-login-cshtml](../../images/overriding-login-cshtml.png)

通常你想要拷贝模块的 `.cshtml` 原文件,然后进行需要的更改. 你可以在[这里](https://github.com/abpframework/abp/blob/dev/modules/account/src/Volo.Abp.Account.Web/Pages/Account/Login.cshtml)找到源文件. 不要拷贝 `Login.cshtml.cs` 文件,它是隐藏razor页面的代码,我们不希望覆盖它(见下节).

这就够了,接下来你可以对文件内容做你想要的更改.

### 完全重写Razo页面

也许你想要完全重写页面,Razor和页面相关的C#文件.

在这种情况下;

1. 像上面描述过的那样重写C#页面模型类,但不需要替换已存在的页面模型类.

2. 像上面描述过的那样重写Razor页面,并且更改@model指向新的页面模型

#### 示例

这个示例重写了[账户模块](../../Modules/Account.md)定义的**\*\*登录页面\*\***

创建一个继承自 `LoginModel`(定义在`Volo.Abp.Account.Web.Pages.Account`命名空间下)的页面模型类:

````csharp

public class MyLoginModel : LoginModel

{

public MyLoginModel(

IAuthenticationSchemeProvider schemeProvider,

IOptions<AbpAccountOptions> accountOptions

) : base(

schemeProvider,

accountOptions)

{

}

public override Task<IActionResult> OnPostAsync(string action)

{

//TODO: Add logic

return base.OnPostAsync(action);

}

//TODO: Add new methods and properties...

}

````

如果需要,你可以重写任何方法或添加新的属性/方法

> 注意我们没有使用 `[Dependency(ReplaceServices = true)]` 或 `[ExposeServices(typeof(LoginModel))]`,因为我们不想替换依赖注入中已存在的类,我们定义了一个新的.

拷贝 `Login.cshtml` 到你们解决方案,更改 **\*\*@model\*\*** 指定到 `MyLoginModel`:

````xml

@page

...

@model Acme.BookStore.Web.Pages.Account.MyLoginModel

...

````

这就够了,接下来你可以做任何想要更改.

#### 不使用继承替换页面模型

你不需要继承源页面模型类(像之前的示例). 你可以完全**\*\*重写实现\*\***你自己的页面. 在这种事情下你可以从 `PageModel`,`AbpPageModel` 或任何你需要的合适的基类派生.

## 重写视图组件

在ABP框架,预构建的模块和主题定义了一些**\*\*可重用的视图组件\*\***. 这些视图组件可以像页面一样被替换.

### 示例

下面是应用程序启动模板自带的 **\*\*基本主题\*\*** 的截图.

![bookstore-brand-area-highlighted](../../images/bookstore-brand-area-highlighted.png)

[基本主题](../../Themes/Basic.md) 为layout定义了一些视图组件. 例如上面带有红色矩形的突出显示区域称为 **\*\*Brand组件\*\***, 你可能想添加自己的**\*\*自己的应用程序logo\*\***来自定义此组件. 让我们来看看如何去做.

首先创建你的logo并且放到你的web应用程序文件夹中,我们使用 `wwwroot/logos/bookstore-logo.png` 路径. 然后在 `Themes/Basic/Components/Brand` 文件夹下复制[Brand组件视图](https://github.com/abpframework/abp/blob/dev/modules/basic-theme/src/Volo.Abp.AspNetCore.Mvc.UI.Theme.Basic/Themes/Basic/Components/Brand/Default.cshtml). 结果应该是类似下面的图片:

![bookstore-added-brand-files](../../images/bookstore-added-brand-files.png)

然后对 `Default.cshtml` 文件做你想要的更改. 例如内容可以是这样的:

````xml

<a href="/">

<img src="~/logos/bookstore-logo.png" width="250" height="60"/>

</a>

````

现在你可以运行应用程序看到结果:

![bookstore-added-logo](../../images/bookstore-added-logo.png)

如果你需要,你也可以仅使用依赖注入系统替换组件[背后的C#类代码](https://github.com/abpframework/abp/blob/dev/modules/basic-theme/src/Volo.Abp.AspNetCore.Mvc.UI.Theme.Basic/Themes/Basic/Components/Brand/MainNavbarBrandViewComponent.cs)

### 重写主题

正如上所解释的,你可以更改任何组件,layout或c#类. 参阅[主题文档]了解更多关于主题系统的信息.

## 重写静态资源

重写模块的静态资源(像JavaScript,Css或图片文件)是很简单的. 只需要在解决方案的相同路径创建文件,[虚拟文件系统](../../Virtual-File-System.md)会自动处理它.

## 操作捆绑

[捆绑 & 压缩](Bundling-Minification.md) 系统提供了**\*\*动态可扩展的\*\*** 系统去创建**\*\*script\*\***和**\*\*style\*\***捆绑. 它允许你扩展和操作现有的包.

### 示例: 添加全局CSS文件

例如APP框架定义了一个**\*\*全局样式捆绑\*\***添加到所有的页面(事实上由主题添加layout). 让我们添加一个**\*\*自定义样式文件\*\***到这个捆绑文件的最后,我们可以覆盖任何全局样式.

创建在 `wwwroot` 文件夹下创建一个CSS文件

![bookstore-global-css-file](../../images/bookstore-global-css-file.png)

在CSS文件中定义一些规则. 例如:

````css

.card-title {

color: orange;

font-size: 2em;

text-decoration: underline;

}

.btn-primary {

background-color: red;

}

````

然后在你的[模块](../../Module-Development-Basics.md) `ConfigureServices` 方法添加这个文件到标准的全局样式捆绑包:

````csharp

Configure<AbpBundlingOptions>(options =>

{

options.StyleBundles.Configure(

StandardBundles.Styles.Global, //The bundle name!

bundleConfiguration =>

{

bundleConfiguration.AddFiles("/styles/my-global-styles.css");

}

);

});

````

#### 全局脚本捆绑包

就像 `StandardBundles.Styles.Global` 一样,还有一个 `StandardBundles.Scripts.Global`,你可以添加文件或操作现有文件.

### 示例: 操作捆绑包文件

上面的示例中添加了新文件到捆绑包. 如果你创建 **\*\*bundle contributor\*\*** 类则可以做到更多. 示例:

````csharp

public class MyGlobalStyleBundleContributor : BundleContributor

{

public override void ConfigureBundle(BundleConfigurationContext context)

{

context.Files.Clear();

context.Files.Add("/styles/my-global-styles.css");

}

}

````

然后你可以添加这个contributor到已存在的捆绑中:

````csharp

Configure<AbpBundlingOptions>(options =>

{

options.StyleBundles.Configure(

StandardBundles.Styles.Global,

bundleConfiguration =>

{

bundleConfiguration.AddContributors(typeof(MyGlobalStyleBundleContributor));

}

);

});

````

示例中清除了所有的CSS文件,在现实中这并不是一个好主意,你可以找到某个特定的文件替换成你自己的文件.

### 示例: 为特定页面添加JavaScript文件

上面的示例将全局包添加到布局中. 如果要在依赖模块中为特定页面定义添加CSS/JavaScript文件(或替换文件)怎么做?

假设你想要用户进入身份模块的**\*\*角色管理\*\***页面时运行**\*\*JavaScript代码\*\***.

首先在 `wwwroot`, `Pages` 或 `Views` 文件夹下创建一个标准的JavaScript文件(默认ABP支持这些文件夹下的静态文件). 根据约定我们推荐 `Pages/Identity/Roles` 文件夹:

![bookstore-added-role-js-file](../../images/bookstore-added-role-js-file.png)

该文件的内容很简单:

````js

$(function() {

abp.log.info('My custom role script file has been loaded!');

});

````

然后将这个文件添加到角色管页面理捆绑包中:

````csharp

Configure<AbpBundlingOptions>(options =>

{

options.ScriptBundles

.Configure(

typeof(Volo.Abp.Identity.Web.Pages.Identity.Roles.IndexModel).FullName,

bundleConfig =>

{

bundleConfig.AddFiles("/Pages/Identity/Roles/my-role-script.js");

});

});

````

`typeof(Volo.Abp.Identity.Web.Pages.Identity.Roles.IndexModel).FullName` 是获取角色管理页面捆绑包名称的安全方式:

> 请注意并非每个页面都定义了这个页面的捆绑包. 它们仅在需要时定义.

除了添加新的CSS/JavaScript文件到页面,你也可以以替换(通过捆绑包contributor)已存在.

## 布局定制

布局由主题([参阅主题](Theming.md))定义设计. 它们不包含在下载的应用程序解决方案中. 通过这种方式你可以轻松的**\*\*更改\*\***主题并获取新的功能. 你不能**\*\*直接更改\*\***应用程序中的布局代码,除非你用自己的布局替换它(在下一部分中说明).

有一些通用的方法可以**\*\*自定义布局\*\***,将在下一节中介绍.

### 菜单贡献者

ABP框架定义了两个**\*\*标准菜单\*\***:

![bookstore-menus-highlighted](../../images/bookstore-menus-highlighted.png)

\* `StandardMenus.Main`: 应用程序的主菜单.

\* `StandardMenus.User`: 用户菜单 (通常在屏幕的右上方).

显示菜单是主题的责任,但**\*\*菜单项\*\***由模板和你的应用程序代码决定. 只需要实现 `IMenuContributor` 接口并在 `ConfigureMenuAsync` 方法操作菜单项.

渲染菜单时需要执行菜单贡献者. **\*\*应用程序启动模板\*\*** 已经定义了菜单贡献者,所以你可以使用它. 参阅[导航菜单](Navigation-Menu.md)文档了解更多.

### 工具栏贡献者

[工具栏系统](Toolbars.md)用于在用户界面定义 **\*\*工具栏\*\*** . 模块 (或你的应用程序)可以将 **\*\*项\*\*** 添加到工具栏, 随后主题将在**\*\*布局\*\***上呈现工具栏.

只有一个 **\*\*标准工具栏\*\*** (名称为 "Main" - 定义为常量: `StandardToolbars.Main`). 对于基本主题,按如下呈现:![bookstore-toolbar-highlighted](../../images/bookstore-toolbar-highlighted.png)

在上面的屏幕快照中,主工具栏添加了两个项目:语言开关组件和用户菜单. 你可以在此处添加自己的项.

#### 示例: 添加通知图标

在这个示例中,我们会添加一个**\*\*通知(响铃)图标\*\***到语言切换项的左侧. 工具栏的项项目是一个**\*\*视图组件\*\***. 所以,在你的项目中创建一个新的视图组件:

![bookstore-notification-view-component](../../images/bookstore-notification-view-component.png)

**\*\*NotificationViewComponent.cs\*\***

````csharp

public class NotificationViewComponent : AbpViewComponent

{

public async Task<IViewComponentResult> InvokeAsync()

{

return View("/Pages/Shared/Components/Notification/Default.cshtml");

}

}

````

**\*\*Default.cshtml\*\***

````xml

<div id="MainNotificationIcon" style="color: white; margin: 8px;">

<i class="far fa-bell"></i>

</div>

````

现在,我们创建一个类实现 `IToolbarContributor` 接口:

````csharp

public class MyToolbarContributor : IToolbarContributor

{

public Task ConfigureToolbarAsync(IToolbarConfigurationContext context)

{

if (context.Toolbar.Name == StandardToolbars.Main)

{

context.Toolbar.Items

.Insert(0, new ToolbarItem(typeof(NotificationViewComponent)));

}

return Task.CompletedTask;

}

}

````

这个类向 `Main` 工具栏的第一项添加了 `NotificationViewComponent`.

最后你需要将这个贡献者添加到 `AbpToolbarOptions`,在你模块类的 `ConfigureServices` 方法:

````csharp

Configure<AbpToolbarOptions>(options =>

{

options.Contributors.Add(new MyToolbarContributor());

});

````

这就够了,当你运行应用程序后会看到工具栏上的通知图标:

![bookstore-notification-icon-on-toolbar](../../images/bookstore-notification-icon-on-toolbar.png)

示例中的 `NotificationViewComponent` 返回没有任何数据的视图. 在实际场景中,你可能想**\*\*查询数据库\*\***(或调用HTTP API)获取通知并传递给视图. 如果需要可以将 `JavaScript` 或 `CSS` 文件添加到工具栏的全局捆绑包中(如前所述).

参阅[工具栏文档](Toolbars.md)了解更多关于工具栏系统.

### 布局钩子

[布局钩子](Layout-Hooks.md) 系统允许你在布局页面的某些特定部分 **\*\*添加代码\*\*** . 所有主题的所有布局都应该实现这些钩子. 然后你可以将**\*\*视图组件\*\***添加到钩子.

#### 示例: 添加谷歌统计

假设你想要添加谷歌统计脚本到布局(将适用所有的页面). 首先在你的项目中**\*\*创建一个视图组件\*\***:

![bookstore-google-analytics-view-component](../../images/bookstore-google-analytics-view-component.png)

**\*\*NotificationViewComponent.cs\*\***

````csharp

public class GoogleAnalyticsViewComponent : AbpViewComponent

{

public IViewComponentResult Invoke()

{

return View("/Pages/Shared/Components/GoogleAnalytics/Default.cshtml");

}

}

````

**\*\*Default.cshtml\*\***

````html

<script>

(function(i,s,o,g,r,a,m){i['GoogleAnalyticsObject']=r;i[r]=i[r]||function(){

(i[r].q=i[r].q||[]).push(arguments)},i[r].l=1\*new Date();a=s.createElement(o),

m=s.getElementsByTagName(o)[0];a.async=1;a.src=g;m.parentNode.insertBefore(a,m)

})(window,document,'script','//www.google-analytics.com/analytics.js','ga');

ga('create', 'UA-xxxxxx-1', 'auto');

ga('send', 'pageview');

</script>

````

在你自己的代码中更改 `UA-xxxxxx-1` .

然后你可以在你模块的 `ConfigureServices` 方法将这个组件添加到任何的钩子点:

````csharp

Configure<AbpLayoutHookOptions>(options =>

{

options.Add(

LayoutHooks.Head.Last, //The hook name

typeof(GoogleAnalyticsViewComponent) //The component to add

);

});

````

现在谷歌统计代码将在页面的 `head` 所为最后一项插入. 你(或你在使用的模块)可以将多个项添加到相同的钩子,它们都会添加到布局.

在上面我们添加 `GoogleAnalyticsViewComponent` 到所有的布局,你可能只想添加到指定的布局:

````csharp

Configure<AbpLayoutHookOptions>(options =>

{

options.Add(

LayoutHooks.Head.Last,

typeof(GoogleAnalyticsViewComponent),

layout: StandardLayouts.Application //Set the layout to add

);

});

````

参阅下面的布局部分,以了解有关布局系统的更多信息.

### 布局

布局系统允许主题定义标准,命名布局并且允许任何页面选择使用合适的布局. 有三种预定义的布局:

\* "**\*\*Application\*\***": 应用程序的主要(和默认)布局. 它通常包含页眉,菜单(侧栏),页脚,工具栏等.

\* "**\*\*Account\*\***": 登录,注册和其他类似页面使用此布局. 默认它用于 `/Pages/Account` 文件夹下的页面.

\* "**\*\*Empty\*\***": 空的最小的布局.

这些名称在 `StandardLayouts` 类定义为常量. 这是标准的布局名称,所有的主题开箱即用的实现. 你也可以创建自己的布局.

#### 布局位置

你可以在[这里](https://github.com/abpframework/abp/blob/dev/modules/basic-theme/src/Volo.Abp.AspNetCore.Mvc.UI.Theme.Basic/Themes/Basic/Layouts)找到基本主题的布局文件. 你可以将它们作用构建自己的布局的参考,也可以在必要时覆盖它们.

#### ITheme

ABP框架使用 `ITheme` 服务通过局部名称获取布局位置. 你可以替换此服务动态的选择布局位置.

#### IThemeManager

`IThemeManager` 用于获取当前主题,并得到了布局路径. 任何页面可以都决定自己的布局. 例:

````html

@using Volo.Abp.AspNetCore.Mvc.UI.Theming

@inject IThemeManager ThemeManager

@{

Layout = ThemeManager.CurrentTheme.GetLayout(StandardLayouts.Empty);

}

````

此页面将使用空白布局. 它使用 `ThemeManager.CurrentTheme.GetEmptyLayout()` 扩展方法.

如果你设置特定目录下所有页面的布局,可以在该文件夹下的 `\_ViewStart.cshtml` 文件编写以上代码.

测试

# ASP.NET Core MVC / Razor Pages: 测试

> 你可以参考[ASP.NET Core集成测试文档](https://docs.microsoft.com/en-us/aspnet/core/test/integration-tests)了解ASP.NET Core集成测试的详细内容. 本文档解释了ABP框架提供的附加测试基础设施.

## 应用程序启动模板

应用程序启动模板的`.Web`项目其中包含应用程序的UI视图/页面/组件, 并提供`.Web.Tests`项目来测试这些内容.

![aspnetcore-web-tests-in-solution](../../images/aspnetcore-web-tests-in-solution.png)

## 测试Razor页面

假设你已经创建了一个名为`Issues.cshtml`的Razor页面, 包含以下内容;

**\*\*Issues.cshtml.cs\*\***

````csharp

using System.Collections.Generic;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Mvc.RazorPages;

using MyProject.Issues;

namespace MyProject.Web.Pages

{

public class IssuesModel : PageModel

{

public List<IssueDto> Issues { get; set; }

private readonly IIssueAppService \_issueAppService;

public IssuesModel(IIssueAppService issueAppService)

{

\_issueAppService = issueAppService;

}

public async Task OnGetAsync()

{

Issues = await \_issueAppService.GetListAsync();

}

}

}

````

**\*\*Issues.cshtml\*\***

````html

@page

@model MyProject.Web.Pages.IssuesModel

<h2>Issue List</h2>

<table id="IssueTable" class="table">

<thead>

<tr>

<th>Issue</th>

<th>Closed?</th>

</tr>

</thead>

<tbody>

@foreach (var issue in Model.Issues)

{

<tr>

<td>@issue.Title</td>

<td>

@if (issue.IsClosed)

{

<span>Closed</span>

}

else

{

<span>Open</span>

}

</td>

</tr>

}

</tbody>

</table>

````

本页仅创建一个包含issue的表格:

![issue-list](../../images/issue-list.png)

你可以在`.Web.Tests`项目中编写一个测试类如下所示:

````csharp

using System.Threading.Tasks;

using HtmlAgilityPack;

using Shouldly;

using Xunit;

namespace MyProject.Pages

{

public class Issues\_Tests : MyProjectWebTestBase

{

[Fact]

public async Task Should\_Get\_Table\_Of\_Issues()

{

// Act

var response = await GetResponseAsStringAsync("/Issues");

//Assert

var htmlDocument = new HtmlDocument();

htmlDocument.LoadHtml(response);

var tableElement = htmlDocument.GetElementbyId("IssueTable");

tableElement.ShouldNotBeNull();

var trNodes = tableElement.SelectNodes("//tbody/tr");

trNodes.Count.ShouldBeGreaterThan(0);

}

}

}

````

`GetResponseAsStringAsync`是一个快捷方法, 它来自执行HTTP GET请求的基类, 检查生成的HTTP状态是否为`200`, 并将响应作为`string`返回.

> 你可以使用`Client`对象(类型为`HttpClient`)对服务器执行任何类型的请求, 并读取响应.`GetResponseAsStringAsync`只是一种快捷方法.

本例使用[HtmlAgilityPack](https://html-agility-pack.net/)库来解析传入的HTML并测试它是否包含issue表格.

> 本例假设的数据库中存在一些初始issue. 请参阅[测试文档](../../Testing.md)的*\*种子数据\**部分, 了解如何设置种子数据, 以便可以假定数据库中有一些可用的初始数据.

## 控制器测试

测试控制器也不例外. 只需使用正确的URL向服务器执行请求, 获取响应并做出断言.

### 查看结果

如果控制器返回一个视图, 你可以使用类似的代码来测试返回的HTML. 参见上面的Razor页面示例.

### 对象结果

如果控制器返回对象结果, 则可以使用`GetResponseAsObjectAsync`方法.

假设你有一个如下定义的控制器:

````csharp

using System.Collections.Generic;

using System.Threading.Tasks;

using Microsoft.AspNetCore.Mvc;

using MyProject.Issues;

using Volo.Abp.AspNetCore.Mvc;

namespace MyProject.Web.Controllers

{

[Route("api/issues")]

public class IssueController : AbpController

{

private readonly IIssueAppService \_issueAppService;

public IssueController(IIssueAppService issueAppService)

{

\_issueAppService = issueAppService;

}

[HttpGet]

public async Task<List<IssueDto>> GetAsync()

{

return await \_issueAppService.GetListAsync();

}

}

}

````

你可以编写测试代码来调用API并获得结果:

````csharp

using System.Collections.Generic;

using System.Threading.Tasks;

using MyProject.Issues;

using Shouldly;

using Xunit;

namespace MyProject.Pages

{

public class Issues\_Tests : MyProjectWebTestBase

{

[Fact]

public async Task Should\_Get\_Issues\_From\_Api()

{

var issues = await GetResponseAsObjectAsync<List<IssueDto>>("/api/issues");

issues.ShouldNotBeNull();

issues.Count.ShouldBeGreaterThan(0);

}

}

}

````

## 测试JavaScript代码

ABP框架不提供任何基础设施来测试JavaScript代码. 你可以使用任何测试框架和工具来测试JavaScript代码.

## 测试基础设施

[Volo.Abp.AspNetCore.TestBase](https://www.nuget.org/packages/Volo.Abp.AspNetCore.TestBase) 提供了集成到ABP框架和ASP.NET Core的测试基础设施.

> Volo.Abp.AspNetCore.TestBase 已经安装在 `.Web.Tests` 项目中.

此包提供的`AbpAspNetCoreIntegratedTestBase`作为派生测试类的基类. 上面使用的`MyProjectWebTestBase`继承自`AbpAspNetCoreIntegratedTestBase`, 因此我们间接继承了`AbpAspNetCoreIntegratedTestBase`.

### 基本属性

`AbpAspNetCoreIntegratedTestBase` 提供了测试中使用的以下基本属性:

\* `Server`: 在测试中托管web应用程序的`TestServer`实例.

\* `Client`: 为执行对测试服务器的请求配置`HttpClient`实例.

\* `ServiceProvider`: 可以在你需要时处理服务提供服务.

### 基本方法

`AbpAspNetCoreIntegratedTestBase` 提供了以下方法, 如果需要自定义测试服务器, 可以重写这些方法:

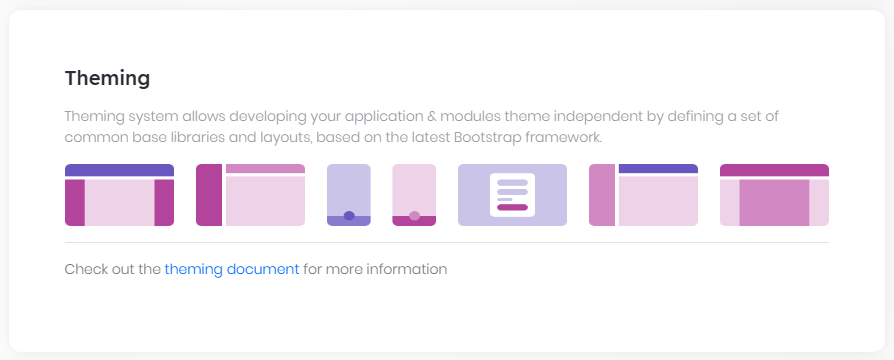
\* `ConfigureServices` 仅为派生测试类注册/替换服务时可以重写使用.

\* `CreateHostBuilder` 可用于自定义生成 `IHostBuilder`.

另请参阅

\* [总览/服务器端测试](../../Testing.md)

主题化



# ASP.NET Core MVC / Razor Pages: UI Theming

## Introduction

ABP Framework provides a complete **\*\*UI Theming\*\*** system with the following goals:

\* Reusable [application modules](../../Modules/Index.md) are developed **\*\*theme-independent\*\***, so they can work with any UI theme.

\* UI theme is **\*\*decided by the final application\*\***.

\* The theme is distributed via NuGet/NPM packages, so it is **\*\*easily upgradable\*\***.

\* The final application can **\*\*customize\*\*** the selected theme.

In order to accomplish these goals, ABP Framework;

\* Determines a set of **\*\*base libraries\*\*** used and adapted by all the themes. So, module and application developers can depend on and use these libraries without depending on a particular theme.

\* Provides a system that consists of [navigation menus](Navigation-Menu.md), [toolbars](Toolbars.md), [layout hooks](Layout-Hooks.md)... that is implemented by all the themes. So, the modules and the application to contribute to the layout to compose a consistent application UI.

### Current Themes

Currently, four themes are **\*\*officially provided\*\***:

\* The [Basic Theme](Basic-Theme.md) is the minimalist theme with the plain Bootstrap style. It is **\*\*open source and free\*\***.

\* The [LeptonX Lite Theme](../../Themes/LeptonXLite/AspNetCore.md) is modern and stylish Bootstrap UI theme. It is ideal if you want to have a production ready UI theme. It is also **\*\*open source and free\*\***.

\* The [Lepton Theme](https://commercial.abp.io/themes) is a **\*\*commercial\*\*** theme developed by the core ABP team and is a part of the [ABP Commercial](https://commercial.abp.io/) license.

\* The [LeptonX Theme](https://docs.abp.io/en/commercial/latest/themes/lepton-x/index) is also a **\*\*commercial\*\*** theme developed by the core ABP theme and is a part of the [ABP Commercial](https://commercial.abp.io/) license. This is the default theme after ABP v6.0.0.

There are also some community-driven themes for the ABP Framework (you can search on the web).

## Overall

### The Base Libraries

All the themes must depend on the [@abp/aspnetcore.mvc.ui.theme.shared](https://www.npmjs.com/package/@abp/aspnetcore.mvc.ui.theme.shared) NPM package, so they are indirectly depending on the following libraries:

\* [Twitter Bootstrap](https://getbootstrap.com/) as the fundamental HTML/CSS framework.

\* [JQuery](https://jquery.com/) for DOM manipulation.

\* [DataTables.Net](https://datatables.net/) for data grids.

\* [JQuery Validation](https://github.com/jquery-validation/jquery-validation) for client side & [unobtrusive](https://github.com/aspnet/jquery-validation-unobtrusive) validation

\* [FontAwesome](https://fontawesome.com/) as the fundamental CSS font library.

\* [SweetAlert](https://sweetalert.js.org/) to show fancy alert message and confirmation dialogs.

\* [Toastr](https://github.com/CodeSeven/toastr) to show toast notifications.

\* [Lodash](https://lodash.com/) as a utility library.

\* [Luxon](https://moment.github.io/luxon/) for date/time operations.

\* [JQuery Form](https://github.com/jquery-form/form) for AJAX forms.

\* [bootstrap-datepicker](https://github.com/uxsolutions/bootstrap-datepicker) to show date pickers.

\* [Select2](https://select2.org/) for better select/combo boxes.

\* [Timeago](http://timeago.yarp.com/) to show automatically updating fuzzy timestamps.

\* [malihu-custom-scrollbar-plugin](https://github.com/malihu/malihu-custom-scrollbar-plugin) for custom scrollbars.

These libraries are selected as the base libraries and available to the applications and modules.

#### Abstractions / Wrappers

There are some abstractions in the ABP Framework to make your code independent from some of these libraries too. Examples;

\* [Tag Helpers](Tag-Helpers/Index.md) makes it easy to generate the Bootstrap UIs.

\* JavaScript [Message](JavaScript-API/Message.md) and [Notification](JavaScript-API/Notify.md) APIs provides abstractions to use the Sweetalert and Toastr.

\* [Forms & Validation](Forms-Validation.md) system automatically handles the validation, so you mostly don't directly type any validation code.

### The Standard Layouts

The main responsibility of a theme is to provide the layouts. There are **\*\*three pre-defined layouts must be implemented by all the themes\*\***:

\* **\*\*Application\*\***: The default layout which is used by the main application pages.

\* **\*\*Account\*\***: Mostly used by the [account module](../../Modules/Account.md) for login, register, forgot password... pages.

\* **\*\*Empty\*\***: The Minimal layout that has no layout components at all.

Layout names are constants defined in the `Volo.Abp.AspNetCore.Mvc.UI.Theming.StandardLayouts` class.

#### The Application Layout

This is the default layout which is used by the main application pages. The following image shows the user management page in the [Basic Theme](Basic-Theme.md) application layout:

![basic-theme-application-layout](../../images/basic-theme-application-layout.png)

And the same page is shown below with the [Lepton Theme](https://commercial.abp.io/themes) application layout:

![lepton-theme-application-layout](../../images/lepton-theme-application-layout.png)

As you can see, the page is the same, but the look is completely different in the themes above.

The application layout typically includes the following parts;

\* A [main menu](Navigation-Menu.md)

\* Main [Toolbar](Toolbars.md) with the following components;

\* User menu

\* Language switch dropdown

\* [Page alerts](Page-Alerts.md)

\* The page content (aka `RenderBody()`)

\* [Layout hooks](Layout-Hooks.md)

Some themes may provide more parts like breadcrumbs, page header & toolbar... etc. See the *\*Layout Parts\** section.

#### The Account Layout

The Account layout is typically used by the [account module](../../Modules/Account.md) for login, register, forgot password... pages.

![basic-theme-account-layout](../../images/basic-theme-account-layout.png)

This layout typically provides the following parts;

\* Language switch dropdown

\* Tenant switch area (if the application is [multi-tenant](../../Multi-Tenancy.md) and the current is resolved by the cookie)

\* [Page alerts](Page-Alerts.md)

\* The page content (aka `RenderBody()`)

\* [Layout hooks](Layout-Hooks.md)

The [Basic Theme](Basic-Theme.md) renders the top navigation bar for this layout too (as shown above)

Here, the account layout of the Lepton Theme:

![lepton-theme-account-layout](../../images/lepton-theme-account-layout.png)

The [Lepton Theme](https://commercial.abp.io/themes) shows the application logo and footer in this layout.

> You can override theme layouts completely or partially in an application to [customize](Customization-User-Interface.md) it.

#### The Empty Layout

The empty layout provides an empty page. It typically includes the following parts;

\* [Page alerts](Page-Alerts.md)

\* The page content (aka `RenderBody()`)

\* [Layout hooks](Layout-Hooks.md)

## Implementing a Theme

### The Easiest Way

The easiest way of creating a new theme is adding [Basic Theme Source Code](https://github.com/abpframework/abp/tree/dev/modules/basic-theme) module with source codes and customizing it.

```bash

abp add-package Volo.Abp.AspNetCore.Mvc.UI.Theme.Basic --with-source-code --add-to-solution-file

```

### The ITheme Interface

`ITheme` interface is used by the ABP Framework to select the layout for the current page. A theme must implement this interface to provide the requested layout path.

This is the `ITheme` implementation of the [Basic Theme](Basic-Theme.md).

````csharp

using Volo.Abp.AspNetCore.Mvc.UI.Theming;

using Volo.Abp.DependencyInjection;

namespace Volo.Abp.AspNetCore.Mvc.UI.Theme.Basic

{

[ThemeName(Name)]

public class BasicTheme : ITheme, ITransientDependency

{

public const string Name = "Basic";

public virtual string GetLayout(string name, bool fallbackToDefault = true)

{

switch (name)

{

case StandardLayouts.Application:

return "~/Themes/Basic/Layouts/Application.cshtml";

case StandardLayouts.Account:

return "~/Themes/Basic/Layouts/Account.cshtml";

case StandardLayouts.Empty:

return "~/Themes/Basic/Layouts/Empty.cshtml";

default:

return fallbackToDefault

? "~/Themes/Basic/Layouts/Application.cshtml"

: null;

}

}

}

}

````

\* `[ThemeName]` attribute is required and a theme must have a unique name, `Basic` in this sample.

\* `GetLayout` method should return a path if the requested layout (`name`) is provided by the theme. *\*The Standard Layouts\** should be implemented if the theme is aimed to be used by a standard application. It may implement additional layouts.

Once the theme implements the `ITheme` interface, it should add the theme to the `AbpThemingOptions` in the `ConfigureServices` method of the [module](../../Module-Development-Basics.md).

````csharp

Configure<AbpThemingOptions>(options =>

{

options.Themes.Add<BasicTheme>();

});

````

#### The IThemeSelector Service

ABP Framework allows to use multiple themes together. This is why `options.Themes` is a list. `IThemeSelector` service selects the theme on the runtime. The application developer can set the `AbpThemingOptions.DefaultThemeName` to set the theme to be used, or replace the `IThemeSelector` service implementation (the default implementation is `DefaultThemeSelector`) to completely control the theme selection on runtime.

### Bundles

[Bundling system](Bundling-Minification.md) provides a standard way to import style & script files into pages. There are two standard bundles defined by the ABP Framework:

\* `StandardBundles.Styles.Global`: The global bundle that includes the style files used in all the pages. Typically, it includes the CSS files of the Base Libraries.

\* `StandardBundles.Scripts.Global`: The global bundle that includes the script files used in all the pages. Typically, it includes the JavaScript files of the Base Libraries.

A theme generally extends these standard bundles by adding theme specific CSS/JavaScript files.

The best way to define new bundles, inherit from the standard bundles and add to the `AbpBundlingOptions` as shown below (this code is from the [Basic Theme](Basic-Theme.md)):

````csharp

Configure<AbpBundlingOptions>(options =>

{

options

.StyleBundles

.Add(BasicThemeBundles.Styles.Global, bundle =>

{

bundle

.AddBaseBundles(StandardBundles.Styles.Global)

.AddContributors(typeof(BasicThemeGlobalStyleContributor));

});

options

.ScriptBundles

.Add(BasicThemeBundles.Scripts.Global, bundle =>

{

bundle

.AddBaseBundles(StandardBundles.Scripts.Global)

.AddContributors(typeof(BasicThemeGlobalScriptContributor));

});

});

````

`BasicThemeGlobalStyleContributor` and `BasicThemeGlobalScriptContributor` are bundle contributors. For example, `BasicThemeGlobalStyleContributor` is defined as shown below:

```csharp

public class BasicThemeGlobalStyleContributor : BundleContributor

{

public override void ConfigureBundle(BundleConfigurationContext context)

{

context.Files.Add("/themes/basic/layout.css");

}

}

```

Then the theme can render these bundles in a layout. For example, you can render the Global Styles as shown below:

````html

<abp-style-bundle name="@BasicThemeBundles.Styles.Global" />

````

See the [Bundle & Minification](Bundling-Minification.md) document to understand the Bundling system better.

### Layout Parts

A typical Layout consists of several parts. The theme should include the necessary parts in each layout.

**\*\*Example: The Basic Theme has the following parts for the Application Layout\*\***

![basic-theme-application-layout-parts](../../images/basic-theme-application-layout-parts.png)

The application code and the modules can only show contents in the Page Content part. If they need to change the other parts (to add a menu item, to add a toolbar item, to change the application name in the branding area...) they should use the ABP Framework APIs.

The following sections explain the fundamental parts pre-defined by the ABP Framework and can be implemented by the themes.

> It is a good practice to split the layout into components/partials, so the final application can override them partially for customization purpose.

#### Branding

`IBrandingProvider` service should be used to get the name and the logo URL of the application to render in the Branding part.

The [Application Startup Template](../../Startup-Templates/Application.md) has an implementation of this interface to set the values by the application developer.

#### Main Menu

`IMenuManager` service is used to get the main menu items and render on the layout.

**\*\*Example: Get the Main Menu to render in a view component\*\***

```csharp

public class MainNavbarMenuViewComponent : AbpViewComponent

{

private readonly IMenuManager \_menuManager;

public MainNavbarMenuViewComponent(IMenuManager menuManager)

{

\_menuManager = menuManager;

}

public async Task<IViewComponentResult> InvokeAsync()

{

var menu = await \_menuManager.GetAsync(StandardMenus.Main);

return View("~/Themes/Basic/Components/Menu/Default.cshtml", menu);

}

}

```

See the [Navigation / Menus](Navigation-Menu.md) document to learn more about the navigation system.

#### Main Toolbar

`IToolbarManager` service is used to get the Main Toolbar items and render on the layout. Each item of this toolbar is a View Component, so it may include any type of UI elements. Inject the `IToolbarManager` and use the `GetAsync` to get the toolbar items:

````csharp

var toolbar = await \_toolbarManager.GetAsync(StandardToolbars.Main);

````

> See the [Toolbars](Toolbars.md) document to learn more on the toolbar system.

The theme has a responsibility to add two pre-defined items to the main toolbar: Language Selection and User Menu. To do that, create a class implementing the `IToolbarContributor` interface and add it to the `AbpToolbarOptions` as shown below:

```csharp

Configure<AbpToolbarOptions>(options =>

{

options.Contributors.Add(new BasicThemeMainTopToolbarContributor());

});

```

##### Language Selection

Language Selection toolbar item is generally a dropdown that is used to switch between languages. `ILanguageProvider` is used to get the list of available languages and `CultureInfo.CurrentUICulture` is used to learn the current language.

`/Abp/Languages/Switch` endpoint can be used to switch the language This endpoint accepts the following query string parameters:

\* `culture`: The selected culture, like `en-US` or `en`.

\* `uiCulture`: The selected UI culture, like `en-US` or `en`.

\* `returnUrl` (optional): Can be used to return a given URL after switching the language.

`culture` and `uiCulture` should match one of the available languages. ABP Framework sets a culture cookie in the `/Abp/Languages/Switch` endpoint.

##### User Menu

User menu includes links related to the user account. `IMenuManager` is used just like the Main Menu, but this time with `StandardMenus.User` parameter like shown below:

````csharp

var menu = await \_menuManager.GetAsync(StandardMenus.User);

````

[ICurrentUser](../../CurrentUser.md) and [ICurrentTenant](../../Multi-Tenancy.md) services can be used to obtain the current user and tenant names.

#### Page Alerts

`IAlertManager` service is used to get the current page alerts to render on the layout. Use the `Alerts` list of the `IAlertManager`. It is generally rendered just before the page content (`RenderBody()`).

See the [Page Alerts](Page-Alerts.md) document to learn more.

#### Layout Hooks

Since the Layout is in the theme package, the final application or any module can't directly manipulate the layout content. The [Layout Hook](Layout-Hooks.md) system allows to inject components to some specific points of the layout.

The theme is responsible to render the hooks in the correct place.

**\*\*Example: Render the** `LayoutHooks.Head.First` **Hook in the Application Layout\*\***

````html

<head>

@await Component.InvokeLayoutHookAsync(LayoutHooks.Head.First, StandardLayouts.Application)

...

````

See the [Layout Hook](Layout-Hooks.md) document to learn the standard layout hooks.

#### Script / Style Sections

Every layout should render the following optional sections:

\* `styles` section is rendered in the end of the `head`, just before the `LayoutHooks.Head.Last`.

\* `scripts` section is rendered in the end of the `body`, just before the `LayoutHooks.Body.Last`.

In this way, the page can import styles and scripts to the layout.

**\*\*Example: Render the** `styles` **section\*\***

````csharp

@await RenderSectionAsync("styles", required: false)

````

#### Content Toolbar Section

Another pre-defined section is the Content Toolbar section which can be used by the pages to add code just before the page content. The Basic Theme renders it as shown below:

````html

<div id="AbpContentToolbar">

<div class="text-end mb-2">

@RenderSection("content\_toolbar", false)

</div>

</div>

````

The container div's id must be `AbpContentToolbar`. This section should come before the `RenderBody()`.

#### Widget Resources

The [Widget System](Widgets.md) allows to define reusable widgets with their own style/script files. All the layouts should render the widget style and scripts.

**\*\*Widget Styles\*\*** is rendered as shown below, just before the `styles` section, after the global style bundle:

````csharp

@await Component.InvokeAsync(typeof(WidgetStylesViewComponent))

````

**\*\*Widget Scripts\*\*** is rendered as shown below, just before the `scripts` section, after the global script bundle:

````csharp

@await Component.InvokeAsync(typeof(WidgetScriptsViewComponent))

````

#### ABP Scripts

ABP has some special scripts those should be included into every layout. They are not included in the global bundles since they are dynamically created based on the current user.

ABP scripts (`ApplicationConfigurationScript` and `ServiceProxyScript`) should be added just after the global script bundle, as shown below:

````html

<script src="~/Abp/ApplicationConfigurationScript"></script>

<script src="~/Abp/ServiceProxyScript"></script>

````

#### Page Title, Selected Menu Item and Breadcrumbs

`IPageLayout` service can be injected by any page to set the Page Title, the selected menu item name and the breadcrumb items. Then the theme can use this service to get these values and render on the UI.

The Basic Theme doesn't implement this service, but the Lepton Theme implements:

![breadcrumbs-example](../../images/breadcrumbs-example.png)

See the [Page Header](Page-Header.md) document for more.

#### Tenant Switch

The Account Layout should allow the user to switch the current tenant if the application is multi-tenant and the tenant was resolved from the cookies. See the [Basic Theme Account Layout](https://github.com/abpframework/abp/blob/dev/modules/basic-theme/src/Volo.Abp.AspNetCore.Mvc.UI.Theme.Basic/Themes/Basic/Layouts/Account.cshtml) as an example implementation.

### Layout Classes

The Standard Layouts (`Application`, `Account` and `Empty`) should add the following CSS classes to the `body` tag:

\* `abp-application-layout` for the `Application` layout.

\* `abp-account-layout` for the `Account` layout.

\* `abp-empty-layout` for the `Empty` layout.

In this way, applications or modules can have selectors based on the current layout.

### RTL

To support Right-To-Left languages, the Layout should check the current culture and add `dir="rtl"` to the `html` tag and `rtl` CSS class the the `body` tag.

You can check `CultureInfo.CurrentUICulture.TextInfo.IsRightToLeft` to understand if the current language is a RTL language.

### The NPM Package

A theme should have a NPM package that depends on the [@abp/aspnetcore.mvc.ui.theme.shared](https://www.npmjs.com/package/@abp/aspnetcore.mvc.ui.theme.shared) package. In this way, it inherits all the Base Libraries. If the theme requires additional libraries, then it should define these dependencies too.

Applications use the [Client Side Package Management](Client-Side-Package-Management.md) system to add client side libraries to the project. So, if an application uses your theme, it should add dependency to your theme's NPM package as well as the NuGet package dependency.

## Angular

v2.x 到 v3 迁移指南

# Angular UI v2.9 迁移到 v3.0 指南

## 在v3.0改变了什么?

### Angular 10

新的ABP Angular UI基于Angular 10和TypeScript 3.9,我们已经放弃了对Angular 8的支持. 不过ABP模块将继续与Angular 9兼容使用. 因此如果你的项目是Angular 9,则无需更新为 Angular10. 更新通常很容易.

#### 如何迁移?

在你的根文件夹中打开一个终端,然后运行以下命令:

```sh

yarn ng update @angular/cli @angular/core --force

```

这会做如下修改:

- 更新你的package.json并安装新的软件包

- 修改tsconfig.json文件创建一个"Solution Style"配置

- 重命名 `browserslist` 为 `.browserslistrc`

另一方面,如果你单独使用 `yarn ng update` 命令检查首先要更新哪些包会更好. Angular会给你一个要更新的包列表.

![Table of packages to update](./images/table-of-packages-to-update.png)

当Angular报告上面的包后,运行命令:

```sh

yarn ng update @angular/cli @angular/core ng-zorro-antd --force

```

> 如果Angular提示你的仓库有中未提交的更改,可以提交/存储它,也可以在命令中添加 `--allow-dirty` 参数.

### 配置模块

在ABP v2.x中,每个延迟加载的模块都有一个可通过单独的程序包使用的配置模块,模块配置如下:

```js

import { AccountConfigModule } from '@abp/ng.account.config';

@NgModule({

imports: [

// other imports

AccountConfigModule.forRoot({ redirectUrl: '/' }),

],

// providers, declarations, and bootstrap

})

export class AppModule {}

```

...在app-routing.module.ts...

```js

const routes: Routes = [

// other route configuration

{

path: 'account',

loadChildren: () => import(

'./lazy-libs/account-wrapper.module'

).then(m => m.AccountWrapperModule),

},

];

```

虽然有效,但有一些缺点:

- 每个模块都有两个独立的程序包,但实际上这些程序包是相互依赖的.

- 配置延迟加载的模块需要包装模块.

- ABP Commercial具有可扩展性系统,在根模块上配置可扩展模块会增加 bundle 的大小.

在ABP v3.0中,我们为每个配置模块引入了辅助入口点,并且提供了一种在没有包装的情况下配置延迟加载的模块的新方法. 现在模块配置如下所示:

```js

import { AccountConfigModule } from '@abp/ng.account/config';

@NgModule({

imports: [

// other imports

AccountConfigModule.forRoot(),

],

// providers, declarations, and bootstrap

})

export class AppModule {}

```

... 在app-routing.module.ts...

```js

const routes: Routes = [

// other route configuration

{

path: 'account',

loadChildren: () => import('@abp/ng.account')

.then(m => m.AccountModule.forLazy({ redirectUrl: '/' })),

},

];

```

这项更改帮助我们减少了捆绑包的大小并大大缩短了构建时间. 我们相信你会注意到你的应用程序有所不同.

#### 一个更好的例子

AppModule:

```js

import { AccountConfigModule } from '@abp/ng.account/config';

import { CoreModule } from '@abp/ng.core';

import { IdentityConfigModule } from '@abp/ng.identity/config';

import { SettingManagementConfigModule } from '@abp/ng.setting-management/config';

import { TenantManagementConfigModule } from '@abp/ng.tenant-management/config';

import { ThemeBasicModule } from '@abp/ng.theme.basic';

import { ThemeSharedModule } from '@abp/ng.theme.shared';

import { NgModule } from '@angular/core';

import { BrowserModule } from '@angular/platform-browser';

import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { NgxsModule } from '@ngxs/store';

import { environment } from '../environments/environment';

import { AppRoutingModule } from './app-routing.module';

@NgModule({

imports: [

BrowserModule,

BrowserAnimationsModule,

AppRoutingModule,

CoreModule.forRoot({

environment,

sendNullsAsQueryParam: false,

skipGetAppConfiguration: false,

}),

ThemeSharedModule.forRoot(),

AccountConfigModule.forRoot(),

IdentityConfigModule.forRoot(),

TenantManagementConfigModule.forRoot(),

SettingManagementConfigModule.forRoot(),

ThemeBasicModule.forRoot(),

NgxsModule.forRoot(),

],

// providers, declarations, and bootstrap

})

export class AppModule {}

```

AppRoutingModule:

```js

import { NgModule } from '@angular/core';

import { RouterModule, Routes } from '@angular/router';

const routes: Routes = [

{

path: '',

pathMatch: 'full',

loadChildren: () => import('./home/home.module').then(m => m.HomeModule),

},

{

path: 'account',

loadChildren: () =>

import('@abp/ng.account').then(m => m.AccountModule.forLazy({ redirectUrl: '/' })),

},

{

path: 'identity',

loadChildren: () => import('@abp/ng.identity').then(m => m.IdentityModule.forLazy()),

},

{

path: 'tenant-management',

loadChildren: () =>

import('@abp/ng.tenant-management').then(m => m.TenantManagementModule.forLazy()),

},

{

path: 'setting-management',

loadChildren: () =>

import('@abp/ng.setting-management').then(m => m.SettingManagementModule.forLazy()),

},

];

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule],

})

export class AppRoutingModule {}

```

> 你可能已经注意到我们在AppComponent模板中使用了 `<abp-dynamic-layout>` 而不是 `<router-outlet>`. 我们这样做是为了避免不必要的渲染和闪烁. 这不是强制的,但是我们建议在你的应用程序路由中做同样的事情.

#### 如何迁移?

- 使用 `yarn remove` 删除你的项目的配置包.

- 从辅助入口点(例如`@abp/ng.identity/config`)导入配置模块.

- 调用所有新配置模块的静态 `forRoot`方法,即使配置没有被传递.

- 调用 `ThemeBasicModule` 的静态 `forRoot` 方法(或商业上的 `ThemeLeptonModule`),并从导入中删除 `SharedModule`(除非已在其中添加了根模块所需的任何内容).

- 在app路由模块中直接导入延迟ABP模块 (如 `() => import('@abp/ng.identity').then(...)`).

- 在所有延迟模块 `then` 中调用的静态 `forLazy` 方法,即使配置没有被传递.

- [可选]添加 `<abp-dynamic-layout></abp-dynamic-layout>` 到AppComponent模板并且删除 `<router-outlet></router-outlet>`,获得更好的性能和UX.

### RoutesService

在ABP v2.x中,通过以下两种方式之一将路由添加到菜单:

- [通过 `AppRoutingModule` 的 `routes` 属性](https://docs.abp.io/en/abp/2.9.0/UI/Angular/Modifying-the-Menu#via-routes-property-in-approutingmodule)

- [通过 ConfigState](https://docs.abp.io/en/abp/2.9.0/UI/Angular/Modifying-the-Menu#via-configstate)

从v3.0开始，我们更改了添加和修改路由的方式. 我们不再将路由存储在 `ConfigState`中(破坏性更改). 而是有一个名为 `RoutesService` 的新服务,该服务用于添加,修补或删除菜单项. 详情请查看[文档](Modifying-the-Menu.md).

#### 如何迁移?

- 检查你是否曾经使用 `ConfigState` 或 `ConfigStateService` 添加路由. 请用 `RoutesService` 的 `add` 方法替换它们.

- 检查你是否曾经修补的路由. 将其替换为 `RoutesService` 的 `patch` 方法.

- 仔细检查你是否使用绝对路径,并在 `add` 或 `patch` 方法调用中为子菜单项提供 `parentName` 而不是 `children` 属性.

### NavItemsService

在ABP v2.x中,[通过LayoutStateService](https://docs.abp.io/en/abp/2.9.0/UI/Angular/Modifying-the-Menu#how-to-add-an-element-to-right-part-of-the-menu)添加导航元素.

从v3.0开始,我们改变了添加和修改导航项的方式,以前的方法不再可用(破坏性更改). 详情请查看[文档](Modifying-the-Menu.md).

#### 如何迁移?

- 用 `NavItemsService` 的 `addItems` 方法替换所有 `dispatchAddNavigationElement` 调用.

### ngx-datatable

在v3之前,我们一直使用自定义组件 `abp-table` 作为默认表. 但是数据表是复杂的组件,要实现功能齐全的数据表需要大量的精力,我们计划将其引入其他功能.

从ABP v3开始,我们已切换到经过严格测试,执行良好的数据表格:[ngx-datatable](https://github.com/swimlane/ngx-datatable). 所有的ABP模块都已经实现了ngx-datatable. `ThemeSharedModule` 已经导出了 `NgxDatatableModule`. 因此如果你在终端运行 `yarn add @swimlane/ngx-datatable` 来安装这个包,它将在你的应用的所有模块中可用.

为了正确设置样式,你需要在angular.json文件的样式部分中添加以下内容(在其他所有元素之上):

```json

"styles": [

{

"input": "node\_modules/@swimlane/ngx-datatable/index.css",

"inject": true,

"bundleName": "ngx-datatable-index"

},

{

"input": "node\_modules/@swimlane/ngx-datatable/assets/icons.css",

"inject": true,

"bundleName": "ngx-datatable-icons"

},

{

"input": "node\_modules/@swimlane/ngx-datatable/themes/material.css",

"inject": true,

"bundleName": "ngx-datatable-material"

},

// other styles

]

```

由于尚未删除 `abp-table`, 因此以前由ABP v2.x构建的模块不会突然丢失所有. 但是它们的外观与内置ABP v3模块有所不同, 因此你可能希望将这些模块中的表转换为ngx-datatable. 为了减少将abp-table转换为ngx-datatable所需的工作量,我们修改了 `ListService` 以使其与 `ngx-datatable` 一起很好地工作,并引入了两个新指令: `NgxDatatableListDirective` 和 `NgxDatatableDefaultDirective`.

这些指令的用法很简单:

```js

@Component({

providers: [ListService],

})

export class SomeComponent {

data$ = this.list.hookToQuery(

query => this.dataService.get(query)

);

constructor(

public readonly list: ListService,

public readonly dataService: SomeDataService,

) {}

}

```

...在组件模板...

```html

<ngx-datatable

[rows]="(data$ | async)?.items || []"

[count]="(data$ | async)?.totalCount || 0"

[list]="list"

default

>

<!-- column templates here -->

</ngx-datatable>

```

通过 `NgxDatatableListDirective` 绑定注入的 `ListService` 实例后,你不再需要担心分页或排序. 同样 `NgxDatatableDefaultDirective` 去除了几个属性绑定,以使ngx-datatable适合我们的样式.

#### 一个更好的例子

```html

<ngx-datatable

[rows]="items"

[count]="count"

[list]="list"

default

>

<!-- the grid actions column -->

<ngx-datatable-column

name=""

[maxWidth]="150"

[width]="150"

[sortable]="false"

>

<ng-template

ngx-datatable-cell-template

let-row="row"

let-i="rowIndex"

>

<abp-grid-actions

[index]="i"

[record]="row"

text="AbpUi::Actions"

></abp-grid-actions>

</ng-template>

</ngx-datatable-column>

<!-- a basic column -->

<ngx-datatable-column

prop="someProp"

[name]="'::SomeProp' | abpLocalization"

[width]="200"

></ngx-datatable-column>

<!-- a column with a custom template -->

<ngx-datatable-column

prop="someOtherProp"

[name]="'::SomeOtherProp' | abpLocalization"

[width]="250"

>

<ng-template

ngx-datatable-cell-template

let-row="row"

let-i="index"

>

<div abpEllipsis>{%{{{ row.someOtherProp }}}%}</div>

</ng-template>

</ngx-datatable-column>

</ngx-datatable>

```

#### 如何迁移?

- 安装 `@swimlane/ngx-datatable` 包.

- 添加ngx-datatable样式到angular.json文件.

- 如果可以的话,根据上面的例子更新你的模.

- 如果你稍后需要这样做,并且打算保留abp-table一段时间,请确保根据此处描述的[破坏性更改](List-Service.md)更新分页.

\*\*重要说明:\*\*abp-table没有被删除,但已被弃用并在以后的版本中删除. 请考虑切换到ngx-datatable.

### 扩展系统[商业版]

扩展程序系统现在是开源的, 可以从 `@abp/ng.theme.shared/extensions` 而不是从 `@volo/abp.commercial.ng.ui` 中获取. 同样,根据config软件包的新结构,如上所述通过 `forLazy` 静态方法进行配置.

#### 如何迁移?

如果你以前从未使用过扩展系统,则无需执行任何操作. 否则请再次检查文档以查看更改. 扩展系统本身的工作原理与以前相同,唯一的变化是你从中导入的包,静态方法以及您将贡献者传递给的模块.

### Lepton 主题Logo [商业版]

在ABP v2.x中,Lepton每个颜色主题都有一个亮徽标和一个暗徽标. 我们意识到我们可以使它仅使用一个浅色和一个深色徽标. 因此我们更改了Lepton查找徽标图像的方式,现在你只需要在项目中包含 `logo-light.png` 和 `logo-dark.png`.

#### 如何迁移?

如果你之前已切换模板徽标PNG,则更改很简单:

- 转到 `/assets/images/logo` 目录.

- 重命名 `theme1.png` 为 `logo-light.png` 并且重命名 `theme1-reverse.png` 为 `logo-dark.png`.

- 删除所有其他 `theme\*.png` 文件.

如果你更换了徽标组件,则更改有些不同,但仍然很简单. `LayoutStateService` 有两个新成员: `primaryLogoColor` 和 `secondaryLogoColor`. 它们有 `'light'` 和 `'dark'` 设置值做为可观察流. 你可以使用 `async` 管道在自定义徽标组件模板中使用它们的值. 这是一个完整的示例,其中涵盖了主要和辅助(帐户)布局徽标.

```js

import { AddReplaceableComponent } from '@abp/ng.core';

import { CommonModule } from '@angular/common';

import { APP\_INITIALIZER, Component, Injector, NgModule } from '@angular/core';

import { Store } from '@ngxs/store';

import { eAccountComponents } from '@volo/abp.ng.account';

import {

AccountLayoutComponent,

eThemeLeptonComponents,

LayoutStateService,

} from '@volo/abp.ng.theme.lepton';

import { Observable } from 'rxjs';

import { map } from 'rxjs/operators';

@Component({

template: `

<div class="account-brand p-4 text-center mb-1" \*ngIf="isAccount; else link">

<ng-template [ngTemplateOutlet]="link"></ng-template>

</div>

<ng-template #link>

<a [style.background-image]="logoUrl | async" class="navbar-brand" routerLink="/"></a>

</ng-template>

`,

})

export class LogoComponent {

isAccount: boolean;

logoColor: Observable<'dark' | 'light'>;

get logoUrl() {

return this.logoColor.pipe(map(color => `url(/assets/images/logo/logo-${color}.png)`));

}

constructor(injector: Injector) {

const layout = injector.get(LayoutStateService);

this.isAccount = Boolean(injector.get(AccountLayoutComponent, false));

this.logoColor = this.isAccount ? layout.secondaryLogoColor : layout.primaryLogoColor;

}

}

@NgModule({

imports: [CommonModule],

declarations: [LogoComponent],

exports: [LogoComponent],

})

export class LogoModule {}

export const APP\_LOGO\_PROVIDER = [

{ provide: APP\_INITIALIZER, useFactory: switchLogos, multi: true, deps: [Store] },

];

export function switchLogos(store: Store) {

return () => {

store.dispatch(

new AddReplaceableComponent({

component: LogoComponent,

key: eThemeLeptonComponents.Logo,

}),

);

store.dispatch(

new AddReplaceableComponent({

component: LogoComponent,

key: eAccountComponents.Logo,

}),

);

};

}

```

只要将 `APP\_LOGO\_PROVIDER` 添加到根模块的提供程序(通常是 `AppModule` ),你就会有一个调整主题颜色的自定义徽标组件.

### 过时的接口

某些接口早已被标记为已弃用,现在已将其删除.

#### 如何迁移?

请检查你是否仍在使用[Issue中列出的任何内容](https://github.com/abpframework/abp/issues/4281).

## 下一步是什么?

\* [环境](./Environment.md)

环境

# Environment

Every application needs some **\*\*environment\*\*** variables. In Angular world, this is usually managed by `environment.ts`, `environment.prod.ts` and so on. It is the same for ABP as well.

Current `Environment` configuration holds sub config classes as follows:

```js

export interface Environment {

apis: Apis;

application: Application;

oAuthConfig: AuthConfig;

production: boolean;

remoteEnv?: RemoteEnv;

}

```

## Apis

```js

export interface Apis {

[key: string]: ApiConfig;

default: ApiConfig;

}

export interface ApiConfig {

[key: string]: string;

rootNamespace?: string;

url: string;

}

```

Api config has to have a default config and it may have some additional ones for different modules.

I.e. you may want to connect to different Apis for different modules.

Take a look at following example

```json

{

// ...

"apis": {

"default": {

"url": "https://localhost:8080"

},

"AbpIdentity": {

"url": "https://localhost:9090"

}

}

// ...

}

```

When an api from `AbpIdentity` is called, the request will be sent to `"https://localhost:9090"`.

Everything else will be sent to `"https://localhost:8080"`

- `rootNamespace` **\*\*(new)\*\*** : Root namespace of the related API. e.g. Acme.BookStore

## Application

```js

export interface Application {

name: string;

baseUrl?: string;

logoUrl?: string;

}

```

- `name`: Name of the backend Application. It is also used by `logo.component` if `logoUrl` is not provided.

- `logoUrl`: Url of the application logo. It is used by `logo.component`

- `baseUrl`: [For detailed information](./Multi-Tenancy.md#domain-tenant-resolver)

## AuthConfig

For authentication, we use angular-oauth2-oidc. Please check their [docs](https://github.com/manfredsteyer/angular-oauth2-oidc) out

## RemoteEnvironment

Some applications need to integrate an existing config into the `environment` used throughout the application.

Abp Framework supports this out of box.

To integrate an existing config json into the `environment`, you need to set `remoteEnv`

```js

export type customMergeFn = (

localEnv: Partial<Config.Environment>,

remoteEnv: any

) => Config.Environment;

export interface RemoteEnv {

url: string;

mergeStrategy: "deepmerge" | "overwrite" | customMergeFn;

method?: string;

headers?: ABP.Dictionary<string>;

}

```

- `url` \\*: Required. The url to be used to retrieve environment config

- `mergeStrategy` \\*: Required. Defines how the local and the remote `environment` json will be merged

- `deepmerge`: Both local and remote `environment` json will be merged recursively. If both configs have same nested path, the remote `environment` will be prioritized.

- `overwrite`: Remote `environment` will be used and local environment will be ignored.

- `customMergeFn`: You can also provide your own merge function as shown in the example. It will take two parameters, `localEnv: Partial<Config.Environment>` and `remoteEnv` and it needs to return a `Config.Environment` object.

- `method`: HTTP method to be used when retrieving environment config. Default: `GET`

- `headers`: If extra headers are needed for the request, it can be set through this field.

## EnvironmentService

` EnvironmentService` is a singleton service, i.e. provided in root level of your application, and keeps the environment in the internal store.

### Before Use

In order to use the `EnvironmentService` you must inject it in your class as a dependency.

```js

import { EnvironmentService } from '@abp/ng.core';

@Component({

/\* class metadata here \*/

})

class DemoComponent {

constructor(private environment: EnvironmentService) {}

}

```

You do not have to provide the `EnvironmentService` at module or component/directive level, because it is already **\*\*provided in root\*\***.

### Get Methods

`EnvironmentService` has numerous get methods which allow you to get a specific value or all environment object.

Get methods with "$" at the end of the method name (e.g. `getEnvironment$`) return an RxJs stream. The streams are triggered when set or patched the state.

#### How to Get Environment Object

You can use the `getEnvironment` or `getEnvironment$` method of `EnvironmentService` to get all of the environment object. It is used as follows:

```js

// this.environment is instance of EnvironmentService

const environment = this.environment.getEnvironment();

// or

this.environment.getEnvironment$().subscribe((environment) => {

// use environment here

});

```

#### How to Get API URL

The `getApiUrl` or `getApiUrl$` method is used to get a specific API URL from the environment object. This is how you can use it:

```js

// this.environment is instance of EnvironmentService

const apiUrl = this.environment.getApiUrl();

// environment.apis.default.url

this.environment.getApiUrl$("search").subscribe((searchUrl) => {

// environment.apis.search.url

});

```

This method returns the `url` of a specific API based on the key given as its only parameter. If there is no key, `'default'` is used.

#### How to Set the Environment

`EnvironmentService` has a method named `setState` which allows you to set the state value.

```js

// this.environment is instance of EnvironmentService

this.environment.setState(newEnvironmentObject);

```

Note that **\*\*you do not have to call this method at application initiation\*\***, because the environment variables are already being stored at start.

#### Environment Properties

Please refer to `Environment` type for all the properties. It can be found in the [environment.ts file](https://github.com/abpframework/abp/blob/dev/npm/ng-packs/packages/core/src/lib/models/environment.ts#L4).

服务代理

## 服务代理

从Angular应用程序中调用服务器中的REST端点是很常见的, 在这种情况下我们通常创建**\*\*服务\*\***(在服务器端具有针对每个服务方法的方法)和**\*\*模型对象\*\***(与服务器端[DTO](../../Data-Transfer-Objects)匹配).

除了手动创建这样的服务器交互服务之外,我们还可以使用[NSWAG](https://github.com/RicoSuter/NSwag)之类的工具来为我们生成服务代理. 但使用NSWAG过程中我们遇到以下问题:

\* 它生成一个**\*\*大的单个.ts文件\*\***,该文件存在一些问题;

\* 当你的应用程序增长时,它会变的**\*\*越来越大\*\***.

\* 它不适合ABP框架的[模块化](../../Module-Development-Basics)方法

\* 它创建了一些**\*\*难看的代码\*\***. 我们希望有一个干净的代码(就像我们手动编写一样).

\* 它不能生成在服务器端声明的相同**\*\*方法签名\*\***(因为swagger.json并不完全反映后端服务的方法签名). 我们已经创建了一个端点公开了服务器端方法信息,以便客户端生成更好的一致的客户端代理.

ABP CLI 的`generate-proxies` 命令在 `src/app` 文件夹中创建按模块名称分隔的文件夹,自动生成typescript客户端代理.

在angular应用程序的**\*\*根文件夹\*\***中运行以下命令:

```bash

abp generate-proxy -t ng

```

它只为你自己的应用程序的服务创建代理. 不会为你正在使用的应用程序模块的服务创建代理(默认情况下). 有几个选项,参见[CLI文档](../../CLI).

使用 `--module all` 选项生成的文件如下所示:

![generated-files-via-generate-proxy](./images/generated-files-via-generate-proxy.png)

### Services

每个生成的服务都与后端控制器匹配. 服务方法通过[RestService](./Http-Requests#restservice)调用后端API.

在每个服务中都定义了一个名为 `apiName` 的变量(自v2.4起可用). `apiName` 与模块的 `RemoteServiceName` 匹配. 在每次请求时该变量将作为参数传递给 `RestService`. 如果环境中未定义微服务API, `RestService` 使用默认值. 请参阅[从应用程序配置中获取特定的API端点](./Http-Requests#how-to-get-a-specific-api-endpoint-from-application-config)

服务的 `providerIn` 属性定义为 `'root'`. 因此无需将服务作为提供程序添加到模块. 你可以通过将服务注入到构造函数中来使用它,如下所示:

```js

import { AbpApplicationConfigurationService } from '../abp/applicationconfiguration/services';

//...

export class HomeComponent{

constructor(private appConfigService: AbpApplicationConfigurationService) {}

ngOnInit() {

this.appConfigService.get().subscribe()

}

}

```

Angular编译器会从最终输出中删除那些没有被注入的服务. 参见[摇树优化的提供者文档](https://angular.cn/guide/dependency-injection-providers#tree-shakable-providers).

### Models

生成的模型与后端中的dto匹配. 每个模型在 `src/app/\*/models` 文件夹生成一个类.

`@abp/ng.core` 包有一些[基类](https://github.com/abpframework/abp/blob/dev/npm/ng-packs/packages/core/src/lib/models/dtos.ts). 一些模型扩展了这些类.

可以如下所示创建一个类的实例:

```js

import { IdentityRoleCreateDto } from '../identity/role/models'

//...

const instance = new IdentityRoleCreateDto({name: 'Role 1', isDefault: false, isPublic: true})

```

可以选择将初始值传递给每个类构造函数.

## 下一步是什么?

\* [HTTP请求](./Http-Requests)

单元测试

# Angular UI 单元测试

ABP Angular UI的测试与其他Angular应用程序一样. 所以, [这里的指南](https://angular.io/guide/testing)也适用于ABP. 也就是说, 我们想指出一些**\*\*特定于ABP Angular应用程序的单元测试内容\*\***.

## 设置

在Angular中, 单元测试默认使用[Karma](https://karma-runner.github.io/)和[Jasmine](https://jasmine.github.io). 虽然我们更喜欢Jest, 但我们选择不偏离这些默认设置, 因此**\*\*你下载的应用程序模板将预先配置Karma和Jasmine\*\***. 你可以在根目录中的 *\_karma.conf.js\_* 文件中找到Karma配置. 你什么都不用做. 添加一个spec文件并运行`npm test`即可.

## 基础

简化版的spec文件如下所示:

```js

import { CoreTestingModule } from "@abp/ng.core/testing";

import { ThemeBasicTestingModule } from "@abp/ng.theme.basic/testing";

import { ThemeSharedTestingModule } from "@abp/ng.theme.shared/testing";

import { ComponentFixture, TestBed, waitForAsync } from "@angular/core/testing";

import { NgxValidateCoreModule } from "@ngx-validate/core";

import { MyComponent } from "./my.component";

describe("MyComponent", () => {

let fixture: ComponentFixture<MyComponent>;

beforeEach(

waitForAsync(() => {

TestBed.configureTestingModule({

declarations: [MyComponent],

imports: [

CoreTestingModule.withConfig(),

ThemeSharedTestingModule.withConfig(),

ThemeBasicTestingModule.withConfig(),

NgxValidateCoreModule,

],

providers: [

/\* mock providers here \*/

],

}).compileComponents();

})

);

beforeEach(() => {

fixture = TestBed.createComponent(MyComponent);

fixture.detectChanges();

});

it("should be initiated", () => {

expect(fixture.componentInstance).toBeTruthy();

});

});

```

如果你看一下导入内容, 你会注意到我们已经准备了一些测试模块来取代内置的ABP模块. 这对于模拟某些特性是必要的, 否则这些特性会破坏你的测试. 请记住**\*\*使用测试模块\*\***并**\*\*调用其**`withConfig`**静态方法\*\***.

## 提示

### Angular测试库

虽然你可以使用Angular TestBed测试代码, 但你可以找到一个好的替代品[Angular测试库](https://testing-library.com/docs/angular-testing-library/intro).

上面的简单示例可以用Angular测试库编写, 如下所示:

```js

import { CoreTestingModule } from "@abp/ng.core/testing";

import { ThemeBasicTestingModule } from "@abp/ng.theme.basic/testing";

import { ThemeSharedTestingModule } from "@abp/ng.theme.shared/testing";

import { ComponentFixture } from "@angular/core/testing";

import { NgxValidateCoreModule } from "@ngx-validate/core";

import { render } from "@testing-library/angular";

import { MyComponent } from "./my.component";

describe("MyComponent", () => {

let fixture: ComponentFixture<MyComponent>;

beforeEach(async () => {

const result = await render(MyComponent, {

imports: [

CoreTestingModule.withConfig(),

ThemeSharedTestingModule.withConfig(),

ThemeBasicTestingModule.withConfig(),

NgxValidateCoreModule,

],

providers: [

/\* mock providers here \*/

],

});

fixture = result.fixture;

});

it("should be initiated", () => {

expect(fixture.componentInstance).toBeTruthy();

});

});

```

正如你所见, 二者非常相似. 当我们使用查询和触发事件时, 真正的区别就显现出来了.

```js

// other imports

import { getByLabelText, screen } from "@testing-library/angular";

import userEvent from "@testing-library/user-event";

describe("MyComponent", () => {

beforeEach(/\* removed for sake of brevity \*/);

it("should display advanced filters", () => {

const filters = screen.getByTestId("author-filters");

const nameInput = getByLabelText(filters, /name/i) as HTMLInputElement;

expect(nameInput.offsetWidth).toBe(0);

const advancedFiltersBtn = screen.getByRole("link", { name: /advanced/i });

userEvent.click(advancedFiltersBtn);

expect(nameInput.offsetWidth).toBeGreaterThan(0);

userEvent.type(nameInput, "fooo{backspace}");

expect(nameInput.value).toBe("foo");

});

});

```

**\*\*Angular测试库中的查询遵循可维护测试\*\***, 用户事件库提供了与DOM的**\*\*类人交互\*\***, 并且该库通常有**\*\*清晰的API\*\***简化组件测试. 下面提供一些有用的链接:

- [查询](https://testing-library.com/docs/dom-testing-library/api-queries)

- [用户事件](https://testing-library.com/docs/ecosystem-user-event)

- [范例](https://github.com/testing-library/angular-testing-library/tree/main/apps/example-app/src/app/examples)

### 在每个Spec之后清除DOM

需要记住的一点是, Karma在真实的浏览器实例中运行测试. 这意味着, 你将能够看到测试代码的结果, 但也会遇到与文档正文连接的组件的问题, 这些组件可能无法在每次测试后都清除, 即使你配置了Karma也一样无法清除.

我们准备了一个简单的函数, 可以在每次测试后清除所有剩余的DOM元素.

```js

// other imports

import { clearPage } from "@abp/ng.core/testing";

describe("MyComponent", () => {

let fixture: ComponentFixture<MyComponent>;

afterEach(() => clearPage(fixture));

beforeEach(async () => {

const result = await render(MyComponent, {

/\* removed for sake of brevity \*/

});

fixture = result.fixture;

});

// specs here

});

```

请确保你使用它, 否则Karma将无法删除对话框, 并且你将有多个模态对话框、确认框等的副本.

### 等待

一些组件, 特别是在检测周期之外工作的模态对话框. 换句话说, 你无法在打开这些组件后立即访问这些组件插入的DOM元素. 同样, 插入的元素在关闭时也不会立即销毁.

为此, 我们准备了一个`wait`函数.

```js

// other imports

import { wait } from "@abp/ng.core/testing";

describe("MyComponent", () => {

beforeEach(/\* removed for sake of brevity \*/);

it("should open a modal", async () => {

const openModalBtn = screen.getByRole("button", { name: "Open Modal" });

userEvent.click(openModalBtn);

await wait(fixture);

const modal = screen.getByRole("dialog");

expect(modal).toBeTruthy();

/\* wait again after closing the modal \*/

});

});

```

`wait`函数接受第二个参数, 即超时(默认值为`0`). 但是尽量不要使用它. 使用大于`0`的超时通常表明某些不正确事情发生了.

## 测试示例

下面是一个测试示例. 它并没有涵盖所有内容, 但却能够对测试有一个更好的了解.

```js

import { clearPage, CoreTestingModule, wait } from "@abp/ng.core/testing";

import { ThemeBasicTestingModule } from "@abp/ng.theme.basic/testing";

import { ThemeSharedTestingModule } from "@abp/ng.theme.shared/testing";

import { ComponentFixture } from "@angular/core/testing";

import {

NgbCollapseModule,

NgbDatepickerModule,

NgbDropdownModule,

} from "@ng-bootstrap/ng-bootstrap";

import { NgxValidateCoreModule } from "@ngx-validate/core";

import { CountryService } from "@proxy/countries";

import {

findByText,

getByLabelText,

getByRole,

getByText,

queryByRole,

render,

screen,

} from "@testing-library/angular";

import userEvent from "@testing-library/user-event";

import { BehaviorSubject, of } from "rxjs";

import { CountryComponent } from "./country.component";

const list$ = new BehaviorSubject({

items: [{ id: "ID\_US", name: "United States of America" }],

totalCount: 1,

});

describe("Country", () => {

let fixture: ComponentFixture<CountryComponent>;

afterEach(() => clearPage(fixture));

beforeEach(async () => {

const result = await render(CountryComponent, {

imports: [

CoreTestingModule.withConfig(),

ThemeSharedTestingModule.withConfig(),

ThemeBasicTestingModule.withConfig(),

NgxValidateCoreModule,

NgbCollapseModule,

NgbDatepickerModule,

NgbDropdownModule,

],

providers: [

{

provide: CountryService,

useValue: {

getList: () => list$,

},

},

],

});

fixture = result.fixture;

});

it("should display advanced filters", () => {

const filters = screen.getByTestId("country-filters");

const nameInput = getByLabelText(filters, /name/i) as HTMLInputElement;

expect(nameInput.offsetWidth).toBe(0);

const advancedFiltersBtn = screen.getByRole("link", { name: /advanced/i });

userEvent.click(advancedFiltersBtn);

expect(nameInput.offsetWidth).toBeGreaterThan(0);

userEvent.type(nameInput, "fooo{backspace}");

expect(nameInput.value).toBe("foo");

userEvent.click(advancedFiltersBtn);

expect(nameInput.offsetWidth).toBe(0);

});

it("should have a heading", () => {

const heading = screen.getByRole("heading", { name: "Countries" });

expect(heading).toBeTruthy();

});

it("should render list in table", async () => {

const table = await screen.findByTestId("country-table");

const name = getByText(table, "United States of America");

expect(name).toBeTruthy();

});

it("should display edit modal", async () => {

const actionsBtn = screen.queryByRole("button", { name: /actions/i });

userEvent.click(actionsBtn);

const editBtn = screen.getByRole("button", { name: /edit/i });

userEvent.click(editBtn);

await wait(fixture);

const modal = screen.getByRole("dialog");

const modalHeading = queryByRole(modal, "heading", { name: /edit/i });

expect(modalHeading).toBeTruthy();

const closeBtn = getByText(modal, "×");

userEvent.click(closeBtn);

await wait(fixture);

expect(screen.queryByRole("dialog")).toBeFalsy();

});

it("should display create modal", async () => {

const newBtn = screen.getByRole("button", { name: /new/i });

userEvent.click(newBtn);

await wait(fixture);

const modal = screen.getByRole("dialog");

const modalHeading = queryByRole(modal, "heading", { name: /new/i });

expect(modalHeading).toBeTruthy();

});

it("should validate required name field", async () => {

const newBtn = screen.getByRole("button", { name: /new/i });

userEvent.click(newBtn);

await wait(fixture);

const modal = screen.getByRole("dialog");

const nameInput = getByRole(modal, "textbox", {

name: /^name/i,

}) as HTMLInputElement;

userEvent.type(nameInput, "x");

userEvent.type(nameInput, "{backspace}");

const nameError = await findByText(modal, /required/i);

expect(nameError).toBeTruthy();

});

it("should delete a country", () => {

const getSpy = spyOn(fixture.componentInstance.list, "get");

const deleteSpy = jasmine.createSpy().and.returnValue(of(null));

fixture.componentInstance.service.delete = deleteSpy;

const actionsBtn = screen.queryByRole("button", { name: /actions/i });

userEvent.click(actionsBtn);

const deleteBtn = screen.getByRole("button", { name: /delete/i });

userEvent.click(deleteBtn);

const confirmText = screen.getByText("AreYouSure");

expect(confirmText).toBeTruthy();

const confirmBtn = screen.getByRole("button", { name: "Yes" });

userEvent.click(confirmBtn);

expect(deleteSpy).toHaveBeenCalledWith(list$.value.items[0].id);

expect(getSpy).toHaveBeenCalledTimes(1);

});

});

```

## CI配置

你的CI环境需要不同的配置. 要为单元测试设置新的配置, 请在测试项目中找到 *\_angular.json\_* 文件, 或者如下所示添加一个:

```json

// angular.json

"test": {

"builder": "@angular-devkit/build-angular:karma",

"options": { /\* several options here \*/ },

"configurations": {

"production": {

"karmaConfig": "karma.conf.prod.js"

}

}

}

```

现在你可以复制 *\_karma.conf.js\_* 作为 *\_karma.conf.prod.js\_* 并在其中使用你喜欢的任何配置. 请查看[Karma配置文档](http://karma-runner.github.io/5.2/config/configuration-file.html)配置选项.

最后, 不要忘记使用以下命令运行CI测试:

```sh

npm test -- --prod

```

## 另请参阅

- [ABP Community Video - Unit Testing with the Angular UI](https://community.abp.io/articles/unit-testing-with-the-angular-ui-p4l550q3)

HTTP请求

## HTTP请求

## 关于 HttpClient

Angular具有很棒的 `HttpClient` 与后端服务进行通信. 它位于顶层,是[XMLHttpRequest Web API](https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest)的封装. 同时也是Angular建议用于任何HTTP请求的代理,在你的ABP项目中使用 `HttpClient` 是最佳做法.

但是 `HttpClient` 将错误处理留给调用方,换句话说HTTP错误是通过手动处理的,通过挂接到返回的 `Observable` 的观察者中来处理.

```js

getConfig() {

this.http.get(this.configUrl).subscribe(

config => this.updateConfig(config),

error => {

// Handle error here

},

);

}

```

上面的代码尽管清晰灵活,但即使将错误处理委派给Store或任何其他注入. 以这种方式处理错误也是重复性的工作.

`HttpInterceptor` 能够捕获 `HttpErrorResponse` 并可用于集中的错误处理. 然而,在必须放置错误处理程序(也就是拦截器)的情况下,需要额外的工作以及对Angular内部机制的理解. 检查[这个issue](https://github.com/angular/angular/issues/20203)了解详情.

## RestService

ABP核心模块有用于HTTP请求的实用程序服务: `RestService`. 除非另有明确配置,否则它将捕获HTTP错误并调度 `RestOccurError` 操作, 然后由 `ThemeSharedModule` 引入的 `ErrorHandler` 捕获此操作. 你应该已经在应用程序中导入了此模块,在使用 `RestService` 时,默认情况下将自动处理所有HTTP错误.

### RestService 入门

为了使用 `RestService`, 你必须将它注入到你的类中.

```js

import { RestService } from '@abp/ng.core';

@Injectable({

/\* class metadata here \*/

})

class DemoService {

constructor(private rest: RestService) {}

}

```

你不必在模块或组件/指令级别提供 `estService`,因为它已经在**\*\*根中\*\***中提供了.

### 如何使用RestService发出请求

你可以使用 `RestService` 的 `request` 方法来处理HTTP请求. 示例:

```js

getFoo(id: number) {

const request: Rest.Request<null> = {

method: 'GET',

url: '/api/some/path/to/foo/' + id,

};

return this.rest.request<null, FooResponse>(request);

}

```

`request` 方法始终返回 `Observable<T>`. 无论何时使用 `getFoo` 方法,都可以执行以下操作:

```js

doSomethingWithFoo(id: number) {

this.demoService.getFoo(id).subscribe(

foo => {

// Do something with foo.

}

)

}

```

**\*\*你不必担心关于取消订阅\*\***. `RestService` 在内部使用 `HttpClient`,因此它返回的每个可观察对象都是有限的可观察对象,成功或出错后将自动关闭订阅.

如你所见,`request` 方法获取一个具有 `Rest.Reques<T>` 类型的请求选项对象. 此泛型类型需要请求主体的接口. 当没有正文时,例如在 `GET` 或 `DELETE` 请求中,你可以传递 `null`. 示例:

```js

postFoo(body: Foo) {

const request: Rest.Request<Foo> = {

method: 'POST',

url: '/api/some/path/to/foo',

body

};

return this.rest.request<Foo, FooResponse>(request);

}

```

你可以在[此处检查](https://github.com/abpframework/abp/blob/dev/npm/ng-packs/packages/core/src/lib/models/rest.ts#L23)完整的 `Rest.Request<T>` 类型,与Angular中的[HttpRequest](https://github.com/abpframework/abp/blob/dev/npm/ng-packs/packages/core/src/lib/models/rest.ts#L23)类相比只有很少的改动.

### 如何禁用RestService的默认错误处理程序

默认 `request` 方法始终处理错误. 让我们看看如何改变这种行为并由自己处理错误:

```js

deleteFoo(id: number) {

const request: Rest.Request<null> = {

method: 'DELETE',

url: '/api/some/path/to/foo/' + id,

};

return this.rest.request<null, void>(request, { skipHandleError: true });

}

```

`skipHandleError` 配置选项设置为 `true` 时,禁用错误处理程序,并返回 `observable` 引发错误,你可以在订阅中捕获该错误.

```js

removeFooFromList(id: number) {

this.demoService.deleteFoo(id).subscribe(

foo => {

// Do something with foo.

},

error => {

// Do something with error.

}

)

}

```

### 如何从应用程序配置获取特定的API端点

`request` 方法接收到的另一个配置选项是 `apiName` (在v2.4中可用),它用于从应用程序配置获取特定的模块端点.

```js

putFoo(body: Foo, id: string) {

const request: Rest.Request<Foo> = {

method: 'PUT',

url: '/' + id,

body

};

return this.rest.request<Foo, void>(request, {apiName: 'foo'});

}

```

上面的putFoo将请求 `https://localhost:44305/api/some/path/to/foo/{id}` 当环境变量如下:

```js

// environment.ts

export const environment = {

apis: {

default: {

url: 'https://localhost:44305',

},

foo: {

url: 'https://localhost:44305/api/some/path/to/foo',

},

},

/\* rest of the environment variables here \*/

}

```

### 如何观察响应对象或HTTP事件而不是正文

`RestService` 假定你通常对响应的正文感兴趣,默认情况下将 `observe` 属性设置为 `body`. 但是有时你可能对其他内容(例如自定义标头)非常感兴趣. 为此, `request` 方法在 `config` 对象中接收 `watch` 属性.

```js

getSomeCustomHeaderValue() {

const request: Rest.Request<null> = {

method: 'GET',

url: '/api/some/path/that/sends/some-custom-header',

};

return this.rest.request<null, HttpResponse<any>>(

request,

{observe: Rest.Observe.Response},

).pipe(

map(response => response.headers.get('Some-Custom-Header'))

);

}

```

你可以在[此处](https://github.com/abpframework/abp/blob/dev/npm/ng-packs/packages/core/src/lib/models/rest.ts#L10)找到 `Rest.Observe` 枚举.

## 下一步是什么?

\* [本地化](./Localization.md)

本地化

# 本地化

在阅读本地化管道和本地化服务之前你应该了解本地化Key.

本地化key格式由两个部分组成,分别是**\*\*资源名\*\***和**\*\*Key\*\***

`ResourceName::Key`

> 如果你没有指定资源名称,它默认是在 `environment.ts` 中声明的 `defaultResourceName`.

```js

const environment = {

//...

localization: {

defaultResourceName: 'MyProjectName',

},

};

```

所以这两个结果是一样的:

```html

<h1>{%{{{ '::Key' | abpLocalization }}}%}</h1>

<h1>{%{{{ 'MyProjectName::Key' | abpLocalization }}}%}</h1>

```

## 使用本地化管道

你可以使用 `abpLocalization` 管道来获取本地化的文本. 例:

```html

<h1>{%{{{ 'Resource::Key' | abpLocalization }}}%}</h1>

```

管道将用本地化的文本替换Key.

你还可以指定一个默认值,如下所示:

```html

<h1>{%{{{ { key: 'Resource::Key', defaultValue: 'Default Value' } | abpLocalization }}}%}</h1>

```

要使用插值,必须将插值作为管道参数给出. 例如:

本地化数据存储在键值对中:

```js

{

//...

AbpAccount: { // AbpAccount is the resource name

Key: "Value",

PagerInfo: "Showing {0} to {1} of {2} entries"

}

}

```

所以我们可以这样使用Key:

```html

<h1>{%{{{ 'AbpAccount::PagerInfo' | abpLocalization:'20':'30':'50' }}}%}</h1>

<!-- Output: Showing 20 to 30 of 50 entries -->

```

### 使用本地化服务

首先应该从 **\*\*@abp/ng.core\*\*** 导入 `LocalizationService`.

```js

import { LocalizationService } from '@abp/ng.core';

class MyClass {

constructor(private localizationService: LocalizationService) {}

}

```

之后你就可以使用本地化服务.

> 你可以将插值参数作为参数添加到 `instant()` 和 `get()` 方法中.

```js

this.localizationService.instant('AbpIdentity::UserDeletionConfirmation', 'John');

// with fallback value

this.localizationService.instant(

{ key: 'AbpIdentity::UserDeletionConfirmation', defaultValue: 'Default Value' },

'John',

);

// Output

// User 'John' will be deleted. Do you confirm that?

```

要获取[*\_Observable\_*](https://rxjs.dev/guide/observable)的本地化文本,应该使用 `get` 方法而不是 `instant`:

```js

this.localizationService.get('Resource::Key');

// with fallback value

this.localizationService.get({ key: 'Resource::Key', defaultValue: 'Default Value' });

```

### 使用配置状态

要使用 `getLocalization` 方法,你应该导入 `ConfigState`.

```js

import { ConfigState } from '@abp/ng.core';

```

然后你可以按以下方式使用它:

```js

this.store.selectSnapshot(ConfigState.getLocalization('ResourceName::Key'));

```

`getLocalization` 方法可以与 `本地化key` 和 [`LocalizationWithDefault`](https://github.com/abpframework/abp/blob/dev/npm/ng-packs/packages/core/src/lib/models/config.ts#L34) 接口一起使用.

```js

this.store.selectSnapshot(

ConfigState.getLocalization(

{

key: 'AbpIdentity::UserDeletionConfirmation',

defaultValue: 'Default Value',

},

'John',

),

);

```

本地化资源存储在 `ConfigState` 的 `localization` 属性中.

## RTL支持

从v2.9开始,ABP支持RTL. 如果使用v2.9及更高版本生成新项目,你无需进行任何更改. 如果要从早期版本迁移项目,请按照以下2个步骤操作:

#### 步骤 1. 为 Bootstrap LRT和RTL创建Chunks

在[angular.json中找到样式配置](https://angular.io/guide/workspace-config#style-script-config)确保项目中的chunks含有 `bootstrap-rtl.min` 和 `bootstrap-ltr.min`:

```json

{

"projects": {

"MyProjectName": {

"architect": {

"build": {

"options": {

"styles": [

{

"input": "node\_modules/@fortawesome/fontawesome-free/css/all.min.css",

"inject": true,

"bundleName": "fontawesome-all.min"

},

{

"input": "node\_modules/@fortawesome/fontawesome-free/css/v4-shims.min.css",

"inject": true,

"bundleName": "fontawesome-v4-shims.min"

},

{

"input": "node\_modules/@abp/ng.theme.shared/styles/bootstrap-rtl.min.css",

"inject": false,

"bundleName": "bootstrap-rtl.min"

},

{

"input": "node\_modules/bootstrap/dist/css/bootstrap.min.css",

"inject": true,

"bundleName": "bootstrap-ltr.min"

},

"apps/dev-app/src/styles.scss"

],

}

}

}

}

}

}

```

#### 步骤 2. 清除AppComponent中延迟加载的Fontawesome

如果你如上所述为Fontawesome创建并且注入了chunks,就不再需要v2.9版本之前的实现的 `AppComponent` 延迟加载. 删除它们即可,新版的 `AppComponent` 如下所示:

```js

import { Component } from '@angular/core';

@Component({

selector: 'app-root',

template: `

<abp-loader-bar></abp-loader-bar>

<router-outlet></router-outlet>

`,

})

export class AppComponent {}

```

## 文化名称到语言环境文件名的映射

.NET中定义的某些文化名称与Angular语言环境不匹配. 在这种情况下Angular应用程序在运行时会引发如下错误:

![locale-error](./images/locale-error.png)

如果你看到这样的错误,你应该像下面这样传递 `cultureNameLocaleFileMap` 属性到CoreModule的forRoot静态方法.

```js

// app.module.ts

@NgModule({

imports: [

// other imports

CoreModule.forRoot({

// other options

cultureNameLocaleFileMap: {

"DotnetCultureName": "AngularLocaleFileName",

"pt-BR": "pt" // example

}

})

//...

```

查看 [Angular中所有的语言环境文件](https://github.com/angular/angular/tree/master/packages/common/locales).

## 另请参阅

\* [ASP.NET Core中的本地化](../../Localization.md)

## 下一步是什么?

\* [权限管理](./Permission-Management.md)

权限管理

# 权限管理

权限是为特定用户,角色或客户端授予或禁止的简单策略. 你可以在[ABP授权文档](../../Authorization.md)中阅读更多信息.

你可以使用 `ConfigState` 的 `getGrantedPolicy` 选择器获取经过身份验证的用户的权限.

你可以获取权限的布尔值:

```js

import { Store } from '@ngxs/store';

import { ConfigState } from '@abp/ng.core';

export class YourComponent {

constructor(private store: Store) {}

ngOnInit(): void {

const canCreate = this.store.selectSnapshot(ConfigState.getGrantedPolicy('AbpIdentity.Roles.Create'));

}

// ...

}

```

或者你可以通过 `ConfigStateService` 获取它:

```js

import { ConfigStateService } from '@abp/ng.core';

export class YourComponent {

constructor(private configStateService: ConfigStateService) {}

ngOnInit(): void {

const canCreate = this.configStateService.getGrantedPolicy('AbpIdentity.Roles.Create');

}

// ...

}

```

## 权限指令

你可以使用 `PermissionDirective` 来根据用户的权限控制DOM元素是否可见.

```html

<div \*abpPermission="'AbpIdentity.Roles'">

仅当用户具有`AbpIdentity.Roles`权限时,此内容才可见.

</div>

```

如上所示,你可以使用 `abpPermission` 结构指令从DOM中删除元素.

该指令也可以用作属性指令,但是我们建议你将其用作结构指令.

## 权限守卫

如果你想要在导航过程中控制经过身份验证的用户对路由的访问权限,可以使用 `PermissionGuard`.

\* 从@abp/ng.core导入PermissionGuard.

\* 添加 `canActivate: [PermissionGuard]` 到你的路由对象.

\* 添加 `requiredPolicy` 到路由模块路由的 `data` 属性.

```js

import { PermissionGuard } from '@abp/ng.core';

// ...

const routes: Routes = [

{

path: 'path',

component: YourComponent,

canActivate: [PermissionGuard],

data: {

requiredPolicy: 'YourProjectName.YourComponent', // policy key for your component

},

},

];

```

授予的策略存储在 `ConfigState` 的 `auth` 属性中.

## 下一步是什么?

\* [多租户](./Multi-Tenancy.md)

多租户

# 在 Angular UI 中的多租户支持

ABP Angular UI 支持多租户，以下是与多租户相关的一些特性：

![Tenants Page](./images/tenants-page.png)

<p style="font-size:small;text-align:center;">租户页面</p>

在上面的页面中，您可以：

- 查看所有租户。

- 创建新的租户。

- 编辑现有租户。

- 删除租户。

![Tenant Switching Component](./images/tenant-switching-box.png)

<p style="font-size:small;text-align:center;">租户切换组件</p>

您可以在 MVC Account Public Module 的子页面（如登录页面）中使用租户切换框来在现有租户之间切换。Angular UI 从 `application-configuration` 响应中获取选定的租户，并在每个请求中将租户 ID 作为 `\_\_tenant` header 发送到后端。

## 域名/子域名租户解析器

> **\*\*注意：\*\*** 如果要执行下面的步骤，您还应该实现后端的域名/子域名租户解析器功能。请参阅[多租户文档中的域名/子域名租户解析器](../../Multi-Tenancy#domain-subdomain-tenant-resolver)以了解后端实现。

Angular UI 可以从运行 URL 中获取租户名称。您可以通过子域名（如 mytenant1.mydomain.com）或整个域名（如 mytenant.com）来确定当前租户。要做到这一点，您需要在环境中设置 `application.baseUrl` 属性：

子域名解析器：

```js

// environment.prod.ts

export const environment = {

//...

application: {

baseUrl: "https://{0}.mydomain.com/",

},

//...

};

```

**\*\*{0}\*\*** 是用于确定当前租户唯一名称的占位符。

在上述配置完成后，如果您的应用程序运行在 `mytenant1.mydomain.com` 上，应用程序将获取租户名称为 **\*\*mytenant1\*\***。接下来，应用程序将调用 `/api/abp/multi-tenancy/tenants/by-name/mytenant1` 端点来检查租户是否存在。如果租户（mytenant1）存在，则应用程序将保留此租户数据，并在每个请求中将其 `id` 作为 `\_\_tenant` header 发送到后端。如果租户不存在，则应用程序不会将 `\_\_tenant` header 发送到后端。

> **\*\*重要提示：\*\*** 如果在 `baseUrl` 中使用了占位符（**\*\*{0}\*\***），则子页面（如登录页面）中 `AccountLayoutComponent` 中的租户切换组件将被隐藏。

域名解析器:

```js

// environment.prod.ts

export const environment = {

//...

application: {

baseUrl: "https://{0}.com/",

},

//...

};

```

配置完成后，如果您的应用程序运行在 `mytenant.com` 上，应用程序将获取租户名称为 **\*\*mytenant\*\***。

### 租户特定的入口

在环境中，可以将占位符 **\*\*{0}\*\*** 放入 API URL 中以确定租户特定的入口。

```js

// environment.prod.ts

export const environment = {

//...

application: {

baseUrl: "https://{0}.mydomain.com/",

//...

},

oAuthConfig: {

issuer: "https://{0}.ids.mydomain.com",

//...

},

apis: {

default: {

url: "https://{0}.api.mydomain.com",

},

AbpIdentity: {

url: "https://{0}.identity.mydomain.com",

},

},

};

```

> **\*\*重要提示:\*\*** `application.baseUrl`和`baseUrl`属性中的 `{0}` 占位符是必需的，以便从运行 URL 中获取租户。API URL 中的其他占位符是可选的。

在上述配置完成后，如果您的应用程序运行在 `mytenant1.mydomain.com`上，应用程序将获取租户名称为 **\*\*mytenant1\*\*** ，并在应用程序初始化时将环境对象替换为 `EnvironmentService` 中的以下内容:

```js

// environment object in EnvironmentService

{

//...

application: {

baseUrl: 'https://mytenant1.mydomain.com/',

//...

},

oAuthConfig: {

issuer: 'https://mytenant1.ids.mydomain.com',

//...

},

apis: {

default: {

url: 'https://mytenant1.api.mydomain.com',

},

AbpIdentity: {

url: 'https://mytenant1.identity.mydomain.com',

},

},

}

```

替换后，应用程序将使用以下 URL:

- `https://mytenant1.ids.mydomain.com` 作为 AuthServer URL。

- `https://mytenant1.api.mydomain.com` 作为默认 URL。

- `https://mytenant1.identity.mydomain.com` 作为 `AbpIdentity` 入点口 URL.

应用程序在每个请求中发送包含当前租户 id 的 `\_\_tenant` header.

## 参见

- [ABP 中的多租户](../../Multi-Tenancy.md)

确认弹层

# 确认弹层

你可以使用@abp/ng.theme.shared包中提供 `ConfirmationService` 放置在你项目的级别来显示确认弹层

## 入门

你不必在模块或组件级别提供 `ConfirmationService`,它已经在**\*\*根\*\***级别提供,你可以在你的组件,指令或服务直接注入并使用它.

```js

import { ConfirmationService } from '@abp/ng.theme.shared';

@Component({

/\* class metadata here \*/

})

class DemoComponent {

constructor(private confirmation: ConfirmationService) {}

}

```

## 用法

你可以使用 `ConfirmationService` 的 `success`, `warn`, `error` 和 `info` 方法显示一个确认弹层.

### 如何显示一个确认弹层

```js

const confirmationStatus$ = this.confirmation.success('Message', 'Title');

```

- `ConfirmationService` 方法接收三个参数,分别是 `message`, `title`, 和 `options`.

- `success`, `warn`, `error`, 和 `info` 方法返回一个[RxJS Subject](https://rxjs-dev.firebaseapp.com/guide/subject)监听确认弹层关闭事件. 事件值类型是 [`Confirmation.Status`](https://github.com/abpframework/abp/blob/master/npm/ng-packs/packages/theme-shared/src/lib/models/confirmation.ts#L24)枚举.

### 如何监听关闭事件

你可以订阅确认弹层关闭事件,例:

```js

import { Confirmation, ConfirmationService } from '@abp/ng.theme.shared';

constructor(private confirmation: ConfirmationService) {}

this.confirmation

.warn('::WillBeDeleted', { key: '::AreYouSure', defaultValue: 'Are you sure?' })

.subscribe((status: Confirmation.Status) => {

// your code here

});

```

- `message` 和 `title` 参数接收字符串,本地化Key或本地化对象. 参阅[本地化文档](./Localization.md)

- `Confirmation.Status` 是一个枚举,具有三个属性;

- `Confirmation.Status.confirm` 是一个关闭事件值,当通过确认按钮关闭弹出窗口时触发此事件.

- `Confirmation.Status.reject` 是一个关闭事件值,当通过取消按钮关闭弹出窗口时触发此事件.

- `Confirmation.Status.dismiss` 是一个关闭事件值,当通过按Escape键关闭弹出窗口时触发此事件.

如果你对确认状态不感兴趣,则不必订阅返回的observable:

```js

this.confirmation.error('You are not authorized.', 'Error');

```

### 如何显示具有给定选项的确认弹层

选项可以作为第三个参数传递给`success`, `warn`, `error`, 和 `info` 方法:

```js

const options: Partial<Confirmation.Options> = {

hideCancelBtn: false,

hideYesBtn: false,

dismissible: false,

cancelText: 'Close',

yesText: 'Confirm',

messageLocalizationParams: ['Demo'],

titleLocalizationParams: [],

};

this.confirmation.warn(

'AbpIdentity::RoleDeletionConfirmationMessage',

'Are you sure?',

options,

);

```

- `hideCancelBtn` 选项为 `true` 时隐藏取消按钮. 默认值为 `false`.

- `hideYesBtn` 选项为 `true` 时隐藏确认按钮. 默认值为 `false`.

- `dismissible`选项允许通过按Escape键或单击背景来取消确认弹出窗口. 默认值为 `true`.

- `cancelText` 是取消按钮的文本,可以传递本地化键或本地化对象. 默认值为 `AbpUi::Cancel`.

- `yesText` 是确定按钮的文本,可以传递本地化键或本地化对象. 默认值为 `AbpUi::Yes`.

- `messageLocalizationParams`是用于消息本地化的插值参数.

- `titleLocalizationParams` 是标题本地化的插值参数.

使用以上选项确认弹层窗口如下所示:

![confirmation](./images/confirmation.png)

你可以传递HTML字符串作为标题,消息或按钮文本. 例如:

```js

const options: Partial<Confirmation.Options> = {

yesText: '<i class="fa fa-trash mr-1"></i>Yes, delete it',

};

this.confirmation.warn(

`

<strong>Role Demo</strong> will be <strong>deleted</strong>

<br>

Do you confirm that?

`,

'<span class="my-custom-title">Are you sure?</span>',

options,

);

```

由于这些值现在是HTML,因此应该手动处理本地化. 参阅[LocalizationService](./Localization#using-the-localization-service)了解如何实现.

> 注意,Angular会清除所有字符串,并且并非每个HTML字符串都可以使用. 仅显示被Angular视为"安全"的值.

### 如何删除一个确认弹层

打开的确认弹出窗口可以通过 `clear` 方法手动删除:

```js

this.confirmation.clear();

```

## API

### success

```js

success(

message: Config.LocalizationParam,

title: Config.LocalizationParam,

options?: Partial<Confirmation.Options>,

): Observable<Confirmation.Status>

```

> 请参见[`Config.LocalizationParam`类型](https://github.com/abpframework/abp/blob/master/npm/ng-packs/packages/core/src/lib/models/config.ts#L46)和[Confirmation名称空间](https://github.com/abpframework/abp/blob/master/npm/ng-packs/packages/theme-shared/src/lib/models/confirmation.ts)

### warn

```js

warn(

message: Config.LocalizationParam,

title: Config.LocalizationParam,

options?: Partial<Confirmation.Options>,

): Observable<Confirmation.Status>

```

### error

```js

error(

message: Config.LocalizationParam,

title: Config.LocalizationParam,

options?: Partial<Confirmation.Options>,

): Observable<Confirmation.Status>

```

### info

```js

info(

message: Config.LocalizationParam,

title: Config.LocalizationParam,

options?: Partial<Confirmation.Options>,

): Observable<Confirmation.Status>

```

### clear

```js

clear(

status: Confirmation.Status = Confirmation.Status.dismiss

): void

```

- `status` 参数是确认关闭事件的值.

## 下一步是什么?

- [Toast Overlay](./Toaster-Service.md)

Toast Overlay

# Toast Overlay

你可以通常将@abp/ng.theme.shared包提供的 `ToasterService` 放置在你项目的根级别下以覆盖显示消息.

## 入门

你不必在模块或组件级别提供 `ToasterService`,它已经在**\*\*根\*\***级别提供,你可以在你的组件,指令或服务直接注入并使用它.

```js

import { ToasterService } from '@abp/ng.theme.shared';

@Component({

/\* class metadata here \*/

})

class DemoComponent {

constructor(private toaster: ToasterService) {}

}

```

## 用法

你可以使用 `ToasterService` 的 `success`, `warn`, `error` 和 `info` 方法显示一个overlay.

### 如何显示一个Toast Overlay

```js

this.toaster.success('Message', 'Title');

```

- `ToasterService` 方法接收三个参数,分别是 `message`, `title`, 和 `options`.

- `success`, `warn`, `error`, 和 `info` 方法返回一个已打开的 toast overlay Id. 可以使用此id删除toast.

### 如何显示具有给定选项的Toast Overlay

选项可以作为第三个参数传递给`success`, `warn`, `error`, 和 `info` 方法:

```js

import { Toaster, ToasterService } from '@abp/ng.theme.shared';

//...

constructor(private toaster: ToasterService) {}

//...

const options: Partial<Toaster.ToastOptions> = {

life: 10000,

sticky: false,

closable: true,

tapToDismiss: true,

messageLocalizationParams: ['Demo', '1'],

titleLocalizationParams: []

};

this.toaster.error('AbpUi::EntityNotFoundErrorMessage', 'AbpUi::Error', options);

```

- `life` 选项是关闭的时间毫秒数. 默认值是 `5000`.

- `sticky` 选项为 `true` 时忽略 `life` 选项,将toast overlay留在屏幕上. 默认值是 `false`.

- `closable` 选项为 `true` 时在toast overlay上显示关闭图标. 默认值是 `true`.

- `tapToDismiss` 选项为 `true` 允许通过单击关闭toast overlay. 默认值是 `false`.

- `yesText` 是确定按钮的文本,可以传递本地化键或本地化对象. 默认值是 `AbpUi::Yes`.

- `messageLocalizationParams` 是用于消息本地化的插值参数.

- `titleLocalizationParams` 是标题本地化的插值参数.

使用上面的选项,toast overlay看起来像这样:

![toast](./images/toast.png)

### 如何删除 Toast Overlay

已打开的toast overlay可以通过手动调用 `remove` 方法传递指定的 toast `id`删除.

```js

const toastId = this.toaster.success('Message', 'Title')

this.toaster.remove(toastId);

```

### 如何删除所有的Toasts

可以手动调用 `clear` 方法删除所有的已打开的toasts.

```js

this.toaster.clear();

```

## API

### success

```js

success(

message: Config.LocalizationParam,

title: Config.LocalizationParam,

options?: Partial<Toaster.ToastOptions>,

): number

```

- `Config` 命令空间可以从 `@abp/ng.core` 导入.

- `Toaster` 命令空间可以从 `@abp/ng.theme.shared` 导入.

> 请参见[`Config.LocalizationParam`类型](https://github.com/abpframework/abp/blob/master/npm/ng-packs/packages/core/src/lib/models/config.ts#L46)和[`Toaster` namespace](https://github.com/abpframework/abp/blob/master/npm/ng-packs/packages/theme-shared/src/lib/models/toaster.ts)

### warn

```js

warn(

message: Config.LocalizationParam,

title: Config.LocalizationParam,

options?: Partial<Toaster.ToastOptions>,

): number

```

### error

```js

error(

message: Config.LocalizationParam,

title: Config.LocalizationParam,

options?: Partial<Toaster.ToastOptions>,

): number

```

### info

```js

info(

message: Config.LocalizationParam,

title: Config.LocalizationParam,

options?: Partial<Toaster.ToastOptions>,

): number

```

### remove

```js

remove(id: number): void

```

按给定的id移除打开的toast.

### clear

```js

clear(): void

```

删除所有打开的toasts.

## 另请参阅

- [Confirmation Popup](./Confirmation-Service.md)

## 确认弹层?

- [Config State](./Config-State.md)

配置状态

## 配置状态

`ConfigStateService` 是一个单例服务,即在应用程序的根级别提供,用于与 `Store` 中的应用程序配置状态进行交互.

## 使用前

为了使用 `ConfigStateService`,你必须将其注入到你的类中.

```js

import { ConfigStateService } from '@abp/ng.core';

@Component({

/\* class metadata here \*/

})

class DemoComponent {

constructor(private config: ConfigStateService) {}

}

```

你不必在模块或组件/指令级别提供 `ConfigStateService`,因为它已经在**\*\*根中\*\***提供.

## 选择器方法

`ConfigStateService` 有许多选择器方法允许你从 `Store` 获取特定或所有的配置.

### 如何从Store获取所有的配置

你可以使用 `ConfigStateService` 的 `getAll` 方法从Store获取所有的配置对象. 用法如下:

```js

// this.config is instance of ConfigStateService

const config = this.config.getAll();

```

### 如何从Store获取特定的配置

你可以使用 `ConfigStateService` 的 `getOne` 方法从Store获取特定的配置属性. 你需要将属性名做为参数传递给方法:

```js

// this.config is instance of ConfigStateService

const currentUser = this.config.getOne("currentUser");

```

有时你想要获取具体信息,而不是当前用户. 例如你只想获取到 `tenantId`:

```js

const tenantId = this.config.getDeep("currentUser.tenantId");

```

或通过提供键数组作为参数:

```js

const tenantId = this.config.getDeep(["currentUser", "tenantId"]);

```

`getDeep` 可以执行 `getOne` 的所有操作. 但 `getOne` 的执行效率要高一些.

#### 配置状态属性

请参阅 `Config.State` 类型,你可以通过 `getOne` 和 `getDeep` 获取所有属性. 你可以在[config.ts 文件](https://github.com/abpframework/abp/blob/dev/npm/ng-packs/packages/core/src/lib/models/config.ts#L7)中找到.

### 如何从Store获取应用程序信息

`getApplicationInfo` 方法从存储为配置状态存储的环境变量中获取应用程序信息. 你可以这样使用它:

```js

// this.config is instance of ConfigStateService

const appInfo = this.config.getApplicationInfo();

```

该方法不会返回 `undefined` 或 `null`,而是会返回一个空对象(`{}`). 换句话说,当你使用上面代码中的 `appInfo` 属性时,永远不会出现错误.

#### 应用程序信息属性

请参阅 `Config.State` 类型,你可以通过 `getApplicationInfo` 获取所有属性. 你可以在[config.ts 文件](https://github.com/abpframework/abp/blob/dev/npm/ng-packs/packages/core/src/lib/models/config.ts#L21)中找到.

### 如何从Store获取

`getApplicationInfo` 方法从存储为配置状态存储的环境变量中获取特定的API URL. 你可以这样使用它:

```js

// this.config is instance of ConfigStateService

const apiUrl = this.config.getApiUrl();

// environment.apis.default.url

const searchUrl = this.config.getApiUrl("search");

// environment.apis.search.url

```

该方法返回给定键的特定的API `url`. 如果没有Key,则使用 `default`.

### 如何从Store获取所有的设置

你可以使用 `ConfigStateService` 的 `getSettings` 获取配置状态所有的设置对象. 你可以这样使用它:

```js

// this.config is instance of ConfigStateService

const settings = this.config.getSettings();

```

实际上该方法可以通过**\*\*传递关键字\*\***来搜索设置.

```js

const localizationSettings = this.config.getSettings("Localization");

/\*

{

'Abp.Localization.DefaultLanguage': 'en'

}

\*/

```

请注意, **\*\*设置搜索区分大小写\*\***.

### 如何从Store获取特定的设置

你可以使用 `ConfigStateService` 的 `getSetting` 获取配置状态特定的设置. 你可以这样使用它:

```js

// this.config is instance of ConfigStateService

const defaultLang = this.config.getSetting("Abp.Localization.DefaultLanguage");

// 'en'

```

### 如何从Store获取特定的权限

你可以使用 `ConfigStateService` 的 `getGrantedPolicy` 获取配置状态特定的权限. 你应该将策略key做为参数传递给方法:

```js

// this.config is instance of ConfigStateService

const hasIdentityPermission = this.config.getGrantedPolicy("Abp.Identity");

// true

```

你还可以使用 **\*\*组合策略key\*\*** 来微调你的选择:

```js

// this.config is instance of ConfigStateService

const hasIdentityAndAccountPermission = this.config.getGrantedPolicy(

"Abp.Identity && Abp.Account"

);

// false

const hasIdentityOrAccountPermission = this.config.getGrantedPolicy(

"Abp.Identity || Abp.Account"

);

// true

```

创建权限选择器时,请考虑以下**\*\*规则\*\***:

- 最多可组合两个键.

- `&&` 操作符查找两个键.

- `||` 操作符查找任意一个键.

- 空字符串 `''` 做为键将返回 `true`

- 使用没有第二个键的操作符将返回 `false`

### 如何从Store中获取翻译

`ConfigStateService` 的 `getLocalization` 方法用于翻译. 这里有一些示例:

```js

// this.config is instance of ConfigStateService

const identity = this.config.getLocalization("AbpIdentity::Identity");

// 'identity'

const notFound = this.config.getLocalization("AbpIdentity::IDENTITY");

// 'AbpIdentity::IDENTITY'

const defaultValue = this.config.getLocalization({

key: "AbpIdentity::IDENTITY",

defaultValue: "IDENTITY"

});

// 'IDENTITY'

```

请参阅[本地化文档](./Localization.md)了解详情.

## 分发方法

`ConfigStateService` 有几种分发方法,让你方便地将预定义操作分发到 `Store`.

### 如何从服务器获取应用程序配置

`dispatchGetAppConfiguration` 触发对端点的请求,该端点使用应用程序状态进行响应,然后将此响应作为配置状态放置到 `Store`中.

```js

// this.config is instance of ConfigStateService

this.config.dispatchGetAppConfiguration();

// returns a state stream which emits after dispatch action is complete

```

请注意,**\*\*你不必在应用程序启动时调用此方法\*\***,因为在启动时已经从服务器收到了应用程序配置.

### 如何修补路由配置

`dispatchPatchRouteByName` 根据名称查找路由, 并将其在 `Store` 中的配置替换为作为第二个参数传递的新配置.

```js

// this.config is instance of ConfigStateService

const newRouteConfig: Partial<ABP.Route> = {

name: "Home",

path: "home",

children: [

{

name: "Dashboard",

path: "dashboard"

}

]

};

this.config.dispatchPatchRouteByName("::Menu:Home", newRouteConfig);

// returns a state stream which emits after dispatch action is complete

```

### 如何添加新路由配置

`dispatchAddRoute` 向 `Store` 的配置状态添加一个新路由. 应该将路由配置做为方法参数传递.

```js

// this.config is instance of ConfigStateService

const newRoute: ABP.Route = {

name: "My New Page",

iconClass: "fa fa-dashboard",

path: "page",

invisible: false,

order: 2,

requiredPolicy: "MyProjectName.MyNewPage"

};

this.config.dispatchAddRoute(newRoute);

// returns a state stream which emits after dispatch action is complete

```

`newRoute` 将被放置在根级别,没有任何父路由,并且其url将存储为 `'/path'`.

如果你想要**\*\*添加一个子路由,你可以这样做:\*\***

```js

import { eIdentityRouteNames } from '@abp/ng.identity';

// this.config is instance of ConfigStateService

const newRoute: ABP.Route = {

parentName: eIdentityRouteNames.IdentityManagement,

name: "My New Page",

iconClass: "fa fa-dashboard",

path: "page",

invisible: false,

order: 2,

requiredPolicy: "MyProjectName.MyNewPage"

};

this.config.dispatchAddRoute(newRoute);

// returns a state stream which emits after dispatch action is complete

```

`newRoute` 做为 `'AbpAccount::Login'` 父路由的子路由被放置,它的url被设置为 `'/account/login/page'`.

#### 路由配置属性

请参阅 `ABP.Route` 类型,获取可在参数中传递给 `dispatchSetEnvironment` 的所有属性. 你可以在[common.ts 文件](https://github.com/abpframework/abp/blob/dev/npm/ng-packs/packages/core/src/lib/models/common.ts#L27)中找到.

### 如何设置环境

`dispatchSetEnvironment` 将传递给它的环境变量放在 `Store` 中的配置状态下. 使用方法如下:

```js

// this.config is instance of ConfigStateService

this.config.dispatchSetEnvironment({

/\* environment properties here \*/

});

// returns a state stream which emits after dispatch action is complete

```

注意,**\*\*你不必在应用程序启动时调用此方法\*\***,因为环境变量已经在启动时存储了.

#### 环境属性

请参阅 `Config.Environment` 类型,获取可在参数中传递给 `dispatchSetEnvironment` 的所有属性. 你可以在[config.ts 文件](https://github.com/abpframework/abp/blob/dev/npm/ng-packs/packages/core/src/lib/models/config.ts#L13)中找到.

## 下一步是什么?

- [修改菜单](./Modifying-the-Menu.md)

修改菜单

# 修改菜单

菜单在 @abp/ng.theme.basic包 `ApplicationLayoutComponent` 内部. 有几种修改菜单的方法,本文档介绍了这些方法. 如果你想完全替换菜单,请参考[组件替换文档]了解如何替换布局.

## 如何添加Logo

环境变量中的 `logoUrl` 是logo的url.

你可以在 `src/assets` 文件夹下添加logo并设置 `logoUrl`:

```js

export const environment = {

// other configurations

application: {

name: 'MyProjectName',

logoUrl: 'assets/logo.png',

},

// other configurations

};

```

## 如何添加导航元素

### 通过 `RoutesService`

你可以通过调用 `RoutesService` 的 `add` 方法添加路由到菜单,它是一个单例的服务,在root中提供,你可以立即注入使用它.

```js

import { RoutesService, eLayoutType } from '@abp/ng.core';

import { Component } from '@angular/core';

@Component(/\* component metadata \*/)

export class AppComponent {

constructor(routes: RoutesService) {

routes.add([

{

path: '/your-path',

name: 'Your navigation',

order: 101,

iconClass: 'fas fa-question-circle',

requiredPolicy: 'permission key here',

layout: eLayoutType.application,

},

{

path: '/your-path/child',

name: 'Your child navigation',

parentName: 'Your navigation',

order: 1,

requiredPolicy: 'permission key here',

},

]);

}

}

```

另一种方法是使用路由提供程序. 首先创建一个提供程序:

```js

// route.provider.ts

import { RoutesService, eLayoutType } from '@abp/ng.core';

import { APP\_INITIALIZER } from '@angular/core';

export const APP\_ROUTE\_PROVIDER = [

{ provide: APP\_INITIALIZER, useFactory: configureRoutes, deps: [RoutesService], multi: true },

];

function configureRoutes(routes: RoutesService) {

return () => {

routes.add([

{

path: '/your-path',

name: 'Your navigation',

requiredPolicy: 'permission key here',

order: 101,

iconClass: 'fas fa-question-circle',

layout: eLayoutType.application,

},

{

path: '/your-path/child',

name: 'Your child navigation',

parentName: 'Your navigation',

requiredPolicy: 'permission key here',

order: 1,

},

]);

};

}

```

...然后在app.module.ts ...

```js

import { NgModule } from '@angular/core';

import { APP\_ROUTE\_PROVIDER } from './route.provider';

@NgModule({

providers: [APP\_ROUTE\_PROVIDER],

// imports, declarations, and bootstrap

})

export class AppModule {}

```

下面是每个属性的工作原理:

- `path` 是导航元素的绝对路径.

- `name` 是导航元素的label. 可以使用本地化Key和本地化对象.

- `parentName` 是菜单中父路由的 `name` 的引用,用于创建多级菜单项.

- `requiredPolicy` 是用于访问该页面的权限Key. 参阅[权限管理文档](Permission-Management.md).

- `order` 是导航元素的排序. `Administration` 的顺序是 `100`. 在排序top级别菜单项时请记得这一点.

- `iconClass` 是 `i` 标签的class, 它放在导航label的左边.

- `layout` 定义路由使用哪个布局加载. (默认: `eLayoutType.empty`).

- `invisible` 使该项在菜单中不可见. (默认: `false`).

### 通过 `AppRoutingModule` 的 `routes` 属性

你可以通过在 `app-routing.module` 中将路由作为子属性添加到路由配置的 `data` 属性来定义路由. `@abp/ng.core` 包组织路由并将其存储在 `ConfigState` 中.`ApplicationLayoutComponent` 从存储中获取路由显示在菜单上.

你可以像以下一样添加 `routes` 属性:

```js

{

path: 'your-path',

data: {

routes: {

name: 'Your navigation',

order: 101,

iconClass: 'fas fa-question-circle',

requiredPolicy: 'permission key here',

children: [

{

path: 'child',

name: 'Your child navigation',

order: 1,

requiredPolicy: 'permission key here',

},

],

},

},

}

```

或者你可以这样做:

```js

{

path: 'your-path',

data: {

routes: [

{

path: '/your-path',

name: 'Your navigation',

order: 101,

iconClass: 'fas fa-question-circle',

requiredPolicy: 'permission key here',

},

{

path: '/your-path/child',

name: 'Your child navigation',

parentName: 'Your navigation',

order: 1,

requiredPolicy: 'permission key here',

},

] as ABP.Route[], // can be imported from @abp/ng.core

},

}

```

第二种方法的优点是你不必绑定到父/子结构,可以使用任何喜欢的路由.

如上所述添加 `routes` 属性后,导航菜单看起来像这样:

![navigation-menu-via-app-routing](./images/navigation-menu-via-app-routing.png)

## 如何修补或删除导航元素

`RoutesService` 的 `patch` 方法通过名称查找路由,并将配置替换为第二个参数传递的新配置. `remove` 方法会找到一个路由并将其连同其子路由一起删除.

```js

// this.routes is instance of RoutesService

// eThemeSharedRouteNames enum can be imported from @abp/ng.theme.shared

const dashboardRouteConfig: ABP.Route = {

path: '/dashboard',

name: '::Menu:Dashboard',

parentName: '::Menu:Home',

order: 1,

layout: eLayoutType.application,

};

const newHomeRouteConfig: Partial<ABP.Route> = {

iconClass: 'fas fa-home',

parentName: eThemeSharedRouteNames.Administration,

order: 0,

};

this.routes.add([dashboardRouteConfig]);

this.routes.patch('::Menu:Home', newHomeRouteConfig);

this.routes.remove(['Your navigation']);

```

- 根据给定的 `parentName` 将 *\_Home\_* 导航移动到 *\_Administration\_* 下拉菜单下.

- 为 *\_Home\_* 添加了图标.

- 指定 *\_Home\_* 的顺序为列表的第一项.

- 为 *\_Home\_* 添加了一个名为 *\_Dashboard\_* 的子路由.

- 删除 *\_Your navigation\_* 与其子路由.

上述操作后,新的菜单看起来如下:

![navigation-menu-after-patching](./images/navigation-menu-after-patching.png)

## 如何在菜单的右侧添加元素

你可以通过调用 `NavItemsService` 的 `addItems` 方法将元素添加到菜单的右侧. 这是一个单例服务,即以根身份提供. 因此你可以立即注入并使用它.

```js

import { NavItemsService } from '@abp/ng.theme.shared';

import { Component } from '@angular/core';

@Component({

template: `

<input type="search" placeholder="Search" class="bg-transparent border-0 color-white" />

`,

})

export class MySearchInputComponent {}

@Component(/\* component metadata \*/)

export class AppComponent {

constructor(private navItems: NavItemsService) {

navItems.addItems([

{

id: 'MySearchInput',

order: 1,

component: MySearchInputComponent,

},

{

id: 'SignOutIcon',

html: '<i class="fas fa-sign-out-alt fa-lg text-white m-2"><i>',

action: () => console.log('Clicked the sign out icon'),

order: 101, // puts as last element

},

]);

}

}

```

上面我们在菜单添加了一个搜索输入和退出登录图标,最终UI如下:

![navigation-menu-search-input](./images/navigation-menu-search-input.png)

> 默认元素的排序为 `100`. 如果要将自定义元素放在默认值之前,请指定一个排序,最高为 `99`. 如果要将自定义元素放在默认值之后,请指定排序从 `101` 开始. 最后如果必须在默认值之间放置一个项目,请按如下所述修补默认元素顺序. 但有一个警告:我们将来可能会添加另一个默认元素,排序也为 `100`.

## 如何修补或删除右侧部分元素

`NavItemsService` 的 `patchItem` 方法通过 `id` 查找元素,并将配置替换为第二个参数传递的新配置. `removeItem` 方法会找到一个元素并删除.

```js

export class AppComponent {

constructor(private navItems: NavItemsService) {

navItems.patchItem(eThemeBasicComponents.Languages, {

requiredPolicy: 'new policy here',

order: 1,

});

navItems.removeItem(eThemeBasicComponents.CurrentUser);

}

}

```

\* 使用新的 `requiredPolicy` 和新的 `order` 修补了语言下拉菜单元素.

\* 删除了当前用户的下拉菜单元素.

## 下一步是什么?

\* [组件替换](./Component-Replacement.md)

替换组件

## 替换组件

你可以将一些ABP的组件替换为你自己的自定义组件.

你可以**\*\*替换\*\***但**\*\*不能自定义\*\***默认ABP组件的原因是禁用或更改该组件的一部分可能会导致问题. 所以我们把这些组件称为可替换组件.

### 如何替换组件

创建一个你想要使用的新组件,添加到 `AppModule` 中的 `declarations` 和`entryComponents` 中.

然后打开 `app.component.ts` 使用 `AddReplaceableComponent` 将你的组件替换ABP组件. 如下所示:

```js

import { AddReplaceableComponent } from '@abp/ng.core'; // imported AddReplaceableComponent action

import { eIdentityComponents } from '@abp/ng.identity'; // imported eIdentityComponents enum

import { Store } from '@ngxs/store'; // imported Store

//...

@Component(/\* component metadata \*/)

export class AppComponent {

constructor(

private store: Store // injected Store

)

{

// dispatched the AddReplaceableComponent action

this.store.dispatch(

new AddReplaceableComponent({

component: YourNewRoleComponent,

key: eIdentityComponents.Roles,

}),

);

}

}

```

![Example Usage](./images/component-replacement.gif)

### 如何替换布局

每个ABP主题模块有3个布局,分别是`ApplicationLayoutComponent`, `AccountLayoutComponent`, `EmptyLayoutComponent`. 这些布局可以用相同的方式替换.

> 一个布局组件模板应该包含 `<router-outlet></router-outlet>` 元素.

下面的例子解释了如何更换 `ApplicationLayoutComponent`:

运行以下命令在 `angular` 文件夹中生成布局:

```bash

yarn ng generate component my-application-layout

```

在你的布局模板(`my-application-layout.component.html`)中添加以下代码:

```html

<router-outlet></router-outlet>

```

打开 `src/app` 文件夹下的 `app.component.ts` 文件添加以下内容:

```js

import { AddReplaceableComponent } from '@abp/ng.core'; // imported AddReplaceableComponent

import { eThemeBasicComponents } from '@abp/ng.theme.basic'; // imported eThemeBasicComponents enum for component keys

import { Store } from '@ngxs/store'; // imported Store

import { MyApplicationLayoutComponent } from './my-application-layout/my-application-layout.component'; // imported MyApplicationLayoutComponent

@Component(/\* component metadata \*/)

export class AppComponent {

constructor(

private store: Store, // injected Store

) {

// dispatched the AddReplaceableComponent action

this.store.dispatch(

new AddReplaceableComponent({

component: MyApplicationLayoutComponent,

key: eThemeBasicComponents.ApplicationLayout,

}),

);

}

}

```

### 布局组件

![Layout Components](./images/layout-components.png)

#### 如何替换LogoComponent

![LogoComponent](./images/logo-component.png)

在 `angular` 目录下运行以下命令创建新的组件 `LogoComponent`:

```bash

yarn ng generate component logo --inlineTemplate --inlineStyle --entryComponent

# You don't need the --entryComponent option in Angular 9

```

打开 `src/app/logo` 目录下生成的 `logo.component.ts` 并使用以下内容替换它:

```js

import { Component } from '@angular/core';

@Component({

selector: 'app-logo',

template: `

<a class="navbar-brand" routerLink="/">

<!-- Change the img src -->

<img

src="https://via.placeholder.com/100x50/343a40/FF0000?text=MyLogo"

alt="logo"

width="100%"

height="auto"

/>

</a>

`,

})

export class LogoComponent {}

```

打开 `src/app` 目录下的 `app.component.ts` 做以下修改:

```js

import { ..., AddReplaceableComponent } from '@abp/ng.core'; // imported AddReplaceableComponent

import { Store } from '@ngxs/store'; // imported Store

import { LogoComponent } from './logo/logo.component'; // imported NavItemsComponent

import { eThemeBasicComponents } from '@abp/ng.theme.basic'; // imported eThemeBasicComponents

//...

@Component(/\* component metadata \*/)

export class AppComponent implements OnInit {

constructor(..., private store: Store) {} // injected Store

ngOnInit() {

//...

// added dispatch

this.store.dispatch(

new AddReplaceableComponent({

component: LogoComponent,

key: eThemeBasicComponents.Logo,

}),

);

}

}

```

最终UI如下:

![New logo](./images/replaced-logo-component.png)

#### 如何替换RoutesComponent

![RoutesComponent](./images/routes-component.png)

在 `angular` 目录下运行以下命令创建新的组件 `RoutesComponent`:

```bash

yarn ng generate component routes --entryComponent

# You don't need the --entryComponent option in Angular 9

```

打开 `src/app/routes` 目录下生成的 `routes.component.ts` 并使用以下内容替换它:

```js

import { ABP, ReplaceableComponents } from '@abp/ng.core';

import {

Component,

HostBinding,

Inject,

Renderer2,

TrackByFunction,

AfterViewInit,

} from '@angular/core';

import { fromEvent } from 'rxjs';

import { debounceTime } from 'rxjs/operators';

@Component({

selector: 'app-routes',

templateUrl: 'routes.component.html',

})

export class RoutesComponent implements AfterViewInit {

@HostBinding('class.mx-auto')

marginAuto = true;

smallScreen = window.innerWidth < 992;

constructor(private renderer: Renderer2) {}

ngAfterViewInit() {

fromEvent(window, 'resize')

.pipe(debounceTime(150))

.subscribe(() => {

this.smallScreen = window.innerWidth < 992;

});

}

}

```

打开 `src/app/routes` 目录下生成的 `routes.component.html` 并使用以下内容替换它:

```html

<ul class="navbar-nav">

<li class="nav-item">

<a class="nav-link" routerLink="/"

><i class="fas fa-home"></i> {%{{{ '::Menu:Home' | abpLocalization }}}%}</a

>

</li>

<li class="nav-item">

<a class="nav-link" routerLink="/my-page"><i class="fas fa-newspaper mr-1"></i>My Page</a>

</li>

<li

#navbarRootDropdown

[abpVisibility]="routeContainer"

class="nav-item dropdown"

display="static"

(click)="

navbarRootDropdown.expand

? (navbarRootDropdown.expand = false)

: (navbarRootDropdown.expand = true)

"

>

<a class="nav-link dropdown-toggle" data-toggle="dropdown" href="javascript:void(0)">

<i class="fas fa-wrench"></i>

{%{{{ 'AbpUiNavigation::Menu:Administration' | abpLocalization }}}%}

</a>

<div

#routeContainer

class="dropdown-menu border-0 shadow-sm"

(click)="$event.preventDefault(); $event.stopPropagation()"

[class.d-block]="smallScreen && navbarRootDropdown.expand"

>

<div

class="dropdown-submenu"

ngbDropdown

#dropdownSubmenu="ngbDropdown"

placement="right-top"

[autoClose]="true"

\*abpPermission="'AbpIdentity.Roles || AbpIdentity.Users'"

>

<div ngbDropdownToggle [class.dropdown-toggle]="false">

<a

abpEllipsis="210px"

[abpEllipsisEnabled]="!smallScreen"

role="button"

class="btn d-block text-left dropdown-toggle"

>

<i class="fa fa-id-card-o"></i>

{%{{{ 'AbpIdentity::Menu:IdentityManagement' | abpLocalization }}}%}

</a>

</div>

<div

#childrenContainer

class="dropdown-menu border-0 shadow-sm"

[class.d-block]="smallScreen && dropdownSubmenu.isOpen()"

>

<div class="dropdown-submenu" \*abpPermission="'AbpIdentity.Roles'">

<a class="dropdown-item" routerLink="/identity/roles">

{%{{{ 'AbpIdentity::Roles' | abpLocalization }}}%}</a

>

</div>

<div class="dropdown-submenu" \*abpPermission="'AbpIdentity.Users'">

<a class="dropdown-item" routerLink="/identity/users">

{%{{{ 'AbpIdentity::Users' | abpLocalization }}}%}</a

>

</div>

</div>

</div>

<div

class="dropdown-submenu"

ngbDropdown

#dropdownSubmenu="ngbDropdown"

placement="right-top"

[autoClose]="true"

\*abpPermission="'AbpTenantManagement.Tenants'"

>

<div ngbDropdownToggle [class.dropdown-toggle]="false">

<a

abpEllipsis="210px"

[abpEllipsisEnabled]="!smallScreen"

role="button"

class="btn d-block text-left dropdown-toggle"

>

<i class="fa fa-users"></i>

{%{{{ 'AbpTenantManagement::Menu:TenantManagement' | abpLocalization }}}%}

</a>

</div>

<div

#childrenContainer

class="dropdown-menu border-0 shadow-sm"

[class.d-block]="smallScreen && dropdownSubmenu.isOpen()"

>

<div class="dropdown-submenu" \*abpPermission="'AbpTenantManagement.Tenants'">

<a class="dropdown-item" routerLink="/tenant-management/tenants">

{%{{{ 'AbpTenantManagement::Tenants' | abpLocalization }}}%}</a

>

</div>

</div>

</div>

</div>

</li>

</ul>

```

打开 `src/app` 目录下的 `app.component.ts` 做以下修改:

```js

import { ..., AddReplaceableComponent } from '@abp/ng.core'; // imported AddReplaceableComponent

import { Store } from '@ngxs/store'; // imported Store

import { RoutesComponent } from './routes/routes.component'; // imported NavItemsComponent

import { eThemeBasicComponents } from '@abp/ng.theme.basic'; // imported eThemeBasicComponents

//...

@Component(/\* component metadata \*/)

export class AppComponent implements OnInit {

constructor(..., private store: Store) {} // injected Store

ngOnInit() {

//...

// added dispatch

this.store.dispatch(

new AddReplaceableComponent({

component: RoutesComponent,

key: eThemeBasicComponents.Routes,

}),

);

}

}

```

最终UI如下:

![New routes](./images/replaced-routes-component.png)

#### 如何替换NavItemsComponent

![NavItemsComponent](./images/nav-items-component.png)

在 `angular` 目录下运行以下命令创建新的组件 `NavItemsComponent`:

```bash

yarn ng generate component nav-items --entryComponent

# You don't need the --entryComponent option in Angular 9

```

打开 `src/app/nav-items` 目录下生成的 `nav-items.component.ts` 并使用以下内容替换它:

```js

import {

ApplicationConfiguration,

AuthService,

ConfigState,

SessionState,

SetLanguage,

} from '@abp/ng.core';

import { Component, AfterViewInit } from '@angular/core';

import { Navigate, RouterState } from '@ngxs/router-plugin';

import { Select, Store } from '@ngxs/store';

import { Observable, fromEvent } from 'rxjs';

import { map, debounceTime } from 'rxjs/operators';

import snq from 'snq';

@Component({

selector: 'app-nav-items',

templateUrl: 'nav-items.component.html',

})

export class NavItemsComponent implements AfterViewInit {

@Select(ConfigState.getOne('currentUser'))

currentUser$: Observable<ApplicationConfiguration.CurrentUser>;

@Select(ConfigState.getDeep('localization.languages'))

languages$: Observable<ApplicationConfiguration.Language[]>;

smallScreen = window.innerWidth < 992;

get defaultLanguage$(): Observable<string> {

return this.languages$.pipe(

map(

languages =>

snq(

() => languages.find(lang => lang.cultureName === this.selectedLangCulture).displayName,

),

'',

),

);

}

get dropdownLanguages$(): Observable<ApplicationConfiguration.Language[]> {

return this.languages$.pipe(

map(

languages =>

snq(() => languages.filter(lang => lang.cultureName !== this.selectedLangCulture)),

[],

),

);

}

get selectedLangCulture(): string {

return this.store.selectSnapshot(SessionState.getLanguage);

}

constructor(private store: Store, private authService: AuthService) {}

ngAfterViewInit() {

fromEvent(window, 'resize')

.pipe(debounceTime(150))

.subscribe(() => {

this.smallScreen = window.innerWidth < 992;

});

}

onChangeLang(cultureName: string) {

this.store.dispatch(new SetLanguage(cultureName));

}

logout() {

this.authService.logout().subscribe(() => {

this.store.dispatch(

new Navigate(['/'], null, {

state: { redirectUrl: this.store.selectSnapshot(RouterState).state.url },

}),

);

});

}

}

```

打开 `src/app/nav-items` 目录下生成的 `nav-items.component.html` 并使用以下内容替换它:

```html

<ul class="navbar-nav">

<input type="search" placeholder="Search" class="bg-transparent border-0 text-white" />

<li \*ngIf="(dropdownLanguages$ | async)?.length > 0" class="nav-item">

<div class="dropdown" ngbDropdown #languageDropdown="ngbDropdown" display="static">

<a

ngbDropdownToggle

class="nav-link"

href="javascript:void(0)"

role="button"

id="dropdownMenuLink"

data-toggle="dropdown"

aria-haspopup="true"

aria-expanded="false"

>

{%{{{ defaultLanguage$ | async }}}%}

</a>

<div

class="dropdown-menu dropdown-menu-right border-0 shadow-sm"

aria-labelledby="dropdownMenuLink"

[class.d-block]="smallScreen && languageDropdown.isOpen()"

>

<a

\*ngFor="let lang of dropdownLanguages$ | async"

href="javascript:void(0)"

class="dropdown-item"

(click)="onChangeLang(lang.cultureName)"

>{%{{{ lang?.displayName }}}%}</a

>

</div>

</div>

</li>

<li class="nav-item">

<ng-template #loginBtn>

<a role="button" class="nav-link" routerLink="/account/login">{%{{{

'AbpAccount::Login' | abpLocalization

}}}%}</a>

</ng-template>

<div

\*ngIf="(currentUser$ | async)?.isAuthenticated; else loginBtn"

ngbDropdown

class="dropdown"

#currentUserDropdown="ngbDropdown"

display="static"

>

<a

ngbDropdownToggle

class="nav-link"

href="javascript:void(0)"

role="button"

id="dropdownMenuLink"

data-toggle="dropdown"

aria-haspopup="true"

aria-expanded="false"

>

{%{{{ (currentUser$ | async)?.userName }}}%}

</a>

<div

class="dropdown-menu dropdown-menu-end border-0 shadow-sm"

aria-labelledby="dropdownMenuLink"

[class.d-block]="smallScreen && currentUserDropdown.isOpen()"

>

<a class="dropdown-item" routerLink="/account/manage-profile"

><i class="fa fa-cog mr-1"></i>{%{{{ 'AbpAccount::MyAccount' | abpLocalization }}}%}</a

>

<a class="dropdown-item" href="javascript:void(0)" (click)="logout()"

><i class="fa fa-power-off mr-1"></i>{%{{{ 'AbpUi::Logout' | abpLocalization }}}%}</a

>

</div>

</div>

</li>

</ul>

```

打开 `src/app` 目录下的 `app.component.ts` 做以下修改:

```js

import { ..., AddReplaceableComponent } from '@abp/ng.core'; // imported AddReplaceableComponent

import { Store } from '@ngxs/store'; // imported Store

import { NavItemsComponent } from './nav-items/nav-items.component'; // imported NavItemsComponent

import { eThemeBasicComponents } from '@abp/ng.theme.basic'; // imported eThemeBasicComponents

//...

@Component(/\* component metadata \*/)

export class AppComponent implements OnInit {

constructor(..., private store: Store) {} // injected Store

ngOnInit() {

//...

// added dispatch

this.store.dispatch(

new AddReplaceableComponent({

component: NavItemsComponent,

key: eThemeBasicComponents.NavItems,

}),

);

}

}

```

最终UI如下:

![New nav-items](./images/replaced-nav-items-component.png)

## 另请参阅

- [如何替换PermissionManagementComponent](./Permission-Management-Component-Replacement.md)

## 下一步是什么?

- [自定义设置页面](./Custom-Setting-Page.md)

懒加载Scripts与Styles

# 如何懒加载 Scripts 与 Styles

你可以使用@abp/ng.core包中的 `LazyLoadService` 以简单明了的方式延迟加载脚本和样式.

## 入门

你不必在模块或组件/指令级别提供 `LazyLoadService`,因为它已经在**\*\*根中\*\***中提供了. 你可以在组件,指令或服务中注入并使用它.

```js

import { LazyLoadService } from '@abp/ng.core';

@Component({

/\* class metadata here \*/

})

class DemoComponent {

constructor(private lazyLoadService: LazyLoadService) {}

}

```

## 用法

你可以使用 `LazyLoadService` 的 `load` 方法在DOM中的所需位置创建 `<script>` 或 `<link>` 元素并强制浏览器下载目标资源.

### 如何加载 Scripts

`load` 方法的第一个参数需要一个 `LoadingStrategy`. 如果传递 `ScriptLoadingStrategy` 实例,`LazyLoadService` 将使用给定的 `src` 创建一个 `<script>` 元素并放置在指定的DOM位置.

```js

import { LazyLoadService, LOADING\_STRATEGY } from '@abp/ng.core';

@Component({

template: `

<some-component \*ngIf="libraryLoaded$ | async"></some-component>

`

})

class DemoComponent {

libraryLoaded$ = this.lazyLoad.load(

LOADING\_STRATEGY.AppendAnonymousScriptToHead('/assets/some-library.js'),

);

constructor(private lazyLoadService: LazyLoadService) {}

}

```

`load` 方法返回一个 `observable`,你可以在组件中或通过 `AsyncPipe` 订阅它. 在上面的示例中**\*\*仅当脚本成功加载或之前已经加载脚本时\*\***, `NgIf` 指令才会呈现 `<some-component>`.

> 你可以使用 `async` 管道在模板中多次订阅,脚本将仅加载一次

请参阅[LoadingStrategy](./Loading-Strategy.md)查看所有可用的加载策略以及如何构建自己的加载策略.

### 如何加载 Styles

如果传递给 `load` 方法第一个参数为 `StyleLoadingStrategy` 实例,`LazyLoadService` 将使用给定的 `href` 创建一个 `<link>` 元素并放置在指定的DOM位置.

```js

import { LazyLoadService, LOADING\_STRATEGY } from '@abp/ng.core';

@Component({

template: `

<some-component \*ngIf="stylesLoaded$ | async"></some-component>

`

})

class DemoComponent {

stylesLoaded$ = this.lazyLoad.load(

LOADING\_STRATEGY.AppendAnonymousStyleToHead('/assets/some-styles.css'),

);

constructor(private lazyLoadService: LazyLoadService) {}

}

```

`load` 方法返回一个 `observable`,你可以在组件中或通过 `AsyncPipe` 订阅它. 在上面的示例中**\*\*仅当样式成功加载或之前已经加载样式时\*\***, `NgIf` 指令才会呈现 `<some-component>`.

> 你可以使用 `async` 管道在模板中多次订阅,样式将仅加载一次

请参阅[LoadingStrategy](./Loading-Strategy.md)查看所有可用的加载策略以及如何构建自己的加载策略.

### 高级用法

你有**\*\*很大的自由度来定义延迟加载的工作方式\*\***. 示例:

```js

const domStrategy = DOM\_STRATEGY.PrependToHead();

const crossOriginStrategy = CROSS\_ORIGIN\_STRATEGY.Anonymous(

'sha384-Vkoo8x4CGsO3+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9MuhOf23Q9Ifjh',

);

const loadingStrategy = new StyleLoadingStrategy(

'https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css',

domStrategy,

crossOriginStrategy,

);

this.lazyLoad.load(loadingStrategy, 1, 2000);

```

此代码将创建具有给定URL和完整性哈希的 `<link>` 元素,将其插入到 `<head>` 元素的顶部,如果第一次尝试失败,则在2秒后重试一次.

一个常见的用例是在**\*\*使用功能之前加载多个脚本/样式\*\***:

```js

import { LazyLoadService, LOADING\_STRATEGY } from '@abp/ng.core';

import { frokJoin } from 'rxjs';

@Component({

template: `

<some-component \*ngIf="scriptsAndStylesLoaded$ | async"></some-component>

`

})

class DemoComponent {

private stylesLoaded$ = forkJoin(

this.lazyLoad.load(

LOADING\_STRATEGY.PrependAnonymousStyleToHead('/assets/library-dark-theme.css'),

),

this.lazyLoad.load(

LOADING\_STRATEGY.PrependAnonymousStyleToHead('/assets/library.css'),

),

);

private scriptsLoaded$ = forkJoin(

this.lazyLoad.load(

LOADING\_STRATEGY.AppendAnonymousScriptToHead('/assets/library.js'),

),

this.lazyLoad.load(

LOADING\_STRATEGY.AppendAnonymousScriptToHead('/assets/other-library.css'),

),

);

scriptsAndStylesLoaded$ = forkJoin(this.scriptsLoaded$, this.stylesLoaded$);

constructor(private lazyLoadService: LazyLoadService) {}

}

```

RxJS `forkJoin` 并行加载所有脚本和样式,并仅在加载所有脚本和样式时才放行. 因此放置 `<some-component>` 时,所有必需的依赖项都可用的.

> 注意到我们在文档头上添加了样式吗? 有时这是必需的因为你的应用程序样式可能会覆盖某些库样式. 在这种情况下你必须注意前置样式的顺序. 它们将一一放置,**\*\*并且在放置前,最后一个放置在最上面\*\***.

另一个常见的用例是**\*\*按顺序加载依赖脚本\*\***:

```js

import { LazyLoadService, LOADING\_STRATEGY } from '@abp/ng.core';

import { concat } from 'rxjs';

@Component({

template: `

<some-component \*ngIf="scriptsLoaded$ | async"></some-component>

`

})

class DemoComponent {

scriptsLoaded$ = concat(

this.lazyLoad.load(

LOADING\_STRATEGY.PrependAnonymousScriptToHead('/assets/library.js'),

),

this.lazyLoad.load(

LOADING\_STRATEGY.AppendAnonymousScriptToHead('/assets/script-that-requires-library.js'),

),

);

constructor(private lazyLoadService: LazyLoadService) {}

}

```

在此示例中,第二个文件需要预先加载第一个文件, RxJS `concat` 函数将允许你以给定的顺序一个接一个地加载所有脚本,并且仅在加载所有脚本时放行.

## API

### loaded

```js

loaded: Set<string>

```

所有以前加载的路径都可以通过此属性访问. 它是一个简单的[JavaScript集]

### load

```js

load(strategy: LoadingStrategy, retryTimes?: number, retryDelay?: number): Observable<Event>

```

- `strategy` 是主要参数,上面已经介绍过.

- `retryTimes` 定义加载失败前再次尝试多少次(默认值:2).

- `retryDelay` 定义重试之间的延迟(默认为1000).

## 下一步是什么?

- [DomInsertionService](./Dom-Insertion-Service.md)

DomInsertionService

# Dom插入(Scripts与Styles)

你可以使用@abp/ng.core包提供的 `DomInsertionService` 以简单的方式的插入脚本与样式.

## 入门

你不必在模块或组件级别提供 `DomInsertionService` ,因为它已经在**\*\*根中\*\***提供. 你可以在组件,指令或服务中直接注入并使用它.

```js

import { DomInsertionService } from '@abp/ng.core';

@Component({

/\* class metadata here \*/

})

class DemoComponent {

constructor(private domInsertionService: DomInsertionService) {}

}

```

## 用法

你可以使用 `DomInsertionService` 提供的 `insertContent` 方法去创建一个 `<script>` 或 `<style>` 元素到DOM的指定位置. 还有 `projectContent` 方法用于渲染组件和模板.

### 如何插入Scripts

`insertContent` 方法的第一个参数需要一个 `ContentStrategy`. 如果传递 `ScriptContentStrategy` 实例, `DomInsertionService` 将创建具有给定内容的 `<script>` 元素并放置在指定的DOM位置.

```js

import { DomInsertionService, CONTENT\_STRATEGY } from '@abp/ng.core';

@Component({

/\* class metadata here \*/

})

class DemoComponent {

constructor(private domInsertionService: DomInsertionService) {}

ngOnInit() {

const scriptElement = this.domInsertionService.insertContent(

CONTENT\_STRATEGY.AppendScriptToBody('alert()')

);

}

}

```

在上面的示例中,将 `<script>alert()</script>` 元素放置在 `<body>`的末尾, `scriptElement` 类型是一个 `HTMLScriptElement`.

请参考[ContentStrategy](./Content-Strategy.md)查看所有可用的内容策略以及如何构建自己的内容策略.

> 重要说明: `DomInsertionService` 不会两次插入相同的内容. 为了再次添加内容你首先应该使用 `removeContent` 方法删除旧内容.

### 如何插入Styles

`insertContent` 方法的第一个参数需要一个 `ContentStrategy`. 如果传递 `StyleContentStrategy` 实例, `DomInsertionService` 将创建具有给定内容的 `<style>` 元素并放置在指定的DOM位置.

```js

import { DomInsertionService, CONTENT\_STRATEGY } from '@abp/ng.core';

@Component({

/\* class metadata here \*/

})

class DemoComponent {

constructor(private domInsertionService: DomInsertionService) {}

ngOnInit() {

const styleElement = this.domInsertionService.insertContent(

CONTENT\_STRATEGY.AppendStyleToHead('body {margin: 0;}')

);

}

}

```

在上面的示例中,将 `<style>body {margin: 0;}</style>` 元素放置在 `<head>`的末尾, `styleElement` 类型是一个 `HTMLStyleElement`.

请参考[ContentStrategy](./Content-Strategy.md)查看所有可用的内容策略以及如何构建自己的内容策略.

.

> 重要说明: `DomInsertionService` 不会两次插入相同的内容. 为了再次添加内容你首先应该使用 `removeContent` 方法删除旧内容.

### 如何删除已插入的 Scripts & Styles

如果你传递 `HTMLScriptElement` 或 `HTMLStyleElement` 做为 `removeContent` 方法的第一个参数, `DomInsertionService` 将删除给定的元素.

```js

import { DomInsertionService, CONTENT\_STRATEGY } from '@abp/ng.core';

@Component({

/\* class metadata here \*/

})

class DemoComponent {

private styleElement: HTMLStyleElement;

constructor(private domInsertionService: DomInsertionService) {}

ngOnInit() {

this.styleElement = this.domInsertionService.insertContent(

CONTENT\_STRATEGY.AppendStyleToHead('body {margin: 0;}')

);

}

ngOnDestroy() {

this.domInsertionService.removeContent(this.styleElement);

}

}

```

在上面的示例中,销毁组件时,将从 `<head>` 中删除 `<style>body {margin: 0;}</style>` 元素.

## API

### insertContent

```js

insertContent<T extends HTMLScriptElement | HTMLStyleElement>(

contentStrategy: ContentStrategy<T>,

): T

```

- `contentStrategy` 是方法的重要参数,已经在上方进行说明.

- 根据给定的策略返回 `HTMLScriptElement` 或 `HTMLStyleElement`.

### removeContent

```js

removeContent(element: HTMLScriptElement | HTMLStyleElement): void

```

- `element` 参数是已插入的 `HTMLScriptElement` 或 `HTMLStyleElement` 元素,它们应由 `insertContent` 方法返回.

### has

```js

has(content: string): boolean

```

`has` 返回一个布尔值,用于表示给定的内容是否插入到DOM.

- `content` 参数是 `HTMLScriptElement` 或 `HTMLStyleElement` 元素的内容.

## 下一步是什么?

- [ContentProjectionService](./Content-Projection-Service.md)

ContentProjectionService

# 内容投影

你可以使用位于@abp/ng.core包中的 `ContentProjectionService` 简单明确的投影内容.

## 入门

你不必在模块或组件级别提供 `ContentProjectionService`,因为它已经在**\*\*根中提供了\*\***. 你可以在组件中注入并开始使用它. 为了获得更好的类型支持,你可以将迭代项目的类型传递给它.

```js

import { ContentProjectionService } from '@abp/ng.core';

@Component({

/\* class metadata here \*/

})

class DemoComponent {

constructor(private contentProjectionService: ContentProjectionService) {}

}

```

## 用法

你可以使用 `ContentProjectionService` 的 `projectContent` 方法在你的项目中动态的渲染组件和模板.

### 如何将组件投影到根级别

如果将 `RootComponentProjectionStrategy` 做为 `projectContent` 方法的第一个参数,那么 `ContentProjectionService` 会解析投影组件并放在根级别,它还为组件传递上下文.

```js

const strategy = PROJECTION\_STRATEGY.AppendComponentToBody(

SomeOverlayComponent,

{ someOverlayProp: "SOME\_VALUE" }

);

const componentRef = this.contentProjectionService.projectContent(strategy);

```

在上面的示例中, `SomeOverlayComponent` 组件放置在 `<body>` 的**\*\*末尾\*\***并返回 `ComponentRef`. 另外将应用给定的上下文,因此组件的 `someOverlayProp` 被设置为 `SOME\_VALUE`.

> 你应该总是返回 `ComponentRef` 实例,因为它是对投影组件的引用,在你需要时使用该引用销毁投影视图和组件实例.

### 如何将组件和模板投影到容器中

如果将 `ComponentProjectionStrategy` 或 `TemplateProjectionStrategy` 做为 `projectContent` 方法的第一个参数,并且传递 `ViewContainerRef` 做为策略的第二个参数传递. 那么 `ContentProjectionService` 把组件或模板投影到给定的容器中,它还为组件或模板传递上下文.

```js

const strategy = PROJECTION\_STRATEGY.ProjectComponentToContainer(

SomeComponent,

viewContainerRefOfTarget,

{ someProp: "SOME\_VALUE" }

);

const componentRef = this.contentProjectionService.projectContent(strategy);

```

在上面的示例中,`viewContainerRefOfTarget`(它是一个`ViewContainerRef` 实例)将被清除,并把 `SomeComponent` 组件放在其中. 另外将应用给定的上下文,因此组件的 `someProp` 被设置为 `SOME\_VALUE`.

> 你应该总是返回 `ComponentRef` 或 `EmbeddedViewRef` ,因为它是对投影内容的引用,在你需要时使用该引用销毁它们.

请参考[ProjectionStrategy](./Projection-Strategy.md)查看所有可用的投影策略以及如何构建自己的投影策略.

## API

### projectContent

```js

projectContent<T extends Type<any> | TemplateRef<any>>(

projectionStrategy: ProjectionStrategy<T>,

injector = this.injector,

): ComponentRef<C> | EmbeddedViewRef<C>

```

- `projectionStrategy` 参数是此处的要点,在上面进行了说明.

- `injector` 参数是 `Injector` 实例,你可以传递到投影内容. 在 `TemplateProjectionStrategy` 并没有使用到它.

## 下一步是什么?

- [TrackByService](./Track-By-Service.md)

TrackByService

# 轻松实现TrackByFunction

`TrackByService` 是一个实用服务,为Angular模板中最常见的需求之一: `TrackByFunction` 提供简单的实现. 在继续下面的内容之前,请参先阅 [Angular 文档](https://angular.io/guide/template-syntax#ngfor-with-trackby).

## 入门

你不必在模块或组件级别提供 `TrackByService`,因为它已经在**\*\*根中提供了\*\***. 你可以在组件中注入并开始使用它. 为了获得更好的类型支持,你可以将迭代项目的类型传递给它.

```js

import { TrackByService } from '@abp/ng.core';

@Component({

/\* class metadata here \*/

})

class DemoComponent {

list: Item[];

constructor(public readonly track: TrackByService<Item>) {}

}

```

> 注意到 `track` 是 `public` 并且 `readonly` 了吗? 因为我们将看到一些在组件模板中直接使用 `TrackByService` 实例的方法示例. 可以把它看做反模式,但它有自身的优势,尤其是在利用组件继承时. 你始终可以使用公共组件属性.

**\*\*成员也被导出做为独立的函数.\*\*** 如果你不想注入 `TrackByService`, 你可以直接在类中导入并使用这些函数.

## 用法

有两种方法可用.

1. 你可以直接注入 `TrackByService` 到你的组件并且使用它的成员.

2. 你可以在直接在组件属性上使用导出的函数.

### 如何通过一个键跟踪项

你可以使用 `by` 获取一个 `TrackByFunction` , 该函数根据它的一个键来跟踪迭代的对象. 你可以将迭代类型传递给它获得类型支持.

```html

<!-- template of DemoComponent -->

<div \*ngFor="let item of list; trackBy: track.by('id')">{%{{{ item.name }}}%}</div>

```

`by` 作为一个独立函数导出,命名为 `trackBy`.

```js

import { trackBy } from "@abp/ng.core";

@Component({

template: `

<div

\*ngFor="let item of list; trackBy: trackById"

>

{%{{{ item.name }}}%}

</div>

`,

})

class DemoComponent {

list: Item[];

trackById = trackBy<Item>('id');

}

```

### 如何通过深度嵌套的键进行跟踪

你可以使用 `byDeep` 获取一个 `TrackByFunction` , 它根据深度嵌套的键跟踪迭代对象. 你可以将迭代类型传递给它获得类型支持.

```html

<!-- template of DemoComponent -->

<div

\*ngFor="let item of list; trackBy: track.byDeep('tenant', 'account', 'id')"

>

{%{{{ item.tenant.name }}}%}

</div>

```

`byDeep` 作为一个独立函数导出,命名为 `trackByDeep`.

```js

import { trackByDeep } from "@abp/ng.core";

@Component({

template: `

<div

\*ngFor="let item of list; trackBy: trackByTenantAccountId"

>

{%{{{ item.name }}}%}

</div>

`,

})

class DemoComponent {

list: Item[];

trackByTenantAccountId = trackByDeep<Item>('tenant', 'account', 'id');

}

```

## 下一步是什么?

- [SubscriptionService](./Subscription-Service.md)

SubscriptionService

# 轻松取消将你的观察取消订阅

`SubscriptionService` 是一个实用工具服务,它提供了一个简单的取消订阅Angular组件和指令中的RxJS可观察对象的功能. 请参见[为什么在实例销毁时要取消订阅可观察对象](https://angular.io/guide/lifecycle-hooks#cleaning-up-on-instance-destruction).

## 入门

你必须在组件或指令级别提供 `SubscriptionService`,因为它没有在**\*\*根中提供\*\***,而且它与组件/指令的生命周期同步. 只有在此之后，您才能注入并开始使用它。

```js

import { SubscriptionService } from '@abp/ng.core';

@Component({

/\* class metadata here \*/

providers: [SubscriptionService],

})

class DemoComponent {

count$ = interval(1000);

constructor(private subscription: SubscriptionService) {

this.subscription.addOne(this.count$, console.log);

}

}

```

由 `count$` 发出的值将被记录下来,直到组件被销毁. 你不必手动退订.

> 请不要尝试使用单例 `SubscriptionService`. 这是行不通的.

## 用法

## 如何订阅可观察对象

你可以传递 `next` 函数和 `error` 函数.

```js

@Component({

/\* class metadata here \*/

providers: [SubscriptionService],

})

class DemoComponent implements OnInit {

constructor(private subscription: SubscriptionService) {}

ngOnInit() {

const source$ = interval(1000);

const nextFn = value => console.log(value \* 2);

const errorFn = error => {

console.error(error);

return of(null);

};

this.subscription.addOne(source$, nextFn, errorFn);

}

}

```

或者,你可以传递一个观察者.

```js

@Component({

/\* class metadata here \*/

providers: [SubscriptionService],

})

class DemoComponent implements OnInit {

constructor(private subscription: SubscriptionService) {}

ngOnInit() {

const source$ = interval(1000);

const observer = {

next: value => console.log(value \* 2),

complete: () => console.log('DONE'),

};

this.subscription.addOne(source$, observer);

}

}

```

`addOne` 方法返回单个订阅以便你以后使用它. 有关详细信息,请参见下面的主题.

### 实例销毁之前如何退订

有两种方法可以做到这一点. 如果你不想再次订阅.可以使用 `closeAll` 方法.

```js

@Component({

/\* class metadata here \*/

providers: [SubscriptionService],

})

class DemoComponent implements OnInit {

constructor(private subscription: SubscriptionService) {}

ngOnInit() {

this.subscription.addOne(interval(1000), console.log);

}

onSomeEvent() {

this.subscription.closeAll();

}

}

```

这将清除所有订阅,你将无法再次订阅. 如果你打算添加另一个订阅,可以使用`reset`方法.

```js

@Component({

/\* class metadata here \*/

providers: [SubscriptionService],

})

class DemoComponent implements OnInit {

constructor(private subscription: SubscriptionService) {}

ngOnInit() {

this.subscription.addOne(interval(1000), console.log);

}

onSomeEvent() {

this.subscription.reset();

this.subscription.addOne(interval(1000), console.warn);

}

}

```

### 如何取消单个订阅

有时你可能需要取消订阅特定的订阅,但保留其他订阅. 在这种情况下,你可以使用 `closeOne` 方法.

```js

@Component({

/\* class metadata here \*/

providers: [SubscriptionService],

})

class DemoComponent implements OnInit {

countSubscription: Subscription;

constructor(private subscription: SubscriptionService) {}

ngOnInit() {

this.countSubscription = this.subscription.addOne(

interval(1000),

console.log

);

}

onSomeEvent() {

this.subscription.closeOne(this.countSubscription);

console.log(this.countSubscription.closed); // true

}

}

```

### 如何从跟踪的订阅中删除单个订阅

你可能需要控制特定的订阅. 在这种情况下你可以使用 `removeOne` 方法将其从跟踪的订阅中删除.

```js

@Component({

/\* class metadata here \*/

providers: [SubscriptionService],

})

class DemoComponent implements OnInit {

countSubscription: Subscription;

constructor(private subscription: SubscriptionService) {}

ngOnInit() {

this.countSubscription = this.subscription.addOne(

interval(1000),

console.log

);

}

onSomeEvent() {

this.subscription.removeOne(this.countSubscription);

console.log(this.countSubscription.closed); // false

}

}

```

### 如何检查是否全部取消订阅

使用 `isClosed` 检查 `closeAll` 是否被调用.

```js

@Component({

/\* class metadata here \*/

providers: [SubscriptionService],

})

class DemoComponent implements OnInit {

constructor(private subscription: SubscriptionService) {}

ngOnInit() {

this.subscription.addOne(interval(1000), console.log);

}

onSomeEvent() {

console.log(this.subscription.isClosed); // false

}

}

```

## 下一步是什么?

- [ListService](./List-Service.md)

ListService

# 使用ListService轻松查询列表

`ListService` 是一种实用程序服务,提供简单的分页,排序和搜索实现.

## 入门

`ListService` **\*\*没有在根提供\*\***. 原因是通过这种方式它会清除组件上的所有订阅. 你可以使用可选的 `LIST\_QUERY\_DEBOUNCE\_TIME` 令牌调整debounce行为.

```js

import { ListService } from '@abp/ng.core';

import { BookDto } from '../models';

import { BookService } from '../services';

@Component({

/\* class metadata here \*/

providers: [

// [Required]

ListService,

// [Optional]

// Provide this token if you want a different debounce time.

// Default is 300. Cannot be 0. Any value below 100 is not recommended.

{ provide: LIST\_QUERY\_DEBOUNCE\_TIME, useValue: 500 },

],

template: `

`,

})

class BookComponent {

items: BookDto[] = [];

count = 0;

constructor(

public readonly list: ListService,

private bookService: BookService,

) {}

ngOnInit() {

// A function that gets query and returns an observable

const bookStreamCreator = query => this.bookService.getList(query);

this.list.hookToQuery(bookStreamCreator).subscribe(

response => {

this.items = response.items;

this.count = response.count;

// If you use OnPush change detection strategy,

// call detectChanges method of ChangeDetectorRef here.

}

); // Subscription is auto-cleared on destroy.

}

}

```

> 注意 `list` 是 `public` 并且 `readonly`. 因为我们将直接在组件的模板中使用 `ListService` 属性. 可以视为反模式,但是实现起来要快得多. 你可以改为使用公共组件属性.

像这样绑定 `ListService` 到 ngx-datatable:

```html

<ngx-datatable

[rows]="items"

[count]="count"

[list]="list"

default

>

<!-- column templates here -->

</ngx-datatable>

```

## 与Observables一起使用

你可以将Observables与Angular的[AsyncPipe](https://angular.io/guide/observables-in-angular#async-pipe)结合使用:

```js

book$ = this.list.hookToQuery(query => this.bookService.getListByInput(query));

```

```html

<!-- simplified representation of the template -->

<ngx-datatable

[rows]="(book$ | async)?.items || []"

[count]="(book$ | async)?.totalCount || 0"

[list]="list"

default

>

<!-- column templates here -->

</ngx-datatable>

<!-- DO NOT WORRY, ONLY ONE REQUEST WILL BE MADE -->

```

...or...

```js

@Select(BookState.getBooks)

books$: Observable<BookDto[]>;

@Select(BookState.getBookCount)

bookCount$: Observable<number>;

ngOnInit() {

this.list.hookToQuery((query) => this.store.dispatch(new GetBooks(query))).subscribe();

}

```

```html

<!-- simplified representation of the template -->

<ngx-datatable

[rows]="(books$ | async) || []"

[count]="(bookCount$ | async) || 0"

[list]="list"

default

>

<!-- column templates here -->

</ngx-datatable>

```

> 我们不建议将NGXS存储用于CRUD页面,除非你的应用程序需要在组件之间共享列表信息或稍后在另一页面中使用它.

## 如何在创建/更新/删除时刷新表

`ListService` 公开了一个 `get` 方法来触发当前查询的请求. 因此基本上每当创建,更新或删除操作解析时,你可以调用 `this.list.get();` 它会调用钩子流创建者.

```ts

this.store.dispatch(new DeleteBook(id)).subscribe(this.list.get);

```

...or...

```ts

this.bookService.createByInput(form.value)

.subscribe(() => {

this.list.get();

// Other subscription logic here

})

```

## 如何在表中实现服务器端搜索

`ListService` 公开一个 `filter` 属性,该属性将使用当前查询和给定的搜索字符串触发一个请求. 你需要做的就是通过双向绑定将其绑定到输入元素.

```html

<!-- simplified representation -->

<input type="text" name="search" [(ngModel)]="list.filter">

```

## ABP v3.0的重大更改

我们必须修改 `ListService` 使其与 `ngx-datatable` 一起使用. 之前 `page` 属性的最小值为 `1`, 你可以像这样使用它:

```html

<!-- other bindings are hidden in favor of brevity -->

<abp-table

[(page)]="list.page"

></abp-table>

```

从v3.0开始, 对于`ngx-datatable`, 初始页面的 `page`属性必须设置为 `0`. 因此如果你以前在表上使用过 `ListService` 并打算保留 `abp-table`,则需要进行以下更改:

```html

<!-- other bindings are hidden in favor of brevity -->

<abp-table

[page]="list.page + 1"

(pageChange)="list.page = $event - 1"

></abp-table>

```

**\*\*重要提示:\*\*** `abp-table` 没有被删除,但是会被弃用,并在将来的版本中移除,请考虑切换到 ngx-datatable.

## React Native

入门

# React Native入门

ABP平台提供了[React Native](https://reactnative.dev/)模板用于开发移动应用程序.

当你按照[入门文档](Getting-Started.md)中所述**\*\*创建新应用程序\*\***时, 你应该使用`-m react-native`选项以在解决方案中包含`react-native`项目.

## 配置你的本地IP地址

运行在Android模拟器或真机上的React Native应用程序无法连接到 `localhost` 上的后.要修复此问题,需要在本地IP上运行后端.

{{ if Tiered == "No"}}

![React Native host project local IP entry](images/rn-host-local-ip.png)

\* 打开 `.HttpApi.Host` 文件夹下的 `appsettings.json` 文件. 将 `SelfUrl` 和 `Authority` 属性的 `localhost` 替换为你本地的IP地址.

\* 打开 `.HttpApi.Host/Properties` 文件夹下的 `launchSettings.json` 文件. 将 `applicationUrl` 属性的 `localhost` 替换为你本地的IP地址.

{{ else if Tiered == "Yes" }}

![React Native tiered project local IP entry](images/rn-tiered-local-ip.png)

\* 打开 `.AuthServer` 文件夹下的 `appsettings.json` 文件. 将 `SelfUrl` 属性的 `localhost` 替换为你本地的IP地址.

\* 打开 `.AuthServer/Properties` 文件夹下的 `launchSettings.json` 文件. 将 `applicationUrl` 属性的 `localhost` 替换为你本地的IP地址.

\* 打开 `.HttpApi.Host` 文件夹下的 `appsettings.json` 文件. 将 `Authority` 属性的 `localhost` 替换为你本地的IP地址.

\* 打开 `.HttpApi.Host/Properties` 文件夹下的 `launchSettings.json` 文件. 将 `applicationUrl` 属性的 `localhost` 替换为你本地的IP地址.

{{ end }}

按照\*\*运行HTTP API Host (服务端口)\*\*那样运行后端.

> React Native应用程序不信任自动生成的.NET HTTPS证书,你可以在开发期间使用HTTP.

在 `react-native` 文件夹打开命令行终端,输入 `yarn` 命令(我们推荐使用[yarn](https://yarnpkg.com/)包管理, `npm install` 在大多数情况下也可以工作).

```bash

yarn

```

\* 打开 `react-nativer` 文件夹下的 `Environment.js` 文件. 将 `apiUrl` 和 `issuer` 属性的 `localhost` 替换为你本地的IP地址:

![react native environment local IP](images/rn-environment-local-ip.png)

{{ if Tiered == "Yes" }}

> 确保 `issuer` 与正在运行的 `.AuthServer` 项目匹配, `apiUrl` 与正在运行的 `.HttpApi.Host` 项目匹配.

{{else}}

> 确保 `issuer` 和 `apiUrl` 与正在运行的 `.HttpApi.Host` 项目匹配

{{ end }}

等到所有node模块加载成功, 执行 `yarn start` (或 `npm start`) 命令:

```bash

yarn start

```

等待Expo CLI启动后Expo CLI在 `http://localhost:19002/` 地址要开管理页面.

![expo-interface](images/rn-expo-interface.png)

在上面的管理界面中,可以通过使用[Expo Client](https://expo.io/tools#client)扫描二维码,使用Android模拟器,iOS模拟器或真机来启动应用程序.

> 请参阅expo.io上的[Android Studio模拟器](https://docs.expo.io/workflow/android-simulator/)和[iOS模拟器文档](https://docs.expo.io/workflow/ios-simulator/).

![React Native login screen on iPhone 11](images/rn-login-iphone.png)

输入用户名 **\*\*admin\*\***,密码 **\*\*1q2w3E\*\***\* 登录到应用程序.

应用程序已经启动并执行,你可以基于该启动模板开发应用程序.

## 通用

Utilities

#### 链表 (双向)

# 链表 (双向)

@abp/utils包提供了称为[双链表](https://en.wikipedia.org/wiki/Doubly\_linked\_list)的实用数据结构. 简而言之双向链表是一系列记录(又称节点),这些记录具有上一个节点,下一个节点及其自身值(或数据)的信息.

## 入门

要创建一个双向链表,你需要做的就是导入和创建它的一个新的实例:

```js

import { LinkedList } from '@abp/utils';

var list = new LinkedList();

```

MVC:

```js

var list = new abp.utils.common.LinkedList();

```

构造函数没有任何参数.

## 用法

### 如何添加新节点

有几种方法可以在链表中创建新节点,这些方法都可以单独使用,也可以通过 `add` 和 `addMany` 方法.

#### addHead(value)

```js

addHead(value: T): ListNode<T>

```

将给定值添加到链表的第一个节点:

```js

list.addHead('a');

// "a"

list.addHead('b');

// "b" <-> "a"

list.addHead('c');

// "c" <-> "b" <-> "a"

```

#### addManyHead(values)

```js

addManyHead(values: T[]): ListNode<T>[]

```

将给定的多个值添加到链表的第一个节点:

```js

list.addManyHead(['a', 'b', 'c']);

// "a" <-> "b" <-> "c"

list.addManyHead(['x', 'y', 'z']);

// "x" <-> "y" <-> "z" <-> "a" <-> "b" <-> "c"

```

#### addTail(value)

```js

addTail(value: T): ListNode<T>

```

将给定值添加到链表的最后一个节点:

```js

list.addTail('a');

// "a"

list.addTail('b');

// "a" <-> "b"

list.addTail('c');

// "a" <-> "b" <-> "c"

```

#### addManyTail(values)

```js

addManyTail(values: T[]): ListNode<T>[]

```

将给定多个值添加到链表的最后一个节点:

```js

list.addManyTail(['a', 'b', 'c']);

// "a" <-> "b" <-> "c"

list.addManyTail(['x', 'y', 'z']);

// "a" <-> "b" <-> "c" <-> "x" <-> "y" <-> "z"

```

#### addAfter(value, previousValue [, compareFn])

```js

addAfter(value: T, previousValue: T, compareFn?: ListComparisonFn<T>): ListNode<T>

```

添加给定值到previousValue节点后:

```js

list.addTail('a');

list.addTail('b');

list.addTail('b');

list.addTail('c');

// "a" <-> "b" <-> "b" <-> "c"

list.addAfter('x', 'b');

// "a" <-> "b" <-> "x" <-> "b" <-> "c"

```

你可以自定义比较器:

```js

list.addTail({ x: 1 });

list.addTail({ x: 2 });

list.addTail({ x: 3 });

// {"x":1} <-> {"x":2} <-> {"x":3}

list.addAfter(

{ x: 0 },

2,

(value, searchedValue) => value.x === searchedValue

);

// {"x":1} <-> {"x":2} <-> {"x":0} <-> {"x":3}

```

> 默认的比较函数检查深度相等性,因此你几乎不需要传递该参数.

#### addManyAfter(values, previousValue [, compareFn])

```js

addManyAfter(values: T[], previousValue: T, compareFn?: ListComparisonFn<T>): ListNode<T>[]

```

添加给定的多个值到previousValue节点后:

```js

list.addManyTail(['a', 'b', 'b', 'c']);

// "a" <-> "b" <-> "b" <-> "c"

list.addManyAfter(['x', 'y'], 'b');

// "a" <-> "b" <-> "x" <-> "y" <-> "b" <-> "c"

```

你可以自定义比较器:

```js

list.addManyTail([{ x: 1 },{ x: 2 },{ x: 3 }]);

// {"x":1} <-> {"x":2} <-> {"x":3}

list.addManyAfter(

[{ x: 4 }, { x: 5 }],

2,

(value, searchedValue) => value.x === searchedValue

);

// {"x":1} <-> {"x":2} <-> {"x":4} <-> {"x":5} <-> {"x":3}

```

> 默认的比较函数检查深度相等性,因此你几乎不需要传递该参数.

#### addBefore(value, nextValue [, compareFn])

```js

addBefore(value: T, nextValue: T, compareFn?: ListComparisonFn<T>): ListNode<T>

```

添加给值到previousValue节点前:

```js

list.addTail('a');

list.addTail('b');

list.addTail('b');

list.addTail('c');

// "a" <-> "b" <-> "b" <-> "c"

list.addBefore('x', 'b');

// "a" <-> "x" <-> "b" <-> "b" <-> "c"

```

你可以自定义比较器:

```js

list.addTail({ x: 1 });

list.addTail({ x: 2 });

list.addTail({ x: 3 });

// {"x":1} <-> {"x":2} <-> {"x":3}

list.addBefore(

{ x: 0 },

2,

(value, searchedValue) => value.x === searchedValue

);

// {"x":1} <-> {"x":0} <-> {"x":2} <-> {"x":3}

```

> 默认的比较函数检查深度相等性,因此你几乎不需要传递该参数.

#### addManyBefore(values, nextValue [, compareFn])

```js

addManyBefore(values: T[], nextValue: T, compareFn?: ListComparisonFn<T>): ListNode<T>[]

```

添加给定的多个值到previousValue节点前:

```js

list.addManyTail(['a', 'b', 'b', 'c']);

// "a" <-> "b" <-> "b" <-> "c"

list.addManyBefore(['x', 'y'], 'b');

// "a" <-> "x" <-> "y" <-> "b" <-> "b" <-> "c"

```

你可以自定义比较器

```js

list.addManyTail([{ x: 1 },{ x: 2 },{ x: 3 }]);

// {"x":1} <-> {"x":2} <-> {"x":3}

list.addManyBefore(

[{ x: 4 }, { x: 5 }],

2,

(value, searchedValue) => value.x === searchedValue

);

// {"x":1} <-> {"x":4} <-> {"x":5} <-> {"x":2} <-> {"x":3}

```

> 默认的比较函数检查深度相等性,因此你几乎不需要传递该参数.

#### addByIndex(value, position)

```js

addByIndex(value: T, position: number): ListNode<T>

```

在链表的指定位置添加节点:

```js

list.addTail('a');

list.addTail('b');

list.addTail('c');

// "a" <-> "b" <-> "c"

list.addByIndex('x', 2);

// "a" <-> "b" <-> "x" <-> "c"

```

它也适用于负索引:

```js

list.addTail('a');

list.addTail('b');

list.addTail('c');

// "a" <-> "b" <-> "c"

list.addByIndex('x', -1);

// "a" <-> "b" <-> "x" <-> "c"

```

#### addManyByIndex(values, position)

```js

addManyByIndex(values: T[], position: number): ListNode<T>[]

```

添加多个节点到链表的指定位置:

```js

list.addManyTail(['a', 'b', 'c']);

// "a" <-> "b" <-> "c"

list.addManyByIndex(['x', 'y'], 2);

// "a" <-> "b" <-> "x" <-> "y" <-> "c"

```

它也适用于负索引:

```js

list.addManyTail(['a', 'b', 'c']);

// "a" <-> "b" <-> "c"

list.addManyByIndex(['x', 'y'], -1);

// "a" <-> "b" <-> "x" <-> "y" <-> "c"

```

#### add(value).head()

```js

add(value: T).head(): ListNode<T>

```

将添加的节点移动到链表头:

```js

list.add('a').head();

// "a"

list.add('b').head();

// "b" <-> "a"

list.add('c').head();

// "c" <-> "b" <-> "a"

```

> 它是 `addHead` 的替代API.

#### add(value).tail()

```js

add(value: T).tail(): ListNode<T>

```

将添加的节点移动到链表尾:

```js

list.add('a').tail();

// "a"

list.add('b').tail();

// "a" <-> "b"

list.add('c').tail();

// "a" <-> "b" <-> "c"

```

> 它是 `addTail` 的替代API.

#### add(value).after(previousValue [, compareFn])

```js

add(value: T).after(previousValue: T, compareFn?: ListComparisonFn<T>): ListNode<T>

```

将添加的节点移动到指定节点后:

```js

list.add('a').tail();

list.add('b').tail();

list.add('b').tail();

list.add('c').tail();

// "a" <-> "b" <-> "b" <-> "c"

list.add('x').after('b');

// "a" <-> "b" <-> "x" <-> "b" <-> "c"

```

你可以自定义比较器

```js

list.add({ x: 1 }).tail();

list.add({ x: 2 }).tail();

list.add({ x: 3 }).tail();

// {"x":1} <-> {"x":2} <-> {"x":3}

list

.add({ x: 0 })

.after(2, (value, searchedValue) => value.x === searchedValue);

// {"x":1} <-> {"x":2} <-> {"x":0} <-> {"x":3}

```

> 它是 `addAfter` 的替代API.

>

> 默认的比较函数检查深度相等性,因此你几乎不需要传递该参数.

#### add(value).before(nextValue [, compareFn])

```js

add(value: T).before(nextValue: T, compareFn?: ListComparisonFn<T>): ListNode<T>

```

将添加的节点移动到指定节点前:

```js

list.add('a').tail();

list.add('b').tail();

list.add('b').tail();

list.add('c').tail();

// "a" <-> "b" <-> "b" <-> "c"

list.add('x').before('b');

// "a" <-> "x" <-> "b" <-> "b" <-> "c"

```

你可以自定义比较器

```js

list.add({ x: 1 }).tail();

list.add({ x: 2 }).tail();

list.add({ x: 3 }).tail();

// {"x":1} <-> {"x":2} <-> {"x":3}

list

.add({ x: 0 })

.before(2, (value, searchedValue) => value.x === searchedValue);

// {"x":1} <-> {"x":0} <-> {"x":2} <-> {"x":3}

```

> 它是 `addBefore` 的替代API.

>

> 默认的比较函数检查深度相等性,因此你几乎不需要传递该参数.

#### add(value).byIndex(position)

```js

add(value: T).byIndex(position: number): ListNode<T>

```

将添加的节点移动到链表指定位置:

```js

list.add('a').tail();

list.add('b').tail();

list.add('c').tail();

// "a" <-> "b" <-> "c"

list.add('x').byIndex(2);

// "a" <-> "b" <-> "x" <-> "c"

```

它也适用于负索引:

```js

list.add('a').tail();

list.add('b').tail();

list.add('c').tail();

// "a" <-> "b" <-> "c"

list.add('x').byIndex(-1);

// "a" <-> "b" <-> "x" <-> "c"

```

> 它是 `addByIndex` 的替代API.

#### addMany(values).head()

```js

addMany(values: T[]).head(): ListNode<T>[]

```

将添加的多个节点移动到链表头:

```js

list.addMany(['a', 'b', 'c']).head();

// "a" <-> "b" <-> "c"

list.addMany(['x', 'y', 'z']).head();

// "x" <-> "y" <-> "z" <-> "a" <-> "b" <-> "c"

```

> 它是 `addManyHead` 的替代API.

#### addMany(values).tail()

```js

addMany(values: T[]).tail(): ListNode<T>[]

```

将添加的多个节点移动到链表尾:

```js

list.addMany(['a', 'b', 'c']).tail();

// "a" <-> "b" <-> "c"

list.addMany(['x', 'y', 'z']).tail();

// "a" <-> "b" <-> "c" <-> "x" <-> "y" <-> "z"

```

> 它是 `addManyTail` 的替代API.

#### addMany(values).after(previousValue [, compareFn])

```js

addMany(values: T[]).after(previousValue: T, compareFn?: ListComparisonFn<T>): ListNode<T>[]

```

将添加的多个节点移动到指定节点后:

```js

list.addMany(['a', 'b', 'b', 'c']).tail();

// "a" <-> "b" <-> "b" <-> "c"

list.addMany(['x', 'y']).after('b');

// "a" <-> "b" <-> "x" <-> "y" <-> "b" <-> "c"

```

你可以自定义比较器

```js

list.addMany([{ x: 1 }, { x: 2 }, { x: 3 }]).tail();

// {"x":1} <-> {"x":2} <-> {"x":3}

list

.addMany([{ x: 4 }, { x: 5 }])

.after(2, (value, searchedValue) => value.x === searchedValue);

// {"x":1} <-> {"x":2} <-> {"x":4} <-> {"x":5} <-> {"x":3}

```

> 它是 `addManyAfter` 的替代API.

>

> 默认的比较函数检查深度相等性,因此你几乎不需要传递该参数.

#### addMany(values).before(nextValue [, compareFn])

```js

addMany(values: T[]).before(nextValue: T, compareFn?: ListComparisonFn<T>): ListNode<T>[]

```

将添加的多个节点移动到指定节点前:

```js

list.addMany(['a', 'b', 'b', 'c']).tail();

// "a" <-> "b" <-> "b" <-> "c"

list.addMany(['x', 'y']).before('b');

// "a" <-> "x" <-> "y" <-> "b" <-> "b" <-> "c"

```

你可以自定义比较器

```js

list.addMany([{ x: 1 }, { x: 2 }, { x: 3 }]).tail();

// {"x":1} <-> {"x":2} <-> {"x":3}

list

.addMany([{ x: 4 }, { x: 5 }])

.before(2, (value, searchedValue) => value.x === searchedValue);

// {"x":1} <-> {"x":4} <-> {"x":5} <-> {"x":2} <-> {"x":3}

```

> 它是 `addManyBefore` 的替代API.

>

> 默认的比较函数检查深度相等性,因此你几乎不需要传递该参数.

#### addMany(values).byIndex(position)

```js

addMany(values: T[]).byIndex(position: number): ListNode<T>[]

```

将添加的多个节点移动到链表的指定位置:

```js

list.addMany(['a', 'b', 'c']).tail();

// "a" <-> "b" <-> "c"

list.addMany(['x', 'y']).byIndex(2);

// "a" <-> "b" <-> "x" <-> "y" <-> "c"

```

它也适用于负索引:

```js

list.addMany(['a', 'b', 'c']).tail();

// "a" <-> "b" <-> "c"

list.addMany(['x', 'y']).byIndex(-1);

// "a" <-> "b" <-> "x" <-> "y" <-> "c"

```

> 它是 `addManyByIndex` 的替代API.

### 如何删除节点

有几种方法可以在链表中删除节点,这些方法都可以单独使用,也可以通过 `drop` 方法.

#### dropHead()

```js

dropHead(): ListNode<T> | undefined

```

删除链表的第一个节点:

```js

list.addMany(['a', 'b', 'c']).tail();

// "a" <-> "b" <-> "c"

list.dropHead();

// "b" <-> "c"

```

#### dropManyHead(count)

```js

dropManyHead(count: number): ListNode<T>[]

```

删除指定数量的链表的头节点:

```js

list.addMany(['a', 'b', 'c']).tail();

// "a" <-> "b" <-> "c"

list.dropManyHead(2);

// "c"

```

#### dropTail()

```js

dropTail(): ListNode<T> | undefined

```

删除链表的最后一个节点:

```js

list.addMany(['a', 'b', 'c']).tail();

// "a" <-> "b" <-> "c"

list.dropTail();

// "a" <-> "b"

```

#### dropManyTail(count)

```js

dropManyTail(count: number): ListNode<T>[]

```

删除指定数量的链表的尾节点:

```js

list.addMany(['a', 'b', 'c']).tail();

// "a" <-> "b" <-> "c"

list.dropManyTail(2);

// "a"

```

#### dropByIndex(position)

```js

dropByIndex(position: number): ListNode<T> | undefined

```

删除链表中给定位置的节点:

```js

list.addMany(['a', 'b', 'c']).tail();

// "a" <-> "b" <-> "c"

list.dropByIndex(1);

// "a" <-> "c"

```

它也适用于负索引:

```js

list.addMany(['a', 'b', 'c']).tail();

// "a" <-> "b" <-> "c"

list.dropByIndex(-2);

// "a" <-> "c"

```

#### dropManyByIndex(count, position)

```js

dropManyByIndex(count: number, position: number): ListNode<T>[]

```

删除链表中给定位置与数量的多个节点:

```js

list.addMany(['a', 'b', 'c', 'd']).tail();

// "a" <-> "b" <-> "c" <-> "d

list.dropManyByIndex(2, 1);

// "a" <-> "d"

```

它也适用于负索引:

```js

list.addMany(['a', 'b', 'c', 'd']).tail();

// "a" <-> "b" <-> "c" <-> "d

list.dropManyByIndex(2, -2);

// "a" <-> "d"

```

#### dropByValue(value [, compareFn])

```js

dropByValue(value: T, compareFn?: ListComparisonFn<T>): ListNode<T> | undefined

```

删除链表中含有给定值的第一个节点:

```js

list.addMany(['a', 'x', 'b', 'x', 'c']).tail();

// "a" <-> "x" <-> "b" <-> "x" <-> "c"

list.dropByValue('x');

// "a" <-> "b" <-> "x" <-> "c"

```

你可以自定义比较器

```js

list.addMany([{ x: 1 }, { x: 0 }, { x: 2 }, { x: 0 }, { x: 3 }]).tail();

// {"x":1} <-> {"x":0} <-> {"x":2} <-> {"x":0} <-> {"x":3}

list.dropByValue(0, (value, searchedValue) => value.x === searchedValue);

// {"x":1} <-> {"x":2} <-> {"x":0} <-> {"x":3}

```

> 默认的比较函数检查深度相等性,因此你几乎不需要传递该参数.

#### dropByValueAll(value [, compareFn])

```js

dropByValueAll(value: T, compareFn?: ListComparisonFn<T>): ListNode<T>[]

```

删除链表中含有给定值的所有节点:

```js

list.addMany(['a', 'x', 'b', 'x', 'c']).tail();

// "a" <-> "x" <-> "b" <-> "x" <-> "c"

list.dropByValueAll('x');

// "a" <-> "b" <-> "c"

```

你可以自定义比较器

```js

list.addMany([{ x: 1 }, { x: 0 }, { x: 2 }, { x: 0 }, { x: 3 }]).tail();

// {"x":1} <-> {"x":0} <-> {"x":2} <-> {"x":0} <-> {"x":3}

list.dropByValue(0, (value, searchedValue) => value.x === searchedValue);

// {"x":1} <-> {"x":2} <-> {"x":3}

```

> 默认的比较函数检查深度相等性,因此你几乎不需要传递该参数.

#### drop().head()

```js

drop().head(): ListNode<T> | undefined

```

删除链表的头节点:

```js

list.addMany(['a', 'b', 'c']).tail();

// "a" <-> "b" <-> "c"

list.drop().head();

// "b" <-> "c"

```

> 它是 `dropHead` 的替代API.

#### drop().tail()

```js

drop().tail(): ListNode<T> | undefined

```

删除链表的尾节点:

```js

list.addMany(['a', 'b', 'c']).tail();

// "a" <-> "b" <-> "c"

list.drop().tail();

// "a" <-> "b"

```

> 它是 `dropTail` 的替代API.

#### drop().byIndex(position)

```js

drop().byIndex(position: number): ListNode<T> | undefined

```

删除链表指定位置的节点:

```js

list.addMany(['a', 'b', 'c']).tail();

// "a" <-> "b" <-> "c"

list.drop().byIndex(1);

// "a" <-> "c"

```

它也适用于负索引:

```js

list.addMany(['a', 'b', 'c']).tail();

// "a" <-> "b" <-> "c"

list.drop().byIndex(-2);

// "a" <-> "c"

```

> 它是 `dropByIndex` 的替代API.

#### drop().byValue(value [, compareFn])

```js

drop().byValue(value: T, compareFn?: ListComparisonFn<T>): ListNode<T> | undefined

```

删除链表中含有给定值的第一个节点:

```js

list.addMany(['a', 'x', 'b', 'x', 'c']).tail();

// "a" <-> "x" <-> "b" <-> "x" <-> "c"

list.drop().byValue('x');

// "a" <-> "b" <-> "x" <-> "c"

```

你可以自定义比较器

```js

list.addMany([{ x: 1 }, { x: 0 }, { x: 2 }, { x: 0 }, { x: 3 }]).tail();

// {"x":1} <-> {"x":0} <-> {"x":2} <-> {"x":0} <-> {"x":3}

list

.drop()

.byValue(0, (value, searchedValue) => value.x === searchedValue);

// {"x":1} <-> {"x":2} <-> {"x":0} <-> {"x":3}

```

> 它是 `dropByValue` 的替代API.

>

> 默认的比较函数检查深度相等性,因此你几乎不需要传递该参数.

#### drop().byValueAll(value [, compareFn])

```js

drop().byValueAll(value: T, compareFn?: ListComparisonFn<T>): ListNode<T>[]

```

删除链表中含有给定值的所有节点:

```js

list.addMany(['a', 'x', 'b', 'x', 'c']).tail();

// "a" <-> "x" <-> "b" <-> "x" <-> "c"

list.drop().byValueAll('x');

// "a" <-> "b" <-> "c"

```

你可以自定义比较器

```js

list.addMany([{ x: 1 }, { x: 0 }, { x: 2 }, { x: 0 }, { x: 3 }]).tail();

// {"x":1} <-> {"x":0} <-> {"x":2} <-> {"x":0} <-> {"x":3}

list

.drop()

.byValueAll(0, (value, searchedValue) => value.x === searchedValue);

// {"x":1} <-> {"x":2} <-> {"x":3}

```

> 它是 `dropByValueAll` 的替代API.

>

> 默认的比较函数检查深度相等性,因此你几乎不需要传递该参数.

#### dropMany(count).head()

```js

dropMany(count: number).head(): ListNode<T>[]

```

删除链表中指定数量的头节点:

```js

list.addMany(['a', 'b', 'c']).tail();

// "a" <-> "b" <-> "c"

list.dropMany(2).head();

// "c"

```

> 它是 `dropManyHead` 的替代API.

#### dropMany(count).tail()

```js

dropMany(count: number).tail(): ListNode<T>[]

```

删除链表中指定数量的尾节点::

```js

list.addMany(['a', 'b', 'c']).tail();

// "a" <-> "b" <-> "c"

list.dropMany(2).tail();

// "a"

```

> 它是 `dropManyTail` 的替代API.

#### dropMany(count).byIndex(position)

```js

dropMany(count: number).byIndex(position: number): ListNode<T>[]

```

删除链表中指定位置和数量的节点:

```js

list.addMany(['a', 'b', 'c', 'd']).tail();

// "a" <-> "b" <-> "c" <-> "d

list.dropMany(2).byIndex(1);

// "a" <-> "d"

```

它也适用于负索引:

```js

list.addMany(['a', 'b', 'c', 'd']).tail();

// "a" <-> "b" <-> "c" <-> "d

list.dropMany(2).byIndex(-2);

// "a" <-> "d"

```

> 它是 `dropManyByIndex` 的替代API.

### 如何查找节点

有几个方法找到链表特定节点.

#### head

```js

head: ListNode<T> | undefined;

```

链表中的第一个节点.

#### tail

```js

tail: ListNode<T> | undefined;

```

链表中的最后一个节点.

#### length

```js

length: number;

```

链表的节点总数.

#### find(predicate)

```js

find(predicate: ListIteratorFunction<T>): ListNode<T> | undefined

```

从链表中找到与给定谓词匹配的第一个节点:

```js

list.addManyTail(['a', 'b', 'b', 'c']);

// "a" <-> "b" <-> "b" <-> "c"

var found = list.find(node => node.value === 'b');

/\*

found.value === "b"

found.previous.value === "a"

found.next.value === "b"

\*/

```

#### findIndex(predicate)

```js

findIndex(predicate: ListIteratorFunction<T>): number

```

从链表中找到与给定谓词匹配的第一个节点的位置:

```js

list.addManyTail(['a', 'b', 'b', 'c']);

// "a" <-> "b" <-> "b" <-> "c"

var i0 = list.findIndex(node => node.next && node.next.value === 'b');

var i1 = list.findIndex(node => node.value === 'b');

var i2 = list.findIndex(node => node.previous && node.previous.value === 'b');

var i3 = list.findIndex(node => node.value === 'x');

/\*

i0 === 0

i1 === 1

i2 === 2

i3 === -1

\*/

```

#### get(position)

```js

get(position: number): ListNode<T> | undefined

```

查找并返回链表中特定位置的节点:

```js

list.addManyTail(['a', 'b', 'c']);

// "a" <-> "b" <-> "c"

var found = list.get(1);

/\*

found.value === "b"

found.previous.value === "a"

found.next.value === "c"

\*/

```

#### indexOf(value [, compareFn])

```js

indexOf(value: T, compareFn?: ListComparisonFn<T>): number

```

在链表中找到匹配给定值的第一个节点位置:

```js

list.addManyTail(['a', 'b', 'b', 'c']);

// "a" <-> "b" <-> "b" <-> "c"

var i0 = list.indexOf('a');

var i1 = list.indexOf('b');

var i2 = list.indexOf('c');

var i3 = list.indexOf('x');

/\*

i0 === 0

i1 === 1

i2 === 3

i3 === -1

\*/

```

你可以自定义比较器

```js

list.addManyTail([{ x: 1 }, { x: 0 }, { x: 2 }, { x: 0 }, { x: 3 }]);

// {"x":1} <-> {"x":0} <-> {"x":2} <-> {"x":0} <-> {"x":3}

var i0 = indexOf(1, (value, searchedValue) => value.x === searchedValue);

var i1 = indexOf(2, (value, searchedValue) => value.x === searchedValue);

var i2 = indexOf(3, (value, searchedValue) => value.x === searchedValue);

var i3 = indexOf(0, (value, searchedValue) => value.x === searchedValue);

var i4 = indexOf(4, (value, searchedValue) => value.x === searchedValue);

/\*

i0 === 0

i1 === 2

i2 === 4

i3 === 1

i4 === -1

\*/

```

> 默认的比较函数检查深度相等性,因此你几乎不需要传递该参数.

### 如何检查所有节点

有几种方法来遍历或显示一个链表.

#### forEach(iteratorFn)

```js

forEach(iteratorFn: ListIteratorFn<T>): void

```

从头到尾在链表中的所有节点上运行回调函数:

```js

list.addManyTail(['a', 'b', 'c']);

// "a" <-> "b" <-> "c"

list.forEach((node, index) => console.log(node.value + index));

// 'a0'

// 'b1'

// 'c2'

```

#### \\*\[Symbol.iterator\]\(\)

链表是可迭代的. 换句话说你可以使用诸如`for ... of`之类的方法.

```js

list.addManyTail(['a', 'b', 'c']);

// "a" <-> "b" <-> "c"

for(var node of list) { /\* ES6 for...of statement \*/

console.log(node.value);

}

// 'a'

// 'b'

// 'c'

```

#### toArray()

```js

toArray(): T[]

```

转换链表值为数组:

```js

list.addManyTail(['a', 'b', 'c']);

// "a" <-> "b" <-> "c"

var arr = list.toArray();

/\*

arr === ['a', 'b', 'c']

\*/

```

#### toNodeArray()

```js

toNodeArray(): T[]

```

转换链表节点为数组:

```js

list.addManyTail(['a', 'b', 'c']);

// "a" <-> "b" <-> "c"

var arr = list.toNodeArray();

/\*

arr[0].value === 'a'

arr[1].value === 'a'

arr[2].value === 'a'

\*/

```

#### toString(mapperFn)

```js

toString(mapperFn: ListMapperFn<T> = JSON.stringify): string

```

将链表转换为节点及其关系的字符串表示形式:

```js

list.addManyTail(['a', 2, 'c', { k: 4, v: 'd' }]);

// "a" <-> 2 <-> "c" <-> {"k":4,"v":"d"}

var str = list.toString();

/\*

str === '"a" <-> 2 <-> "c" <-> {"k":4,"v":"d"}'

\*/

```

你可以在对值进行字符串化之前通过自定义映射器函数来映射值:

```js

list.addMany([{ x: 1 }, { x: 2 }, { x: 3 }, { x: 4 }, { x: 5 }]).tail();

// {"x":1} <-> {"x":2} <-> {"x":3} <-> {"x":4} <-> {"x":5}

var str = list.toString(value => value.x);

/\*

str === '1 <-> 2 <-> 3 <-> 4 <-> 5'

\*/

```

## API

### Classes

#### LinkedList

```js

export class LinkedList<T = any> {

// properties and methods are explained above

}

```

#### ListNode

```js

export class ListNode<T = any> {

next: ListNode | undefined;

previous: ListNode | undefined;

constructor(public readonly value: T) {}

}

```

`ListNode` 是存储在 `LinkedList` 中的每个记录的节点.

- `value` value是存储在节点中的值,通过构造函数传递.

- `next` 引用列表中的下一个节点.

- `previous`引用列表中的上一个节点.

```js

list.addManyTail([ 0, 1, 2 ]);

console.log(

list.head.value, // 0

list.head.next.value, // 1

list.head.next.next.value, // 2

list.head.next.next.previous.value, // 1

list.head.next.next.previous.previous.value, // 0

list.tail.value, // 2

list.tail.previous.value, // 1

list.tail.previous.previous.value, // 0

list.tail.previous.previous.next.value, // 1

list.tail.previous.previous.next.next.value, // 2

);

```

### Types

#### ListMapperFn

```js

type ListMapperFn<T = any> = (value: T) => any;

```

该函数在 `toString` 方法中用于在生成列表的字符串形式之前映射节点值.

#### ListComparisonFn

```js

type ListComparisonFn<T = any> = (nodeValue: T, comparedValue: any) => boolean;

```

该函数用于根据比较值添加,删除和查找节点.

#### ListIteratorFn

```js

type ListIteratorFn<T = any, R = boolean> = (

node: ListNode<T>,

index?: number,

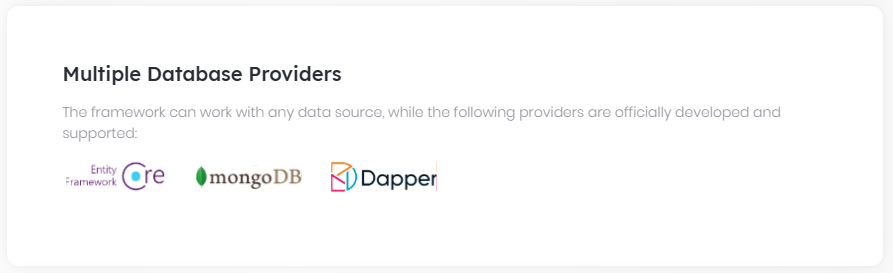
list?: LinkedList,

) => R;

```

该函数在遍历列表时使用,可以对每个节点执行某些操作,也可以查找某个节点.

# 数据访问



## 概述

# 数据访问

ABP框架被设计为与数据库无关, 它通过[仓储](Repositories.md)和[工作单元](Unit-Of-Work.md)抽象处理来自任何类型的数据源.

目前实现了以下数据库访问提供程序:

\* [Entity Framework Core](Entity-Framework-Core.md) (与各种[DBMS和提供程序](https://docs.microsoft.com/zh-cn/ef/core/providers/?tabs=dotnet-core-cli)一起使用.)

\* [MongoDB](MongoDB.md)

\* [Dapper](Dapper.md)

## 另请参阅

\* [连接字符串](Connection-Strings.md)

\* [种子数据](Data-Seeding.md)

\* [数据过滤](Data-Filtering.md)

## Entity Framework Core

# Entity Framework Core 集成

本文介绍了如何将EF Core作为ORM提供程序集成到基于ABP的应用程序以及如何对其进行配置.

## 安装

`Volo.Abp.EntityFrameworkCore` 是EF Core 集成的主要nuget包. 将其安装到你的项目中(在分层应用程序中适用于 数据访问/基础设施层):

```shell

Install-Package Volo.Abp.EntityFrameworkCore

```

然后添加 `AbpEntityFrameworkCoreModule` 模块依赖项(`DependsOn` Attribute) 到 [module](Module-Development-Basics.md)(项目中的Mudole类):

````C#

using Volo.Abp.EntityFrameworkCore;

using Volo.Abp.Modularity;

namespace MyCompany.MyProject

{

[DependsOn(typeof(AbpEntityFrameworkCoreModule))]

public class MyModule : AbpModule

{

//...

}

}

````

> 注: 你可以直接下载预装EF Core的[启动模板](https://abp.io/Templates).

### 数据库管理系统选择

EF Core支持多种数据库管理系统([查看全部](https://docs.microsoft.com/zh-cn/ef/core/providers/)). ABP框架和本文档不依赖于任何特定的DBMS. 如果要创建一个可重用的[应用程序模块](Modules/Index.md),应避免依赖于特定的DBMS包.但在最终的应用程序中,始终会选择一个DBMS.

参阅[为Entity Framework Core切换到其他DBMS](Entity-Framework-Core-Other-DBMS.md)文档学习如何切换DBMS.

## 创建 DbContext

你可以平常一样创建DbContext,它需要继承自 `AbpDbContext<T>`. 如下所示:

````C#

using Microsoft.EntityFrameworkCore;

using Volo.Abp.EntityFrameworkCore;

namespace MyCompany.MyProject

{

public class MyDbContext : AbpDbContext<MyDbContext>

{

//...在这里添加 DbSet properties

public MyDbContext(DbContextOptions<MyDbContext> options)

: base(options)

{

}

}

}

````

### 关于EF Core Fluent Mapping

[应用程序启动模板](Startup-Templates/Application.md)已配置使用[EF Core fluent configuration API](https://docs.microsoft.com/zh-cn/ef/core/modeling/)映射你的实体到数据库表.

你依然为你的实体属性使用**\*\*data annotation attributes\*\***(像`[Required]`),而ABP文档通常遵循**\*\*fluent mapping API\*\*** approach方法. 如何使用取决与你.

ABP框架有一些**\*\*实体基类\*\***和**\*\*约定\*\***(参阅[实体文档](Entities.md))提供了一些有用的扩展方法来配置从基本实体类继承的属性.

#### ConfigureByConvention 方法

`ConfigureByConvention()` 是主要的扩展方法,它对你的实体**\*\*配置所有的基本属性\*\***和约定. 所以在你的流利映射代码中为你所有的实体调用这个方法是 **\*\*最佳实践\*\***,

**\*\*示例\*\***: 假设你有一个直接继承 `AggregateRoot<Guid>` 基类的 `Book` 实体:

````csharp

public class Book : AuditedAggregateRoot<Guid>

{

public string Name { get; set; }

}

````

你可以在你的 `DbContext` 重写 `OnModelCreating` 方法并且做以下配置:

````csharp

protected override void OnModelCreating(ModelBuilder builder)

{

//Always call the base method

base.OnModelCreating(builder);

builder.Entity<Book>(b =>

{

b.ToTable("Books");

//Configure the base properties

b.ConfigureByConvention();

//Configure other properties (if you are using the fluent API)

b.Property(x => x.Name).IsRequired().HasMaxLength(128);

});

}

````

\* 这里调用了 `b.ConfigureByConvention()` 它对于**\*\*配置基本属性\*\***非常重要.

\* 你可以在这里配置 `Name` 属性或者使用**\*\*data annotation attributes\*\***(参阅[EF Core 文档](https://docs.microsoft.com/zh-cn/ef/core/modeling/entity-properties)).

> 尽管有许多扩展方法可以配置基本属性,但如果需要 `ConfigureByConvention()` 内部会调用它们. 因此仅调用它就足够了.

### 配置连接字符串选择

如果你的应用程序有多个数据库,你可以使用 `[connectionStringName]` Attribute为你的DbContext配置连接字符串名称.例:

```csharp

[ConnectionStringName("MySecondConnString")]

public class MyDbContext : AbpDbContext<MyDbContext>

{

}

```

如果不进行配置,则使用`Default`连接字符串. 如果你配置特定的连接字符串的名称,但在应用程序配置中没有定义这个连接字符串名称,那么它会回退到`Default`连接字符串(参阅[连接字符串文档](Connection-Strings.md)了解更多信息).

### AbpDbContextOptions

`AbpDbContextOptions` 用于配置 `DbContext`. 当你使用ABP的应用程序启动模板新建解决方案时, 你会看到一个简单的配置 (在 `EntityFrameworkCore` 集成项目模块类) 如下:

````csharp

Configure<AbpDbContextOptions>(options =>

{

options.UseSqlServer();

});

````

上面的配置为应用程序的所有 `DbContext`使用SQL Server作为默认DBMS. 上面的配置是简化的写法, 它也可以使用下面的方法进行配置:

````csharp

Configure<AbpDbContextOptions>(options =>

{

options.Configure(opts =>

{

opts.UseSqlServer();

});

});

````

`options.Configure(...)` 方法有更多的选项进行配置. 例如, 你可以设置 `DbContextOptions` (EF Core自有的配置):

````csharp

Configure<AbpDbContextOptions>(options =>

{

options.Configure(opts =>

{

opts.DbContextOptions.UseQueryTrackingBehavior(QueryTrackingBehavior.NoTracking);

});

});

````

如果你有唯一的 `DbContext` 或者有多个 `DbContext`, 但是希望对所有的 `DbContext` 使用相同的DBMS, 你无需更多的配置. 但是, 如果你需要为某个特定的 `DbContext` 配置不同的DBMS或对配置进行定制, 你可以进行如下定制:

````csharp

Configure<AbpDbContextOptions>(options =>

{

// Default configuration for all DbContexts

options.Configure(opts =>

{

opts.UseSqlServer();

});

// Customized configuration for a specific DbContext

options.Configure<MyOtherDbContext>(opts =>

{

opts.UseMySQL();

});

});

````

> 参阅 [为Entity Framework Core切换DBMS](Entity-Framework-Core-Other-DBMS.md) 文档学习如何配置DBMS.

## 将DbContext注册到依赖注入

在module中的ConfigureServices方法使用 `AddAbpDbContext` 在[依赖注入](Dependency-Injection.md)系统注册DbContext类.

````C#

using Microsoft.Extensions.DependencyInjection;

using Volo.Abp.EntityFrameworkCore;

using Volo.Abp.Modularity;

namespace MyCompany.MyProject

{

[DependsOn(typeof(AbpEntityFrameworkCoreModule))]

public class MyModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

context.Services.AddAbpDbContext<MyDbContext>();

//...

}

}

}

````

### 添加默认仓储

ABP会自动为DbContext中的实体创建[默认仓储](Repositories.md). 需要在注册的时使用options添加`AddDefaultRepositories()`:

````C#

services.AddAbpDbContext<MyDbContext>(options =>

{

options.AddDefaultRepositories();

});

````

默认情况下为每个[聚合根实体](Entities.md)(`AggregateRoot`派生的子类)创建一个仓储. 如果想要为其他实体也创建仓储, 请将`includeAllEntities` 设置为 `true`:

````C#

services.AddAbpDbContext<MyDbContext>(options =>

{

options.AddDefaultRepositories(includeAllEntities: true);

});

````

然后你就可以在服务中注入和使用 `IRepository<TEntity>` 或 `IQueryableRepository<TEntity>`. 假如你有一个主键是Guid名为Book实体(聚合根)

```csharp

public class Book : AggregateRoot<Guid>

{

public string Name { get; set; }

public BookType Type { get; set; }

}

```

在[领域服务](Domain-Services.md)中创建一个新的Book实例并且使用仓储持久化到数据库中

````csharp

public class BookManager : DomainService

{

private readonly IRepository<Book, Guid> \_bookRepository;

//inject default repository to the constructor

public BookManager(IRepository<Book, Guid> bookRepository)

{

\_bookRepository = bookRepository;

}

public async Task<Book> CreateBook(string name, BookType type)

{

Check.NotNullOrWhiteSpace(name, nameof(name));

var book = new Book

{

Id = GuidGenerator.Create(),

Name = name,

Type = type

};

//Use a standard repository method

await \_bookRepository.InsertAsync(book);

return book;

}

}

````

在这个示例中使用 `InsertAsync` 将新实例插入到数据库中

#### 添加自定义仓储

默认通用仓储可以满足大多数情况下的需求(它实现了`IQueryable`),但是你可能会需要自定义仓储与仓储方法. 假设你需要根据图书类型删除所有的书籍.

建议为自定义仓储定义一个接口:

````csharp

public interface IBookRepository : IRepository<Book, Guid>

{

Task DeleteBooksByType(BookType type);

}

````

你通常希望从IRepository派生以继承标准存储库方法. 然而,你没有必要这样做. 仓储接口在分层应用程序的领域层中定义,它在数据访问/基础设施层([启动模板](https://abp.io/Templates)中的`EntityFrameworkCore`项目)中实现

IBookRepository接口的实现示例:

````csharp

public class BookRepository

: EfCoreRepository<BookStoreDbContext, Book, Guid>, IBookRepository

{

public BookRepository(IDbContextProvider<BookStoreDbContext> dbContextProvider)

: base(dbContextProvider)

{

}

public async Task DeleteBooksByType(BookType type)

{

var dbContext = await GetDbContextAsync();

await dbContext.Database.ExecuteSqlRawAsync(

$"DELETE FROM Books WHERE Type = {(int)type}"

);

}

}

````

现在可以在需要时[注入](Dependency-Injection.md)`IBookRepository`并使用`DeleteBooksByType`方法.

#### 覆盖默认通用仓储

即使创建了自定义仓储,仍可以注入使用默认通用仓储(在本例中是 `IRepository<Book, Guid>`). 默认仓储实现不会使用你创建的自定义仓储类.

如果要将默认仓储实现替换为自定义仓储,请在`AddAbpDbContext`使用options执行:

````csharp

context.Services.AddAbpDbContext<BookStoreDbContext>(options =>

{

options.AddDefaultRepositories();

//Replaces IRepository<Book, Guid>

options.AddRepository<Book, BookRepository>();

});

````

在你想要覆盖默认仓储方法对其自定义时,这一点非常需要. 例如你可能希望自定义`DeleteAsync`方法覆盖默认实现, 以使用更有效的方式删除特定的实体.

````csharp

public async override Task DeleteAsync(

Guid id,

bool autoSave = false,

CancellationToken cancellationToken = default)

{

//TODO: Custom implementation of the delete method

}

````

## 加载关联实体

假设你拥有带有`OrderLine`集合的`Order`,并且`OrderLine`具有`Order`的导航属性:

````csharp

using System;

using System.Collections.Generic;

using System.Collections.ObjectModel;

using Volo.Abp.Auditing;

using Volo.Abp.Domain.Entities;

namespace MyCrm

{

public class Order : AggregateRoot<Guid>, IHasCreationTime

{

public Guid CustomerId { get; set; }

public DateTime CreationTime { get; set; }

public ICollection<OrderLine> Lines { get; set; } //子集合

public Order()

{

Lines = new Collection<OrderLine>();

}

}

public class OrderLine : Entity<Guid>

{

public Order Order { get; set; } //导航属性

public Guid OrderId { get; set; }

public Guid ProductId { get; set; }

public int Count { get; set; }

public double UnitPrice { get; set; }

}

}

````

然后象下面显示的这样定义数据库映射:

````csharp

builder.Entity<Order>(b =>

{

b.ToTable("Orders");

b.ConfigureByConvention();

//定义关系

b.HasMany(x => x.Lines)

.WithOne(x => x.Order)

.HasForeignKey(x => x.OrderId)

.IsRequired();

});

builder.Entity<OrderLine>(b =>

{

b.ToTable("OrderLines");

b.ConfigureByConvention();

});

````

当你查询一个 `Order`, 你可能想要在单个查询中**\*\*包含\*\***所有的 `OrderLine`s 或根据需要在**\*\*以后加载它们\*\***.

> 实际上这与ABP框架没有直接关系. 你可以按照 [EF Core 文档](https://docs.microsoft.com/zh-cn/ef/core/querying/related-data/) 了解全部细节. 本节将涵盖与 ABP 框架相关的一些主题.

### 预先加载 / 包含子对象的加载

当你想加载一个带有关联实体的实体时,可以使用不同的选项.

#### Repository.WithDetails

`IRepository.WithDetailsAsync(...)` 可以通过包含一个关系收集/属性来获得 `IQueryable<T>` .

**\*\*示例: 获取一个带有** `lines` **的** `order` **对象\*\***

````csharp

using System;

using System.Linq;

using System.Threading.Tasks;

using Volo.Abp.Domain.Repositories;

using Volo.Abp.Domain.Services;

namespace AbpDemo.Orders

{

public class OrderManager : DomainService

{

private readonly IRepository<Order, Guid> \_orderRepository;

public OrderManager(IRepository<Order, Guid> orderRepository)

{

\_orderRepository = orderRepository;

}

public async Task TestWithDetails(Guid id)

{

//通过包含子集合获取一个 IQueryable<T>

var queryable = await \_orderRepository.WithDetailsAsync(x => x.Lines);

//应用其他的 LINQ 扩展方法

var query = queryable.Where(x => x.Id == id);

//执行此查询并获取结果

var order = await AsyncExecuter.FirstOrDefaultAsync(query);

}

}

}

````

> `AsyncExecuter` 用于执行异步 LINQ 扩展,而无需依赖 EF Core. 如果你将 EF Core NuGet 包引用添加到你的项目中,则可以直接使用 `await query.FirstOrDefaultAsync()`. 但是, 这次你依赖于域层中的 EF Core. 请参阅 [仓储文档](Repositories.md) 以了解更多.

**\*\*示例: 获取一个包含** `lines` **的** `orders` **列表\*\***

````csharp

public async Task TestWithDetails()

{

//通过包含一个子集合获取一个 IQueryable<T>

var queryable = await \_orderRepository.WithDetailsAsync(x => x.Lines);

//执行此查询并获取结果

var orders = await AsyncExecuter.ToListAsync(queryable);

}

````

> 如果你需要包含多个导航属性或集合,`WithDetailsAsync`方法可以获得多个表达参数.

#### DefaultWithDetailsFunc

如果你没有将任何表达式传递到 `WithDetailsAsync` 方法,则它包括使用你提供的 `DefaultWithDetailsFunc` 选项的所有详细信息.

你可以在你的 `EntityFrameworkCore` 项目[模块](Module-Development-Basics.md)的 `ConfigureServices` 方法为一个实体配置 `DefaultWithDetailsFunc`.

**\*\*示例: 在查询一个** `Order` **时包含** `Lines`**\*\***

````csharp

Configure<AbpEntityOptions>(options =>

{

options.Entity<Order>(orderOptions =>

{

orderOptions.DefaultWithDetailsFunc = query => query.Include(o => o.Lines);

});

});

````

> 你可以在这里完全使用 EF Core API,因为这位于 EF Core集成项目中.

然后你可以不带任何参数地调用 `WithDetails` 方法:

````csharp

public async Task TestWithDetails()

{

//通过包含一个子集合获取一个 IQueryable<T>

var queryable = await \_orderRepository.WithDetailsAsync();

//执行此查询并获取结果

var orders = await AsyncExecuter.ToListAsync(queryable);

}

````

`WithDetailsAsync()` 执行你已经在 `DefaultWithDetailsFunc` 中设置的表达式.

#### 仓储 Get/Find 方法

有些标准的 [仓储](Repositories.md) 方法带有可选的 `includeDetails` 参数;

\* `GetAsync` 和 `FindAsync` 方法带有默认值为 `true` 的 `includeDetails`.

\* `GetListAsync` 和 `GetPagedListAsync` 方法带有默认值为 `false` 的 `includeDetails`.

这意味着,默认情况下返回**\*\*包含子对象的单个实体\*\***,而列表返回方法则默认不包括子对象信息.你可以明确通过 `includeDetails` 来更改此行为.

> 这些方法使用上面解释的 `DefaultWithDetailsFunc` 选项.

**\*\*示例:获取一个包含子对象的** `order`**\*\***

````csharp

public async Task TestWithDetails(Guid id)

{

var order = await \_orderRepository.GetAsync(id);

}

````

**\*\*示例:获取一个不包含子对象的** `order`**\*\***

````csharp

public async Task TestWithoutDetails(Guid id)

{

var order = await \_orderRepository.GetAsync(id, includeDetails: false);

}

````

**\*\*示例:获取一个包含子对象的实体列表\*\***

````csharp

public async Task TestWithDetails()

{

var orders = await \_orderRepository.GetListAsync(includeDetails: true);

}

````

#### 更多的替代选择

存储库模式尝试封装 EF Core, 因此你的选项是有限的. 如果你需要高级方案,你可以按照其中一个选项执行:

\* 创建自定义存储库方法并使用完整的 EF Core API.

\* 在你的项目中引用 `Volo.Abp.EntityFrameworkCore` . 通过这种方式,你可以直接在代码中使用 `Include` 和 `ThenInclude` .

请参阅 EF Core 的 [预先加载文档](https://docs.microsoft.com/zh-cn/ef/core/querying/related-data/eager).

### 显式 / 延迟加载

如果你在查询实体时不包括关系,并且以后需要访问导航属性或集合,则你有不同的选择.

#### EnsurePropertyLoadedAsync / EnsureCollectionLoadedAsync

仓储提供 `EnsurePropertyLoadedAsync` 和 `EnsureCollectionLoadedAsync` 扩展方法来**\*\*显示加载\*\***一个导航属性或子集合.

**\*\*示例: 在需要时加载一个** `Order` **的** `Lines`**\*\***

````csharp

public async Task TestWithDetails(Guid id)

{

var order = await \_orderRepository.GetAsync(id, includeDetails: false);

//order.Lines 此时是空的

await \_orderRepository.EnsureCollectionLoadedAsync(order, x => x.Lines);

//order.Lines 被填充

}

````

如果导航属性或集合已经被加载那么 `EnsurePropertyLoadedAsync` 和 `EnsureCollectionLoadedAsync` 方法不做任何处理. 所以,调用多次也没有问题.

请参阅 EF Core 的[显示加载文档](https://docs.microsoft.com/zh-cn/ef/core/querying/related-data/explicit).

#### 使用代理的延时加载

在某些情况下,可能无法使用显式加载,尤其是当你没有引用 `Repository` 或 `DbContext`时.延时加载是 EF Core 加载关联属性/集合的一个功能, 当你第一次访问它.

启用延时加载:

1. 安装 [Microsoft.EntityFrameworkCore.Proxies](https://www.nuget.org/packages/Microsoft.EntityFrameworkCore.Proxies/) 包到你的项目(通常是 EF Core 集成项目)

2. 为你的 `DbContext` 配置 `UseLazyLoadingProxies` (在 EF Core 项目的模块的 `ConfigureServices` 方法中). 例如:

````csharp

Configure<AbpDbContextOptions>(options =>

{

options.PreConfigure<MyCrmDbContext>(opts =>

{

opts.DbContextOptions.UseLazyLoadingProxies(); //启用延时加载

});

options.UseSqlServer();

});

````

3. 使你的导航属性和集合是 `virtual`. 例如:

````csharp

public virtual ICollection<OrderLine> Lines { get; set; } //虚集合

public virtual Order Order { get; set; } //虚导航属性

````

启用延时加载并整理实体后,你可以自由访问导航属性和集合:

````csharp

public async Task TestWithDetails(Guid id)

{

var order = await \_orderRepository.GetAsync(id);

//order.Lines 此时是空的

var lines = order.Lines;

//order.Lines 被填充 (延时加载)

}

````

每当你访问属性/集合时,EF Core 都会自动执行额外的查询,从数据库中加载属性/集合.

> 应谨慎使用延时加载,因为它可能会在某些特定情况下导致性能问题.

请参阅 EF Core 的[延时加载文档](https://docs.microsoft.com/zh-cn/ef/core/querying/related-data/lazy).

## 访问 EF Core API

大多数情况下应该隐藏仓储后面的EF Core API(这也是仓储的设计目的). 但是如果想要通过仓储访问DbContext实现,则可以使用`GetDbContext()`或`GetDbSet()`扩展方法. 例如:

````csharp

public async Task TestAsync()

{

var dbContext = await \_orderRepository.GetDbContextAsync();

var dbSet = await \_orderRepository.GetDbSetAsync();

//var dbSet = dbContext.Set<Order>(); //Alternative, when you have the DbContext

}

````

\* `GetDbContext` 返回 `DbContext` 引用,而不是 `BookStoreDbContext`. 你可以强制转化它, 但大多数情况下你不会需要它.

> 要点: 你必须在使用`DbContext`的项目里引用`Volo.Abp.EntityFrameworkCore`包. 这会破坏封装,但在这种情况下,这就是你需要的.

## 额外属性 & Object Extension Manager

额外属性系统允许你为实现了 `IHasExtraProperties` 的实体set/get动态属性. 当你想将自定义属性添加到[应用程序模块](Modules/Index.md)中定义的实体时,它特别有用.

默认情况下, 实体的所有额外属性存储在数据库的一个 `JSON` 对象中.

实体扩展系统允许你存储额外属性在数据库的单独字段中. 有关额外属性和实体扩展系统的更多信息,请参阅下列文档:

\* [自定义应用模块: 扩展实体](Customizing-Application-Modules-Extending-Entities.md)

\* [实体](Entities.md)

本节只解释了 EF Core相关的 `ObjectExtensionManager` 及其用法.

### ObjectExtensionManager.Instance

`ObjectExtensionManager` 实现单例模式,因此你需要使用静态的 `ObjectExtensionManager.Instance` 来执行所有操作.

### MapEfCoreProperty

`MapEfCoreProperty` 是一种快捷扩展方法,用于定义实体的扩展属性并映射到数据库.

**\*\*示例\*\***: 添加 `Title` 属性 (数据库字段)到 `IdentityRole` 实体:

````csharp

ObjectExtensionManager.Instance

.MapEfCoreProperty<IdentityRole, string>(

"Title",

(entityBuilder, propertyBuilder) =>

{

propertyBuilder.HasMaxLength(64);

}

);

````

### MapEfCoreEntity

`MapEfCoreEntity` 一个配置 `Entity` 的快捷扩展方法.

**\*\*示例\*\***: 设置 `IdentityRole` 实体的 `Name` 的最大长度:

````csharp

ObjectExtensionManager.Instance

.MapEfCoreEntity<IdentityRole>(builder =>

{

builder.As<EntityTypeBuilder<IdentityRole>>().Property(x => x.Name).HasMaxLength(200);

});

````

### MapEfCoreDbContext

`MapEfCoreDbContext` 一个配置 `DbContext` 的快捷扩展方法.

**\*\*示例\*\***: 设置 `IdentityDbContext` 的 `IdentityRole` 实体的 `IdentityRole` 的最大长度:

````csharp

ObjectExtensionManager.Instance.MapEfCoreDbContext<IdentityDbContext>(b =>

{

b.Entity<IdentityRole>().Property(x => x.Name).HasMaxLength(200);

});

````

如果相关模块已实现此功能(通过使用下面说明的 `ConfigureEfCoreEntity`)则将新属性添加到模型中. 然后你需要运行标准的 `Add-Migration` 和 `Update-Database` 命令更新数据库以添加新字段.

>`MapEfCoreProperty`, `MapEfCoreEntity` and `MapEfCoreDbContext` 方法必须在使用相关的 `DbContext` 之前调用,它是一个静态方法. 最好的方法是尽早的应用程序中使用它. 应用程序启动模板含有 `YourProjectNameEfCoreEntityExtensionMappings` 类,可以在放心的在此类中使用此方法.

### ConfigureEfCoreEntity, ApplyObjectExtensionMappings 和 TryConfigureObjectExtensions

如果你正在开发一个可重用使用的模块,并允许应用程序开发人员将属性添加到你的实体,你可以在实体映射使用 `ConfigureEfCoreEntity` 扩展方法,但是在配置实体映射时可以使用快捷的扩展方法 `ConfigureObjectExtensions`:

**\*\*示例\*\***:

````csharp

public static class QADbContextModelCreatingExtensions

{

public static void ConfigureQA(

this ModelBuilder builder,

Action<QAModelBuilderConfigurationOptions> optionsAction = null)

{

Check.NotNull(builder, nameof(builder));

var options = new QAModelBuilderConfigurationOptions(

QADatabaseDbProperties.DbTablePrefix,

QADatabaseDbProperties.DbSchema

);

optionsAction?.Invoke(options);

builder.Entity<QA\_Question>(b =>

{

b.ToTable(options.TablePrefix + "Questions", options.Schema);

b.ConfigureByConvention();

//...

//Call this in the end of buildAction.

b.ApplyObjectExtensionMappings();

});

//...

//Call this in the end of ConfigureQA.

builder.TryConfigureObjectExtensions<QADbContext>();

}

}

````

如果你调用 `ConfigureByConvention()` 扩展方法(在此示例中 `b.ConfigureByConvention`),ABP框架内部会调用 `ConfigureObjectExtensions` 和 `ConfigureEfCoreEntity` 方法. 使用 `ConfigureByConvention` 方法是**\*\*最佳实践\*\***,因为它还按照约定配置基本属性的数据库映射.

参阅上面提到的 "*\*ConfigureByConvention 方法\**" 了解更多信息.

## 高级主题

### 控制多租户

如果你的方案是基于 [多租户](Multi-Tenancy.md)的, 租户可以拥有 **\*\*独立数据库\*\***, 你在解决方案中可以拥有 **\*\*多个\*\*** `DbContext` 类, 并且其中的一些 `DbContext` 类 **\*\*只能在主机端\*\*** 可用, 这种情况下建议在 `DbContext` 类上添加 `[IgnoreMultiTenancy]` 属性. ABP 保证相关的 `DbContext` 始终使用主机 [连接字符串](Connection-Strings.md), 即使你在租户上下文中.

**\*\*示例:\*\***

````csharp

[IgnoreMultiTenancy]

public class MyDbContext : AbpDbContext<MyDbContext>

{

...

}

````

不要使用 `[IgnoreMultiTenancy]` 特性如果 `DbContext` 中任何一个实体可以被持久化到多租户数据库中.

> 当你使用repositories时, ABP 已经为未实现`IMultiTenant`接口的实体使用了主机数据库. 所以, 如果你使用repositories访问数据库, 多数时候你不需要 `[IgnoreMultiTenancy]` 特性.

### 设置默认仓储类

默认的通用仓储的默认实现是`EfCoreRepository`类,你可以创建自己的实现,并将其做为默认实现

首先,像这样定义仓储类:

```csharp

public class MyRepositoryBase<TEntity>

: EfCoreRepository<BookStoreDbContext, TEntity>

where TEntity : class, IEntity

{

public MyRepositoryBase(IDbContextProvider<BookStoreDbContext> dbContextProvider)

: base(dbContextProvider)

{

}

}

public class MyRepositoryBase<TEntity, TKey>

: EfCoreRepository<BookStoreDbContext, TEntity, TKey>

where TEntity : class, IEntity<TKey>

{

public MyRepositoryBase(IDbContextProvider<BookStoreDbContext> dbContextProvider)

: base(dbContextProvider)

{

}

}

```

第一个用于具有[复合主键的实体](Entities.md),第二个用于具有单个主键的实体

建议从`EfCoreRepository`类继承并在需要时重写方法. 否则,你需要手动实现所有标准仓储方法.

现在,你可以使用SetDefaultRepositoryClasses Options

```csharp

context.Services.AddAbpDbContext<BookStoreDbContext>(options =>

{

options.SetDefaultRepositoryClasses(

typeof(MyRepositoryBase<,>),

typeof(MyRepositoryBase<>)

);

//...

});

```

### 为默认仓储设置Base DbContext类或接口

如果你的DbContext继承了另外一个DbContext或实现了一个接口,你可以使用这个基类或接口作为默认仓储的DbContext. 例:

````csharp

public interface IBookStoreDbContext : IEfCoreDbContext

{

DbSet<Book> Books { get; }

}

````

`IBookStoreDbContext`接口是由`BookStoreDbContext`实现的. 然后你可以使用`AddDefaultRepositories`的泛型重载.

````csharp

context.Services.AddAbpDbContext<BookStoreDbContext>(options =>

{

options.AddDefaultRepositories<IBookStoreDbContext>();

//...

});

````

现在,你的自定义仓储也可以使用`IBookStoreDbContext`接口:

````csharp

public class BookRepository : EfCoreRepository<IBookStoreDbContext, Book, Guid>, IBookRepository

{

//...

}

````

使用DbContext接口的一个优点是它可以被其他实现替换.

### 替换其他仓储

正确定义并使用DbContext接口后,任何其他实现都可以使用以下方法替换它:

**\*\*ReplaceDbContextAttribute\*\***

```csharp

[ReplaceDbContext(typeof(IBookStoreDbContext))]

public class OtherDbContext : AbpDbContext<OtherDbContext>, IBookStoreDbContext

{

//...

}

```

**\*\*ReplaceDbContext option\*\***

````csharp

context.Services.AddAbpDbContext<OtherDbContext>(options =>

{

//...

options.ReplaceDbContext<IBookStoreDbContext>();

});

````

在这个例子中,`OtherDbContext`实现了`IBookStoreDbContext`. 此功能允许你在开发时使用多个DbContext(每个模块一个),但在运行时可以使用单个DbContext(实现所有DbContext的所有接口).

### 拆分查询

ABP 为了更好的性能, 默认全局启用 [拆分查询](https://docs.microsoft.com/en-us/ef/core/querying/single-split-queries). 你可以按需修改.

**\*\*示例\*\***

````csharp

Configure<AbpDbContextOptions>(options =>

{

options.UseSqlServer(optionsBuilder =>

{

optionsBuilder.UseQuerySplittingBehavior(QuerySplittingBehavior.SingleQuery);

});

});

````

### 自定义批量操作

如果你有更好的逻辑或使用外部库实现批量操作, 你可以通过实现 `IEfCoreBulkOperationProvider` 覆写这个逻辑.

- 你可以使用下面的示例模板:

```csharp

public class MyCustomEfCoreBulkOperationProvider

: IEfCoreBulkOperationProvider, ITransientDependency

{

public async Task DeleteManyAsync<TDbContext, TEntity>(

IEfCoreRepository<TEntity> repository,

IEnumerable<TEntity> entities,

bool autoSave,

CancellationToken cancellationToken)

where TDbContext : IEfCoreDbContext

where TEntity : class, IEntity

{

// Your logic here.

}

public async Task InsertManyAsync<TDbContext, TEntity>(

IEfCoreRepository<TEntity> repository,

IEnumerable<TEntity> entities,

bool autoSave,

CancellationToken cancellationToken)

where TDbContext : IEfCoreDbContext

where TEntity : class, IEntity

{

// Your logic here.

}

public async Task UpdateManyAsync<TDbContext, TEntity>(

IEfCoreRepository<TEntity> repository,

IEnumerable<TEntity> entities,

bool autoSave,

CancellationToken cancellationToken)

where TDbContext : IEfCoreDbContext

where TEntity : class, IEntity

{

// Your logic here.

}

}

```

## 另请参阅

\* [实体](Entities.md)

\* [仓储](Repositories.md)

数据库迁移

# EF Core数据库迁移

本文首先介绍[应用程序启动模板](Startup-Templates/Application.md)提供的**\*\*默认结构\*\***,并讨论你可能希望为自己的应用程序实现的**\*\*各种场景\*\***.

> 本文档适用于希望完全理解和自定义[应用程序启动模板](Startup-Templates/Application.md)附带的数据库结构的人员. 如果你只是想创建实体和管理代码优先(code first)迁移,只需要遵循[启动教程](Tutorials/Part-1.md).

### 源码

你可以在[这里](https://github.com/abpframework/abp-samples/tree/master/EfCoreMigrationDemo)找到本文引用的示例项目的源代码; 但是为了理解示例项目的源代码,你需要阅读和理解这个文档.

## 关于EF Core 代码优先迁移

Entity Framework Core 提供了一种简单强大[数据库迁移系统](https://docs.microsoft.com/zh-cn/ef/core/managing-schemas/migrations/). ABP框架[启动模板](Startup-Templates/Index.md)使用这个系统,让你以标准的方式开发你的应用程序.

但是EF Core迁移系统在**\*\*模块化环境中不是很好\*\***,在模块化环境中,每个模块都维护**\*\*自己的数据库模式\*\***,而实际上两个或多个模块可以**\*\*共享一个数据库\*\***.

由于ABP框架在所有方面都关心模块化,所以它为这个问题提供了**\*\*解决方案\*\***. 如果你需要**\*\*自定义数据库结构\*\***,那么应当了解这个解决方案.

> 参阅[EF Core文档](https://docs.microsoft.com/en-us/ef/core/managing-schemas/migrations/)充分了解EF Core Code First迁移,以及为什么需要这样的系统.

## 默认解决方案与数据库配置

当你[创建一个新的Web应用程序](https://abp.io/get-started)(使用EF Core,它是默认的数据库提供程序),你的解决方案结构类似下图:

![bookstore-visual-studio-solution-v3](images/bookstore-visual-studio-solution-v3.png)

实际的解决方案结构可能会根据你的偏好有所不同,但是数据库部分是相同的.

> 本文档将使用 `Acme.BookStore` 示例项目名称来引用项目和类. 你需要在解决方案中找到相应的类/项目.

### 数据库模式

启动模板已预安装了一些[应用程序模块](Modules/Index.md). 解决方案的每一层都有相应的模块包引用. 所以 `.EntityFrameworkCore` 项目含有使用 `EntityFrameworkCore` 模块的Nuget的引用:

![bookstore-efcore-dependencies](images/bookstore-efcore-dependencies.png)

通过这种方式,你可以看到所有的 `.EntityFrameworkCore` 项目下的**\*\*EF Core的依赖\*\***.

> 除了模块引用之外,它还引用了 `Volo.Abp.EntityFrameworkCore.SqlServer` 包,因为启动模板预配置的是Sql Server. 参阅文档了解如何[切换到其它DBMS](Entity-Framework-Core-Other-DBMS.md).

虽然每个模块在设计上有自己的 `DbContext` 类,并且可以使用其自己的**\*\*物理数据库\*\***,但解决方案的配置是使用**\*\*单个共享数据库\*\***如下图所示:

![single-database-usage](images/single-database-usage.png)

这是**\*\*最简单的配置\*\***,适用于大部分的应用程序. `appsettings.json` 文件有名为`Default`**\*\*单个连接字符串\*\***:

````json

"ConnectionStrings": {

"Default": "..."

}

````

所以你有一个**\*\*单一的数据库模式\*\***,其中包含**\*\*共享\*\***此数据库的模块的所有表.

ABP框架的[连接字符串](Connection-Strings.md)系统允许你轻松为所需的模块**\*\*设置不同的连接字符串\*\***:

````json

"ConnectionStrings": {

"Default": "...",

"AbpAuditLogging": "..."

}

````

示例配置告诉ABP框架[审计日志模块](Modules/Audit-Logging.md)应使用第二个连接字符串.

**\*\*然而这仅仅只是开始\*\***. 你还需要创建第二个数据库以及里面审计日志表并使用code frist迁移的方法维护数据库表. 本文档的主要目的之一就是指导你了解这样的数据库分离场景.

#### 模块表

每个模块都使用**\*\*自己的数据库表\*\***. 例如[身份模块](Modules/Identity.md)有一些表来管理系统中的用户和角色.

##### 表前缀

由于所有模块都允许共享一个数据库(这是默认配置),所以模块通常使用**\*\*表前缀\*\***来对自己的表进行分组.

基础模块(如[身份](Modules/Identity.md), [租户管理](Modules/Tenant-Management.md) 和 [审计日志](Modules/Audit-Logging.md))使用 `Abp` 前缀, 其他的模块使用自己的前缀. 如[Identity Server](Modules/IdentityServer.md) 模块使用前缀 `IdentityServer`.

如果你愿意,你可以为你的应用程序的模块**\*\*更改数据库表前缀\*\***.

例:

````csharp

Volo.Abp.IdentityServer.AbpIdentityServerDbProperties.DbTablePrefix = "Ids";

````

这段代码更改了[Identity Server](Modules/IdentityServer.md)的前缀. 在应用程序的**\*\*最开始\*\***编写这段代码.

> 每个模块还定义了 `DbSchema` 属性,你可以在支持schema的数据库中使用它.

### 项目

从数据库的角度来看.有三个重要的项目将在下一节中解释.

#### .EntityFrameworkCore 项目

这个项目有应用程序的 `DbContext`类(本例中的 `BookStoreDbContex` ).

**\*\*每个模块都使用自己的** `DbContext` **类\*\***来访问数据库.同样你的应用程序有它自己的 `DbContext`. 通常在应用程序中使用这个 `DbContet`(如果你遵循最佳实践,应该在[仓储](Repositories.md)中使用). 它几乎是一个空的 `DbContext`,因为你的应用程序在一开始没有任何实体,除了预定义的 `AppUser` 实体:

````csharp

[ConnectionStringName("Default")]

public class BookStoreDbContext : AbpDbContext<BookStoreDbContext>

{

public DbSet<AppUser> Users { get; set; }

/\* Add DbSet properties for your Aggregate Roots / Entities here. \*/

public BookStoreDbContext(DbContextOptions<BookStoreDbContext> options)

: base(options)

{

}

protected override void OnModelCreating(ModelBuilder builder)

{

base.OnModelCreating(builder);

/\* Configure the shared tables (with included modules) here \*/

builder.Entity<AppUser>(b =>

{

//Sharing the same Users table with the IdentityUser

b.ToTable(AbpIdentityDbProperties.DbTablePrefix + "Users");

b.ConfigureByConvention();

b.ConfigureAbpUser();

/\* Configure mappings for your additional properties

\* Also see the MyProjectNameEntityExtensions class

\*/

});

/\* Configure your own tables/entities inside the ConfigureBookStore method \*/

builder.ConfigureBookStore();

}

}

````

这个简单的 `DbContext` 类仍然需要一些解释:

\* 它定义了一个 `[connectionStringName]` Attribute,它告诉ABP始终为此 `Dbcontext` 使用 `Default` 连接字符串.

\* 它从 `AbpDbContext<T>` 而不是标准的 `DbContext` 类继承. 你可以参阅[EF Core集成](Entity-Framework-Core.md)文档了解更多. 现在你需要知道 `AbpDbContext<T>` 基类实现ABP框架的一些约定,为你自动化一些常见的任务.

\* 它为 `AppUser` 实体定义了 `DbSet` 属性. `AppUser` 与[身份模块](Modules/Identity.md)的 `IdentityUser` 实体共享同一个表(默认名为 `AbpUsers`). 启动模板在应用程序中提供这个实体,因为我们认为用户实体一般需要应用程序中进行定制.

\* 构造函数接受一个 `DbContextOptions<T>` 实例.

\* 它覆盖了 `OnModelCreating` 方法定义EF Core 映射.

\* 首先调用 `base.OnModelCreating` 方法让ABP框架为我们实现基础映射.

\* 然后它配置了 `AppUser` 实体的映射. 这个实体有一个特殊的情况(它与Identity模块共享一个表),在下一节中进行解释.

\* 最后它调用 `builder.ConfigureBookStore()` 扩展方法来配置应用程序的其他实体.

在介绍其他数据库相关项目之后,将更详细地说明这个设计.

#### .EntityFrameworkCore.DbMigrations 项目

正如前面所提到的,每个模块(和你的应用程序)有**\*\*它们自己\*\***独立的 `DbContext` 类. 每个 `DbContext` 类只定义了自身模块的实体到表的映射,每个模块(包括你的应用程序)在**\*\*运行时\*\***都使用相关的 `DbContext` 类.

如你所知,EF Core Code First迁移系统依赖于 `DbContext` 类来跟踪和生成Code First迁移. 那么我们应该使用哪个 `DbContext` 进行迁移? 答案是它们都不是. `.EntityFrameworkCore.DbMigrations` 项目中定义了另一个 `DbContext` (示例解决方案中的 `BookStoreMigrationsDbContext`).

##### MigrationsDbContext

`MigrationsDbContext` 仅用于创建和应用数据库迁移. **\*\*不在运行时使用\*\***. 它将所有使用的模块的所有实体到表的映射以及应用程序的映射**\*\*合并\*\***.

通过这种方式你可以创建和维护**\*\*单个数据库迁移路径\*\***. 然而这种方法有一些困难,接下来的章节将解释ABP框架如何克服这些困难. 首先以 `BookStoreMigrationsDbContext` 类为例:

````csharp

/\* This DbContext is only used for database migrations.

\* It is not used on runtime. See BookStoreDbContext for the runtime DbContext.

\* It is a unified model that includes configuration for

\* all used modules and your application.

\*/

public class BookStoreMigrationsDbContext : AbpDbContext<BookStoreMigrationsDbContext>

{

public BookStoreMigrationsDbContext(

DbContextOptions<BookStoreMigrationsDbContext> options)

: base(options)

{

}

protected override void OnModelCreating(ModelBuilder builder)

{

base.OnModelCreating(builder);

/\* Include modules to your migration db context \*/

builder.ConfigurePermissionManagement();

builder.ConfigureSettingManagement();

builder.ConfigureBackgroundJobs();

builder.ConfigureAuditLogging();

builder.ConfigureIdentity();

builder.ConfigureIdentityServer();

builder.ConfigureFeatureManagement();

builder.ConfigureTenantManagement();

/\* Configure your own tables/entities inside the ConfigureBookStore method \*/

builder.ConfigureBookStore();

}

}

````

##### 共享映射代码

第一个问题是: 一个模块使用自己的 `DbContext` 这就需要到数据库的映射. 该 `MigrationsDbContext` 也需要相同的映射创建此模块的数据库表. 我们绝对**\*\*不希望复制\*\***的映射代码.

解决方案是定义一个**\*\*扩展方法\*\***(在`ModelBuilder`)由两个 `DbContext` 类调用. 所以每个模块都定义了这样的扩展方法.

例如 `builder.ConfigureBackgroundJobs()` 方法调用[后台作业模块](Modules/Background-Jobs.md)配置数据库表. 扩展方法的定义如下:

````csharp

public static class BackgroundJobsDbContextModelCreatingExtensions

{

public static void ConfigureBackgroundJobs(

this ModelBuilder builder,

Action<BackgroundJobsModelBuilderConfigurationOptions> optionsAction = null)

{

var options = new BackgroundJobsModelBuilderConfigurationOptions(

BackgroundJobsDbProperties.DbTablePrefix,

BackgroundJobsDbProperties.DbSchema

);

optionsAction?.Invoke(options);

builder.Entity<BackgroundJobRecord>(b =>

{

b.ToTable(options.TablePrefix + "BackgroundJobs", options.Schema);

b.ConfigureCreationTime();

b.ConfigureExtraProperties();

b.Property(x => x.JobName)

.IsRequired()

.HasMaxLength(BackgroundJobRecordConsts.MaxJobNameLength);

//...

});

}

}

````

此扩展方法还提供了选项用于更改此模块的数据库表前缀和模式,但在这里并不重要.

最终的应用程序在 `MigrationsDbContext` 类中调用扩展方法, 因此它可以确定此 `MigrationsDbContext` 维护的数据库中包含哪些模块. 如果要创建第二个数据库并将某些模块表移动到第二个数据库,则需要有第二个`MigrationsDbContext` 类,该类仅调用相关模块的扩展方法. 下一部分将详细介绍该主题.

同样 `ConfigureBackgroundJobs` 方法也被后台作业模块的 `DbContext` 调用:

````csharp

[ConnectionStringName(BackgroundJobsDbProperties.ConnectionStringName)]

public class BackgroundJobsDbContext

: AbpDbContext<BackgroundJobsDbContext>, IBackgroundJobsDbContext

{

public DbSet<BackgroundJobRecord> BackgroundJobs { get; set; }

public BackgroundJobsDbContext(DbContextOptions<BackgroundJobsDbContext> options)

: base(options)

{

}

protected override void OnModelCreating(ModelBuilder builder)

{

base.OnModelCreating(builder);

//Reuse the same extension method!

builder.ConfigureBackgroundJobs();

}

}

````

通过这种方式,可以在 `DbContext` 类之间共享模块的映射配置. 上面的代码在相关的模块NuGet包中,所以你不需要关心它.

##### 重用模块的表

你可能想在应用程序中**\*\*重用依赖模块的表\*\***. 在这种情况下你有两个选择:

1. 你可以**\*\*直接使用模块定义的实体\*\***(你仍然可以在某种程度上[扩展实体](Customizing-Application-Modules-Extending-Entities.md)).

2. 你可以**\*\*创建一个新的实体\*\***映射到同一个数据库表.

###### 使用由模块定义的实体

使用实体定义的模块有标准用法非常简单. 例如身份模块定义了 `IdentityUser` 实体. 你可以为注入 `IdentityUser` 仓储,为此实体执行标准仓储操作.

例:

````csharp

using System;

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.Domain.Repositories;

using Volo.Abp.Identity;

namespace Acme.BookStore

{

public class MyService : ITransientDependency

{

private readonly IRepository<IdentityUser, Guid> \_identityUserRepository;

public MyService(IRepository<IdentityUser, Guid> identityUserRepository)

{

\_identityUserRepository = identityUserRepository;

}

public async Task DoItAsync()

{

//Get all users

var users = await \_identityUserRepository.GetListAsync();

}

}

}

````

示例注入了 `IRepository<IdentityUser,Guid>`(默认仓储). 它定义了标准的存储库方法并实现了 `IQueryable` 接口.

另外,身份模块定义了 `IIdentityUserRepository`(自定义仓储),你的应用程序也可以注入和使用它. `IIdentityUserRepository` 为 `IdentityUser` 实体提供了额外的定制方法,但它没有实现 `IQueryable`.

###### 创建一个新的实体

如果你只是按原样使用实体,那么使用模块的实体是很容易的. 但是在下面的情况下你可能需要定义自己的实体并映射到模块实体相同的数据库表.

\* 你想要**\*\*添加一个新的字段\*\***到表中并将其映射到实体的属性. 你无法使用模块的实体,因为它没有相关的属性.

\* 你想要**\*\*使用的表字段的一个子集\*\***. 你不希望访问实体的所有属性并隐藏不相关的属性(从安全的角度或仅出于设计目的).

\* 你不希望直接**\*\*依赖于\*\***模块实体类.

无论哪一种情况,流程都是一样的. 假如你想要创建一个名为 `AppRole` 的实体,将其映射到与[身份模块](Modules/Identity.md)的 `IdentityRole` 实体相同的表

在这里我们将介绍实现方式, 然后**\*\*将讨论这种方法的局限性\*\***.

首先,在你的 `.Domain` 项目一个新的 `AppRole` 类:

````csharp

using System;

using Volo.Abp.Domain.Entities;

using Volo.Abp.MultiTenancy;

namespace Acme.BookStore.Roles

{

public class AppRole : AggregateRoot<Guid>, IMultiTenant

{

// Properties shared with the IdentityRole class

public Guid? TenantId { get; private set; }

public string Name { get; private set; }

//Additional properties

public string Title { get; set; }

private AppRole()

{

}

}

}

````

\* 它继承了[`AggregateRoot<Guid>`类](Entities.md)和实现了[`IMultiTenant`]接口(Multi-Tenancy.md),因为 `IdentityRole` 也做了同样的继承.

\* 你可以添加 `IdentityRole` 实体定义的任何属性. 本例只加了 `TenantId` 和 `Name` 属性,因为我们这里只需要它们. 你可以把setters设置为私有(如同本例)以防意外更改身份模块的属性.

\* 你可以添加自定义(附加)属性. 本例添加了 `Title` 属性.

\* **\*\*构造函数是私有的\*\***,所以它不允许直接创建一个新的 `AppRole` 实体.创建角色身份模块的责任. 你可以查询角色,设置/更新自定义属性,但做为最佳实践你不应该在代码中创建和删除角色(尽管没有强制的限制).

现在是时候定义EF Core映射. 打开应用程序的 `DbContext` (此示例中是 `BookStoreDbContext` )添加以下属性:

````csharp

public DbSet<AppRole> Roles { get; set; }

````

然后在 `OnModelCreating` 方法中配置映射(调用 `base.OnModelCreating(builder)` 之后):

````csharp

protected override void OnModelCreating(ModelBuilder builder)

{

base.OnModelCreating(builder);

/\* Configure the shared tables (with included modules) here \*/

//CONFIGURE THE AppRole ENTITY

builder.Entity<AppRole>(b =>

{

b.ToTable("AbpRoles");

b.ConfigureByConvention();

b.Property(x => x.Title).HasMaxLength(128);

});

...

/\* Configure your own tables/entities inside the ConfigureBookStore method \*/

builder.ConfigureBookStore();

}

````

我们添加了以下几行:

````csharp

builder.Entity<AppRole>(b =>

{

b.ToTable("AbpRoles");

b.ConfigureByConvention();

b.Property(x => x.Title).HasMaxLength(128);

});

````

\* 它映射到 `AbpRoles` 表,与 `IdentityRole` 实体共享.

\* `ConfigureByConvention()` 配置了标准/基本属性(像`TenantId`),建议总是调用它.

你已经为你的 `DbContext` 配置自定义属性,该属性在应用程序运行时使用.

与其直接更改 `MigrationsDbContext`,我们应该使用ABP框架的实体扩展系统,在解决方案的 `.EntityFrameworkCore` 项目中找到 `YourProjectNameEntityExtensions` 类(本示例中是 `BookStoreEntityExtensions`)并且进行以下更改:

````csharp

public static class MyProjectNameEntityExtensions

{

private static readonly OneTimeRunner OneTimeRunner = new OneTimeRunner();

public static void Configure()

{

OneTimeRunner.Run(() =>

{

ObjectExtensionManager.Instance

.MapEfCoreProperty<IdentityRole, string>(

"Title",

(entityBuilder, propertyBuilder) =>

{

propertyBuilder.HasMaxLength(128);

}

);

});

}

}

````

> 我们建议使用 `nameof(AppRole.Title)` 而不是硬编码 "Title" 字符串

`ObjectExtensionManager` 用于添加属性到现有的实体. 由于 `ObjectExtensionManager.Instance` 是静态实例(单例),因此应调用一次. `OneTimeRunner` 是ABP框架定义简单的工具类.

参阅[EF Core集成文档](Entity-Framework-Core.md)了解更多关于实体扩展系统.

我们在两个类中都重复了类似的数据库映射代码,例如 `HasMaxLength(128)`.

现在你可以在包管理控制台(记得选择 `.EntityFrameworkCore.DbMigrations` 做为PMC的默认项目并将 `.Web` 项目设置为启动项目)使用标准的 `Add-Migration` 命令添加一个新的EF Core数据库迁移.

![pmc-add-migration-role-title](images/pmc-add-migration-role-title.png)

命令将创建一个新的Code First迁移类,如下所示:

````csharp

public partial class Added\_Title\_To\_Roles : Migration

{

protected override void Up(MigrationBuilder migrationBuilder)

{

migrationBuilder.AddColumn<string>(

name: "Title",

table: "AbpRoles",

maxLength: 128,

nullable: true);

}

protected override void Down(MigrationBuilder migrationBuilder)

{

migrationBuilder.DropColumn(

name: "Title",

table: "AbpRoles");

}

}

````

全部完成,只运行在PMC运行`Update-Database` 命令或运行你解决方案中的`.DbMigrator` 项目应用数据库更改.

现在你可以像使用应用程序的其他实体一样使用 `AppRole` 实体. 下面是查询和更新角色的[应用服务](Application-Services.md).

示例:

````csharp

public class AppRoleAppService : ApplicationService, IAppRoleAppService

{

private readonly IRepository<AppRole, Guid> \_appRoleRepository;

public AppRoleAppService(IRepository<AppRole, Guid> appRoleRepository)

{

\_appRoleRepository = appRoleRepository;

}

public async Task<List<AppRoleDto>> GetListAsync()

{

var roles = await \_appRoleRepository.GetListAsync();

return roles

.Select(r => new AppRoleDto

{

Id = r.Id,

Name = r.Name,

Title = r.Title

})

.ToList();

}

public async Task UpdateTitleAsync(Guid id, string title)

{

var role = await \_appRoleRepository.GetAsync(id);

role.Title = title;

await \_appRoleRepository.UpdateAsync(role);

}

}

````

创建新实体并将其映射到依赖模块的表存在一些**\*\*限制\*\***:

\* 你的**\*\*自定义属性必须是可空的\*\***. 例如 `AppRole.Title` 在这里是可空的. 否则身份模块会抛出异常,因为它在向数据库插入新角色时不知道如何如何填充标题.

\* 好的做法是,你不应更新**\*\*模块定义的属性\*\***,尤其是在需要业务逻辑的情况下. 通常你可以管理自己(自定义)的属性.

##### 替代方法

你可以使用以下方法替代创建实体添加自定义属性.

###### 使用ExtraProperties

所有从 `AggregateRoot` 派生的实体都可以在 `ExtraProperties` 属性(因为它们都实现了 `IHasExtraProperties` 接口)中存储键值对, 它是 `Dictionary<string, object>` 类型在数据库中被序列化为JSON. 所以你可以在字典中添加值用于查询,无需更改实体.

例如你可以将查询属性 `Title` 存储在 `IdentityRole` 中,而不是创建一个新的实体.

例:

````csharp

public class IdentityRoleExtendingService : ITransientDependency

{

private readonly IIdentityRoleRepository \_identityRoleRepository;

public IdentityRoleExtendingService(IIdentityRoleRepository identityRoleRepository)

{

\_identityRoleRepository = identityRoleRepository;

}

public async Task<string> GetTitleAsync(Guid id)

{

var role = await \_identityRoleRepository.GetAsync(id);

return role.GetProperty<string>("Title");

}

public async Task SetTitleAsync(Guid id, string newTitle)

{

var role = await \_identityRoleRepository.GetAsync(id);

role.SetProperty("Title", newTitle);

await \_identityRoleRepository.UpdateAsync(role);

}

}

````

\* `GetProperty` 和 `SetProperty` 方法是在 `role.ExtraProperties` 词典中获取和设置值的简化写法,推荐使用.

通过这种方式你可以轻松任何类型的值附加到依赖模块的实体. 不过这种方法也有一些缺点:

\* 所有的额外属性都存储在数据库中的一个**\*\*JSON对象\*\***,它们不是作为表的字段存储,与简单的表字段相比创建索引和针对此属性使用SQL查询将更加困难.

\* 属性名称是字符串,他们**\*\*不是类型安全的\*\***. 建议这些类型的属性定义常量,以防止拼写错误.

###### 使用实体扩展系统

实体扩展系统解决了额外属性主要的问题: 它可以将额外属性做为**\*\*标准表字段\*\***存储到数据库.

你需要做的就是如上所诉使用 `ObjectExtensionManager` 定义额外属性, 然后你就可以使得 `GetProperty` 和 `SetProperty` 方法对实体的属性进行get/set,但是这时它存储在数据库表的单独字段中.

参阅[实体扩展系统](Customizing-Application-Modules-Extending-Entities.md)了解更多.

###### 创建新表

你可以创建**\*\*自己的表\*\***来存储属性,而不是创建新实体并映射到同一表. 你通常复制原始实体的一些值. 例如可以将 `Name` 字段添加到你自己的表中,它是原表中 `Name` 字段的副本.

在这种情况下你不需要处理迁移问题,但是需要处理数据复制问题. 当重复的值发生变化时,你应该在表中同步相同的变化. 你可以使用本地或分布式[事件总线](Event-Bus.md)订阅原始实体的更改事件. 这是根据来自另一个微服务的数据推荐的方法,特别是如果它们有单独的物理数据库(你可以在网络中搜索关于微服务设计的数据共享,这是一个广泛的主题).

#### 讨论另一种场景:每个模块管理自己的迁移路径

上面提到, `.EntityFrameworkCore.DbMigrations` 合并所有模块的所有数据库映射(加上应用程序映射)创建统一的迁移路径.

另一种方法是允许每个模块拥有自己的迁移来维护其数据库表. 虽然它在一开始看起来更模块化,但它有一些重要的缺点:

\* **\*\*EF Core迁移系统依赖于数据库DBMS提供程序\*\***. 例如,如果一个模块为SQL Server创建迁移,那么你就不能使用MySQL使用这个迁移代码,对于模块而言为所有可用的DBMS提供程序维护迁移是不切实际的. 将迁移留给应用程序代码(如本文档中所解释的),你就可以在**\*\*应用程序代码中选择DBMS\*\***.

\* 在模块之间 **\*\*共享表\*\*** 或在应用程序中 **\*\*重用模块的表\*\*** 会更加困难. 因为EF Core迁移系统不能处理它,并会抛出异常,例如"数据库中已存在表XXX".

\* **\*\*自定义/增强\*\*** 映射和生成迁移代码会更加困难.

\* 当你使用多个模块时更难跟踪和**\*\*应用数据库更改\*\***.

## 使用多个数据库

默认的启动模板被组织为所有模块和应用程序使用单个数据库. 但是ABP框架和所有预构建的模块都被设计为**\*\*可以使用多个数据库\*\***. 每个模块可以使用自己的数据库,也可以将模块分组到几个数据库中.

本节将说明如何将审计日志,设置管理和权限管理模块表移至**\*\*第二个数据库\*\***,而其余模块继续使用主("Default")数据库.

结构如下图所示:

![single-database-usage](images/multiple-database-usage.png)

### 更改连接字符串部分

第一步是更改所有 `appsettings.json` 文件的连接字符串部分. 最初它是这样的:

````json

"ConnectionStrings": {

"Default": "Server=localhost;Database=BookStore;Trusted\_Connection=True"

}

````

更改如下所示:

````json

"ConnectionStrings": {

"Default": "Server=localhost;Database=BookStore;Trusted\_Connection=True",

"AbpPermissionManagement": "Server=localhost;Database=BookStore\_SecondDb;Trusted\_Connection=True",

"AbpSettingManagement": "Server=localhost;Database=BookStore\_SecondDb;Trusted\_Connection=True",

"AbpAuditLogging": "Server=localhost;Database=BookStore\_SecondDb;Trusted\_Connection=True"

}

````

为相关模块添加了**\*\*另外三个连接字符串\*\***,以针对 `BookStore\_SecondDb` 数据库(它们都是相同的). 例如 `AbpPermissionManagement` 是权限管理模块的连接字符串.

`AbpPermissionManagement` 是权限管理模块[定义](https://github.com/abpframework/abp/blob/97eaa6ff5a044f503465455c86332e5a277b077a/modules/permission-management/src/Volo.Abp.PermissionManagement.Domain/Volo/Abp/PermissionManagement/AbpPermissionManagementDbProperties.cs#L11)的一个常量. 如果定义值,ABP框架[连接字符串选择系统](Connection-Strings.md)将为权限管理模块选择此连接字符串. 如果不定义它将回退到 `Default` 连接字符串.

### 创建第二个迁移项目

在**\*\*运行时\*\***上定义如上所述的连接字符串就足够了. 但是 `BookStore\_SecondDb` 数据库还不存在. 你需要为相关模块创建数据库表.

就像主数据库我们要使用的EF Core Code Frist迁移系统创建和维护第二个数据库.

一种简单的方法是为第二个迁移 `DbContext` 创建第二个项目(`.csproj`).

在你的解决方案中创建一个名为 `Acme.BookStore.EntityFrameworkCore.DbMigrationsForSecondDb` 的**\*\*类库项目\*\***(如果不喜欢你可以进行其他命名).

`.csproj` 内容应该和以下类似:

````xml

<Project Sdk="Microsoft.NET.Sdk">

<Import Project="..\..\common.props" />

<PropertyGroup>

<TargetFramework>netcoreapp3.1</TargetFramework>

<RootNamespace>Acme.BookStore.DbMigrationsForSecondDb</RootNamespace>

</PropertyGroup>

<ItemGroup>

<ProjectReference Include="..\Acme.BookStore.EntityFrameworkCore\Acme.BookStore.EntityFrameworkCore.csproj" />

</ItemGroup>

<ItemGroup>

<PackageReference Include="Microsoft.EntityFrameworkCore.Design" Version="3.1.0" />

</ItemGroup>

</Project>

````

你可以仅复制和修改原始 `.DbMigrations` 项目的内容. 该项目引用 `.EntityFrameworkCore` 项目. **\*\*唯一的区别\*\***是 `RootNamespace` 值.

从 `.Web` 项目中添加对该项目的引用(否则,EF Core Tool不允许使用`Add-migration`命令).

### 创建第二个DbMigrationDbContext

为迁移创建一个新的 `DbContext` 并调用模块的**\*\*扩展方法\*\***为相关模块配置数据库表:

````csharp

[ConnectionStringName("AbpPermissionManagement")]

public class BookStoreSecondMigrationsDbContext :

AbpDbContext<BookStoreSecondMigrationsDbContext>

{

public BookStoreSecondMigrationsDbContext(

DbContextOptions<BookStoreSecondMigrationsDbContext> options)

: base(options)

{

}

protected override void OnModelCreating(ModelBuilder builder)

{

base.OnModelCreating(builder);

/\* Include modules to your migration db context \*/

builder.ConfigurePermissionManagement();

builder.ConfigureSettingManagement();

builder.ConfigureAuditLogging();

}

}

````

> `[ConnectionStringName(...)` Attribute 在这里非常重,它告诉ABP框架应该为这个 `DbContext` 使用哪个连接字符串. 我们使用了 `AbpPermissionManagement`.

创建一个**\*\*Design Time Db Factory\*\***类, EF Core Tool需要使用它(例如`Add-Migration`和`Update-Database` PCM命令):

````csharp

/\* This class is needed for EF Core console commands

\* (like Add-Migration and Update-Database commands) \*/

public class BookStoreSecondMigrationsDbContextFactory

: IDesignTimeDbContextFactory<BookStoreSecondMigrationsDbContext>

{

public BookStoreSecondMigrationsDbContext CreateDbContext(string[] args)

{

var configuration = BuildConfiguration();

var builder = new DbContextOptionsBuilder<BookStoreSecondMigrationsDbContext>()

.UseSqlServer(configuration.GetConnectionString("AbpPermissionManagement"));

return new BookStoreSecondMigrationsDbContext(builder.Options);

}

private static IConfigurationRoot BuildConfiguration()

{

var builder = new ConfigurationBuilder()

.SetBasePath(Directory.GetCurrentDirectory())

.AddJsonFile("appsettings.json", optional: false);

return builder.Build();

}

}

````

它与 `.EntityFrameworCore.DbMigrations` 项目中的类相似, 除了它使用 `AbpPermissionManagement` 连接字符串.

现在你可以打开包管理器控制台, 选择 `.EntityFrameworkCore.DbMigrationsForSecondDb` 项目为默认项目(确保`.Web`项目是启动项目),运行 `Add-Migration "Initial` 和 `Update-Database` 命令,如下所示:

![pmc-add-migration-initial-update-database](images/pmc-add-migration-initial-update-database.png)

现在你应该有一个包含相关数据库表的新数据库.

![bookstore-second-database](images/bookstore-second-database.png)

### 从主数据库中删除模块

我们 **\*\*创建第二个数据库\*\*** 包含了审计日志,权限管理和设置管理模块表. 我们应该从**\*\*主数据库中删除这些表\*\***,这是很容易的.

S

首先移除从 `MigrationsDbContext` 类以下行(本例中是`BookStoreMigrationsDbContext`):

````csharp

builder.ConfigurePermissionManagement();

builder.ConfigureSettingManagement();

builder.ConfigureAuditLogging();

````

打开包管理器控制台, 选择 `.EntityFrameworkCore.DbMigrations` 为默认的项目(确保`.Web`项目是启动项目)运行以下命令:

````

Add-Migration "Removed\_Audit\_Setting\_Permission\_Modules"

````

此命令将创建一个如下所示的迁移类:

````csharp

public partial class Removed\_Audit\_Setting\_Permission\_Modules : Migration

{

protected override void Up(MigrationBuilder migrationBuilder)

{

migrationBuilder.DropTable(

name: "AbpAuditLogActions");

migrationBuilder.DropTable(

name: "AbpEntityPropertyChanges");

migrationBuilder.DropTable(

name: "AbpPermissionGrants");

migrationBuilder.DropTable(

name: "AbpSettings");

migrationBuilder.DropTable(

name: "AbpEntityChanges");

migrationBuilder.DropTable(

name: "AbpAuditLogs");

}

...

}

````

在这一步需要小心:

\* 如果你是 **\*\*已经在生产的系统\*\***, 那么你应该关心**\*\*数据丢失\*\***,在删除表之前需要将表的内容移动到第二个数据库.

\* 如果你还 **\*\*没有开始\*\*** 你的项目, 你可以考虑 **\*\*删除所有的迁移\*\*** 并且重新生成初始版本,让迁移历史更加清晰.

运行 `Update-Database` 命令从主数据库中删除表.

请注意如果你还没有将其复制到新数据库,那么还会**\*\*删除了一些初始种子数据\*\***(例如授予管理员角色的权限). 如果你运行应用程序可能无法再登录. 解决方案很简单:在解决方案中重新运行 `.DbMigrator` 控制台应用程序,它会为新数据库添加种子数据.

### 自动化第二个数据库模式迁移

`.DbMigrator` 控制台应用程序可以跨多个数据库运行数据库种子代码无需任何其他配置. 但它不能在第二个数据库迁移项目中运行EF Core First迁移. 下面介绍如何配置控制台迁移应用程序以处理两个数据库.

#### 实现 IBookStoreDbSchemaMigrator

`Acme.BookStore.EntityFrameworkCore.DbMigrations` 项目中的 `EntityFrameworkCoreBookStoreDbSchemaMigrator` 类负责为 `BookStoreMigrationsDbContext` 迁移数据库模式. 它应该是这样的:

````csharp

[Dependency(ReplaceServices = true)]

public class EntityFrameworkCoreBookStoreDbSchemaMigrator

: IBookStoreDbSchemaMigrator, ITransientDependency

{

private readonly IServiceProvider \_serviceProvider;

public EntityFrameworkCoreBookStoreDbSchemaMigrator(

IServiceProvider serviceProvider)

{

\_serviceProvider = serviceProvider;

}

public async Task MigrateAsync()

{

/\* We are intentionally resolving the BookStoreMigrationsDbContext

\* from IServiceProvider (instead of directly injecting it)

\* to properly get the connection string of the current tenant in the

\* current scope.

\*/

await \_serviceProvider

.GetRequiredService<BookStoreMigrationsDbContext>()

.Database

.MigrateAsync();

}

}

````

它实现了 `IBookStoreDbSchemaMigrator` 并**\*\*替换现有服务\*\***(见第一行).

删除 `[Dependency(ReplaceServices = true)]`,因为我们有这个接口的两个实现并且希望同时使用它们.

在新的迁移项目(`Acme.BookStore.EntityFrameworkCore.DbMigrationsForSecondDb`)创建它的一个副本,内部使用 `BookStoreSecondMigrationsDbContext` . 示例实现:

````csharp

public class EntityFrameworkCoreSecondBookStoreDbSchemaMigrator

: IBookStoreDbSchemaMigrator, ITransientDependency

{

private readonly IServiceProvider \_serviceProvider;

public EntityFrameworkCoreSecondBookStoreDbSchemaMigrator(

IServiceProvider serviceProvider)

{

\_serviceProvider = serviceProvider;

}

public async Task MigrateAsync()

{

/\* We are intentionally resolving the BookStoreSecondMigrationsDbContext

\* from IServiceProvider (instead of directly injecting it)

\* to properly get the connection string of the current tenant in the

\* current scope.

\*/

await \_serviceProvider

.GetRequiredService<BookStoreSecondMigrationsDbContext>()

.Database

.MigrateAsync();

}

}

````

> 类的名称对于[依赖注入](Dependency-Injection.md)非常重要. 为了可以使用 `IBookStoreDbSchemaMigrator` 引用注入它的结尾应该是 `BookStoreDbSchemaMigrator`.

现在我们有 `IBookStoreDbSchemaMigrator` 接口的两个实现,每个实现负责迁移相关的数据库模式.

#### 为第二个迁移项目定义一个模块类

是时候为第二次迁移(`Acme.BookStore.EntityFrameworkCore.DbMigrationsForSecondDb`)项目定义[模块](Module-Development-Basics.md)类了:

````csharp

[DependsOn(

typeof(BookStoreEntityFrameworkCoreModule)

)]

public class BookStoreEntityFrameworkCoreSecondDbMigrationsModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

context.Services.AddAbpDbContext<BookStoreSecondMigrationsDbContext>();

}

}

````

在 `Acme.BookStore.DbMigrator` 项目中引用 `Acme.BookStore.EntityFrameworkCore.DbMigrationsForSecondDb` 项目并且将 `typeof(BookStoreEntityFrameworkCoreSecondDbMigrationsModule)` 依赖项添加到 `BookStoreDbMigratorModule` 的依赖项列表中. `BookStoreDbMigratorModule` 类应该是这样的:

````csharp

[DependsOn(

typeof(AbpAutofacModule),

typeof(BookStoreEntityFrameworkCoreDbMigrationsModule),

typeof(BookStoreEntityFrameworkCoreSecondDbMigrationsModule), // ADDED THIS!

typeof(BookStoreApplicationContractsModule)

)]

public class BookStoreDbMigratorModule : AbpModule

{

...

}

````

#### 运行数据库迁移程序

你可以运行 `.DbMigrator` 应用程序应用迁移和初始化种子数据. 你可以删除这两个数据库进行测试,然后再次运行 `.DbMigrator` 应用程序,它会创建两个数据库.

## 结论

本文档说明了如何拆分数据库以及管理Entity Framework Core解决方案的数据库迁移. 简而言之,你需要为每个不同的数据库创建一个单独的迁移项目.

切换DMBS

# 切换到EF Core 其它DBMS提供程序

**\*\*[**应用程序启动模板**](Startup-Templates/Application.md)\*\*** 为EF Core预配置了Sql Server提供程序,EF Core支持许多其它DBMS,你可以在基于ABP的应用程序使用它们.

ABP框架为一些常见的DMBS提供了简化配置的**\*\*集成包\*\***,你可以通过以下文档来学习如何**\*\*切换到你喜欢的DBMS\*\***:

\* [MySQL](Entity-Framework-Core-MySQL.md)

\* [PostgreSQL](Entity-Framework-Core-PostgreSQL.md)

\* [Oracle](Entity-Framework-Core-Oracle.md)

\* [SQLite](Entity-Framework-Core-SQLite.md)

你也可以不使用集成包配置DBMS提供程序,虽然总是建议使用集成包(它也使不同模块之间的依赖版本成为标准版本),但是如果没有用于DBMS提供程序的集成包,也可以手动集成.

本文介绍了如何在不使用[MySQL集成包](Entity-Framework-Core-MySQL.md)的情况下切换到MySQL.

## 替换SQL Server依赖

\* 删除 `.EntityFrameworkCore` 项目依赖的 [Volo.Abp.EntityFrameworkCore.SqlServer](https://www.nuget.org/packages/Volo.Abp.EntityFrameworkCore.SqlServer) NuGet 包.

\* 添加 [Pomelo.EntityFrameworkCore.MySql](https://www.nuget.org/packages/Pomelo.EntityFrameworkCore.MySql/) NuGet 包到 `.EntityFrameworkCore` 项目.

## 删除模块依赖项

从 *\*****\*\*YourProjectName\**EntityFrameworkCoreModule\*\*** 类的依赖列表中删除`AbpEntityFrameworkCoreSqlServerModule`.

## 更改UseSqlServer()调用

在*\*YourProjectName\**EntityFrameworkCoreModule类中找到以下代码:

````csharp

Configure<AbpDbContextOptions>(options =>

{

options.UseSqlServer();

});

````

替换成以下代码:

````csharp

Configure<AbpDbContextOptions>(options =>

{

options.Configure(ctx =>

{

if (ctx.ExistingConnection != null)

{

ctx.DbContextOptions.UseMySql(ctx.ExistingConnection);

}

else

{

ctx.DbContextOptions.UseMySql(ctx.ConnectionString);

}

});

});

````

\* 调用的 `UseMySql` 代码是在 Pomelo.EntityFrameworkCore.MySql 包中定义的,方法还有附加选项,如果需要可以使用它.

\* 这段代码首先检查当前请求中是否存在到相同数据库的现有(活动)连接,并在可能的情况下重用它. 这允许在不同的DbContext类型之间共享单个事务. ABP处理其余的事情.

\* 如果没有活动的连接,它将把 `ctx.ConnectionString` 传递给UseMySql(这将创建新的数据库连接). 这里使用 `ctx.ConnectionString` 很重要. 不要传递静态连接字符串(或配置中的连接字符串). 因为ABP在多数据库或[多租户](Multi-Tenancy.md)环境中[动态确定正确的连接字符串](Connection-Strings.md).

## 更改连接连接字符串

MySQL连接字符串与SQL Server连接字符串不同. 所以检查你的解决方案中所有的 `appsettings.json` 文件,更改其中的连接字符串. 有关MySQL连接字符串选项的详细内容请参见[connectionstrings.com](https://www.connectionstrings.com/mysql/).

通常需要更改 `.DbMigrator` 和 `.Web` 项目里面的 `appsettings.json` ,但它取决于你的解决方案结构.

## 更改迁移DbContext Factory

启动模板包含*\*****\*\*YourProjectName\**MigrationsDbContextFactory\*\***类，这是EF Core控制台命令所必须的类（比如[Add-Migration](https://docs.microsoft.com/en-us/ef/ef6/modeling/code-first/migrations/#generating--running-migrations)和[Update-Database](https://docs.microsoft.com/en-us/ef/ef6/modeling/code-first/migrations/#generating--running-migrations)），在切换到MySql数据库时，我们同时也需要修改`DbContextOptionsBuilder`

在 *\*YourProjectName\**MigrationsDbContextFactory 类中找到以下代码：

````csharp

var builder = new DbContextOptionsBuilder<YourProjectNameMigrationsDbContext>()

.UseSqlServer(configuration.GetConnectionString("Default"));

````

将其替换为：

````csharp

var builder = new DbContextOptionsBuilder<YourProjectNameMigrationsDbContext>()

.UseMySql(configuration.GetConnectionString("Default"));

````

如果在创建或执行数据库迁移时遇到任何问题，请参考相关模块文档

## 重新生成迁移

启动模板使用[Entity Framework Core的Code First迁移](https://docs.microsoft.com/zh-cn/ef/core/managing-schemas/migrations/). EF Core迁移取决于所选的DBMS提供程序. 因此更改DBMS提供程序会导致迁移失败.

\* 删除 `.EntityFrameworkCore.DbMigrations` 项目下的Migrations文件夹,并重新生成解决方案.

\* 在包管理控制台中运行 `Add-Migration "Initial"`(在解决方案资源管理器选择 `.DbMigrator` (或 `.Web`) 做为启动项目并且选择 `.EntityFrameworkCore.DbMigrations` 做为默认项目).

这将创建一个配置所有数据库对象(表)的数据库迁移.

运行 `.DbMigrator` 项目创建数据库和初始种子数据.

## 运行应用程序

它已准备就绪, 只需要运行该应用程序与享受编码.

## DBMS限制

不同的DBMS可能存在一些限制, 如字段名称的最大长度, 索引长度等等.

模块可能提供了一些内置的解决方案. 你可以通过`ModelBuilder`来配置它 如: Identity Server模块

```csharp

builder.ConfigureIdentityServer(options =>

{

options.DatabaseProvider = EfCoreDatabaseProvider.MySql;

});

```

v2.9+版本无需手动设置 ([版本历史](https://github.com/abpframework/abp/blob/dev/modules/identityserver/src/Volo.Abp.IdentityServer.EntityFrameworkCore/Volo/Abp/IdentityServer/EntityFrameworkCore/IdentityServerModelBuilderConfigurationOptions.cs))

相关讨论: https://github.com/abpframework/abp/issues/1920

#### 到MySql

# 切换到EF Core MySql提供程序

本文介绍如何将预配置为SqlServer提供程序的 **\*\*[**应用程序启动模板**](Startup-Templates/Application.md)\*\*** 切换到 **\*\*MySql\*\*** 数据库提供程序

## 替换Volo.Abp.EntityFrameworkCore.SqlServer包

解决方案中的 `.EntityFrameworkCore` 项目依赖于 [Volo.Abp.EntityFrameworkCore.SqlServer](https://www.nuget.org/packages/Volo.Abp.EntityFrameworkCore.SqlServer) NuGet包. 删除这个包并且添加相同版本的 [Volo.Abp.EntityFrameworkCore.MySQL](https://www.nuget.org/packages/Volo.Abp.EntityFrameworkCore.MySQL) 包.

## 替换模块依赖项

在 `.EntityFrameworkCore` 项目中找到 **\*\*YourProjectName\*EntityFrameworkCoreModule\*\*** 类, 删除 `DependsOn` attribute 上的`typeof(AbpEntityFrameworkCoreSqlServerModule)`, 添加 `typeof(AbpEntityFrameworkCoreMySQLModule)` (并且替换 `using Volo.Abp.EntityFrameworkCore.SqlServer;` 为 `using Volo.Abp.EntityFrameworkCore.MySQL;`).

## UseMySQL()

查找你的解决方案中 `UseSqlServer()`调用,替换为 `UseMySQL()`. 检查下列文件:

\* `.EntityFrameworkCore` 项目中的*\*YourProjectName\**EntityFrameworkCoreModule.cs.

\* `.EntityFrameworkCore.DbMigrations` 项目中的*\*YourProjectName\**MigrationsDbContextFactory.cs.

> 根据你的解决方案的结构,你可能发现更多需要改变代码的文件.

## 更改连接字符串

MySQL连接字符串与SQL Server连接字符串不同. 所以检查你的解决方案中所有的 `appsettings.json` 文件,更改其中的连接字符串. 有关MySQL连接字符串选项的详细内容请参见[connectionstrings.com](https://www.connectionstrings.com/mysql/).

通常需要更改 `.DbMigrator` 和 `.Web` 项目里面的 `appsettings.json` ,但它取决于你的解决方案结构.

## 重新生成迁移

启动模板使用[Entity Framework Core的Code First迁移](https://docs.microsoft.com/zh-cn/ef/core/managing-schemas/migrations/). EF Core迁移取决于所选的DBMS提供程序. 因此更改DBMS提供程序会导致迁移失败.

\* 删除 `.EntityFrameworkCore.DbMigrations` 项目下的Migrations文件夹,并重新生成解决方案.

\* 在包管理控制台中运行 `Add-Migration "Initial"`(在解决方案资源管理器选择 `.DbMigrator` (或 `.Web`) 做为启动项目并且选择 `.EntityFrameworkCore.DbMigrations` 做为默认项目).

这将创建一个配置所有数据库对象(表)的数据库迁移.

运行 `.DbMigrator` 项目创建数据库和初始种子数据.

## 运行应用程序

它已准备就绪, 只需要运行该应用程序与享受编码.

#### 到PostgreSQL

# 切换到EF Core PostgreSQL提供程序

本文介绍如何将预配置为SqlServer提供程序的 **\*\*[**应用程序启动模板**](Startup-Templates/Application.md)\*\*** 切换到 **\*\*PostgreSQL\*\*** 数据库提供程序

## 替换Volo.Abp.EntityFrameworkCore.SqlServer包

解决方案中的 `.EntityFrameworkCore` 项目依赖于 [Volo.Abp.EntityFrameworkCore.SqlServer](https://www.nuget.org/packages/Volo.Abp.EntityFrameworkCore.SqlServer) NuGet包. 删除这个包并且添加相同版本的 [Volo.Abp.EntityFrameworkCore.PostgreSql](https://www.nuget.org/packages/Volo.Abp.EntityFrameworkCore.PostgreSql) 包.

## 替换模块依赖项

在 `.EntityFrameworkCore` 项目中找到 **\*\*YourProjectName\*EntityFrameworkCoreModule\*\*** 类, 删除 `DependsOn` attribute 上的`typeof(AbpEntityFrameworkCoreSqlServerModule)`, 添加 `typeof(AbpEntityFrameworkCorePostgreSqlModule)` (并且替换 `using Volo.Abp.EntityFrameworkCore.SqlServer;` 为 `using Volo.Abp.EntityFrameworkCore.PostgreSql;`).

## UseNpgsql()

查找你的解决方案中 `UseSqlServer()`调用,替换为 `UseNpgsql()`. 检查下列文件:

\* `.EntityFrameworkCore` 项目中的*\*YourProjectName\**EntityFrameworkCoreModule.cs.

\* `.EntityFrameworkCore.DbMigrations` 项目中的*\*YourProjectName\**MigrationsDbContextFactory.cs.

> 根据你的解决方案的结构,你可能发现更多需要改变代码的文件.

## 更改连接字符串

PostgreSql连接字符串与SQL Server连接字符串不同. 所以检查你的解决方案中所有的 `appsettings.json` 文件,更改其中的连接字符串. 有关PostgreSql连接字符串选项的详细内容请参见[connectionstrings.com](https://www.connectionstrings.com/postgresql/).

通常需要更改 `.DbMigrator` 和 `.Web` 项目里面的 `appsettings.json` ,但它取决于你的解决方案结构.

## 重新生成迁移

启动模板使用[Entity Framework Core的Code First迁移](https://docs.microsoft.com/zh-cn/ef/core/managing-schemas/migrations/). EF Core迁移取决于所选的DBMS提供程序. 因此更改DBMS提供程序会导致迁移失败.

\* 删除 `.EntityFrameworkCore.DbMigrations` 项目下的Migrations文件夹,并重新生成解决方案.

\* 在包管理控制台中运行 `Add-Migration "Initial"`(在解决方案资源管理器选择 `.DbMigrator` (或 `.Web`) 做为启动项目并且选择 `.EntityFrameworkCore.DbMigrations` 做为默认项目).

这将创建一个配置所有数据库对象(表)的数据库迁移.

运行 `.DbMigrator` 项目创建数据库和初始种子数据.

## 运行应用程序

它已准备就绪, 只需要运行该应用程序与享受编码.

#### Oracle

# 切换到EF Core Oracle 提供程序

本文介绍如何将预配置为SqlServer提供程序的 **\*\*[**应用程序启动模板**](Startup-Templates/Application.md)\*\*** 切换到 **\*\*Oracle\*\*** 数据库提供程序

ABP框架提供了两种不同的Oracle包集成. 你可以选择以下其中一个:

\* **\*\*[**Volo.Abp.EntityFrameworkCore.Oracle**](Entity-Framework-Core-Oracle-Official.md)\*\*** 使用官方 & 免费的oracle驱动 ( **\*\*当前处于 beta\*\***).

\* **\*\*[**Volo.Abp.EntityFrameworkCore.Oracle.Devart**](Entity-Framework-Core-Oracle-Devart.md)\*\*** 使用[Devart](https://www.devart.com/)公司提供的商业(付费)驱动.

> 你可以选择一个你想要的包,如果你不知道它们之间的区别,请在网站上进行搜索. ABP框架仅提供集成,不提供第三库类库的支持.

# 切换到EF Core Oracle提供程序

本文介绍如何将预配置为SqlServer提供程序的 **\*\*[**应用程序启动模板**](Startup-Templates/Application.md)\*\*** 切换到 **\*\*Oracle\*\*** 数据库提供程序

> 本文档使用[Devart](https://www.devart.com/dotconnect/oracle/)公司的付费库,因为它是oracle唯一支持EF Core 3.x的库

## 替换Volo.Abp.EntityFrameworkCore.SqlServer包

解决方案中的 `.EntityFrameworkCore` 项目依赖于 [Volo.Abp.EntityFrameworkCore.SqlServer](https://www.nuget.org/packages/Volo.Abp.EntityFrameworkCore.SqlServer) NuGet包. 删除这个包并且添加相同版本的 [Volo.Abp.EntityFrameworkCore.Oracle.Devart](https://www.nuget.org/packages/Volo.Abp.EntityFrameworkCore.Oracle.Devart) 包.

## 替换模块依赖项

在 `.EntityFrameworkCore` 项目中找到 **\*\*YourProjectName\*EntityFrameworkCoreModule\*\*** 类, 删除 `DependsOn` attribute 上的`typeof(AbpEntityFrameworkCoreSqlServerModule)`, 添加 `typeof(AbpEntityFrameworkCoreOracleDevartModule)` (并且替换 `using Volo.Abp.EntityFrameworkCore.SqlServer;` 为 `using Volo.Abp.EntityFrameworkCore.Oracle.Devart;`).

## UseOracle()

Find `UseSqlServer()` calls in your solution, replace with `UseOracle()`. Check the following files:

\* *\*YourProjectName\**EntityFrameworkCoreModule.cs inside the `.EntityFrameworkCore` project.

\* *\*YourProjectName\**MigrationsDbContextFactory.cs inside the `.EntityFrameworkCore.DbMigrations` project.

In the `CreateDbContext()` method of the *\*YourProjectName\**MigrationsDbContextFactory.cs, replace the following code block

查找你的解决方案中 `UseSqlServer()`调用,替换为 `UseOracle()`. 检查下列文件:

\* `.EntityFrameworkCore` 项目中的*\*YourProjectName\**EntityFrameworkCoreModule.cs.

\* `.EntityFrameworkCore.DbMigrations` 项目中的*\*YourProjectName\**MigrationsDbContextFactory.cs.

使用以下代码替换*\*YourProjectName\**MigrationsDbContextFactory.cs中的 `CreateDbContext()` 方法:

```csharp

var builder = new DbContextOptionsBuilder<YourProjectNameMigrationsDbContext>()

.UseSqlServer(configuration.GetConnectionString("Default"));

```

与这个

```csharp

var builder = (DbContextOptionsBuilder<YourProjectNameMigrationsDbContext>)

new DbContextOptionsBuilder<YourProjectNameMigrationsDbContext>().UseOracle

(

configuration.GetConnectionString("Default")

);

```

> 根据你的解决方案的结构,你可能发现更多需要改变代码的文件.

## 更改连接字符串

Oracle连接字符串与SQL Server连接字符串不同. 所以检查你的解决方案中所有的 `appsettings.json` 文件,更改其中的连接字符串. 有关oracle连接字符串选项的详细内容请参见[connectionstrings.com](https://www.connectionstrings.com/oracle/).

通常需要更改 `.DbMigrator` 和 `.Web` 项目里面的 `appsettings.json` ,但它取决于你的解决方案结构.

## 重新生成迁移

启动模板使用[Entity Framework Core的Code First迁移](https://docs.microsoft.com/zh-cn/ef/core/managing-schemas/migrations/). EF Core迁移取决于所选的DBMS提供程序. 因此更改DBMS提供程序会导致迁移失败.

\* 删除 `.EntityFrameworkCore.DbMigrations` 项目下的Migrations文件夹,并重新生成解决方案.

\* 在包管理控制台中运行 `Add-Migration "Initial"`(在解决方案资源管理器选择 `.DbMigrator` (或 `.Web`) 做为启动项目并且选择 `.EntityFrameworkCore.DbMigrations` 做为默认项目).

这将创建一个配置所有数据库对象(表)的数据库迁移.

运行 `.DbMigrator` 项目创建数据库和初始种子数据.

## 运行应用程序

它已准备就绪, 只需要运行该应用程序与享受编码.

# 切换到 EF Core Oracle Devart 提供程序

本文介绍如何将预配置为SqlServer提供程序的 **\*\*[**应用程序启动模板**](Startup-Templates/Application.md)\*\*** 切换到 **\*\*Oracle\*\*** 数据库提供程序

> 本文档使用[Devart](https://www.devart.com/dotconnect/oracle/)公司的付费库,有关其他选项,请参见[文档](Entity-Framework-Core-Oracle.md).

## 替换Volo.Abp.EntityFrameworkCore.SqlServer包

解决方案中的 `.EntityFrameworkCore` 项目依赖于 [Volo.Abp.EntityFrameworkCore.SqlServer](https://www.nuget.org/packages/Volo.Abp.EntityFrameworkCore.SqlServer) NuGet包. 删除这个包并且添加相同版本的[Volo.Abp.EntityFrameworkCore.Oracle.Devart](https://www.nuget.org/packages/Volo.Abp.EntityFrameworkCore.Oracle.Devart) 包.

## 替换模块依赖项

在 `.EntityFrameworkCore` 项目中找到 **\*\*YourProjectName\*EntityFrameworkCoreModule\*\*** 类, 删除 `DependsOn` attribute 上的`typeof(AbpEntityFrameworkCoreSqlServerModule)`, 添加 `typeof(AbpEntityFrameworkCoreOracleDevartModule)` (并且替换 `using Volo.Abp.EntityFrameworkCore.SqlServer;` 为 `using Volo.Abp.EntityFrameworkCore.Oracle.Devart;`).

## UseOracle()

查找你的解决方案中 `UseSqlServer()`调用,替换为 `UseOracle()`. 检查下列文件:

\* `.EntityFrameworkCore` 项目中的*\*YourProjectName\**EntityFrameworkCoreModule.cs.

\* `.EntityFrameworkCore.DbMigrations` 项目中的*\*YourProjectName\**MigrationsDbContextFactory.cs.

找到 *\*YourProjectName\**MigrationsDbContextFactory.cs 的 `CreateDbContext()` 方法,将以下代码块

```csharp

var builder = new DbContextOptionsBuilder<YourProjectNameMigrationsDbContext>()

.UseSqlServer(configuration.GetConnectionString("Default"));

```

替换为:

```csharp

var builder = (DbContextOptionsBuilder<YourProjectNameMigrationsDbContext>)

new DbContextOptionsBuilder<YourProjectNameMigrationsDbContext>().UseOracle

(

configuration.GetConnectionString("Default")

);

```

> 根据你的解决方案的结构,你可能发现更多需要改变代码的文件.

## 更改连接字符串

Oracle连接字符串与SQL Server连接字符串不同. 所以检查你的解决方案中所有的 `appsettings.json` 文件,更改其中的连接字符串. 有关Oracle连接字符串选项的详细内容请参见[connectionstrings.com](https://www.connectionstrings.com/oracle/).

通常需要更改 `.DbMigrator` 和 `.Web` 项目里面的 `appsettings.json` ,但它取决于你的解决方案结构.

## 重新生成迁移

启动模板使用[Entity Framework Core的Code First迁移](https://docs.microsoft.com/zh-cn/ef/core/managing-schemas/migrations/). EF Core迁移取决于所选的DBMS提供程序. 因此更改DBMS提供程序会导致迁移失败.

\* 删除 `.EntityFrameworkCore.DbMigrations` 项目下的Migrations文件夹,并重新生成解决方案.

\* 在包管理控制台中运行 `Add-Migration "Initial"`(在解决方案资源管理器选择 `.DbMigrator` (或 `.Web`) 做为启动项目并且选择 `.EntityFrameworkCore.DbMigrations` 做为默认项目).

这将创建一个配置所有数据库对象(表)的数据库迁移.

运行 `.DbMigrator` 项目创建数据库和初始种子数据.

## 运行应用程序

它已准备就绪, 只需要运行该应用程序与享受编码.

#### 到SQLite

# 切换到EF Core SQLite提供程序

本文介绍如何将预配置为SqlServer提供程序的 **\*\*[**应用程序启动模板**](Startup-Templates/Application.md)\*\*** 切换到 **\*\*SQLite\*\*** 数据库提供程序.

## 替换Volo.Abp.EntityFrameworkCore.SqlServer包

解决方案中的 `.EntityFrameworkCore` 项目依赖于 [Volo.Abp.EntityFrameworkCore.SqlServer](https://www.nuget.org/packages/Volo.Abp.EntityFrameworkCore.SqlServer) NuGet包. 删除这个包并且添加相同版本的 [Volo.Abp.EntityFrameworkCore.SQLite](https://www.nuget.org/packages/Volo.Abp.EntityFrameworkCore.SQLite) 包.

## 替换模块依赖项

在 `.EntityFrameworkCore` 项目中找到 **\*\*YourProjectName\*EntityFrameworkCoreModule\*\*** 类, 删除 `DependsOn` attribute 上的`typeof(AbpEntityFrameworkCoreSqlServerModule)`, 添加 `typeof(AbpEntityFrameworkCoreSqliteModule)` (并且替换 `using Volo.Abp.EntityFrameworkCore.SqlServer;` 为 `using Volo.Abp.EntityFrameworkCore.Sqlite;`).

## UseSqlite()

查找你的解决方案中 `UseSqlServer()`调用,替换为 `UseSqlite()`. 检查下列文件:

\* `.EntityFrameworkCore` 项目中的*\*YourProjectName\**EntityFrameworkCoreModule.cs.

\* `.EntityFrameworkCore.DbMigrations` 项目中的*\*YourProjectName\**MigrationsDbContextFactory.cs.

> 根据你的解决方案的结构,你可能发现更多需要改变代码的文件.

## 更改连接字符串

SQLite连接字符串与SQL Server连接字符串不同. 所以检查你的解决方案中所有的 `appsettings.json` 文件,更改其中的连接字符串. 有关SQLite连接字符串选项的详细内容请参见[connectionstrings.com](https://www.connectionstrings.com/sqlite/).

通常需要更改 `.DbMigrator` 和 `.Web` 项目里面的 `appsettings.json` ,但它取决于你的解决方案结构.

## 重新生成迁移

启动模板使用[Entity Framework Core的Code First迁移](https://docs.microsoft.com/zh-cn/ef/core/managing-schemas/migrations/). EF Core迁移取决于所选的DBMS提供程序. 因此更改DBMS提供程序会导致迁移失败.

\* 删除 `.EntityFrameworkCore.DbMigrations` 项目下的Migrations文件夹,并重新生成解决方案.

\* 在包管理控制台中运行 `Add-Migration "Initial"`(在解决方案资源管理器选择 `.DbMigrator` (或 `.Web`) 做为启动项目并且选择 `.EntityFrameworkCore.DbMigrations` 做为默认项目).

这将创建一个配置所有数据库对象(表)的数据库迁移.

运行 `.DbMigrator` 项目创建数据库和初始种子数据.

## 运行应用程序

它已准备就绪, 只需要运行该应用程序与享受编码.

## MongoDB

## MongoDB 集成

本文会介绍如何将MongoDB集成到基于ABP的应用程序中以及如何配置它

### 安装

集成MongoDB需要用到`Volo.Abp.MongoDB`这个包.将它安装到你的项目中(如果是多层架构,安装到数据层和基础设施层):

```

Install-Package Volo.Abp.MongoDB

```

然后添加 `AbpMongoDbModule` 依赖到你的 [模块](Module-Development-Basics.md)中:

```c#

using Volo.Abp.MongoDB;

using Volo.Abp.Modularity;

namespace MyCompany.MyProject

{

[DependsOn(typeof(AbpMongoDbModule))]

public class MyModule : AbpModule

{

//...

}

}

```

### 创建一个Mongo Db Context

ABP中引入了 **\*\*Mongo Db Context\*\*** 的概念(跟Entity Framework Core的DbContext很像)让使用和配置集合变得更简单.举个例子:

```c#

public class MyDbContext : AbpMongoDbContext

{

public IMongoCollection<Question> Questions => Collection<Question>();

public IMongoCollection<Category> Categories => Collection<Category>();

protected override void CreateModel(IMongoModelBuilder modelBuilder)

{

modelBuilder.Entity<Question>(b =>

{

b.CollectionName = "Questions";

});

}

}

```

\* 继承 `AbpMongoDbContext` 类

\* 为每一个mongo集合添加一个公共的 `IMongoCollection<TEntity>` 属性.ABP默认使用这些属性创建默认的仓储

\* 重写 `CreateModel` 方法,可以在方法中配置集合(如设置集合在数据库中的名字)

### 将 Db Context 注入到依赖注入中

在你的模块中使用 `AddAbpDbContext` 方法将Db Context注入到[依赖注入](Dependency-Injection.md)系统中.

```c#

using Microsoft.Extensions.DependencyInjection;

using Volo.Abp.MongoDB;

using Volo.Abp.Modularity;

namespace MyCompany.MyProject

{

[DependsOn(typeof(AbpMongoDbModule))]

public class MyModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

context.Services.AddMongoDbContext<MyDbContext>();

//...

}

}

}

```

#### 添加默认的仓储

在注入的时候使用 `AddDefaultRepositories()`, ABP就能自动为Db Context中的每一个实体创建[仓储](Repositories.md):

````C#

services.AddMongoDbContext<MyDbContext>(options =>

{

options.AddDefaultRepositories();

});

````

这样就会默认为每一个聚合根实体(继承自AggregateRoot的类)创建一个仓储.如果你也想为其他的实体创建仓储,将 `includeAllEntities` 设置为 `true`就可以了:

```c#

services.AddMongoDbContext<MyDbContext>(options =>

{

options.AddDefaultRepositories(includeAllEntities: true);

});

```

现在可以在你的服务中注入并使用`IRepository<TEntity>` 或 `IQueryableRepository<TEntity>`了.比如你有一个主键类型为`Guid`的`Book`实体:

```csharp

public class Book : AggregateRoot<Guid>

{

public string Name { get; set; }

public BookType Type { get; set; }

}

```

(`BookType`是个枚举)你想在[领域服务](Domain-Services.md)中创建一个`Book`实体:

```csharp

public class BookManager : DomainService

{

private readonly IRepository<Book, Guid> \_bookRepository;

public BookManager(IRepository<Book, Guid> bookRepository) //注入默认的仓储

{

\_bookRepository = bookRepository;

}

public async Task<Book> CreateBook(string name, BookType type)

{

Check.NotNullOrWhiteSpace(name, nameof(name));

var book = new Book

{

Id = GuidGenerator.Create(),

Name = name,

Type = type

};

await \_bookRepository.InsertAsync(book);

//使用仓储中的方法

return book;

}

}

```

这是一个使用`InsertAsync`方法将一个实体插入到数据库的例子.

#### 添加自定义仓储

大多数情况下默认的泛型仓储已经足够用了(因为它们实现了`IQueryable`).然而,你可能需要创建自定义的仓库并添加自己的仓储方法.

比如你想要通过books类型删除书籍.建议像下面这样为你的仓储定义一个接口:

```csharp

public interface IBookRepository : IRepository<Book, Guid>

{

Task DeleteBooksByType(

BookType type,

CancellationToken cancellationToken = default(CancellationToken)

);

}

```

通常你希望从`IRepository`中继承标准的仓储方法.其实,你不必那么做.仓储接口定义在领域层,在数据层/基础设施层实现.([启动模板](https://abp.io/Templates)中的`MongoDB`项目)

实现`IBookRepository`接口的例子:

```csharp

public class BookRepository :

MongoDbRepository<BookStoreMongoDbContext, Book, Guid>,

IBookRepository

{

public BookRepository(IMongoDbContextProvider<BookStoreMongoDbContext> dbContextProvider)

: base(dbContextProvider)

{

}

public async Task DeleteBooksByType(

BookType type,

CancellationToken cancellationToken = default(CancellationToken))

{

await Collection.DeleteManyAsync(

Builders<Book>.Filter.Eq(b => b.Type, type),

cancellationToken

);

}

}

```

现在,就能在需要的时候[注入](Dependency-Injection.md)`IBookRepository`并使用`DeleteBooksByType`方法了.

##### 重写默认的泛型仓储

即使你创建了自定义仓储,你仍然可以注入默认的泛型仓储(本例中的`IRepository<Book, Guid>`).默认的仓储实现不会使用你创建的类.

如果你想用自定义的仓储替换默认的仓储实现,在`AddMongoDbContext`中做:

```csharp

context.Services.AddMongoDbContext<BookStoreMongoDbContext>(options =>

{

options.AddDefaultRepositories();

options.AddRepository<Book, BookRepository>(); //替换 IRepository<Book, Guid>

});

```

当你想**\*\*重写基础仓储方法\*\***时,这一点尤为重要.例如,你想要重写`DeleteAsync`方法,以便更有效的删除实体:

```csharp

public async override Task DeleteAsync(

Guid id,

bool autoSave = false,

CancellationToken cancellationToken = default)

{

//TODO: 自定义实现删除方法

}

```

#### 访问MongoDB API

大多数情况下,你想要将MongoDB API隐藏在仓储后面(这是仓储的主要目的).如果你想在仓储之上访问MongoDB API,你可以使用`GetDatabaseAsync()`, `GetAggregateAsync()` 或`GetCollectionAsync()`方法.例如:

```csharp

public class BookService

{

private readonly IRepository<Book, Guid> \_bookRepository;

public BookService(IRepository<Book, Guid> bookRepository)

{

\_bookRepository = bookRepository;

}

public async Task FooAsync()

{

IMongoDatabase database = await \_bookRepository.GetDatabaseAsync();

IMongoCollection<Book> books = await \_bookRepository.GetCollectionAsync();

IAggregateFluent<Book> bookAggregate = await \_bookRepository.GetAggregateAsync();

}

}

```

> 重要:如果你想访问MongoDB API,你需要在你的项目中引用`Volo.Abp.MongoDB`.这会破坏封装,但在这种情况下,这就是你想要的.

#### 事务

MongoDB在4.0版本开始支持事务, ABP在3.2版本加入了对MongoDb事务的支持. 如果你升级到3.2版本,需要将[MongoDbSchemaMigrator](https://github.com/abpframework/abp/blob/dev/templates/app/aspnet-core/src/MyCompanyName.MyProjectName.MongoDB/MongoDb/MongoDbMyProjectNameDbSchemaMigrator.cs)添加到你的 `.MongoDB` 项目中.

[启动模板](Startup-templates/Index.md)默认在 `.MongoDB` 项目中**\*\*禁用\*\***了工作单元事务. 如果你的MongoDB服务器支持事务,你可以手动启用工作单元的事务:

```csharp

Configure<AbpUnitOfWorkDefaultOptions>(options =>

{

options.TransactionBehavior = UnitOfWorkTransactionBehavior.Enabled;

});

```

#### 高级主题

##### 设置默认的仓储类

默认的泛型仓储默认被`MongoDbRepository`类实现.你可以创建自己的实现并在默认的仓储中使用.

首先,像下面这样定义你的仓储类:

```csharp

public class MyRepositoryBase<TEntity>

: MongoDbRepository<BookStoreMongoDbContext, TEntity>

where TEntity : class, IEntity

{

public MyRepositoryBase(IMongoDbContextProvider<BookStoreMongoDbContext> dbContextProvider)

: base(dbContextProvider)

{

}

}

public class MyRepositoryBase<TEntity, TKey>

: MongoDbRepository<BookStoreMongoDbContext, TEntity, TKey>

where TEntity : class, IEntity<TKey>

{

public MyRepositoryBase(IMongoDbContextProvider<BookStoreMongoDbContext> dbContextProvider)

: base(dbContextProvider)

{

}

}

```

第一个是[复合主键的实体](Entities.md),第二个是只有一个主键的实体.

如果需要重写方法建议继承`MongoDbRepository`类,否则,你需要手动实现所有的仓储方法.

现在,你可以使用`SetDefaultRepositoryClasses`:

```csharp

context.Services.AddMongoDbContext<BookStoreMongoDbContext>(options =>

{

options.SetDefaultRepositoryClasses(

typeof(MyRepositoryBase<,>),

typeof(MyRepositoryBase<>)

);

//...

});

```

##### 为默认的仓储设置基类或接口

如果你的MongoDbContext继承自另一个MongoDbContext或者实现了某个接口,你可以使用这个基类或者接口作为默认仓储的类型.如:

```csharp

public interface IBookStoreMongoDbContext : IAbpMongoDbContext

{

Collection<Book> Books { get; }

}

```

`IBookStoreMongoDbContext`被`BookStoreMongoDbContext`类实现.然后你就可以在`AddDefaultRepositories`中使用:

```csharp

context.Services.AddMongoDbContext<BookStoreMongoDbContext>(options =>

{

options.AddDefaultRepositories<IBookStoreMongoDbContext>();

//...

});

```

现在,你自定义的`BookRepository`类也可以使用`IBookStoreMongoDbContext`接口:

```csharp

public class BookRepository

: MongoDbRepository<IBookStoreMongoDbContext, Book, Guid>,

IBookRepository

{

//...

}

```

为MongoDbContext使用接口的优点就是它可以被另一个实现替换.

##### 替换其他的DbContexts

一旦你正确定义并为MongoDbContext使用了接口,任何其他实现都可以使用以下方法替换它:

**\*\*ReplaceDbContextAttribute\*\***

```csharp

[ReplaceDbContext(typeof(IBookStoreMongoDbContext))]

public class OtherMongoDbContext : AbpMongoDbContext, IBookStoreMongoDbContext

{

//...

}

```

**\*\*ReplaceDbContext option\*\***

```csharp

context.Services.AddMongoDbContext<OtherMongoDbContext>(options =>

{

//...

options.ReplaceDbContext<IBookStoreMongoDbContext>();

});

```

这个例子中,`OtherMongoDbContext`实现了`IBookStoreMongoDbContext`.这个特性允许你在发开的时候使用多个MongoDbContext(每个模块一个),但是运行的时候只能使有一个MongoDbContext(实现所有MongoDbContexts的所有接口)

## Dapper

# Dapper 集成

Dapper 是轻量化的数据库提供者,使用Dapper主要的好处是编写T-SQL查询,它为 `IDbConnection` 接口提供了一些扩展方法.

ABP没有为Dapper封装许多功能, ABP Dapper库在ABP EntityFrameworkCore的基础上提供了 `DapperRepository<TDbContext>` 基类,在其中提供了Dapper需要的`IDbConnection`和`IDbTransaction`属性,这两个属性可以和[工作单元](Unit-Of-Work.md)很好的配合.

## 安装

请先根据[EF Core的集成文档](Entity-Framework-Core.md)安装并配置好EF Core.

`Volo.Abp.Dapper`是Dapper集成的主要[nuget包](https://www.nuget.org/packages/Volo.Abp.Dapper).

将其安装到你的项目中(在分层应用程序中适用于 数据访问/基础设施层):

```shell

Install-Package Volo.Abp.Dapper

```

然后添加 `AbpDapperModule` 模块依赖项(`DependsOn` Attribute) 到 [module](Module-Development-Basics.cn.md)(项目中的Mudole类):

````C#

using Volo.Abp.Dapper;

using Volo.Abp.Modularity;

namespace MyCompany.MyProject

{

[DependsOn(typeof(AbpDapperModule))]

public class MyModule : AbpModule

{

//...

}

}

````

## 实现Dapper仓储

下面的代码实现了`Person`仓储, 它需要EF Core的`DbContext`(MyAppDbContext). 你可以注入`PersonDapperRepository`来调用它的方法.

`DbConnection`和`DbTransaction`来自于`DapperRepository`基类.

```C#

public class PersonDapperRepository : DapperRepository<MyAppDbContext>, ITransientDependency

{

public PersonDapperRepository(IDbContextProvider<MyAppDbContext> dbContextProvider)

: base(dbContextProvider)

{

}

public virtual async Task<List<string>> GetAllPersonNames()

{

return (await DbConnection.QueryAsync<string>("select Name from People", transaction: DbTransaction))

.ToList();

}

public virtual async Task<int> UpdatePersonNames(string name)

{

return await DbConnection.ExecuteAsync("update People set Name = @NewName", new { NewName = name },

DbTransaction);

}

}

```

# 实时

## SignalR集成

# SignalR 集成

> 你可以按照[标准的微软教程](https://docs.microsoft.com/en-us/aspnet/core/tutorials/signal)添加[SignalR](https://docs.microsoft.com/en-us/aspnet/core/signalr/introduction)到你的应用程序,但ABP提供了简化集成的SignalR集成包.

## 安装

### 服务器端

建议使用[ABP CLI](CLI.md)安装包.

#### 使用 ABP CLI

在项目的文件夹(.csproj文件)中打开命令行窗口,然后输入以下命令:

```bash

abp add-package Volo.Abp.AspNetCore.SignalR

```

> 你通常需要将此软件包添加到应用程序的Web或API层,具体取决于你的架构.

#### 手动安装

如果你想手动安装:

1. 添加[Volo.Abp.AspNetCore.SignalR](https://www.nuget.org/packages/Volo.Abp.AspNetCore.SignalR)NuGet包到你的项目:

```

Install-Package Volo.Abp.AspNetCore.SignalR

```

或者使用VisualStudio提供的UI安装

2. 添加 `AbpAspNetCoreSignalRModule` 到你的模块的依赖列表.

```csharp

[DependsOn(

//...other dependencies

typeof(AbpAspNetCoreSignalRModule) //Add the new module dependency

)]

public class YourModule : AbpModule

{

}

```

> 你不需要 `services.AddSignalR()` 和 `app.UseEndpoints(...)`,它们在 `AbpAspNetCoreSignalRModule` 中已经添加了.

### 客户端

客户端安装取决于你的UI框架/客户端类型.

#### ASP.NET Core MVC / Razor Pages UI

在你的Web项目的根文件夹中运行以下命令:

````bash

yarn add @abp/signalr

````

> 需要 [yarn](https://yarnpkg.com/) 环境.

它会添加 `@abp/signalr` 到你的项目中的 `package.json` 依赖项:

````json

{

...

"dependencies": {

...

"@abp/signalr": "~2.7.0"

}

}

````

在你的Web项目的根文件夹中运行 `abp install-libs`:

````bash

abp install-libs

````

它会将SignalR JavaScript文件拷贝到你的项目:

![signal-js-file](images/signal-js-file.png)

最后将以下代码添加到页面/视图中, 添加包含 `signalr.js` 文件:

````xml

@section scripts {

<abp-script type="typeof(SignalRBrowserScriptContributor)" />

}

````

它需要将 `@using Volo.Abp.AspNetCore.Mvc.UI.Packages.SignalR` 添加到你的页面/视图.

> 你可以用标准方式添加 `signalr.js` 文件. 但是使用 `SignalRBrowserScriptContributor` 具有其他好处. 有关详细信息,请参见[客户端程序包管理](UI/AspNetCore/Client-Side-Package-Management.md)和[捆绑和压缩文档](UI/AspNetCore/Bundling-Minification.md).

这就是全部了,你可以在你的页面使用[SignalR JavaScript API](https://docs.microsoft.com/en-us/aspnet/core/signalr/javascript-client).

#### 其他的UI框架/客户端

其他类型的客户端请参考[微软文档](https://docs.microsoft.com/en-us/aspnet/core/signalr/introduction).

## ABP框架集成

本节介绍了使用ABP框架集成包的其他好处.

### Hub 路由与Mapping

ABP自动将所有集线器注册到[依赖注入](Dependency-Injection.md)(做为transient)并映射集线器端点. 因此你不需要使用 `app.UseEndpoints(...)` 即可映射你的集线器.集线器路由(URL)通常是根据你的集线器名称确定.

示例:

````csharp

public class MessagingHub : Hub

{

//...

}

````

`MessagingHub` 集线器的路由为 `/signalr-hubs/messaging`:

\* 添加了标准 `/signalr-hubs/` 前缀.

\* 使用**\*\*驼峰命名\*\***集线器名称,不包含 `Hub` 后缀.

如果你想指定路由,你可以使用 `HubRoute` attribute:

````csharp

[HubRoute("/my-messaging-hub")]

public class MessagingHub : Hub

{

//...

}

````

### AbpHub 基类

你可以从 `AbpHub` 或 `AbpHub<T>` 继承标准的 `Hub` 和 `Hub<T>` 类,它们具有实用的基本属性,如 `CurrentUser`.

示例:

````csharp

public class MessagingHub : AbpHub

{

public async Task SendMessage(string targetUserName, string message)

{

var currentUserName = CurrentUser.UserName; //Access to the current user info

var txt = L["MyText"]; //Localization

}

}

````

> 虽然可以将相同的属性注入到集线器构造函数中,但是这种方式简化了集线器类.

### 手动注册/Mapping

ABP会自动将所有集线器注册到[依赖注入](Dependency-Injection.md)作为**\*\*transient service\*\***. 如果想要禁用集线器类**\*\*自动添加依赖注入\*\***,只需要使用 `DisableConventionalRegistration` attribute. 如果愿意,你仍然可以在模块的 `ConfigureServices` 方法中注册集线器类:

````csharp

context.Services.AddTransient<MessagingHub>();

````

当**\*\*你或ABP\*\***将类注册到依赖注入时,如前几节所述,它会自动映射到端点路由配置. 如果要手动映射集线器类,你可以使用 `DisableAutoHubMap` attribute.

对于手动映射,你有两个选择:

1. 使用 `AbpSignalROptions` 添加map配置(在[模块](Module-Development-Basics.md)的 `ConfigureServices` 方法中),ABP会为集线器执行端点映射:

````csharp

Configure<AbpSignalROptions>(options =>

{

options.Hubs.Add(

new HubConfig(

typeof(MessagingHub), //Hub type

"/my-messaging/route", //Hub route (URL)

hubOptions =>

{

//Additional options

hubOptions.LongPolling.PollTimeout = TimeSpan.FromSeconds(30);

}

)

);

});

````

这是提供其他SignalR选项的好方式.

如果你不想禁用自动集线器map,但仍想执行其他SignalR配置,可以使用 `options.Hubs.AddOrUpdate(...)` 方法:

````csharp

Configure<AbpSignalROptions>(options =>

{

options.Hubs.AddOrUpdate(

typeof(MessagingHub), //Hub type

config => //Additional configuration

{

config.RoutePattern = "/my-messaging-hub"; //override the default route

config.ConfigureActions.Add(hubOptions =>

{

//Additional options

hubOptions.LongPolling.PollTimeout = TimeSpan.FromSeconds(30);

});

}

);

});

````

你可以通过这种方式修改在依赖模块(没有源代码访问权限)中定义的集线器类的选项.

2. 在[模块](Module-Development-Basics.md)的 `OnApplicationInitialization` 方法中更改 `app.UseConfiguredEndpoints`(添加了lambda方法作为参数).

````csharp

app.UseConfiguredEndpoints(endpoints =>

{

endpoints.MapHub<MessagingHub>("/my-messaging-hub", options =>

{

options.LongPolling.PollTimeout = TimeSpan.FromSeconds(30);

});

});

````

### UserIdProvider

ABP实现 `SignalR` 的 `IUserIdProvider` 接口,从ABP框架的 `ICurrentUser` 服务提供当前用户ID(请参阅[当前用户服务](CurrentUser.md)),它将集成到应用程序的身份验证系统中,实现类是 `AbpSignalRUserIdProvider` (如果你想更改/覆盖它).

## 示例应用程序

参阅 [SignalR集成Demo](https://github.com/abpframework/abp-samples/tree/master/SignalRDemo),它有一个简单的聊天页面,可以在(经过身份验证的)用户之间发送消息.

![signalr-demo-chat](images/signalr-demo-chat.png)

## 备注

ABP框架不会更改SignalR. 就像在其他ASP.NET Core应用程序中一样,它也可以在基于ABP框架的应用程序中工作.

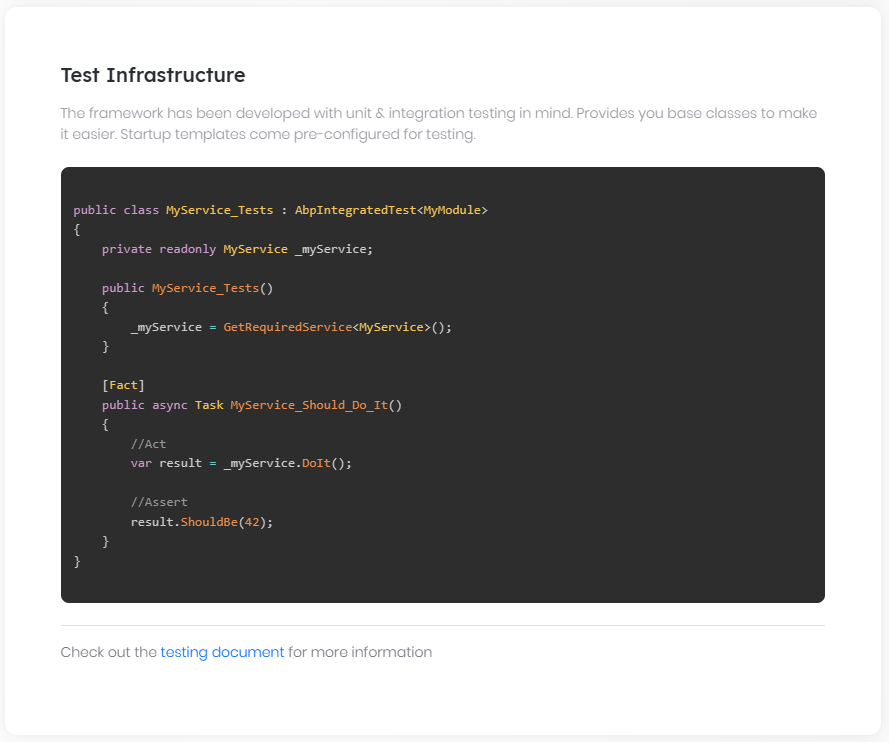
参考[微软文档](https://docs.microsoft.com/zh-cn/aspnet/core/signalr/scale)托管和扩展您的应用程序,集成[Azure](https://docs.microsoft.com/zh-cn/aspnet/core/signalr/publish-to-azure-web-app)或[Redis底版](https://docs.microsoft.com/zh-cn/aspnet/core/signalr/redis-backplane)...等.

## 另请参阅

\* [微软SignalR文档](https://docs.microsoft.com/zh-cn/aspnet/core/signalr/introduction)

\* [使用ABP,SignalR和RabbitMQ在分布式体系结构中的实时消息传递](https://volosoft.com/blog/RealTime-Messaging-Distributed-Architecture-Abp-SingalR-RabbitMQ)

# 测试



# 自动化测试

## 介绍

ABP框架的设计考虑了可测试性. 有一些不同级别的自动化测试:

\* **\*\*单元测试\*\***: 通常只测试一个类(或者一起测试几个类). 这些测试会很快. 然而, 你通常需要处理对服务依赖项的模拟.

\* **\*\*集成测试\*\***: 你通常会测试一个服务, 但这一次你不会模拟基本的基础设施和服务, 以查看它们是否正确地协同工作.

\* **\*\*用户界面测试\*\***: 测试应用程序的UI, 就像用户与应用程序交互一样.

### 单元测试 vs 集成测试

与单元测试相比, 集成测试有一些显著的**\*\*优势\*\***:

\* **\*\*编写更加简单\*\*** 因为你不需要模拟和处理依赖关系.

\* 你的测试代码运行于所有真正的服务和基础设施(包括数据库映射和查询), 因此它更接近于**\*\*真正的应用程序测试\*\***.

同时它们有一些缺点:

\* 与单元测试相比, 它们**\*\*更慢\*\***, 因为所有的基础设施都准备好了测试用例.

\* 服务中的一个bug可能会导致多个测试用例失败, 因此在某些情况下, 可能会**\*\*更难找到真正的问题\*\***.

我们建议混合使用: 在必要的地方编写单元测试或集成测试, 并且有效的编写和维护它.

## 应用程序启动模板

测试基础设施提供[应用程序启动模板](Startup-Templates/Application.md) , 并已经正确安装和配置.

### 测试项目

请参见Visual Studio中的以下解决方案:

![solution-test-projects](images/solution-test-projects.png)

按层级系统分为多个测试项目:

\* `Domain.Tests` 用于测试领域层对象 (例如[领域服务](Domain-Services.md) 和 [实体](Entities.md)).

\* `Application.Tests` 用于测试应用层对象 (例如[应用服务](Application-Services.md)).

\* `EntityFrameworkCore.Tests` 用于测试你的自定义仓储实现或EF Core映射(如果你使用其他[数据访问](Data-Access.md))的话, 该项目将有所不同).

\* `Web.Tests` 用于测试UI层(如页面、控制器和视图组件). 该项目仅适用于MVC / Razor页面应用程序.

\* `TestBase` 包含一些由其他项目共享/使用的类.

> `HttpApi.Client.ConsoleTestApp` 不是自动化测试的应用程序. 它是一个示例的控制台应用程序, 展示了如何从.NET控制台应用程序中调用HTTP API.

以下的部分将介绍这些项目中包含的基类和其他基础设施.

### 测试基础设施

解决方案中已经安装了以下库:

\* [xUnit](https://xunit.net/) 作为测试框架.

\* [NSubstitute](https://nsubstitute.github.io/) 用于模拟.

\* [Shouldly](https://github.com/shouldly/shouldly) 用于断言.

虽然你可以用自己喜欢的工具替换它们, 但本文档和示例将基于这些工具.

## 测试资源管理器

你可以在Visual Studio中使用测试资源管理器查看和运行测试. 其他IDE, 请参阅它们自己的文档.

### 打开测试资源管理器

打开*\*测试\**菜单下的*\*测试资源管理器\**(如果尚未打开):

![vs-test-explorer](images/vs-test-explorer.png)

### 运行测试

然后, 你可以单击在视图中运行所有测试或运行按钮来运行测试. 初始启动模板为你提供了一些测试用例:

![vs-startup-template-tests](images/vs-startup-template-tests.png)

### 并行运行测试

支持并行运行测试. **\*\*强烈建议\*\***并行运行所有测试, 这比逐个运行测试要快得多.

要启用它, 请单击设置(齿轮)按钮附近的插入符号图标, 然后选择*\*并行运行测试\**.

![vs-run-tests-in-parallel](images/vs-run-tests-in-parallel.png)

## 单元测试

对于单元测试, 不需要太多的配置. 通常会实例化你的类, 并对要测试的对象提供一些预先配置的模拟对象.

### 没有依赖项的类

要测试的类没有依赖项是最简单的情况, 你可以直接实例化类, 调用其方法并做出断言.

#### 示例: 测试实体

假设你有一个 `Issue` [实体](Entities.md), 如下所示:

````csharp

using System;

using Volo.Abp.Domain.Entities;

namespace MyProject.Issues

{

public class Issue : AggregateRoot<Guid>

{

public string Title { get; set; }

public string Description { get; set; }

public bool IsLocked { get; set; }

public bool IsClosed { get; private set; }

public DateTime? CloseDate { get; private set; }

public void Close()

{

IsClosed = true;

CloseDate = DateTime.UtcNow;

}

public void Open()

{

if (!IsClosed)

{

return;

}

if (IsLocked)

{

throw new IssueStateException("You can not open a locked issue!");

}

IsClosed = true;

CloseDate = null;

}

}

}

````

请注意, `IsClosed`和`CloseDate`属性具有私有setter, 可以使用`Open()`和`Close()`方法强制执行某些业务逻辑:

\* 无论何时关闭issue, `CloseDate`都应设置为[当前时间](Timing.md).

\* 如果issue被锁定, 则无法重新打开. 如果它被重新打开, `CloseDate`应该设置为`null`.

由于`Issue`实体是领域层的一部分, 所以我们应该在`Domain.Tests`项目中测试它. 在`Domain.Tests`项目中创建一个`Issue\_Tests`类:

````csharp

using Shouldly;

using Xunit;

namespace MyProject.Issues

{

public class Issue\_Tests

{

[Fact]

public void Should\_Set\_The\_CloseDate\_Whenever\_Close\_An\_Issue()

{

// Arrange

var issue = new Issue();

issue.CloseDate.ShouldBeNull(); // null at the beginning

// Act

issue.Close();

// Assert

issue.IsClosed.ShouldBeTrue();

issue.CloseDate.ShouldNotBeNull();

}

}

}

````

这个测试遵循AAA(Arrange-Act-Assert)模式:

\* **\*\*Arrange\*\*** 部分创建一个`Issue`实体, 并确保`CloseDate`在初始值为`null`.

\* **\*\*Act\*\*** 部分执行我们想要测试的方法.

\* **\*\*Assert\*\*** 部分检查`Issue`属性是否与我们预期的相同.

`[Fact]`属性由[xUnit](https://xunit.net/)并将方法标记为测试方法. `Should...`扩展方法由[Shouldly](https://github.com/shouldly/shouldly)提供. 你可以直接使用xUnit中的`Assert`类, 使用Shouldly让它更舒适、更直观.

当你执行测试时, 你将看到它成功通过:

![issue-first-test](images/issue-first-test.png)

让我们再添加两种测试方法:

````csharp

[Fact]

public void Should\_Allow\_To\_ReOpen\_An\_Issue()

{

// Arrange

var issue = new Issue();

issue.Close();

// Act

issue.Open();

// Assert

issue.IsClosed.ShouldBeFalse();

issue.CloseDate.ShouldBeNull();

}

[Fact]

public void Should\_Not\_Allow\_To\_ReOpen\_A\_Locked\_Issue()

{

// Arrange

var issue = new Issue();

issue.Close();

issue.IsLocked = true;

// Act & Assert

Assert.Throws<IssueStateException>(() =>

{

issue.Open();

});

}

````

`Assert.Throws` 检查执行的代码是否匹配引发的异常.

> 有关这些库的更多信息, 请参阅xUnit & Shoudly的文档.

### 具有依赖项的类

如果你的服务中有依赖项, 并且你想对该服务进行单元测试, 那么你需要模拟这些依赖项.

#### 示例: 测试领域服务

假设你有一个`IssueManager` [领域服务](Domain-Services.md), 定义如下:

````csharp

using System;

using System.Threading.Tasks;

using Volo.Abp;

using Volo.Abp.Domain.Services;

namespace MyProject.Issues

{

public class IssueManager : DomainService

{

public const int MaxAllowedOpenIssueCountForAUser = 3;

private readonly IIssueRepository \_issueRepository;

public IssueManager(IIssueRepository issueRepository)

{

\_issueRepository = issueRepository;

}

public async Task AssignToUserAsync(Issue issue, Guid userId)

{

var issueCount = await \_issueRepository.GetIssueCountOfUserAsync(userId);

if (issueCount >= MaxAllowedOpenIssueCountForAUser)

{

throw new BusinessException(

code: "IM:00392",

message: $"You can not assign more" +

$"than {MaxAllowedOpenIssueCountForAUser} issues to a user!"

);

}

issue.AssignedUserId = userId;

}

}

}

````

`IssueManager`依赖于`IssueRepository`服务, 在本例中将模拟该服务.

**\*\*业务逻辑\*\***: 示例`AssignToUserAsync`不允许向用户分配超过3个issue (`MaxAllowedOpenIssueCountForAUser`常量). 在这种情况下, 如果要分配issue, 首先需要取消现有issue的分配.

下面的测试用例给出一个有效的赋值:

````csharp

using System;

using System.Threading.Tasks;

using NSubstitute;

using Shouldly;

using Volo.Abp;

using Xunit;

namespace MyProject.Issues

{

public class IssueManager\_Tests

{

[Fact]

public async Task Should\_Assign\_An\_Issue\_To\_A\_User()

{

// Arrange

var userId = Guid.NewGuid();

var fakeRepo = Substitute.For<IIssueRepository>();

fakeRepo.GetIssueCountOfUserAsync(userId).Returns(1);

var issueManager = new IssueManager(fakeRepo);

var issue = new Issue();

// Act

await issueManager.AssignToUserAsync(issue, userId);

//Assert

issue.AssignedUserId.ShouldBe(userId);

await fakeRepo.Received(1).GetIssueCountOfUserAsync(userId);

}

}

}

````

\* `Substitute.For<IIssueRepository>` 创建一个模拟(假)对象, 该对象被传递到`IssueManager`构造函数中.

\* `fakeRepo.GetIssueCountOfUserAsync(userId).Returns(1)` 确保仓储中的`GetIssueContofuseRasync`方法返回`1`.

\* `issueManager.AssignToUserAsync` 不会引发任何异常, 因为仓储统计当前分配的issue数量并且返回`1`.

\* `issue.AssignedUserId.ShouldBe(userId);` 行检查`AssignedUserId`的值是否正确.

\* `await fakeRepo.Received(1).GetIssueCountOfUserAsync(userId);` 检查 `IssueManager` 实际只调用了 `GetIssueCountOfUserAsync` 方法一次.

让我们添加第二个测试, 看看它是否能阻止将issue分配给超过分配数量的用户:

````csharp

[Fact]

public async Task Should\_Not\_Allow\_To\_Assign\_Issues\_Over\_The\_Limit()

{

// Arrange

var userId = Guid.NewGuid();

var fakeRepo = Substitute.For<IIssueRepository>();

fakeRepo

.GetIssueCountOfUserAsync(userId)

.Returns(IssueManager.MaxAllowedOpenIssueCountForAUser);

var issueManager = new IssueManager(fakeRepo);

// Act & Assert

var issue = new Issue();

await Assert.ThrowsAsync<BusinessException>(async () =>

{

await issueManager.AssignToUserAsync(issue, userId);

});

issue.AssignedUserId.ShouldBeNull();

await fakeRepo.Received(1).GetIssueCountOfUserAsync(userId);

}

````

> 有关模拟的更多信息, 请参阅[NSubstitute](https://nsubstitute.github.io/)文档.

模拟单个依赖项相对容易. 但是, 当依赖关系增长时, 设置测试对象和模拟所有依赖关系变得越来越困难. 请参阅不需要模拟依赖项的*\*Integration Tests\**部分.

### 提示: 共享测试类构造函数

[xUnit](https://xunit.net/) 为每个测试方法创建一个**\*\*新测试类实例\*\***(本例中为`IssueManager\_Tests`). 因此, 你可以将一些*\*Arrange\**代码移动到构造函数中, 以减少代码重复. 构造函数将针对每个测试用例执行, 并且不会相互影响, 即使它们是并行工作.

**\*\*示例: 重构**`IssueManager\_Tests`**以减少代码重复\*\***

````csharp

using System;

using System.Threading.Tasks;

using NSubstitute;

using Shouldly;

using Volo.Abp;

using Xunit;

namespace MyProject.Issues

{

public class IssueManager\_Tests

{

private readonly Guid \_userId;

private readonly IIssueRepository \_fakeRepo;

private readonly IssueManager \_issueManager;

private readonly Issue \_issue;

public IssueManager\_Tests()

{

\_userId = Guid.NewGuid();

\_fakeRepo = Substitute.For<IIssueRepository>();

\_issueManager = new IssueManager(\_fakeRepo);

\_issue = new Issue();

}

[Fact]

public async Task Should\_Assign\_An\_Issue\_To\_A\_User()

{

// Arrange

\_fakeRepo.GetIssueCountOfUserAsync(\_userId).Returns(1);

// Act

await \_issueManager.AssignToUserAsync(\_issue, \_userId);

//Assert

\_issue.AssignedUserId.ShouldBe(\_userId);

await \_fakeRepo.Received(1).GetIssueCountOfUserAsync(\_userId);

}

[Fact]

public async Task Should\_Not\_Allow\_To\_Assign\_Issues\_Over\_The\_Limit()

{

// Arrange

\_fakeRepo

.GetIssueCountOfUserAsync(\_userId)

.Returns(IssueManager.MaxAllowedOpenIssueCountForAUser);

// Act & Assert

await Assert.ThrowsAsync<BusinessException>(async () =>

{

await \_issueManager.AssignToUserAsync(\_issue, \_userId);

});

\_issue.AssignedUserId.ShouldBeNull();

await \_fakeRepo.Received(1).GetIssueCountOfUserAsync(\_userId);

}

}

}

````

> 保持测试代码整洁, 以创建可维护的测试组件.

## 集成测试

> 你还可以按照[Web应用程序开发教程](Tutorials/Part-1.md)学习开发全栈应用程序, 包括集成测试.

### 集成测试基础

ABP为编写集成测试提供了完整的基础设施. 所有ABP基础设施和服务都将在你的测试中执行. 应用程序启动模板附带了为你预先配置的必要基础设施;

#### 数据库

启动模板使用EF Core配置**\*\*内存中的SQLite\*\***数据库(对于MongoDB, 它使用[Mongo2Go](https://github.com/Mongo2Go/Mongo2Go)). 因此, 所有配置和查询都是针对真实数据库执行的, 你甚至可以测试数据库事务.

使用内存中的SQLite数据库有两个主要优点:

\* 它比外部DBMS更快.

\* 它会为每个测试用例创建一个**\*\*新的数据库\*\***, 这样测试就不会相互影响.

> **\*\*提示\*\***: 不要将EF Core的内存数据库用于高级集成测试. 它不是一个真正的DBMS, 在细节上有很多不同. 例如, 它不支持事务和回滚场景, 因此无法真正测试失败的场景. 另一方面, 内存中的SQLite是一个真正的DBMS, 支持SQL数据库的基本功能.

### 种子数据

针对空数据库编写测试是不现实的. 在大多数情况下, 需要在数据库中保存一些初始数据. 例如, 如果你编写了一个查询、更新和删除产品的测试类, 那么在执行测试用例之前, 在数据库中有一些产品数据会很有帮助.

ABP的[种子数据](Data-Seeding.md)系统是一种强大的初始化数据的方法. 应用程序启动模板在`.TestBase`项目中有一个*\*YourProject\**TestDataSeedContributor类. 你可以在其中添加, 以获得可用于每个测试方法的初始数据.

**\*\*示例: 创建一些Issue作为种子数据\*\***

````csharp

using System.Threading.Tasks;

using MyProject.Issues;

using Volo.Abp.Data;

using Volo.Abp.DependencyInjection;

namespace MyProject

{

public class MyProjectTestDataSeedContributor

: IDataSeedContributor, ITransientDependency

{

private readonly IIssueRepository \_issueRepository;

public MyProjectTestDataSeedContributor(IIssueRepository issueRepository)

{

\_issueRepository = issueRepository;

}

public async Task SeedAsync(DataSeedContext context)

{

await \_issueRepository.InsertAsync(

new Issue

{

Title = "Test issue one",

Description = "Test issue one description",

AssignedUserId = TestData.User1Id

});

await \_issueRepository.InsertAsync(

new Issue

{

Title = "Test issue two",

Description = "Test issue two description",

AssignedUserId = TestData.User1Id

});

await \_issueRepository.InsertAsync(

new Issue

{

Title = "Test issue three",

Description = "Test issue three description",

AssignedUserId = TestData.User1Id

});

await \_issueRepository.InsertAsync(

new Issue

{

Title = "Test issue four",

Description = "Test issue four description",

AssignedUserId = TestData.User2Id

});

}

}

}

````

还创建了一个静态类来存储用户的 `Id`:

````csharp

using System;

namespace MyProject

{

public static class TestData

{

public static Guid User1Id = Guid.Parse("41951813-5CF9-4204-8B18-CD765DBCBC9B");

public static Guid User2Id = Guid.Parse("2DAB4460-C21B-4925-BF41-A52750A9B999");

}

}

````

通过这种方式, 我们可以使用这些已知Issue和用户的`Id`来运行测试.

### 示例: 测试领域服务

`AbpIntegratedTest<T>`类 (定义在[Volo.Abp.TestBase](https://www.nuget.org/packages/Volo.Abp.TestBase)) 用于编写集成到ABP框架的测试. `T`是用于设置和初始化应用程序的根模块的类型.

应用程序启动模板在每个测试项目中都有基类, 因此你可以从这些基类派生, 以使其更简单.

`IssueManager`测试将被重写成集成测试

````csharp

using System.Threading.Tasks;

using Shouldly;

using Volo.Abp;

using Xunit;

namespace MyProject.Issues

{

public class IssueManager\_Integration\_Tests : MyProjectDomainTestBase

{

private readonly IssueManager \_issueManager;

private readonly Issue \_issue;

public IssueManager\_Integration\_Tests()

{

\_issueManager = GetRequiredService<IssueManager>();

\_issue = new Issue

{

Title = "Test title",

Description = "Test description"

};

}

[Fact]

public async Task Should\_Not\_Allow\_To\_Assign\_Issues\_Over\_The\_Limit()

{

// Act & Assert

await Assert.ThrowsAsync<BusinessException>(async () =>

{

await \_issueManager.AssignToUserAsync(\_issue, TestData.User1Id);

});

\_issue.AssignedUserId.ShouldBeNull();

}

[Fact]

public async Task Should\_Assign\_An\_Issue\_To\_A\_User()

{

// Act

await \_issueManager.AssignToUserAsync(\_issue, TestData.User2Id);

//Assert

\_issue.AssignedUserId.ShouldBe(TestData.User2Id);

}

}

}

````

\* 第一个测试方法将issue分配给User1, 其中User1已经分配了种子数据代码中的3个issue. 因此, 它抛出了一个`BusinessException`.

\* 第二种测试方法将issue分配给User2, User2只分配了一个issue. 因此, 该方法成功了.

这个类通常位于`.Domain.Tests`项目中, 因为它测试位于`.Domain`项目中的类. 它派生自`MyProjectDomainTestBase`, 并已经为正确运行测试进行了配置.

编写这样一个集成测试类非常简单. 另一个好处是, 在以后向`IssueManager`类添加另一个依赖项时, 不需要更改测试类.

### 示例: 测试应用服务

测试[应用服务](Application-Services.md)并没有太大的不同. 假设你已经创建了一个`IssueAppService`, 定义如下:

````csharp

using System.Collections.Generic;

using System.Threading.Tasks;

using Volo.Abp.Application.Services;

namespace MyProject.Issues

{

public class IssueAppService : ApplicationService, IIssueAppService

{

private readonly IIssueRepository \_issueRepository;

public IssueAppService(IIssueRepository issueRepository)

{

\_issueRepository = issueRepository;

}

public async Task<List<IssueDto>> GetListAsync()

{

var issues = await \_issueRepository.GetListAsync();

return ObjectMapper.Map<List<Issue>, List<IssueDto>>(issues);

}

}

}

````

*\*(假设你还定义了`IIssueAppService`和`IssueDto`, 并在`Issue`和`IssueDto`之间创建了[对象映射](Object-To-Object-Mapping.md))\**

现在, 你可以在`.Application.Tests`项目中编写一个测试类:

````csharp

using System.Threading.Tasks;

using Shouldly;

using Xunit;

namespace MyProject.Issues

{

public class IssueAppService\_Tests : MyProjectApplicationTestBase

{

private readonly IIssueAppService \_issueAppService;

public IssueAppService\_Tests()

{

\_issueAppService = GetRequiredService<IIssueAppService>();

}

[Fact]

public async Task Should\_Get\_All\_Issues()

{

//Act

var issueDtos = await \_issueAppService.GetListAsync();

//Assert

issueDtos.Count.ShouldBeGreaterThan(0);

}

}

}

````

就这么简单. 此测试方法测试的所有内容, 包括应用服务、EF Core映射、对象到对象映射和仓储实现. 通过这种方式, 你可以完全测试解决方案的应用层和领域层.

### 处理集成测试中的工作单元

ABP的[工作单元](Unit-Of-Work.md)系统控制应用程序中的数据库连接和事务管理. 它可以在你编写应用程序代码时无缝工作, 因此你可能没有意识到它.

在ABP框架中, 所有数据库操作都必须在一个工作单元作用域内执行. 当你测试[应用服务](Application-Services.md)方法时, 工作单元的作用域将是应用服务方法的作用域. 如果你正在测试[仓储](Repositories.md)方法, 那么工作单元作用域将是你的仓储方法的作用域.

在某些情况下, 你可能需要手动控制工作单元作用域. 可以考虑下面的测试方法:

````csharp

public class IssueRepository\_Tests : MyProjectDomainTestBase

{

private readonly IRepository<Issue, Guid> \_issueRepository;

public IssueRepository\_Tests()

{

\_issueRepository = GetRequiredService<IRepository<Issue, Guid>>();

}

public async Task Should\_Query\_By\_Title()

{

IQueryable<Issue> queryable = await \_issueRepository.GetQueryableAsync();

var issue = queryable.FirstOrDefaultAsync(i => i.Title == "My issue title");

issue.ShouldNotBeNull();

}

}

````

我们正在使用`\_issueRepository.GetQueryableAsync`获取`IQueryable<Issue>` 对象. 然后, 我们使用`FirstOrDefaultAsync`方法按标题查询issue. 此时执行数据库查询, 你将会得到一个异常, 表明没有起作用的工作单元.

要使该测试正常工作, 你应该手动启动工作单元作用域, 如下所示:

````csharp

public class IssueRepository\_Tests : MyProjectDomainTestBase

{

private readonly IRepository<Issue, Guid> \_issueRepository;

private readonly IUnitOfWorkManager \_unitOfWorkManager;

public IssueRepository\_Tests()

{

\_issueRepository = GetRequiredService<IRepository<Issue, Guid>>();

\_unitOfWorkManager = GetRequiredService<IUnitOfWorkManager>();

}

public async Task Should\_Query\_By\_Title()

{

using (var uow = \_unitOfWorkManager.Begin())

{

IQueryable<Issue> queryable = await \_issueRepository.GetQueryableAsync();

var issue = queryable.FirstOrDefaultAsync(i => i.Title == "My issue title");

issue.ShouldNotBeNull();

await uow.CompleteAsync();

}

}

}

````

我们已经使用了`IUnitOfWorkManager`服务来创建一个工作单元作用域, 然后在该作用域内调用了`FirstOrDefaultAsync`方法, 所以不再有问题了.

> 请注意, 我们测试了`FirstOrDefaultAsync`来演示工作单元的问题. 作为一个好的标准, 编写自己的代码.

### 使用DbContext

在某些情况下, 你可能希望使用Entity Framework的`DbContext`对象来执行测试方法中的数据库操作. 在这种情况下, 可以使用`IDbContextProvider<T>`服务在工作单元内获取`DbContext`实例.

下面的示例展示了如何在测试方法中创建`DbContext`对象:

````csharp

public class MyDbContext\_Tests : MyProjectDomainTestBase

{

private readonly IDbContextProvider<MyProjectDbContext> \_dbContextProvider;

private readonly IUnitOfWorkManager \_unitOfWorkManager;

public IssueRepository\_Tests()

{

\_dbContextProvider = GetRequiredService<IDbContextProvider<MyProjectDbContext>>();

\_unitOfWorkManager = GetRequiredService<IUnitOfWorkManager>();

}

public async Task Should\_Query\_By\_Title()

{

using (var uow = \_unitOfWorkManager.Begin())

{

var dbContext = await \_dbContextProvider.GetDbContextAsync();

var issue = await dbContext.Issues.FirstOrDefaultAsync(i => i.Title == "My issue title");

issue.ShouldNotBeNull();

await uow.CompleteAsync();

}

}

}

````

就像我们在*\*集成测试中处理工作单元\**一节中所做的那样, 我们应该在起作用的工作单元内执行`DbContext`操作.

对于[MongoDB](MongoDB.md), 你可以使用`IMongoDbContextProvider<T>`服务获取`DbContext`对象, 并在测试方法中直接使用MongoDB APIs.

## 用户界面测试

一般来说, 有两种类型的UI测试:

### 非可视化测试

此类测试完全取决于UI框架的选择:

\* 对于MVC / Razor页面UI, 通常向服务器发出请求, 获取HTML, 并测试返回的结果中是否存在一些预期的DOM元素.

\* Angular有自己的基础设施和实践来测试组件、视图和服务.

请参阅以下文档以了解非可视化UI测试:

\* [Testing in ASP.NET Core MVC / Razor Pages](UI/AspNetCore/Testing.md)

\* [Testing in Angular](UI/Angular/Testing.md)

\* [Testing in Blazor](UI/Blazor/Testing.md)

### 可视化测试

与真实用户一样, 可视化测试用于与应用程序UI交互. 它全面测试应用程序, 包括页面和组件的外观.

可视化UI测试超出了ABP框架的范围. 行业中有很多工具(比如[Selenium](https://www.selenium.dev/))可以用来测试应用程序的UI.

# 部署

# 部署

部署ABP应用程序与部署其他.NET或ASP.NET Core应用程序并没有什么不同. 你可以将其部署到云服务提供商(例如Azure、AWS、Google)或内部部署服务器、IIS或任何其他web服务器. ABP的文档中没有太多关于部署的信息. 你可以参考提供商的文档.

但是, 在部署应用程序时, 有些主题是你应该注意的. 其中大多数是一般的软件部署注意事项, 但你应该了解如何在基于ABP的应用程序中处理它们. 我们为此准备了指南, 建议你在设计部署配置之前仔细阅读这些指南.

## 指南

\* [部署到集群环境](Clustered-Environment.md): 讲解了当你希望同时运行应用程序的多个实例时, 如何来配置应用程序.

## 部署到集群环境

# 部署到集群环境

本文档介绍了在将应用程序部署到**\*\*多个应用程序实例同时运行\*\***的集群环境中时应注意的内容, 并解释了如何在基于ABP的应用程序中处理这些内容.

> 无论你使用的是单体式应用程序还是微服务解决方案, 本文档均有效. 适用于一个流程. 应用程序可以是单体式web应用程序、微服务解决方案中的服务、控制台应用程序或其他类型的可执行进程.

>

> 例如, 如果你将应用程序部署到Kubernetes并把应用程序或服务在多个POD中运行, 那么应用程序或服务将在集群环境中运行.

## 了解集群环境

> 如果你已经熟悉集群部署和负载均衡器, 可以跳过本节.

### 单实例部署

考虑作为**\*\*单个实例\*\***部署的应用程序, 如下图所示:

![deployment-single-instance](../images/deployment-single-instance.png)

浏览器和其他客户端应用程序可以直接向应用程序发出HTTP请求. 你可以在客户端和应用程序之间放置一个web服务器(例如IIS或NGINX), 但仍有一个应用程序实例在单个服务器或容器中运行. 单实例的配置**\*\*限于规模\*\***, 因为它在一台服务器上运行, 并且你受到服务器容量的限制.

### 集群部署

**\*\*集群部署\*\***是在一台或多台服务器上**\*\*同时运行\*\***应用程序**\*\*多个实例\*\***的方式. 通过这种方式, 不同的实例可以满足不同的请求, 并且可以通过在系统中添加新服务器来扩展. 下图显示了集群使用**\*\*负载均衡器\*\***的典型实现:

![deployment-clustered](../images/deployment-clustered.png)

### 负载均衡器

[负载均衡器](https://en.wikipedia.org/wiki/Load\_balancing\_(computing)) 有很多特性, 但它们基本上会将**\*\*传入的HTTP请求转发\*\***给应用程序的实例, 并将响应返回给客户端应用程序.

负载平衡器可以使用不同的算法来选择应用程序实例, 同时确定用于传递传入请求的应用程序实例. **\*\*循环\*\***是最简单、最常用的算法之一. 请求被轮流传递到应用程序实例. 第一个实例得到第一个请求, 第二个实例得到第二个请求, 依此类推. 在所有实例都被使用之后, 它返回到第一个实例, 并且下一个请求的算法也是类似的.

### 潜在问题

一旦应用程序的多个实例并行运行, 你应该仔细考虑以下内容:

\* 当你有多个实例时, 存储在应用程序 **\*\*内存中的任何状态(数据)\*\*** 都将成为问题. 存储在应用程序实例内存中的状态可能在下一个请求中不可用, 因为下一个请求将由不同的应用程序实例处理. 虽然有一些解决方案(比如粘性会话)可以解决这个问题, 但如果你想在集群、容器或云中运行应用程序, **\*\*最好将其设计为无状态\*\***.

\* **\*\*内存缓存\*\*** 是一种内存状态, 不应在集群应用程序中使用. 你应该使用**\*\*分布式缓存\*\***.

\* 你不应该在**\*\*本地文件系统\*\***中存储应用程序所有实例都可以使用的数据. 不同的应用程序实例可能在不同的容器或服务器中运行, 并且它们可能无法访问同一个文件系统. 你可以使用**\*\*云或外部存储提供商\*\***作为解决方案.

\* 如果你有**\*\*后台工作者\*\***或**\*\*作业队列管理器\*\***, 则应小心, 因为多个实例可能会尝试执行同一作业或同时执行同一工作. 因此, 你可能会多次完成相同的工作, 或者在尝试访问和更改相同的资源时可能会出现很多错误.

集群部署可能会有更多问题, 但这些是最常见的问题. ABP被设计为与集群部署场景兼容. 以下各节介绍了将基于ABP的应用程序部署到集群环境时应执行的操作.

## 切换分布式缓存

ASP.NET Core提供了不同类型的缓存功能. [内存缓存](https://docs.microsoft.com/en-us/aspnet/core/performance/caching/memory)将对象存储在本地服务器的内存中, 并且仅对存储该对象的应用程序可用. 集群环境中的非粘性会话应使用[分布式缓存](https://docs.microsoft.com/en-us/aspnet/core/performance/caching/distributed), 除了一些特定场景(例如, 你可以将本地CSS文件缓存到内存中. 它是只读数据, 在所有应用程序实例中都是相同的. 出于性能原因, 你可以将其缓存到内存中, 而不会出现任何问题).

[ABP的分布式缓存](../Caching.md)扩展了[ASP.NET Core的分布式缓存](https://docs.microsoft.com/en-us/aspnet/core/performance/caching/distributed)的基础设施. 默认情况下, 它在内存中工作. 当你要将应用程序部署到集群环境时, 应该配置实际的分布式缓存提供程序.

> 即使应用程序不直接使用`IDistributedCache`, 也应该为集群部署配置缓存提供程序. 因为ABP框架和预构建的[应用程序模块](../Modules/Index.md)正在使用分布式缓存.

ASP.NET Core提供了可以用作分布式缓存提供程序的多种集成, 如[Redis](https://redis.io/)和[NCache](https://www.alachisoft.com/ncache/). 你可以按照[微软文档](https://docs.microsoft.com/en-us/aspnet/core/performance/caching/distributed)了解如何在应用程序中使用它们.

如果你决定使用Redis作为分布式缓存提供程序, **\*\*请遵循[**ABP的Redis缓存集成文档**](../Redis-Cache.md)\*\*** 了解将其安装到应用程序并配置Redis所需遵循的步骤.

> 根据你在创建新ABP解决方案时的偏好, Redis缓存可能会预先安装在你的解决方案中. 例如, 如果你在MVC UI中选择了*\*Tiered\**选项, Redis缓存将进行预装. 因为, 在这种情况下, 解决方案中有两个应用程序, 它们应该使用相同的缓存源来保持一致.

## 使用合适的BLOB存储提供程序

如果你在[文件系统提供程序](../Blob-Storing-File-System.md)中使用了ABP的[BLOB存储](../Blob-Storing.md)功能, 则应该在集群环境中使用另一个提供程序, 因为文件系统提供程序使用应用程序的本地文件系统.

[数据库BLOB提供程序](../Blob-Storing-Database)是最简单的方法, 因为它使用应用程序的主数据库(或另一个数据库, 如果你配置的话)来存储BLOB. 但是, 你应该记住, BLOB是大型对象, 可能会迅速增加数据库的大小.

> [ABP商业版](https://commercial.abp.io/)启动解决方案模板预装了数据库BLOB提供程序, 并将BLOB存储在应用程序的数据库中.

查看[BLOB Storing](../Blob-Storing.md)文档以查看所有可用的BLOB存储提供程序.

## 配置后台作业

ABP的[后台作业系统](../Background-Jobs.md)将要在后台执行的任务进行排队. 后台作业队列是持久性的, 排队的任务能够保证执行(如果失败, 将重新尝试).

ABP的默认后台作业管理器与集群环境兼容. 它使用[分布式锁](../Distributed-Locking.md)来确保一次只能在单个应用程序实例中执行作业. 请参阅下面的*\*配置分布式锁提供程序\**部分, 了解如何为应用程序配置分布式锁提供程序, 以便默认后台作业管理器在集群环境中正常工作.

如果不想使用分布式锁提供程序, 可以使用以下选项:

\* 停止所有应用程序实例中的后台作业管理器(将`AbpBackgroundJobOptions.IsJobExecutionEnabled`设置为`false`)只保留其中一个应用程序实例, 以便只有单个实例执行作业(而其他应用程序实例仍可以对作业进行排队).

\* 在所有应用程序实例中停止后台作业管理器(将`AbpBackgroundJobOptions.IsJobExecutionEnabled`设置为`false`), 并创建一个专用的应用程序(可能是在自己的容器中运行的控制台应用程序或在后台运行的Windows服务)来执行所有后台作业. 如果你的后台作业占用大量系统资源(CPU、RAM或磁盘), 那么这是一个不错的选择, 这样你就可以将该后台应用程序部署到专用服务器上, 并且后台作业不会影响应用程序的性能.

> 如果你使用的是外部后台作业集成(例如[Hangfire](../Background-Workers-Hangfire.md)或[Quartz](../Background-Workers-Quartz.md))而不是默认的后台作业管理器, 请参阅提供程序的文档, 了解如何为集群环境配置它.

## 配置分布式锁提供程序

ABP通过[分布式锁](https://github.com/madelson/DistributedLock)库实现了一个抽象的分布式锁. 分布式锁用于控制多个应用程序对共享资源的并发访问, 以防止由于并发写入而导致资源损坏. ABP框架和一些预构建的[应用程序模块](../Modules/Index.md)出于一些原因正在使用分布式锁.

但是, 分布式锁系统默认在进程中工作. 这意味着它实际上不是分布式的, 除非配置分布式锁提供程序. 因此, 如果尚未配置应用程序的提供程序, 请按照[分布式锁](../Distributed-Locking.md)文档为其配置提供程序.

## 实现后台工作者

ASP.NET Core[托管服务](https://docs.microsoft.com/en-us/aspnet/core/fundamentals/host/hosted-services), ABP提供[后台工作者](../Background-Workers.md)在应用程序的后台线程中执行任务.

如果你的应用程序有在后台运行的任务, 你应该注意它们在集群环境中的行为, 尤其是在后台任务使用相同资源的情况下. 你应该设计后台任务, 以便它们在集群环境中继续正常工作.

假设SaaS应用程序中的后台工作者检查用户订阅, 并在订阅续订日期临近时发送电子邮件. 如果后台任务在多个应用程序实例中运行, 可能会多次向某些用户发送同一封电子邮件, 这会影响他们.

我们建议你使用以下方法之一来解决此问题:

\* 实现你的后台工作者, 以便他们在集群环境中工作时不会出现任何问题. 使用[分布式锁](../Distributed-Locking.md)来确保并发控制是一种方法. 应用程序实例中的后台工作者可能会处理分布式锁, 因此其他应用程序实例中的工作者将等待该锁. 这样, 只有一个工作者在实际工作, 而其他的则在等待. 如果你实现了这一点, 你的后台工作者就可以安全地运行, 不必关心应用程序是如何部署的.

\* 停止所有应用程序实例中的后台工作者(将`AbpBackgroundWorkerOptions.IsEnabled`设置为`false`), 只保留其中一个应用程序实例, 因此只有单个实例运行这些后台工作者.

\* 停止所有应用程序实例中的后台工作者(将`AbpBackgroundWorkerOptions.IsEnabled`设置为`false`), 并创建一个专用的应用程序(可能是在自己的容器中运行的控制台应用程序或在后台运行的Windows服务)来执行所有后台任务. 如果你的后台工作者消耗大量系统资源(CPU、RAM或磁盘), 那么这是一个不错的选择, 这样你就可以将该后台应用程序部署到专用服务器上, 并且你的后台任务不会影响应用程序的性能.

# 示例

## 所有示例

# 示例应用

这些是ABP框架创建的官方示例. 这些示例大部分在[abpframework/abp-samples](https://github.com/abpframework/abp-samples) GitHub 仓库.

### 微服务示例

演示如何基于微服务体系结构构建系统的完整解决方案.

\* [示例的文档](Microservice-Demo.md)

\* [源码](https://github.com/abpframework/abp-samples/tree/master/MicroserviceDemo)

\* [微服务架构文档](../Microservice-Architecture.md)

### Book Store

一个简单的CRUD应用程序,展示了使用ABP框架开发应用程序的基本原理. 使用不同的技术实现了相同的示例:

\* **\*\*Book Store: Razor Pages UI & Entity Framework Core\*\***

\* [教程](https://docs.abp.io/en/abp/latest/Tutorials/Part-1?UI=MVC&DB=EF)

\* [源码](https://github.com/abpframework/abp-samples/tree/master/BookStore-Mvc-EfCore)

\* **\*\*Book Store: Angular UI & MongoDB\*\***

\* [教程](https://docs.abp.io/en/abp/latest/Tutorials/Part-1?UI=NG&DB=Mongo)

\* [源码](https://github.com/abpframework/abp-samples/tree/master/BookStore-Angular-MongoDb)

\* **\*\*Book Store: Modular application (Razor Pages UI & EF Core)\*\***

\* [源码](https://github.com/abpframework/abp-samples/tree/master/BookStore-Modular)

如果没有Razor Pages & MongoDB 结合,但你可以检查两个文档来理解它,因为DB和UI不会互相影响.

### 其他示例

\* **\*\*Entity Framework 迁移\*\***: 演示如何将应用程序拆分为多个数据库的解决方案. 每个数据库包含不同的模块.

\* [源码](https://github.com/abpframework/abp-samples/tree/master/EfCoreMigrationDemo)

\* [EF Core数据库迁移文档](../Entity-Framework-Core-Migrations.md)

\* **\*\*SignalR Demo\*\***: A simple chat application that allows to send and receive messages among authenticated users.

\* [源码](https://github.com/abpframework/abp-samples/tree/master/SignalRDemo)

\* [SignalR 集成文档](../SignalR-Integration.md)

\* **\*\*分布式架构中的实时消息\*\*** (使用 SingalR & RabbitMQ)

\* [源码](https://github.com/abpframework/abp-samples/tree/master/SignalRTieredDemo)

\* [文章](https://community.abp.io/articles/real-time-messaging-in-a-distributed-architecture-using-abp-framework-singalr-rabbitmq-daf47e17)

\* **\*\*Dashboard Demo\*\***: 一个简单的应用程序,展示了如何在ASP.NET Core MVC UI中使用widget系统.

\* [源码](https://github.com/abpframework/abp-samples/tree/master/DashboardDemo)

\* [Widget 文档](../UI/AspNetCore/Widgets.md)

\* **\*\*RabbitMQ 事件总线 Demo\*\***: 由两个通过RabbitMQ集成的分布式事件相互通信的应用程序组成的解决方案.

\* [源码](https://github.com/abpframework/abp-samples/tree/master/RabbitMqEventBus)

\* [分布式事件总线文档](../Distributed-Event-Bus.md)

\* [RabbitMQ 分布式事件总线集成文档](../Distributed-Event-Bus-RabbitMQ-Integration.md)

\* **\*\*文本模板 Demo\*\***: 文本模板系统的不同用例.

\* [源码](https://github.com/abpframework/abp-samples/tree/master/TextTemplateDemo)

\* [文本模板文档](../Text-Templating.md)

\* **\*\*存储过程 Demo\*\***: 演示如何以最佳实践使用存储过程,数据库视图和函数.

\* [源码](https://github.com/abpframework/abp-samples/tree/master/StoredProcedureDemo)

\* **\*\*无密码认证\*\***: 演示如何添加自定义令牌提供者使用链接验证用户身份,而不是输入密码.

\* [源码](https://github.com/abpframework/abp-samples/tree/master/PasswordlessAuthentication)

\* [文章](https://community.abp.io/articles/implementing-passwordless-authentication-with-asp.net-core-identity-c25l8koj)

\* **\*\*自定义认证\*\***: 如何为ASP.NET Core MVC / Razor Pages应用程序自定义身份验证的解决方案.

\* [源码](https://github.com/abpframework/abp-samples/tree/master/Authentication-Customization)

\* 相关文章:

\* [Azure Active Directory 认证](https://community.abp.io/articles/how-to-use-the-azure-active-directory-authentication-for-mvc-razor-page-applications-4603b9cf)

\* [自定义登录页面](https://community.abp.io/articles/how-to-customize-the-login-page-for-mvc-razor-page-applications-9a40f3cd)

\* [自定义 SignIn Manager](https://community.abp.io/articles/how-to-customize-the-signin-manager-3e858753)

\* **\*\*空的ASP.NET Core应用程序\*\***: 从基本的ASP.NET Core应用程序使用ABP框架.

\* [源码](https://github.com/abpframework/abp-samples/tree/master/BasicAspNetCoreApplication)

\* [文档](../Getting-Started-AspNetCore-Application.md)

\* **\*\*GRPC Demo\*\***: 演示如何将gRPC服务添加到基于ABP框架的Web应用程序以及如何从控制台应用程序使用它.

\* [源码](https://github.com/abpframework/abp-samples/tree/master/GrpcDemo)

\* **\*\*空的控制台应用程序\*\***: 从基本的控制台应用程序安装ABP框架.

\* [源码](https://github.com/abpframework/abp-samples/tree/master/BasicConsoleApplication)

\* [文档](../Getting-Started-Console-Application.md)

## 微服务示例

# 微服务解决方案示例

*\*"作为****\*\*面向服务架构\*\*****(SOA)的一个变体,微服务是一种将应用程序分解成****\*\*松散耦合服务\*\*****的新型架构风格. 通过****\*\*细粒度\*\*****的服务和****\*\*轻量级\*\*****的协议,微服务提供了更多的****\*\*模块化\*\*****,使应用程序更容易理解,开发,测试,并且更容易抵抗架构侵蚀. 它使小型团队能够****\*\*开发,部署和扩展\*\*****各自的服务,实现开发的****\*\*并行化\*\*****.它还允许通过****\*\*连续重构\*\*****形成单个服务的架构. 基于微服务架构可以实现****\*\*持续交付和部署\*\*****."\**

— [维基百科](https://zh.wikipedia.org/wiki/Microservices)

## 介绍

ABP框架的主要目标之一就是提供[便捷的基础设施来创建微服务解决方案](../Microservice-Architecture.md).

此示例演示了一个简单而完整的微服务解决方案;

\* 拥有多个可独立可单独部署的**\*\*微服务\*\***.

\* 多个**\*\*Web应用程序\*\***, 每一个都使用不同的API网关.

\* 使用[Ocelot](https://github.com/ThreeMammals/Ocelot)库开发了多个**\*\*网关\*\*** / BFFs ([用于前端的后端](https://docs.microsoft.com/zh-cn/azure/architecture/patterns/backends-for-frontends)).

\* 包含使用[IdentityServer](https://identityserver.io/)框架开发的 **\*\*身份认证服务\*\***. 它也是一个带有UI的SSO(单点登陆)应用程序.

\* 有**\*\*多个数据库\*\***. 一些微服务有自己的数据库,也有一些服务/应用程序共享同一个数据库(以演示不同的用例).

\* 有不同类型的数据库: **\*\*SQL Server\*\*** (与 **\*\*Entity Framework Core\*\*** ORM) 和 **\*\*MongoDB\*\***.

\* 有一个**\*\*控制台应用程序\*\***使用身份验证展示使用服务最简单的方法.

\* 使用[Redis](https://redis.io/)做**\*\*分布式缓存\*\***.

\* 使用[RabbitMQ](https://www.rabbitmq.com/)做服务间的**\*\*消息\*\***传递.

\* 使用 [Docker](https://www.docker.com/) & [Kubernates](https://kubernetes.io/) 来**\*\*部署\*\***&**\*\*运行\*\***所有的服务和应用程序.

\* 使用 [Elasticsearch](https://www.elastic.co/products/elasticsearch) & [Kibana](https://www.elastic.co/products/kibana) 来存储和可视化日志 (使用[Serilog](https://serilog.net/)写日志).

下图展示了该系统:

![microservice-sample-diagram](../images/microservice-sample-diagram-3.png)

### 源码

你可以从[GitHub仓库](https://github.com/abpframework/abp-samples/tree/master/MicroserviceDemo)获取源码.

## 运行解决方案

### 预先要求

为了能够从源代码运行解决方案, 应在你的计算机上安装并运行以下工具:

\* [SQL Server](https://www.microsoft.com/en-us/sql-server/sql-server-downloads) 2015+ (可以是 [express edition](https://www.microsoft.com/en-us/sql-server/sql-server-editions-express))

\* [Redis](https://redis.io/download) 5.0+

\* [RabbitMQ](https://www.rabbitmq.com/install-windows.html) 3.7.11+

\* [MongoDB](https://www.mongodb.com/download-center) 4.0+

\* [ElasticSearch](https://www.elastic.co/downloads/elasticsearch) 6.6+

\* [Kibana](https://www.elastic.co/downloads/kibana) 6.6+ (可选,建议显示日志)

### 运行基础设施

- 默认情况下运行Docker-compose可以轻松运行预先要求. 如果你没有安装,在windows环境下你可以在[这里](https://docs.docker.com/docker-for-windows/install/)下载并使用[Docker for Windows](https://docs.docker.com/docker-for-windows/).

\* 在 `MicroserviceDemo` 目录下运行命令 `docker-compose -f docker-compose.infrastructure.yml -f docker-compose.infrastructure.override.yml up -d` 或使用powershell运行 `MicroserviceDemo/\_run` 文件夹下的 `\_\_Run\_Infrastructure.ps1` 脚本.

\* 如果你不想使用docker运行预先要求的服务在你的本地环境,你需要更改微服务示例解决方案项目中的 `appsettings.json` 文件.

### 打开并构建Visual Studio解决方案

\* 在Visual Studio 2017 (15.9.0+)中打开`samples\MicroserviceDemo\MicroserviceDemo.sln`.

\* 在`samples\MicroserviceDemo`文件夹中的命令行运行`dotnet restore`命令.

\* 在Visual Studio中构建解决方案.

### 创建数据库

MongoDB 数据库是动态创建的,但是你需要创建 SQL server 数据库的结构.其实你可以很轻松的创建数据库,因为这个解决方案配置了使用 Entity Core Code First 来做迁移.

这个解决方案中有两个 SQL server 数据库.

#### MsDemo\_Identity 数据库

\* 右键 `AuthServer.Host` 项目,然后点击 `设置为启动项目`.

\* 打开 **\*\*程序包管理器控制台\*\*** (工具 -> NuGet 包管理器 -> 程序包管理器控制台)

\* 选择 `AuthServer.Host` 成为 **\*\*默认项目\*\***.

\* 执行 `Update-Database` 命令.

![microservice-sample-update-database-authserver](../images/microservice-sample-update-database-authserver.png)

#### MsDemo\_ProductManagement

\* 右键 `ProductService.Host` 项目,然后点击 `设置为启动项目`.

\* 打开 **\*\*程序包管理器控制台\*\*** (工具 -> NuGet 包管理器 -> 程序包管理器控制台)

\* 选择 `ProductService.Host` 成为 **\*\*默认项目\*\***.

\* 执行 `Update-Database` 命令.

![microservice-sample-update-database-products](../images/microservice-sample-update-database-products.png)

### 运行项目

按以下顺序运行项目(右键单击每个项目设置为启动项目,按Ctrl+F5运行,无需调试):

\* AuthServer.Host

\* IdentityService.Host

\* TenantManagementService.Host

\* BloggingService.Host

\* ProductService.Host

\* InternalGateway.Host

\* BackendAdminAppGateway.Host

\* PublicWebSiteGateway.Host

\* BackendAdminApp.Host

\* PublicWebSite.Host

## 解决方案简介

Visual Studio解决方案由多个项目组成,每个项目在系统中具有不同的角色:

![microservice-sample-solution](../images/microservice-sample-solution-2.png)

### 应用程序(Applications)

这些是具有用户界面以与用户交互并使用系统的实际应用程序.

- **\*\*AuthServer.Host\*\***: 托管IdentityServer4以向其他服务和应用程序提供身份验证服务. 它是一个单点登录服务器,包含登录页面.

- **\*\*BackendAdminApp.Host\*\***: 这是一个后端管理应用程序,用于托管身份和产品管理模块的UI.

- **\*\*PublicWebSite.Host\*\***: 作为包含简单产品列表页面和博客模块UI的公共网站.

- **\*\*ConsoleClientDemo\*\***: 一个简单的控制台应用程序,用于演示C＃应用程序中使用服务.

### 网关/BFF(前端后端)

网关用于为应用程序提供单一入口点.它还可以用于速率限制,负载平衡等. 使用[Ocelot](https://github.com/ThreeMammals/Ocelot)类库.

\* **\*\*BackendAdminAppGateway.Host\*\***: 由BackendAdminApp.Host应用程序用作后端.

\* **\*\*PublicWebSiteGateway.Host\*\***: 由PublicWebSite.Host应用程序用作后端.

\* **\*\*InternalGateway.Host\*\***: 用于服务间通信(微服务之间的通信).

### 微服务

微服务没有UI,但暴露了一些REST API.

- **\*\*IdentityService.Host\*\***: 托管用于管理用户和角色的ABP Identity模块. 它没有其他服务,仅托管Identity模块的API.

- **\*\*TenantManagementService.Host\*\***: 托管用于管理角色的ABP租户管理模块. 它没有其他服务,仅托管租户管理模块的API.

- **\*\*BloggingService.Host\*\***: 托管ABP博客模块,该模块用于管理博客和帖子(典型的博客应用程序). 它没有其他服务,仅托管Blogging模块的API.

- **\*\*ProductService.Host\*\***: 托管用于管理产品的产品模块(位于解决方案内). 它还包含用于创建/更新产品管理数据库架构的EF Core迁移.

### 模块

\* **\*\*产品\*\***: 使用[模块开发最佳实践](../Best-Practices/Index.md)开发的分层模块. 它可以嵌入到单个应用程序中,也可以通过单独部署API和UI作为微服务托管(如本演示解决方案中所述).

### 数据库

此解决方案使用多个数据库:

\* **\*\*MsDemo\_Identity\*\***: 一个SQL数据库. 默认使用**\*\*SQL Server\*\***,但可以是EF Core支持的任何DBMS. 由AuthServer,IdentityService和TenantManagementService共享. 审计日志,权限和设置也存储在此数据库中(虽然它们可以轻松拥有自己的数据库,共享相同的数据库以保持简单).

\* **\*\*MsDemo\_ProductManagement\*\***: 一个SQL数据库. 同样默认使用 **\*\*SQL Server\*\***,但可以是EF Core支持的任何DBMS. 由ProductService用作专用数据库.

\* **\*\*MsDemo\_Blogging\*\***: **\*\*MongoDB\*\***数据库. 由BloggingService使用.

\* **\*\*Elasticsearch\*\***: 用于在Serilog上写日志.

## 应用

### 认证服务器 (AuthServer.Host)

所有其他服务和应用程序都使用此项目进行身份验证和单点登录. 主要使用**\*\*IdentityServer4\*\***来提供这些服务. 它使用了一些[预构建ABP模块](../Modules/Index) 如 *\*Identity\**, *\*Audit Logging\** 和 *\*Permission Management\**.

#### 数据库和EF Core配置

此应用程序使用SQL数据库(将其命名为**\*\*MsDemo\_Identity\*\***)并通过**\*\*Entity Framework Core迁移\*\***维护其架构.

它有一个名为**\*\*AuthServerDbContext\*\***的DbContext,定义如下:

````csharp

public class AuthServerDbContext : AbpDbContext<AuthServerDbContext>

{

public AuthServerDbContext(DbContextOptions<AuthServerDbContext> options)

: base(options)

{

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

base.OnModelCreating(modelBuilder);

modelBuilder.ConfigureIdentity();

modelBuilder.ConfigureIdentityServer();

modelBuilder.ConfigureAuditLogging();

modelBuilder.ConfigurePermissionManagement();

modelBuilder.ConfigureSettingManagement();

}

}

````

在**\*\*OnModelCreating\*\***方法中, 你会看到 **\*\*ConfigureX()\*\*** 方法调用. 具有数据库模式的模块通常声明这样的扩展方法,以便为其自己的实体配置EF Core映射. 这是一种灵活的方法, 可以在其中安排数据库和模块; 可以为每个模块使用不同的数据库,或者将它们中的一些组合在一个共享数据库中. 在AuthServer项目中,我们决定在单个物理数据库中将单个EF Core DbContext中的多个模块模式组合在一起. 这些模块是Identity,IdentityServer,AuditLogging,PermissionManagement和SettingManagement模块.

请注意,此DbContext仅用于数据库迁移. 所有模块都有自己的`DbContext`类,模块在运行时使用这些类.

#### 用户界面

AuthServer有一个简单的主页,如果当前用户已登录,则显示当前用户信息:

![microservice-sample-authserver-home](../images/microservice-sample-authserver-home.png)

它还提供登录和注册页面:

![microservice-sample-authserver-login](../images/microservice-sample-authserver-login.png)

这些页面不包含在项目本身中. 相反,AuthServer项目使用带有IdentityServer扩展的预构建ABP[帐户模块](https://github.com/abpframework/abp/tree/master/modules/account). 这意味着它还可以充当具有必要UI和逻辑的OpenId Connect服务器.

#### 依赖

\* **\*\*RabbitMQ\*\*** 用于向其他服务发送消息.

\* **\*\*Redis\*\*** 用于分布式/共享缓存.

\* **\*\*Elasticsearch\*\*** 用于存储日志.

### 后端管理应用程序 (BackendAdminApp.Host)

这是一个Web应用程序,用于管理系统中的用户,角色,权限和产品.

#### 认证

BackendAdminApp重定向到AuthServer进行身份验证. 用户输入正确的用户名和密码后,页面将再次重定向到后端应用程序. 身份验证配置在`BackendAdminAppHostModule`类中设置:

````charp

context.Services.AddAuthentication(options =>

{

options.DefaultScheme = "Cookies";

options.DefaultChallengeScheme = "oidc";

})

.AddCookie("Cookies", options =>

{

options.Cookie.Expiration = TimeSpan.FromDays(365);

options.ExpireTimeSpan = TimeSpan.FromDays(365);

})

.AddOpenIdConnect("oidc", options =>

{

options.Authority = configuration["AuthServer:Authority"];

options.ClientId = configuration["AuthServer:ClientId"];

options.ClientSecret = configuration["AuthServer:ClientSecret"];

options.RequireHttpsMetadata = false;

options.ResponseType = OpenIdConnectResponseType.CodeIdToken;

options.SaveTokens = true;

options.GetClaimsFromUserInfoEndpoint = true;

options.Scope.Add("role");

options.Scope.Add("email");

options.Scope.Add("phone");

options.Scope.Add("BackendAdminAppGateway");

options.Scope.Add("IdentityService");

options.Scope.Add("ProductService");

options.ClaimActions.MapAbpClaimTypes();

});

````

\* 它将"Cookies"身份验证添加为主要身份验证类型.

\* "oidc"身份验证配置为使用AuthServer应用程序作为身份验证服务器.

\* 它需要额外的身份范围(scopes) *\*role\**, *\*email\** 和 *\*phone\**.

\* 它需要API资源范围 *\*BackendAdminAppGateway\**, *\*IdentityService\** 和 *\*ProductService\**,因为它将这些服务用作API.

IdentityServer客户端设置存储在`appsettings.json`文件中:

````json

"AuthServer": {

"Authority": "http://localhost:64999",

"ClientId": "backend-admin-app-client",

"ClientSecret": "1q2w3e\*"

}

````

#### 用户界面

BackendAdminApp.Host项目本身没有单个UI元素/页面. 它仅用于提供身份和产品管理模块的UI页面.

`BackendAdminAppHostModule`将依赖关系添加到`AbpIdentityWebModule`(*\*[Volo.Abp.Identity.Web](https://www.nuget.org/packages/Volo.Abp.Identity.Web)\** 包)和`ProductManagementWebModule`(*\*ProductManagement.Web\**项目)为此目的.

用户管理页面的屏幕截图:

![microservice-sample-backend-ui](../images/microservice-sample-backend-ui.png)

来自权限的权限管理模式的屏幕截图:

![microservice-sample-backend-ui-permissions](../images/microservice-sample-backend-ui-permissions.png)

#### 使用微服务

后端管理应用程序通过后端管理网关对所有操作使用Identity和Product微服务(BackendAdminAppGateway.Host).

##### 远程端点

`appsettings.json`文件包含`RemoteServices`部分,用于声明远程服务端点. 每个微服务通常都有不同的端点. 但是,此解决方案使用API网关模式为应用程序提供单个端点:

````json

"RemoteServices": {

"Default": {

"BaseUrl": "http://localhost:65115/"

}

}

````

`http://localhost:65115/` 是 *\*BackendAdminAppGateway.Host\** 项目的URL. 它知道身份和产品服务的位置.

##### HTTP Clients

ABP应用程序模块通常提供C＃客户端库以轻松地使用服务(API)(它们通常使用ABP框架的[Dynamic C# API客户端](../API/Dynamic-CSharp-API-Clients.md)). 这意味着如果你需要使用Identity Service API, 你可以引用其客户端软件包,并通过提供的接口轻松使用API.

为此`BackendAdminAppHostModule`类声明了`AbpIdentityHttpApiClientModule`和`ProductManagementHttpApiClientModule`的依赖关系.

一旦引用这些客户端软件包,就可以直接注入应用程序服务接口(例如`IIdentityUserAppService`)并使用其方法,如本地方法调用. 它实际上通过HTTP调用到相关服务端点的远程服务调用.

##### 传递访问令牌(Access Token)

由于微服务需要身份验证和授权,因此每个远程服务调用都应包含Authentication头. 该头是从当前用户的当前`HttpContext`中的`access\_token`获得的. 当你使用`Volo.Abp.Http.Client.IdentityModel`包时,会自动执行此操作. `BackendAdminAppHostModule`声明对此包和相关的`AbpHttpClientIdentityModelModule`类的依赖. 它集成到上面解释的HTTP客户端.

#### 依赖

- **\*\*Redis\*\*** 用于分布式/共享缓存.

- **\*\*Elasticsearch\*\*** 用于存储日志.

### 公共网站 (PublicWebSite.Host)

这是一个公共网站项目,具有Web博客和产品列表页面.

#### 认证

公共网站可以在不登录的情况下显示博客文章和产品列表. 如果你登录,你还可以管理博客. 它重定向到AuthServer进行身份验证. 用户输入正确的用户名和密码后,页面将再次重定向到公共网站应用程序. 身份验证配置在`PublicWebSiteHostModule`类中设置:

```charp

context.Services.AddAuthentication(options =>

{

options.DefaultScheme = "Cookies";

options.DefaultChallengeScheme = "oidc";

})

.AddCookie("Cookies", options =>

{

options.Cookie.Expiration = TimeSpan.FromDays(365);

options.ExpireTimeSpan = TimeSpan.FromDays(365);

})

.AddOpenIdConnect("oidc", options =>

{

options.Authority = configuration["AuthServer:Authority"];

options.ClientId = configuration["AuthServer:ClientId"];

options.ClientSecret = configuration["AuthServer:ClientSecret"];

options.RequireHttpsMetadata = false;

options.ResponseType = OpenIdConnectResponseType.CodeIdToken;

options.SaveTokens = true;

options.GetClaimsFromUserInfoEndpoint = true;

options.Scope.Add("role");

options.Scope.Add("email");

options.Scope.Add("phone");

options.Scope.Add("PublicWebSiteGateway");

options.Scope.Add("ProductService");

options.Scope.Add("BloggingService");

options.ClaimActions.MapAbpClaimTypes();

});

```

- 它将"Cookies"身份验证添加为主要身份验证类型.

- "oidc"身份验证配置为使用AuthServer应用程序作为身份验证服务器.

- 它需要额外的身份范围 *\*role\**, *\*email\** and *\*phone\**.

- 它需要API资源范围 *\*PublicWebSiteGateway\**,*\*BloggingService\**和*\*ProductService\**,因为它将这些服务用作API.

IdentityServer客户端设置存储在`appsettings.json`文件中:

```json

"AuthServer": {

"Authority": "http://localhost:64999",

"ClientId": "public-website-client",

"ClientSecret": "1q2w3e\*"

}

```

#### 用户界面

PublicWebSite.Host项目有一个列出产品的页面 (`Pages/Products.cshtml`). 它还使用博客模块中的UI. 为此`PublicWebSiteHostModule`加入了`BloggingWebModule`(*\*[Volo.Blogging.Web](https://www.nuget.org/packages/Volo.Blogging.Web)\** 包)的依赖项.

产品页面的屏幕截图:

![microservice-sample-public-product-list](../images/microservice-sample-public-product-list.png)

#### 使用微服务

公共网站应用程序使用Blogging和Product微服务通过公共网站网关进行所有操作(PublicWebSiteGateway.Host).

##### 远程端点

`appsettings.json`文件包含`RemoteServices`部分,用于声明远程服务端点. 每个微服务通常都有不同的端点. 但是,此解决方案使用API网关模式为应用程序提供单个端点:

```json

"RemoteServices": {

"Default": {

"BaseUrl": "http://localhost:64897/"

}

}

```

`http://localhost:64897/` 是*\*PublicWebSiteGateway.Host\**项目的URL. 它知道Blogging和产品服务的位置.

##### HTTP Clients

`PublicWebSiteHostModule`类声明`BloggingHttpApiClientModule`和`ProductManagementHttpApiClientModule`的依赖关系,以便能够为这些服务使用远程HTTP API.

##### 传递访问令牌(Access Token)

正如后端管理应用程序部分中所述, Public Web Site项目还使用`AbpHttpClientIdentityModelModule`将`access\_token`传递给调用服务进行身份验证.

#### Dependencies

- **\*\*Redis\*\*** 用于分布式/共享缓存.

- **\*\*Elasticsearch\*\*** 用于存储日志.

### 控制台客户端演示

最后,该解决方案包括一个非常简单的控制台应用程序,名为ConsoleClientDemo,它通过AuthServer进行身份验证来使用Identity和Product服务. 它使用内部网关(InternalGateway.Host)来执行HTTP API调用.

#### 远程服务配置

`appsettings.json`文件中的`RemoteService`配置很简单:

````json

"RemoteServices": {

"Default": {

"BaseUrl": "http://localhost:65129/"

}

}

````

`http://localhost:65129/` 是内部网关的URL. 对服务的所有API调用都是通过此URL执行的.

#### 身份验证(IdentityServer客户端)配置

`appsettings.json`还有一个IdentityServer身份验证配置:

````json

"IdentityClients": {

"Default": {

"GrantType": "client\_credentials",

"ClientId": "console-client-demo",

"ClientSecret": "1q2w3e\*",

"Authority": "http://localhost:64999",

"Scope": "InternalGateway IdentityService ProductService"

}

}

````

此示例使用`client\_credentials` 授予类型,该类型需要`ClientId`和`ClientSecret`进行身份验证过程. 还有[其他授予类型](http://docs.identityserver.io/en/latest/topics/grant\_types.html). 例如, 你可以使用以下配置切换到`password`(Resource Owner Password)授予类型:

````json

"IdentityClients": {

"Default": {

"GrantType": "password",

"ClientId": "console-client-demo",

"ClientSecret": "1q2w3e\*",

"UserName": "admin",

"UserPassword": "1q2w3E\*",

"Authority": "http://localhost:64999",

"Scope": "InternalGateway IdentityService ProductService"

}

}

````

除客户端凭据外,Resource Owner Password还需要`UserName` 和 `UserPassword`. 此授权类型对于代表用户调用远程服务很有用.

`Scope` 声明API(和网关)以授予访问权限. 此应用程序使用内部网关.

#### HTTP Client依赖

`ConsoleClientDemoModule`与`AbpIdentityHttpApiClientModule`和`ProductManagementHttpApiClientModule`有依赖关系,以便使用Identity和Product API. 它还具有“AbpHttpClientIdentityModelModule”依赖性,可通过IdentityServer进行身份验证.

#### 使用服务

使用这些服务非常简单. 请参阅`ClientDemoService`类,它只是注入`IIdentityUserAppService`和`IProductAppService`并使用它们. 该类还显示了使用`HttpClient`对象的手动HTTP调用. 有关详细信息,请参阅`ClientDemoService`的源代码.

## API网关/ BFF(前端后端)

网关用于为应用程序提供 **\*\*单一入口点\*\***. 通过这种方式,应用程序仅处理单个服务地址(API端点),而不是每个服务的不同地址. 网关还用于速率限制,安全性,身份验证,负载平衡和更多要求.

"**\*\*后端前端\*\***"(BFF)是一种常见的架构模式,可为每种不同的应用程序/客户端类型构建**\*\*专注而专业的\*\***网关. 此解决方案使用此模式并具有多个网关.

此解决方案使用[Ocelot](https://github.com/ThreeMammals/Ocelot)库来构建API网关. 它是ASP.NET核心广泛接受的API网关库.

### 后端管理应用程序网关 (BackendAdminAppGateway.Host)

这是"后端管理应用程序"的后端(服务器端API)(不要混淆命名;后端管理应用程序实际上是前端Web应用程序,但系统管理员而不是普通用户使用).

#### 认证

此网关使用IdentityServer `Bearer`身份验证并配置如下:

````csharp

context.Services.AddAuthentication("Bearer")

.AddIdentityServerAuthentication(options =>

{

options.Authority = configuration["AuthServer:Authority"];

options.ApiName = configuration["AuthServer:ApiName"];

options.RequireHttpsMetadata = false;

options.InboundJwtClaimTypeMap["sub"] = AbpClaimTypes.UserId;

options.InboundJwtClaimTypeMap["role"] = AbpClaimTypes.Role;

options.InboundJwtClaimTypeMap["email"] = AbpClaimTypes.Email;

options.InboundJwtClaimTypeMap["email\_verified"] = AbpClaimTypes.EmailVerified;

options.InboundJwtClaimTypeMap["phone\_number"] = AbpClaimTypes.PhoneNumber;

options.InboundJwtClaimTypeMap["phone\_number\_verified"] =

AbpClaimTypes.PhoneNumberVerified;

options.InboundJwtClaimTypeMap["name"] = AbpClaimTypes.UserName;

});

````

`AddIdentityServerAuthentication` 扩展方法来自 [IdentityServer4.AccessTokenValidation](https://www.nuget.org/packages/IdentityServer4.AccessTokenValidation) 包, IdentityServer4项目的一部分 (参见 [identityserver文档](http://docs.identityserver.io/en/latest/topics/apis.html)).

`ApiName`是受保护的API,在这种情况下是`BackendAdminAppGateway`. 因此,此解决方案将网关定义为API资源. 其余配置与声明映射(计划在下一个ABP版本中自动化)相关. `appsettings.json`中的身份验证相关的配置很简单:

````json

"AuthServer": {

"Authority": "http://localhost:64999",

"ApiName": "BackendAdminAppGateway"

}

````

#### Ocelot配置

Ocelot需要知道微服务的真实URL才能重定向HTTP请求. 此网关的配置如下所示:

````json

"ReRoutes": [

{

"DownstreamPathTemplate": "/api/identity/{everything}",

"DownstreamScheme": "http",

"DownstreamHostAndPorts": [

{

"Host": "localhost",

"Port": 63568

}

],

"UpstreamPathTemplate": "/api/identity/{everything}",

"UpstreamHttpMethod": [ "Put", "Delete", "Get", "Post" ]

},

{

"DownstreamPathTemplate": "/api/productManagement/{everything}",

"DownstreamScheme": "http",

"DownstreamHostAndPorts": [

{

"Host": "localhost",

"Port": 60244

}

],

"UpstreamPathTemplate": "/api/productManagement/{everything}",

"UpstreamHttpMethod": [ "Put", "Delete", "Get", "Post" ]

}

],

"GlobalConfiguration": {

"BaseUrl": "http://localhost:65115"

}

````

`ReRoutes`是一个URL映射数组. `GlobalConfiguration`部分中的`BaseUrl`是该网关的URL(Ocelot需要知道自己的URL). 参见 [ocelot文档](https://ocelot.readthedocs.io/en/latest/features/configuration.html) 更好地了解配置.

Ocelot是一个终结ASP.NET核心中间件,应该写成管道中的最后一项:

````csharp

app.UseOcelot().Wait();

````

它根据上面的配置处理和重定向请求.

#### ABP 配置端点

ABP提供了一些内置API,以从服务器获取一些配置和信息. 例子:

\* `/api/abp/application-configuration` 返回本地化文本,权限和设置值 (http://localhost:65115/api/abp/application-configuration).

\* `/Abp/ServiceProxyScript` 返回动态javascript代理以从javascript客户端调用服务 (http://localhost:65115/Abp/ServiceProxyScript ).

这些端点应由网关服务提供,而不是由微服务提供. 微服务只能知道与该微服务相关的权限. 但是,一旦正确配置,网关就可以将多个服务的权限值聚合为一个更适合客户端的列表.

为此, ASP.NET Core管道配置为通过MVC而不是Ocelot处理某些特定路由. 为了实现这一点,MapWhen扩展方法使用如下:

````csharp

app.MapWhen(ctx => ctx.Request.Path.ToString().StartsWith("/api/abp/") ||

ctx.Request.Path.ToString().StartsWith("/Abp/"),

app2 =>

{

app2.UseConfiguredEndpoints();

});

app.UseOcelot().Wait();

````

当请求路径以 `/api/abp/` 或 `/Abp/`.开头时,此配置使用标准MVC中间件.

#### Swagger

此网关配置为使用[swagger UI](https://swagger.io/tools/swagger-ui/), 这是一种用于发现和测试HTTP API的流行工具. 通常,Ocelot不支持在swagger上显示API,因为它无法了解每个微服务API的详细信息. 但是当你遵循ABP分层模块架构[最佳实践](../Best-Practices/Index.md)时,它是可能的.

`BackendAdminAppGatewayHostModule`将依赖性添加到`AbpIdentityHttpApiModule`(*\*[Volo.Abp.Identity.HttpApi](https://www.nuget.org/packages/Volo.Abp.Identity.HttpApi)\** 包)和`ProductManagementHttpApiModule`(*\*ProductManagement.HttpApi\** 项目)以包含它们的HTTP API控制器. 通过这种方式,swagger可以发现它们. 虽然它引用了API层,但它没有引用应用程序服务的实现,因为它们将在相关的微服务端点中运行,并由Ocelot根据请求URL重定向.

无论如何,当你打开URL`http://localhost:65115/swagger/index.html`时, 你将看到所有已配置的微服务的API.

#### 权限管理

后端管理应用程序提供权限管理UI(之前见过),并使用此网关获取/设置权限. 权限管理API托管在网关内,而不是单独的服务. 这是一个设计决策,但如果你愿意,它可以作为另一个微服务托管.

#### Dependencies

- **\*\*RabbitMQ\*\*** 用于向其他服务发送消息.

- **\*\*Redis\*\*** 用于分布式/共享缓存.

- **\*\*Elasticsearch\*\*** 用于存储日志.

### 公共网站网关 (PublicWebSiteGateway.Host)

这是"公共网站"应用程序的后端(服务器端API网关).

#### 认证

此网关使用IdentityServer `Bearer`身份验证并配置如下:

```csharp

context.Services.AddAuthentication("Bearer")

.AddIdentityServerAuthentication(options =>

{

options.Authority = configuration["AuthServer:Authority"];

options.ApiName = configuration["AuthServer:ApiName"];

options.RequireHttpsMetadata = false;

options.InboundJwtClaimTypeMap["sub"] = AbpClaimTypes.UserId;

options.InboundJwtClaimTypeMap["role"] = AbpClaimTypes.Role;

options.InboundJwtClaimTypeMap["email"] = AbpClaimTypes.Email;

options.InboundJwtClaimTypeMap["email\_verified"] = AbpClaimTypes.EmailVerified;

options.InboundJwtClaimTypeMap["phone\_number"] = AbpClaimTypes.PhoneNumber;

options.InboundJwtClaimTypeMap["phone\_number\_verified"] =

AbpClaimTypes.PhoneNumberVerified;

options.InboundJwtClaimTypeMap["name"] = AbpClaimTypes.UserName;

});

```

`AddIdentityServerAuthentication` 扩展方法来自 [IdentityServer4.AccessTokenValidation](https://www.nuget.org/packages/IdentityServer4.AccessTokenValidation)包, IdentityServer4项目的一部分 (参见 [identityserver文档](http://docs.identityserver.io/en/latest/topics/apis.html)).

`ApiName`是受保护的API,在这种情况下是`PublicWebSiteGateway`. 其余配置与声明映射(计划在下一个ABP版本中自动化)相关. 与`appsettings.json`中的身份验证相关的配置很简单:

```json

"AuthServer": {

"Authority": "http://localhost:64999",

"ApiName": "PublicWebSiteGateway"

}

```

#### Ocelot配置

Ocelot需要知道微服务的真实URL才能重定向HTTP请求. 此网关的配置如下所示:

```json

"ReRoutes": [

{

"DownstreamPathTemplate": "/api/productManagement/{everything}",

"DownstreamScheme": "http",

"DownstreamHostAndPorts": [

{

"Host": "localhost",

"Port": 60244

}

],

"UpstreamPathTemplate": "/api/productManagement/{everything}",

"UpstreamHttpMethod": [ "Put", "Delete", "Get", "Post" ]

},

{

"DownstreamPathTemplate": "/api/blogging/{everything}",

"DownstreamScheme": "http",

"DownstreamHostAndPorts": [

{

"Host": "localhost",

"Port": 62157

}

],

"UpstreamPathTemplate": "/api/blogging/{everything}",

"UpstreamHttpMethod": [ "Put", "Delete", "Get", "Post" ]

}

],

"GlobalConfiguration": {

"BaseUrl": "http://localhost:64897"

}

```

参见 [ocelot文档](https://ocelot.readthedocs.io/en/latest/features/configuration.html) 更好地了解配置.

#### 其它

请参阅"后端管理应用程序网关"部分中的"ABP配置端点"和"Swagger"主题,这些主题与此网关非常相似.

#### 依赖

- **\*\*RabbitMQ\*\*** 用于向其他服务发送消息.

- **\*\*Redis\*\*** 用于分布式/共享缓存.

- **\*\*Elasticsearch\*\*** 用于存储日志.

### 内部网关 (InternalGateway.Host)

该网关不是BFF. 它专为微服务间通信而设计,不会公开.

#### 认证

此网关使用IdentityServer `Bearer`身份验证并配置如下:

```csharp

context.Services.AddAuthentication("Bearer")

.AddIdentityServerAuthentication(options =>

{

options.Authority = configuration["AuthServer:Authority"];

options.ApiName = configuration["AuthServer:ApiName"];

options.RequireHttpsMetadata = false;

options.InboundJwtClaimTypeMap["sub"] = AbpClaimTypes.UserId;

options.InboundJwtClaimTypeMap["role"] = AbpClaimTypes.Role;

options.InboundJwtClaimTypeMap["email"] = AbpClaimTypes.Email;

options.InboundJwtClaimTypeMap["email\_verified"] = AbpClaimTypes.EmailVerified;

options.InboundJwtClaimTypeMap["phone\_number"] = AbpClaimTypes.PhoneNumber;

options.InboundJwtClaimTypeMap["phone\_number\_verified"] = AbpClaimTypes.PhoneNumberVerified;

options.InboundJwtClaimTypeMap["name"] = AbpClaimTypes.UserName;

});

```

`AddIdentityServerAuthentication` 扩展方法来自 [IdentityServer4.AccessTokenValidation](https://www.nuget.org/packages/IdentityServer4.AccessTokenValidation)包, IdentityServer4项目的一部分 (参见 [identityserver文档](http://docs.identityserver.io/en/latest/topics/apis.html)).

`ApiName`是受保护的API,在这种情况下是`InternalGateway`. 其余配置与声明映射(计划在下一个ABP版本中自动化)相关. 与`appsettings.json`中的身份验证相关的配置很简单:

```json

"AuthServer": {

"Authority": "http://localhost:64999",

"ApiName": "InternalGateway"

}

```

#### Ocelot 配置

Ocelot需要知道微服务的真实URL才能重定向HTTP请求. 此网关的配置如下所示:

```json

"ReRoutes": [

{

"DownstreamPathTemplate": "/api/identity/{everything}",

"DownstreamScheme": "http",

"DownstreamHostAndPorts": [

{

"Host": "localhost",

"Port": 63568

}

],

"UpstreamPathTemplate": "/api/identity/{everything}",

"UpstreamHttpMethod": [ "Put", "Delete", "Get", "Post" ]

},

{

"DownstreamPathTemplate": "/api/productManagement/{everything}",

"DownstreamScheme": "http",

"DownstreamHostAndPorts": [

{

"Host": "localhost",

"Port": 60244

}

],

"UpstreamPathTemplate": "/api/productManagement/{everything}",

"UpstreamHttpMethod": [ "Put", "Delete", "Get", "Post" ]

},

{

"DownstreamPathTemplate": "/api/blogging/{everything}",

"DownstreamScheme": "http",

"DownstreamHostAndPorts": [

{

"Host": "localhost",

"Port": 62157

}

],

"UpstreamPathTemplate": "/api/blogging/{everything}",

"UpstreamHttpMethod": [ "Put", "Delete", "Get", "Post" ]

}

],

"GlobalConfiguration": {

"BaseUrl": "http://localhost:65129"

}

```

`ReRoutes`配置涵盖了系统中的所有微服务. 参见 [ocelot文档](https://ocelot.readthedocs.io/en/latest/features/configuration.html) 更好地了解配置.

#### 其它

请参阅"后端管理应用程序网关"部分中的"ABP配置端点"和"Swagger"主题,这些主题与此网关非常相似.

#### 依赖

- **\*\*RabbitMQ\*\*** 用于向其他服务发送消息.

- **\*\*Redis\*\*** 用于分布式/共享缓存.

- **\*\*Elasticsearch\*\*** 用于存储日志.

## 微服务

微服务是独立的HTTP API,它们以分布式方式实现系统业务.

\* 它们由应用程序和其他微服务通过网关和HTTP API使用.

\* 他们可以提升或注册系统中的事件.

\* 它们可以通过异步消息传递相互通信.

### Identity Service (IdentityService.Host)

此服务提供用户和角色管理API.

#### 数据库

与AuthServer应用程序共享相同的数据库(MsDemo\_Identity).

#### Identity模块

该服务实际上只托管ABP身份包/模块. 不包含任何API本身. 为了托管它,添加以下依赖项:

\* `AbpIdentityHttpApiModule` (*\*[Volo.Abp.Identity.HttpApi](https://www.nuget.org/packages/Volo.Abp.Identity.HttpApi)\** 包) 提供身份API.

\* `AbpIdentityApplicationModule` (*\*[Volo.Abp.Identity.Application](https://www.nuget.org/packages/Volo.Abp.Identity.Application)\** 包)承载模块的应用程序和域层的实现.

\* `AbpIdentityEntityFrameworkCoreModule` (*\*[Volo.Abp.Identity.EntityFrameworkCore](https://www.nuget.org/packages/Volo.Abp.Identity.EntityFrameworkCore)\** 包) 使用EF Core作为数据库API.

请参阅[module architecture best practice guide](../Best-Practices/Module-Architecture)以更好地理解分层.

#### 认证

这个微服务使用IdentityServer`Bearer`身份验证,并配置如下:

```csharp

context.Services.AddAuthentication("Bearer")

.AddIdentityServerAuthentication(options =>

{

options.Authority = configuration["AuthServer:Authority"];

options.ApiName = configuration["AuthServer:ApiName"];

options.RequireHttpsMetadata = false;

options.InboundJwtClaimTypeMap["sub"] = AbpClaimTypes.UserId;

options.InboundJwtClaimTypeMap["role"] = AbpClaimTypes.Role;

options.InboundJwtClaimTypeMap["email"] = AbpClaimTypes.Email;

options.InboundJwtClaimTypeMap["email\_verified"] = AbpClaimTypes.EmailVerified;

options.InboundJwtClaimTypeMap["phone\_number"] = AbpClaimTypes.PhoneNumber;

options.InboundJwtClaimTypeMap["phone\_number\_verified"] =

AbpClaimTypes.PhoneNumberVerified;

options.InboundJwtClaimTypeMap["name"] = AbpClaimTypes.UserName;

});

```

`ApiName`是受保护的API,在这种情况下是`IdentityService`. 其余配置与声明映射(计划在下一个ABP版本中自动化)相关. 与`appsettings.json`中的身份验证相关的配置很简单:

```json

"AuthServer": {

"Authority": "http://localhost:64999",

"ApiName": "IdentityService"

}

```

#### Swagger

Swagger UI已配置,是此服务的默认页面. 如果你导航到URL`http://localhost:63568/`, 你将被重定向到swagger页面以查看和测试API.

#### 依赖

- **\*\*RabbitMQ\*\*** 用于向其他服务发送消息.

- **\*\*Redis\*\*** 用于分布式/共享缓存.

- **\*\*Elasticsearch\*\*** 用于存储日志.

### 博客服务 (BloggingService.Host)

此服务托管博客API.

#### 数据库

它有一个专门的MongoDB数据库(MsDemo\_Blogging)来存储博客和帖子. 它还使用MsDemo\_Identity SQL数据库来审核日志,权限和设置. 因此,`appsettings.json`文件中有两个连接字符串:

````json

"ConnectionStrings": {

"Default": "Server=localhost;Database=MsDemo\_Identity;Trusted\_Connection=True",

"Blogging": "mongodb://localhost/MsDemo\_Blogging"

}

````

#### 博客模块

该服务实际上只托管ABP Blogging包/模块. 不包含任何API本身. 为了托管它,添加以下依赖项:

- `BloggingHttpApiModule` (*\*[Volo.Blogging.HttpApi](https://www.nuget.org/packages/Volo.Blogging.HttpApi)\** 包) 提供Blogging API.

- `BloggingApplicationModule` (*\*[Volo.Blogging.Application](https://www.nuget.org/packages/Volo.Blogging.Application)\** 包) 承载模块的应用程序和域层的实现.

- `BloggingMongoDbModule` (*\*[Volo.Blogging.MongoDB](https://www.nuget.org/packages/Volo.Abp.Identity.EntityFrameworkCore)\** 包) 使用MongoDB作为数据库.

请参阅[module architecture best practice guide](../Best-Practices/Module-Architecture)以更好地理解分层.

#### 认证

这个微服务使用IdentityServer `Bearer`身份验证,并配置如下:

```csharp

context.Services.AddAuthentication("Bearer")

.AddIdentityServerAuthentication(options =>

{

options.Authority = configuration["AuthServer:Authority"];

options.ApiName = configuration["AuthServer:ApiName"];

options.RequireHttpsMetadata = false;

options.InboundJwtClaimTypeMap["sub"] = AbpClaimTypes.UserId;

options.InboundJwtClaimTypeMap["role"] = AbpClaimTypes.Role;

options.InboundJwtClaimTypeMap["email"] = AbpClaimTypes.Email;

options.InboundJwtClaimTypeMap["email\_verified"] = AbpClaimTypes.EmailVerified;

options.InboundJwtClaimTypeMap["phone\_number"] = AbpClaimTypes.PhoneNumber;

options.InboundJwtClaimTypeMap["phone\_number\_verified"] =

AbpClaimTypes.PhoneNumberVerified;

options.InboundJwtClaimTypeMap["name"] = AbpClaimTypes.UserName;

});

```

`ApiName`是受保护的API,在这种情况下是`BloggingService`. 其余配置与声明映射(计划在下一个ABP版本中自动化)相关.与`appsettings.json`中的身份验证相关的配置很简单:

```json

"AuthServer": {

"Authority": "http://localhost:64999",

"ApiName": "BloggingService"

}

```

#### IdentityServer Client

此微服务还通过内部网关使用Identity微服务API, 因为在某些情况下它需要查询用户详细信息(username, email, phone, name 和 surname). 因此,它也是IdentityServer的客户端,并在`appsettings.json`文件中定义了一个部分:

````json

"IdentityClients": {

"Default": {

"GrantType": "client\_credentials",

"ClientId": "blogging-service-client",

"ClientSecret": "1q2w3e\*",

"Authority": "http://localhost:64999",

"Scope": "InternalGateway IdentityService"

}

}

````

由于它使用内部网关, 因此它还应配置网关的远程端点:

````json

"RemoteServices": {

"Default": {

"BaseUrl": "http://localhost:65129/",

"UseCurrentAccessToken": "false"

}

}

````

当你将`UseCurrentAccessToken`设置为`false`时,ABP会忽略当前`HttpContext`中的当前`access\_token`,并使用上面定义的凭据对AuthServer进行身份验证.

为什么不在当前请求中使用当前用户的令牌? 因为,用户可能没有Identity模块所需的权限,因此它不能直接将当前身份验证令牌传递给Identity服务. 此外,某些博客服务API是匿名的(不需要经过身份验证的用户),因此在某些情况下,HTTP请求中没有"当前用户". 出于这些原因,应将Blogging服务定义为具有自己的凭据和权限的Identity服务的客户端.

如果检查`MsDemo\_Identity`数据库中的`AbpPermissionGrants`表,则可以看到`blogging-service-client`的相关权限.

![microservice-sample-blogservice-permission-in-database](../images/microservice-sample-blogservice-permission-in-database.png)

#### Swagger

Swagger UI已配置,是此服务的默认页面. 如果你导航到URL`http://localhost:62157/`, 你将被重定向到swagger页面以查看和测试API.

#### 依赖

- **\*\*RabbitMQ\*\*** 用于向其他服务发送消息.

- **\*\*Redis\*\*** 用于分布式/共享缓存.

- **\*\*Elasticsearch\*\*** 用于存储日志.

### 产品服务 (ProductService.Host)

此服务托管产品管理API.

#### 数据库和EF核心迁移

它有一个单独的SQL数据库,名为**\*\*MsDemo\_ProductManagement\*\***,用于产品管理模块. 它使用EF Core作为数据库提供程序,并具有名为`ProductServiceMigrationDbContext`的DbContext:

````csharp

public class ProductServiceMigrationDbContext : AbpDbContext<ProductServiceMigrationDbContext>

{

public ProductServiceMigrationDbContext(

DbContextOptions<ProductServiceMigrationDbContext> options

) : base(options)

{

}

protected override void OnModelCreating(ModelBuilder modelBuilder)

{

base.OnModelCreating(modelBuilder);

modelBuilder.ConfigureProductManagement();

}

}

````

实际模型配置在`modelBuilder.ConfigureProductManagement()`扩展方法内完成. 此项目使用EF Core迁移维护数据库模式.

请注意,此DbContext仅用于数据库迁移. Product Management模块有自己的`DbContext`类,它在运行时使用(参见ProductManagement.EntityFrameworkCore项目中的`ProductManagementDbContext`类).

`appsettings.json`文件中有两个连接字符串:

````json

"ConnectionStrings": {

"Default": "Server=localhost;Database=MsDemo\_Identity;Trusted\_Connection=True",

"ProductManagement": "Server=localhost;Database=MsDemo\_ProductManagement;Trusted\_Connection=True"

}

````

`Default`连接字符串指向MsDemo\_Identity数据库,该数据库用于审计日志记录,权限和设置存储. 产品模块使用`ProductManagement`连接字符串.

#### 产品模块

该服务实际上只托管产品管理模块. 不包含任何API本身. 为了托管它,添加以下依赖项:

- `ProductManagementHttpApiModule` 提供产品管理API.

- `ProductManagementApplicationModule` 承载模块的应用程序和域层的实现.

- `ProductManagementEntityFrameworkCoreModule` 使用EF Core作为数据库API.

请参阅[module architecture best practice guide](../Best-Practices/Module-Architecture)以更好地理解分层. 有关此模块的更多信息,请参阅下面的"产品管理"模块部分

#### 认证

这个微服务使用IdentityServer `Bearer`身份验证,并配置如下:

```csharp

context.Services.AddAuthentication("Bearer")

.AddIdentityServerAuthentication(options =>

{

options.Authority = configuration["AuthServer:Authority"];

options.ApiName = configuration["AuthServer:ApiName"];

options.RequireHttpsMetadata = false;

options.InboundJwtClaimTypeMap["sub"] = AbpClaimTypes.UserId;

options.InboundJwtClaimTypeMap["role"] = AbpClaimTypes.Role;

options.InboundJwtClaimTypeMap["email"] = AbpClaimTypes.Email;

options.InboundJwtClaimTypeMap["email\_verified"] = AbpClaimTypes.EmailVerified;

options.InboundJwtClaimTypeMap["phone\_number"] = AbpClaimTypes.PhoneNumber;

options.InboundJwtClaimTypeMap["phone\_number\_verified"] =

AbpClaimTypes.PhoneNumberVerified;

options.InboundJwtClaimTypeMap["name"] = AbpClaimTypes.UserName;

});

```

`ApiName`是受保护的API,在这种情况下是`ProductService`. 其余配置与声明映射(计划在下一个ABP版本中自动化)相关. 与`appsettings.json`中的身份验证相关的配置很简单:

```json

"AuthServer": {

"Authority": "http://localhost:64999",

"ApiName": "ProductService"

}

```

#### Swagger

Swagger UI已配置,是此服务的默认页面. 如果你导航到URL`http://localhost:60244/`, 你将被重定向到swagger页面以查看和测试API.

#### 依赖

- **\*\*RabbitMQ\*\*** 用于向其他服务发送消息.

- **\*\*Redis\*\*** 用于分布式/共享缓存.

- **\*\*Elasticsearch\*\*** 用于存储日志.

## 模块

ABP提供强大的基础架构,通过提供服务和架构,使模块化应用程序开发更容易(参见[模块开发最佳实践指南](../Best-Practices/Index.md)).

此解决方案演示了如何在分布式体系结构中使用[预构建的应用程序模块](../Modules/Index.md). 该解决方案还包括一个简单的"产品管理"模块,用于显示分层模块示例的实现.

### 产品管理

产品管理是一个由多个层和包/项目组成的模块:

![microservice-sample-product-module-in-solution](../images/microservice-sample-product-module-in-solution.png)

\* `ProductManagement.Domain.Shared` 包含所有层之间共享的常量和类型.

\* `ProductManagement.Domain` 包含域逻辑并定义实体,域服务,域事件,业务/域异常.

\* `ProductManagement.Application.Contracts` 包含应用程序服务接口和DTO.

\* `ProductManagement.Application` 包含应用程序服务的实现.

\* `ProductManagement.EntityFrameworkCore` 包含DbContext和其他与EF Core相关的类和配置.

\* `ProductManagement.HttpApi` 包含API控制器.

\* `ProductManagement.HttpApi.Client` 包含C#代理以远程直接使用HTTP API. 使用ABP的[Dynamic C＃API客户端](../API/Dynamic-CSharp-API-Clients.md)功能.

\* `ProductManagement.Web` 包含UI元素(页面,脚本,样式..等).

通过此分层的帮助,可以在单个应用程序中使用相同的模块作为包引用,或者用作在另一个服务器中运行的服务. 可以分离UI(Web)和API层,因此它们可以在不同的服务器中运行.

在此解决方案中,Web层在后端管理应用程序中运行,而API层由产品微服务托管.

本教程将重点介绍该模块的一些重要方面. 但是,建议查看源代码以便更好地理解.

#### Domain Layer

`Product`是这个模块的主要[聚合根](../Entities.md):

````csharp

public class Product : AuditedAggregateRoot<Guid>

{

/// <summary>

/// A unique value for this product.

/// ProductManager ensures the uniqueness of it.

/// It can not be changed after creation of the product.

/// </summary>

[NotNull]

public string Code { get; private set; }

[NotNull]

public string Name { get; private set; }

public float Price { get; private set; }

public int StockCount { get; private set; }

//...

}

````

它的所有属性都有私有的set方法,可以防止属性从类中直接更改. 产品类通过自己的构造函数和方法确保其自身的完整性和有效性.

它有两个构造函数:

````csharp

private Product()

{

//Default constructor is needed for ORMs.

}

internal Product(

Guid id,

[NotNull] string code,

[NotNull] string name,

float price = 0.0f,

int stockCount = 0)

{

Check.NotNullOrWhiteSpace(code, nameof(code));

if (code.Length >= ProductConsts.MaxCodeLength)

{

throw new ArgumentException(

$"Product code can not be longer than {ProductConsts.MaxCodeLength}"

);

}

Id = id;

Code = code;

SetName(Check.NotNullOrWhiteSpace(name, nameof(name)));

SetPrice(price);

SetStockCountInternal(stockCount, triggerEvent: false);

}

````

默认(**\*\*无参数\*\***)构造函数是私有的,不在应用程序代码中使用. 这是必需的,因为大多数ORM在从数据库获取时需要在反序列化实体时使用无参数构造函数.

第二个构造函数是**\*\*internal\*\***,这意味着它只能在域层内使用. 这强制在创建新的`Product`时使用`ProductManager`. 因为`ProductManager`应该在新产品创建上实现业务规则. 此构造函数仅需要最少的必需参数来创建具有一些可选参数的新产品. 它会检查一些简单的业务规则,以确保将实体创建为有效产品.

该类的其余部分具有操纵实体属性的方法. 例:

````csharp

public Product SetPrice(float price)

{

if (price < 0.0f)

{

throw new ArgumentException($"{nameof(price)} can not be less than 0.0!");

}

Price = price;

return this;

}

````

`SetPrice`方法用于以安全的方式更改产品的价格(通过检查验证规则).

`SetStockCount`是另一种用于更改产品库存数量的方法:

````csharp

public Product SetStockCount(int stockCount)

{

return SetStockCountInternal(stockCount);

}

private Product SetStockCountInternal(int stockCount, bool triggerEvent = true)

{

if (StockCount < 0)

{

throw new ArgumentException($"{nameof(stockCount)} can not be less than 0!");

}

if (StockCount == stockCount)

{

return this;

}

if (triggerEvent)

{

AddDistributedEvent(

new ProductStockCountChangedEto(

Id, StockCount, stockCount

)

);

}

StockCount = stockCount;

return this;

}

````

此方法还触发**\*\*分布式事件\*\***,其中带有`ProductStockCountChangedEto`参数(Eto是传统的后缀代表**\*\*E\*\***vent **\*\*T\*\***ransfer **\*\*O\*\***bject,但不是必需的)通知产品库存数量的听众已发生变化. 任何订户都可以接收此事件并根据该知识执行操作.

RabbitMQ为此解决方案分发事件. 但是ABP通过提供必要的抽象来独立于消息代理[Event Bus](../Event-Bus.md)文档).

如前所述,该模块强制始终使用`ProductManager`来创建新的`Product`. `ProductManager`是一个简单的域服务,定义如下:

````csharp

public class ProductManager : DomainService

{

private readonly IRepository<Product, Guid> \_productRepository;

public ProductManager(IRepository<Product, Guid> productRepository)

{

\_productRepository = productRepository;

}

public async Task<Product> CreateAsync(

[NotNull] string code,

[NotNull] string name,

float price = 0.0f,

int stockCount = 0)

{

var existingProduct =

await \_productRepository.FirstOrDefaultAsync(p => p.Code == code);

if (existingProduct != null)

{

throw new ProductCodeAlreadyExistsException(code);

}

return await \_productRepository.InsertAsync(

new Product(

GuidGenerator.Create(),

code,

name,

price,

stockCount

)

);

}

}

````

\* 它检查之前是否使用过给定的代码. 否则抛出`ProductCodeAlreadyExistsException`.

\* 使用`GuidGenerator`(`IGuidGenerator`)服务来创建一个新的`Guid`.

\* 它将实体插入存储库.

因此,通过这种设计,产品代码的唯一性得到保证.

`ProductCodeAlreadyExistsException`是一个域/业务异常,定义如下:

````csharp

public class ProductCodeAlreadyExistsException : BusinessException

{

public ProductCodeAlreadyExistsException(string productCode)

: base("PM:000001", $"A product with code {productCode} has already exists!")

{

}

}

````

`PM:000001`是发送给客户端的异常类型的代码,因此他们可以理解错误类型. 在这种情况下没有实现,但也可以本地化业务异常. 请参阅[异常处理文档](../Exception-Handling.md).

#### 应用层

该模块的应用层有两个服务:

\* `ProductAppService`主要由后端管理应用程序用于管理(创建,更新,删除.)产品. 它需要许可才能执行任何操作.

\* 公共网站使用`PublicProductAppService`来向访问者显示产品列表. 它不需要任何权限,因为大多数访问者都没有登录到应用程序.

请注意; 每个应用程序分离应用程序层可能是更好的原则, 而不是将两个应用程序服务放入同一个项目中. 但是我们在这个解决方案中简化了它们.

例如,`ProductAppService`具有以下更新产品的方法:

````csharp

[Authorize(ProductManagementPermissions.Products.Update)]

public async Task<ProductDto> UpdateAsync(Guid id, UpdateProductDto input)

{

var product = await \_productRepository.GetAsync(id);

product.SetName(input.Name);

product.SetPrice(input.Price);

product.SetStockCount(input.StockCount);

return ObjectMapper.Map<Product, ProductDto>(product);

}

````

\* 它定义了所需的权限(*\*ProductManagementPermissions.Products.Update\**是一个值为`ProductManagement.Update`的常量)来执行此操作.

\* 获取产品的ID,DTO包含要更新的值.

\* 从存储库中获取相关的产品实体.

\* 使用`Product`类的相关方法(如`SetName`)来更改属性,因为它们是私有set方法,更改值的唯一方法是使用实体方法.

\* 通过使用[ObjectMapper](../Object-To-Object-Mapping.md)向客户端返回更新的`ProductDto`(客户端可能由于某种原因需要它).

实施可能会根据要求而有所不同. 此实现遵循此处提供的[最佳实践](../Best-Practices/Application-Services.md).

#### 其他层

请参阅源代码中的其他层.

## 基础设施

### 消息和RabbitMQ

异步消息传递是分布式系统中的关键概念. 它可以以松散耦合的方式与容错进行通信. 在发送消息时,它不要求双方都在线. 因此,它是微服务架构中广泛使用的通信模式.

#### 分布式事件总线

分布式事件(事件总线)是一种消息传递方式,其中服务引发/触发事件,而其他服务注册/侦听这些事件,以便在发生重要事件时得到通知. ABP通过提供约定,服务和集成使分布式事件更易于使用.

你已经看到`Product`类使用以下代码行发布事件:

````csharp

AddDistributedEvent(new ProductStockCountChangedEto(Id, StockCount, stockCount));

````

`ProductStockCountChangedEto` 定义如下:

````csharp

[Serializable]

public class ProductStockCountChangedEto : EtoBase

{

public Guid Id { get; }

public int OldCount { get; set; }

public int CurrentCount { get; set; }

private ProductStockCountChangedEto()

{

//Default constructor is needed for deserialization.

}

public ProductStockCountChangedEto(Guid id, int oldCount, int currentCount)

{

Id = id;

OldCount = oldCount;

CurrentCount = currentCount;

}

}

````

该对象存储有关该事件的必要信息. 通过使用通用的`ProductStockCountChangedEto`参数实现`IDistributedEventHandler`接口,另一个服务可以轻松注册到此事件:

````csharp

public class MyHandler : IDistributedEventHandler<ProductStockCountChangedEto>

{

public async Task HandleEventAsync(ProductStockCountChangedEto eventData)

{

var productId = eventData.Id;

//...

}

}

````

当你使用[Volo.Abp.EventBus.RabbitMQ](https://www.nuget.org/packages/Volo.Abp.EventBus.RabbitMQ)包时,所有集成和通信都由ABP框架完成.如果需要从实体发布事件,只需注入`IDistributedEventBus`并使用`PublishAsync`方法.

有关分布式事件系统的更多信息, 请参见[Event Bus](../Event-Bus.md)文档.

#### RabbitMQ配置

在此解决方案中, [RabbitMQ](https://www.rabbitmq.com/)用于消息传递和分布式事件.

[Volo.Abp.EventBus.RabbitMQ](https://www.nuget.org/packages/Volo.Abp.EventBus.RabbitMQ)包需要集成到RabbitMQ以用于分布式事件系统. 然后你需要为模块的`AbpEventBusRabbitMqModule`添加依赖项. 例如`ProductServiceHostModule`声明了这种依赖.

默认情况下,`abpEventBusRabbitMqModule`从`appsettings.json`获取配置. 例如产品服务具有以下配置:

````json

"RabbitMQ": {

"Connections": {

"Default": {

"HostName": "localhost"

}

},

"EventBus": {

"ClientName": "MsDemo\_ProductService",

"ExchangeName": "MsDemo"

}

}

````

### 缓存和Redis

分布式系统显然需要分布式和共享缓存,而不是每个服务的隔离内存缓存.

[Redis](https://redis.io/)在此解决方案中用作分布式缓存. 该解决方案使用Microsoft的标准[Microsoft.Extensions.Caching.Redis](https://www.nuget.org/packages/Microsoft.Extensions.Caching.Redis) 包进行集成. 使用和配置此程序包时,所有应用程序和服务都使用Redis缓存. 有关详细信息请参阅[Microsoft的文档](https://docs.microsoft.com/en-us/aspnet/core/performance/caching/distributed).

该解决方案还使用[Microsoft.AspNetCore.DataProtection.StackExchangeRedis](https://www.nuget.org/packages/Microsoft.AspNetCore.DataProtection.StackExchangeRedis)包在Redis缓存上共享应用程序和服务之间的数据保护密钥.

### Logging,Serilog,Elasticsearch和Kibana

该解决方案使用[Serilog](https://serilog.net/)作为日志库. 它是一个广泛使用的库,有许多数据源集成,包括[Elasticsearch](https://www.elastic.co/products/elasticsearch).

使用类似于下面给出的代码块在`Program.cs`文件中完成日志配置:

````csharp

Log.Logger = new LoggerConfiguration()

.MinimumLevel.Debug()

.MinimumLevel.Override("Microsoft", LogEventLevel.Information)

.Enrich.WithProperty("Application", "ProductService")

.Enrich.FromLogContext()

.WriteTo.File("Logs/logs.txt")

.WriteTo.Elasticsearch(

new ElasticsearchSinkOptions(new Uri(configuration["ElasticSearch:Url"]))

{

AutoRegisterTemplate = true,

AutoRegisterTemplateVersion = AutoRegisterTemplateVersion.ESv6,

IndexFormat = "msdemo-log-{0:yyyy.MM}"

})

.CreateLogger();

````

这会配置多个日志输出目标:File和Elasticsearch.对于此示例,`Application`属性设置为`ProductService`. 这是一种区分单个数据库中多个服务的日志的方法. 然后, 你可以通过`Application`名称查询日志.

从`appsettings.json`配置文件中读取Elasticsearch URL:

````json

"ElasticSearch": {

"Url": "http://localhost:9200"

}

````

如果你使用Kibana, 它是一个与Elasticsearch完美集成的可视化工具,可以看到有关你的日志的一些UI:

![microservice-sample-kibana-2](../images/microservice-sample-kibana-2.png)

*\*Figure - 一个仪表板,显示服务/应用程序的日志和错误计数.\**

![microservice-sample-kibana-1](../images/microservice-sample-kibana-1.png)

*\*Figure - 日志条目列表\**

Kibana URL默认为`http://localhost:5601/`.

### 审计日志

ABP提供自动审计日志记录,详细保存每个请求(当前用户,浏览器/客户端,执行了哪些操作,哪些实体更改,甚至实体的哪些属性已更新). 有关详细信息,请参阅[审计日志文档](../Audit-Logging.md).

所有服务和应用程序都配置为编写审核日志. 审核日志将保存到MsDemo\_Identity SQL数据库中. 因此,你可以从单个点查询所有应用程序的所有审核日志.

审核日志记录具有`CorrelationId`属性,可用于跟踪请求. 当服务在单个Web请求中调用另一个服务时,它们都会使用相同的`CorrelationId`保存审核日志. 请参阅数据库中的`AbpAuditLogs`表.

### 多租户

该解决方案已配置提供[多租户](../Multi-Tenancy.md)系统,其中每个租户可以拥有其隔离的用户,角色,权限和其他数据.

# 应用模块

## 概述

# 应用程序模块

ABP是一个 **\*\*模块化的应用程序框架\*\*** 由十多个 **\*\*NuGet & NPM packages\*\*** 组成. 它提供了一个完整的基础设施来构建你自己的应用程序模块,这些模块包含实体,服务,数据库集成,API,UI组件等.

**\*\*有两种类型的模块.\*\*** 它们没有任何结构上的差异,只是按照功能和目地分类:

\* [\*\*框架模块\*\*](https://github.com/abpframework/abp/tree/master/framework/src): 这些是 **\*\*框架的核心模块\*\***,像缓存,邮件,主题,安全性,序列化,验证,Ef Core集成,MongoDB集成...等等. 它们没有应用程序/业务功能,但通过提供通用基础架构,集成和抽象会使你的日常开发更加容易.

\* [\*\*应用程序模块\*\*](https://github.com/abpframework/abp/tree/master/modules): 这些模块是实现特定的应用程序/业务功能,像 博客, 文档管理, 身份管理, 租户管理... 等等. 它是通常有自己的实体,服务,API和UI组件.

## 开源的应用程序模块

有一些由ABP框架开发和维护的 **\*\*开源免费\*\*** 的应用程序模块:

\* [\*\*Account\*\*](Account.md): 提供账户管理UI,并允许用户登录/注册应用程序.

\* [\*\*Audit Logging\*\*](Audit-Logging.md): 用于将审计日志持久化到数据库.

\* [\*\*Background Jobs\*\*](Background-Jobs.md): 用于在使用默认后台作业管理器时保存后台作业.

\* [\*\*Blogging\*\*](Blogging.md): 用于创建精美的博客. ABP的[博客](https://blog.abp.io/) 就使用了此模块.

\* [\*\*Docs\*\*](Docs.md): 用于创建技术文档页面. ABP的[文档](https://abp.io/documents/) 就使用了此模块.

\* [\*\*Feature Management\*\*](Feature-Management.md): 用于保存和管理功能.

\* [\*\*Identity\*\*](Identity.md): 基于Microsoft Identity管理组织单元,角色,用户和他们的权限.

\* [\*\*OpenIddict\*\*](OpenIddict.md): 集成了OpenIddict.

\* [\*\*Identity Server\*\*](IdentityServer.md): 集成了IdentityServer4.

\* [\*\*Permission Management\*\*](Permission-Management.md): 用于保存权限.

\* [\*\*Setting Management\*\*](Setting-Management.md): 用于保存设置.

\* [\*\*Tenant Management\*\*](Tenant-Management.md): 管理[多租户](../Multi-Tenancy.md)应用程序的租户.

\* [\*\*Users\*\*](Users.md): 抽象用户, 因此其他模块可以依赖此模块而不是Identity模块.

\* [\*\*Virtual File Explorer\*\*](Virtual-File-Explorer.md): 提供简单的UI查看[虚拟文件系统](../Virtual-File-System.md)中的文件.

模块化文档正在编写中. 请参阅[这个仓库](https://github.com/abpframework/abp/tree/master/modules)获取所有模块的源代码.

## 商业应用模块

[ABP商业](https://commercial.abp.io/)许可证在ABP框架上提供了额外的预构建应用程序模块. 参见ABP商业版提供的[模块列表](https://commercial.abp.io/modules).

## 账户

# 账户模块

该模块提供必要的UI页面与组件使用户登录和注册到应用程序.

> 文档正在更新

## 社交/外部登录

### 示例: Facebook 认证

按照[ASP.NET Core Facebook集成文档](https://docs.microsoft.com/zh-cn/aspnet/core/security/authentication/social/facebook-logins)向你应用程序添加Facebook登录.

#### 添加NuGet包

添加[Microsoft.AspNetCore.Authentication.Facebook]包到你的项目. 基于你的架构,可能是 `.Web`,`.AuthServer`(对于分层启动)或 `.Host` 项目.

#### 配置提供程序

在你模块的 `ConfigureServices` 方法中使用 `.AddFacebook(...)` 扩展方法来配置客户端:

````csharp

context.Services.AddAuthentication()

.AddFacebook(facebook =>

{

facebook.AppId = "...";

facebook.AppSecret = "...";

facebook.Scope.Add("email");

facebook.Scope.Add("public\_profile");

});

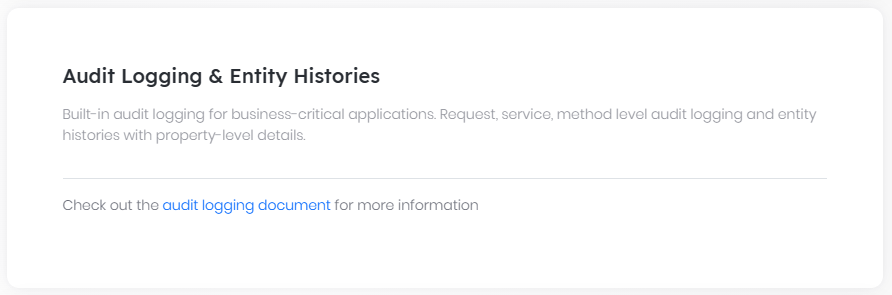
````

> 最佳实践是使用 `appsettings.json` 或ASP.NET Core用户机密系统来存储你的凭据,而不是像这样硬编码值. 请参阅[微软](https://docs.microsoft.com/en-us/aspnet/core/security/authentication/social/facebook-logins)文档了解如何使用用户机密.

### Angular UI

从v3.1开始,Angular UI使用授权码流程(作为最佳实践)通过重定向到MVC UI登录页面来对用户进行身份验证. 因此,即使你使用的是Angular UI,社交/外部登录集成也与上面说明的相同.并且可以开箱即用.

## 审计日志



# 审计日志模块

审计日志模块实现了 `IAuditingStore` 将审计日志对象保存到数据库中.

> [启动模板](../Startup-Templates/Index.md)已经安装并配置了审计日志模块,所以你不需要手动安装到你的应用程序.

参阅[审计日志系统](../Audit-Logging.md)文档了解更多关于审计日志的内容.

## 后台作业

# 后台作业模块

后台作业模块实现了 `IBackgroundJobStore` 接口,并且可以使用ABP框架的默认后台作业管理.如果你不想使用这个模块,那么你需要自己实现 `IBackgroundJobStore` 接口.

> 本文档仅介绍后台作业模块,该模块将后台作业持久化到数据库.有关后台作业系统的更多信息,请参阅[后台作业](../Background-Jobs.md)文档.

## 如何使用

当你使用ABP框架[创建一个新的解决方案](https://abp.io/get-started)时,这个模块是（作为NuGet/NPM包）预先安装的.你可以继续将其作为软件包使用并轻松获取更新,也可以将其源代码包含到解决方案中（请参阅 `get-source` [CLI](../CLI.md)命令）以开发自定义模块.

### 源代码

此模块的源代码可在[此处](https://github.com/abpframework/abp/tree/dev/modules/background-jobs)访问.源代码是由[MIT](https://choosealicense.com/licenses/mit/)授权的,所以你可以自由使用和定制它.

## 内部结构

### 领域层

#### 聚合

- `BackgroundJobRecord` (聚合根): 表示后台工作记录.

#### 仓储

为该模块定义了以下自定义仓储:

- `IBackgroundJobRepository`

### 数据库提供程序

#### 通用

##### 表/集合的前缀与架构

默认情况下,所有表/集合都使用 `Abp` 前缀.如果需要更改表前缀或设置架构名称（如果数据库提供程序支持）,请在 `BackgroundJobsDbProperties` 类上设置静态属性.

##### 连接字符串

此模块使用 `AbpBackgroundJobs` 作为连接字符串名称.如果不使用此名称定义连接字符串,它将返回 `Default` 连接字符串.有关详细信息,请参阅[连接字符串](https://docs.abp.io/en/abp/latest/Connection-Strings)文档.

#### Entity Framework Core

##### 表

- **\*\*AbpBackgroundJobs\*\***

#### MongoDB

##### 集合

- **\*\*AbpBackgroundJobs\*\***

## 另请参阅

\* [后台作业系统](../Background-Jobs.md)

## CMS Kit

# 内容管理系统套件模块

此模块为您的应用程序提供内容管理系统 (Content Management System, CMS) 功能. 它提供 **\*\*核心构建块\*\*** 和完整工作的 **\*\*子系统\*\***, 以创建启用 CMS 功能的您自己的网站, 或出于任何目的使用网站中的构建块.

> **\*\*此模块目前仅适用于 MVC / Razor 页面 UI\*\***. 虽然没有官方的 Blazor 软件包, 但它也可以在 Blazor 服务器 UI 中工作, 因为实际上 Blazor 服务器 UI 实际上是一个运行在 ASP.NET Core MVC / Razor 页面应用程序的混合型应用程序.

目前提供以下功能:

\* 提供 [\*\*页面\*\*](Pages.md) 管理系统来管理具有动态 URL 的动态页面.

\* 提供 [\*\*博客\*\*](Blogging.md) 系统来创建发表具有多种博客支持的博客文章.

\* 提供 [\*\*标签\*\*](Tags.md) 系统来标记任何资源, 如博客文章.

\* 提供 [\*\*评论\*\*](Comments.md) 系统来添加对任何资源的评论功能, 如博客文章或产品评价页面.

\* 提供 [\*\*反应\*\*](Reactions.md) 系统来添加对任何资源的反应 (表情符号) 功能, 如博客文章或评论.

\* 提供 [\*\*评级\*\*](Ratings.md) 系统来添加对任何资源的评级功能.

\* 提供 [\*\*菜单\*\*](Menus.md) 系统来动态管理公共菜单.

\* 提供 [\*\*动态部件\*\*](Dynamic-Widget.md) 系统在页面和博客文章中创建动态部件.

点击功能以了解和学习如何去使用它.

所有功能均可单独使用. 如果你禁用了一个功能, 则在 [全局功能](../../Global-Features.md) 系统的帮助下, 该功能会从你的应用程序甚至数据库表中完全消失.

## 预备要求

- 此模块依赖于 [Blob 存储](../../Blob-Storing.md) 模块来保存媒体内容.

> 确保 `BlobStoring` 模块已安装并至少正确地配置了一个提供程序. 请查阅 [文档](../../Blob-Storing.md) 了解更多信息.

- CMS Kit 使用 [分布式缓存](../../Caching.md) 来提高响应速度.

> 强烈建议在分布式/集群部署中为实现数据一致性使用分布式缓存, 如 [Redis](../../Redis-Cache.md).

## 如何安装

可以使用 [ABP CLI](../../CLI.md) 的 `add-module` 命令为解决方案安装模块. 您可以使用以下命令在命令行中安装 CMS Kit 模块:

```bash

abp add-module Volo.CmsKit

```

> 默认情况下, Cms-Kit `GlobalFeature` 被禁用. 因此初始迁移将为空. 所以, 当你使用 EF Core 安装时,你可以添加 `--skip-db-migrations` 命令来跳过迁移. 启用 Cms-Kit 全局功能后, 请添加新的迁移.

安装过程完成后, 在您的解决方案 `Domain.Shared` 项目中打开 `GlobalFeatureConfigurator` 类, 并将以下代码写入 `Configure` 方法中, 以启用 CMS Kit 模块的全部功能.

```csharp

GlobalFeatureManager.Instance.Modules.CmsKit(cmsKit =>

{

cmsKit.EnableAll();

});

```

你可能更愿意逐个启用这些功能, 而不是启用全部功能. 以下示例仅启用了 [标签](Tags.md) 和 [评论](Comments.md) 功能:

````csharp

GlobalFeatureManager.Instance.Modules.CmsKit(cmsKit =>

{

cmsKit.Tags.Enable();

cmsKit.Comments.Enable();

});

````

> 如果你使用 EF Core, 不要忘记添加一个新的迁移并更新你的数据库.

## 软件包

此模块遵循 [模块开发最佳实践指南](https://docs.abp.io/zh-Hans/abp/latest/Best-Practices/Index), 由多个 NuGet 和 NPM 软件包组成. 如果你想了解软件包及其之间的关系, 请参阅指南.

CMS Kit 软件包专为各种使用场景而设计. 如果您查阅了 [CMS Kit 软件包](https://www.nuget.org/packages?q=Volo.CmsKit) 您将看到一些有 `Admin` 和 `Public` 后缀的软件包. 该模块有两个应用程序层, 原因是他们可能被用于不同类型的应用程序. 这些应用程序层仅使用一个领域层.

- `Volo.CmsKit.Admin.\*` 软件包包括管理员 (后台) 应用程序所必须的功能.

- `Volo.CmsKit.Public.\*` 软件包包括被用于用户阅读博客文章和发表评论的公共网站上的功能.

- `Volo.CmsKit.\*` (不带 Admin/Public 后缀) 软件包称为统一包. 统一包分别是添加 Admin 和 Public (相关层的) 软件包的快照. 如果您有一个用于管理和公共网站的单应用程序, 您可以使用这些软件包.

## 内部结构

### 表/集合 前缀&架构

所有表/集合使用 `Cms` 作为默认前缀. 如果需要更改表的前缀或者设置一个架构名称 (如果你的数据库提供程序支持), 请在 `CmsKitDbProperties` 类中设置静态属性.

### 连接字符串

此模块使用 `CmsKit` 作为连接字符串的名称. 如果您未使用此名称定义连接字符串, 它将回退为 `Default` 连接字符串.

有关详细信息, 请参阅 [连接字符串](https://docs.abp.io/en/abp/latest/Connection-Strings) 文档.

## 文档

# 文档模块

## 什么是文档模块?

文档模块是ABP框架的一个应用程序模块. 它简化了软件文档的制作. 这个模块是开源免费的.

### 集成

目前文档模块提供提供了两种支持的存储,Github与文件系统.

### 托管

文档模块是一个应用程序模块,不提供任何托管的解决方案,你可以在本地或云上托管文档.

### 版本

当你使用GitHub存储文档时,文档模块支持多版本. 如果你的文档具有多个版本, UI上有一个组合框,用于切换版本. 如果你选择使用文件系统存储文档, 那么它不支持多版本.

ABP框架的[文档](docs.abp.io)也是使用的此模块.

> 文档模块遵循 [模块化架构最佳实践](../Best-Practices/Module-Architecture.md) 指南.

## 安装

### 1- 下载

如果你没有现有的ABP项目, 这个步骤向你展示如何在[abp.io](https://abp.io)创建一个新项目并添加文档模块. 如果你本地已经有了一个ABP项目, 那么你可以跳过这一步.

推荐使用ABP CLI创建新项目,使用以下命令行:

`abp new Acme.MyProject`

你也可以在浏览器中导航到 https://abp.io/get-started. 输入项目名称为 `Acme.MyProject`, 其它保持默认选项.

请注意,本文档包含了 `Entity Framework Core` 提供者 不过你也可以选择 `MongoDB` 做为数据库提供者.

![创建新项目](../images/docs-module\_download-new-abp-project.png)

### 2- 运行这个空项目

下载项目后, 解压压缩文档并且打开 `Acme.MyProject.sln`. 你可以看到这个解决方案包含了 `Application`, `Application.Contrawcts`, `DbMigrator`, `Domain`, `Domain.Shared`, `EntityFrameworkCore`, `EntityFrameworkCore.DbMigations`, `HttpApi`, `HttpApi.Client` 和 `Web` 项目. 右键选择 `Acme.MyProject.Web` 项目**\*\*设置为启动项目\*\***.

![创建新项目](../images/docs-module\_solution-explorer.png)

数据库连接字符串位于`Acme.MyProject.Web`项目的`appsettings.json`中. 如果你有不同的数据库配置, 可以修改这个连接字符串.

```json

{

"ConnectionStrings": {

"Default": "Server=(LocalDb)\\MSSQLLocalDB;Database=MyProject;Trusted\_Connection=True"

}

}

```

运行 `Acme.MyProject.DbMigrator` 项目,它会负责应用迁移与初始化种子数据. 数据库`MyProject`将在数据库服务器中创建.

现在一个空的ABP项目已经创建完成! 现在你可以运行项目并且查看网站.

输入用户名 `admin` 密码 `1q2w3E\*` 登录到网站.

### 3- 安装模块

文档模块包托管在Nuget上面. 需要有四个包安装到你的应用程序中. 每个包必须安装到相关的项目.

#### 3.1- 使用ABP CLI

建议使用ABP CLI安装模块,在解决方案文件 (`.sln`) 目录打开 `CMD` 窗口,运行以下命令:

`abp add-module Volo.Docs`

#### 3.2- 手动安装

或者你也可以手动安装nuget包到每个项目:

\* 安装[Volo.Docs.Domain](https://www.nuget.org/packages/Volo.Docs.Domain/) nuget包到 `Acme.MyProject.Domain` 项目.

`Install-Package Volo.Docs.Domain`

\* 安装[Volo.Docs.EntityFrameworkCore](https://www.nuget.org/packages/Volo.Docs.EntityFrameworkCore/) nuget包到 `Acme.MyProject.EntityFrameworkCore` 项目.

`Install-Package Volo.Docs.EntityFrameworkCore`

\* 安装[Volo.Docs.Application](https://www.nuget.org/packages/Volo.Docs.Application/) nuget包到 `Acme.MyProject.Application` 项目.

`Install-Package Volo.Docs.Application`

\* 安装[Volo.Docs.Web](https://www.nuget.org/packages/Volo.Docs.Domain/) nuget包到 `Acme.MyProject.Web` 项目.

`Install-Package Volo.Docs.Web`

##### 3.2.1- 添加模块依赖

一个ABP模块必须声明 `[DependsOn]` attribute 如果它依赖于另一个模块. 每个模块都必须在相关的项目的`[DependsOn]`Attribute 中添加.

\* 打开 `MyProjectDomainModule.cs`并且添加 `typeof(DocsDomainModule)` 如下所示;

```csharp

[DependsOn(

typeof(DocsDomainModule),

typeof(AbpIdentityDomainModule),

typeof(AbpAuditingModule),

typeof(BackgroundJobsDomainModule),

typeof(AbpAuditLoggingDomainModule)

)]

public class MyProjectDomainModule : AbpModule

{

//...

}

```

\* 打开 `MyProjectEntityFrameworkCoreModule.cs`并且添加 `typeof(DocsEntityFrameworkCoreModule)` 如下所示;

```csharp

[DependsOn(

typeof(DocsEntityFrameworkCoreModule),

typeof(MyProjectDomainModule),

typeof(AbpIdentityEntityFrameworkCoreModule),

typeof(AbpPermissionManagementEntityFrameworkCoreModule),

typeof(AbpSettingManagementEntityFrameworkCoreModule),

typeof(AbpEntityFrameworkCoreSqlServerModule),

typeof(BackgroundJobsEntityFrameworkCoreModule),

typeof(AbpAuditLoggingEntityFrameworkCoreModule)

)]

public class MyProjectEntityFrameworkCoreModule : AbpModule

{

//...

}

```

\* 打开 `MyProjectApplicationModule.cs`并且添加 `typeof(DocsApplicationModule)` 如下所示;

```csharp

[DependsOn(

typeof(DocsApplicationModule),

typeof(MyProjectDomainModule),

typeof(AbpIdentityApplicationModule))]

public class MyProjectApplicationModule : AbpModule

{

public override void ConfigureServices(ServiceConfigurationContext context)

{

Configure<PermissionOptions>(options =>

{

options.DefinitionProviders.Add<MyProjectPermissionDefinitionProvider>();

});

Configure<AbpAutoMapperOptions>(options =>

{

options.AddProfile<MyProjectApplicationAutoMapperProfile>();

});

}

}

```

\* 打开 `MyProjectWebModule.cs`并且添加 `typeof(DocsWebModule)` 如下所示;

```csharp

[DependsOn(

typeof(DocsWebModule),

typeof(MyProjectApplicationModule),

typeof(MyProjectEntityFrameworkCoreModule),

typeof(AbpAutofacModule),

typeof(AbpIdentityWebModule),

typeof(AbpAccountWebModule),

typeof(AbpAspNetCoreMvcUiBasicThemeModule)

)]

public class MyProjectWebModule : AbpModule

{

//...

}

```

##### 3.2.2- 添加NPM包

打开 `package.json` 添加 `@abp/docs` 如下所示:

```json

{

"version": "1.0.0",

"name": "my-app",

"private": true,

"dependencies": {

"@abp/aspnetcore.mvc.ui.theme.basic": "^5.0.0",

"@abp/docs": "^5.0.0"

}

}

```

然后在 `Acme.MyProject.Web` 项目目录打开命令行终端运行以下命令:

````bash

abp install-libs

````

### 4- 数据库集成

#### 4.1- Entity Framework 集成

如果你选择了Entity Framework 做为数据库供应者,你需要配置文档模块. 做以下操作;

\* 打开 `MyProjectMigrationsDbContext.cs` 并且添加 `builder.ConfigureDocs()` 到 `OnModelCreating()` 方法中

```csharp

public class MyProjectMigrationsDbContext : AbpDbContext<MyProjectMigrationsDbContext>

{

public MyProjectMigrationsDbContext(DbContextOptions<MyProjectMigrationsDbContext> options)

: base(options)

{

}

protected override void OnModelCreating(ModelBuilder builder)

{

base.OnModelCreating(builder);

/\* Include modules to your migration db context \*/

builder.ConfigurePermissionManagement();

builder.ConfigureSettingManagement();

builder.ConfigureBackgroundJobs();

builder.ConfigureAuditLogging();

builder.ConfigureIdentity();

builder.ConfigureIdentityServer();

builder.ConfigureFeatureManagement();

builder.ConfigureTenantManagement();

builder.ConfigureDocs(); //Add this line to configure the Docs Module

/\* Configure customizations for entities from the modules included \*/

builder.Entity<IdentityUser>(b =>

{

b.ConfigureCustomUserProperties();

});

/\* Configure your own tables/entities inside the ConfigureQaDoc method \*/

builder.ConfigureMyProject();

}

}

```

\* 打开 `Visual Studio` 的 `包管理控制台` 选择 `Acme.MyProject.EntityFrameworkCore.DbMigrations` 做为默认项目. 然后编写以下命令为文档模块添加迁移.

```csharp

add-migration Added\_Docs\_Module

```

当命令执行成功后 , 你会看到`Acme.MyProject.EntityFrameworkCore.DbMigrations\Migrations` 目录下有名为 `20181221111621\_Added\_Docs\_Module` 的迁移文件.

现在更新数据库. 在 `Visual Studio` 的 `包管理控制台` 中执行以下代码. 要确认已 `Acme.MyProject.EntityFrameworkCore.DbMigrations` 项目设置为默认项目.

```csharp

update-database

```

最后你可以查看数据库中创建的新表,例如你可以看到 `DocsProjects` 表已经添加到数据库中.

### 5- 链接文档模块

文档模块的默认路由是;

```

/Documents

```

添加文档模块的链接到你的应用程序菜单中;

\* 打开 `MyProjectMenuContributor.cs` 并且在 `ConfigureMainMenuAsync()` 方法方法中添加以下代码.

```csharp

context.Menu.Items.Add(new ApplicationMenuItem("MyProject.Docs", l["Menu:Docs"], "/Documents"));

```

最后 **\*\*MyProjectMenuContributor.cs\*\*** 有以下内容

```csharp

private async Task ConfigureMainMenuAsync(MenuConfigurationContext context)

{

var l = context.ServiceProvider.GetRequiredService<IStringLocalizer<MyProjectResource>>();

context.Menu.Items.Insert(0, new ApplicationMenuItem("MyProject.Home", l["Menu:Home"], "/"));

context.Menu.Items.Add(new ApplicationMenuItem("MyProject.Docs", l["Menu:Docs"], "/Documents"));

}

```

`Menu:Docs` 关键词是本地化的Key. 要本地化菜单文本, 打开`Acme.MyProject.Domain` 中的 `Localization\MyProject\zh-Hans.json`. 添加以下行.

```json

"Menu:Docs": "文档"

```

最后 **\*\*zh-Hans.json\*\*** 有以下内容

```json

{

"culture": "zh-Hans",

"texts": {

"Menu:Home": "首页",

"Welcome": "欢迎",

"LongWelcomeMessage": "欢迎来到该应用程序. 这是一个基于ABP框架的启动项目. 有关更多信息, 请访问 abp.io.",

"Menu:Docs": "文档"

}

}

```

现在菜单中已经添加了文档模块项. 运行Web应用程序并且在浏览器中打开 `http://localhost:YOUR\_PORT\_NUMBER/documents` URL.

你会看到一个警告;

```

There are no projects yet!

```

这个警告是正常的,因为我们还没有添加任何项目.

### 6- 添加文档项目

在数据库中打开 `DocsProjects`, 并且插入包含以下字段的新记录;

\* **\*\*Name\*\***: 在Web页面上文档的显示名称.

\* **\*\*ShortName\*\***: 在文档URL中使用的友好的简短URL名称.

\* **\*\*Format\*\***: 文档的格式 ( Markdown: `md`, HTML: `html`)

\* **\*\*DefaultDocumentName\*\***: 文档的初始页面.

\* **\*\*NavigationDocumentName\*\***: 导航菜单(索引)的文档.

\* **\*\*MinimumVersion\*\***: 显示文档的最低版本. 低于此的版本不会列出.

\* **\*\*DocumentStoreType\*\***: 文档的来源 ( GitHub:`GitHub`,文件系统`FileSystem`).

\* **\*\*ExtraProperties\*\***: 序列化的`JSON`, 它存储所选 `DocumentStoreType` 的特殊配置.

\* **\*\*MainWebsiteUrl\*\***: 用户单击文档模块页面Logo时跳转的URL.你只需设置为`/`即可链接到网站根地址.

\* **\*\*LatestVersionBranchName\*\***: 这是GitHub的配置.它是检索文档的分支名称.你可以将其设置为`master`.

#### "GitHub" 项目的示例记录

你可以使用 [ABP Framework](https://github.com/abpframework/abp/) GitHub文档来配置Github文档存储.

- Name: `ABP framework (GitHub)`

- ShortName: `abp`

- Format: `md`

- DefaultDocumentName: `Index`

- NavigationDocumentName: `docs-nav.json`

- MinimumVersion: `<NULL>` (no minimum version)

- DocumentStoreType: `GitHub`

- ExtraProperties:

```json

{"GitHubRootUrl":"https://github.com/abpframework/abp/tree/{version}/docs/zh-Hans/","GitHubAccessToken":"\*\*\*","GitHubUserAgent":""}

```

注意 `GitHubAccessToken` 用 `\*\*\*` 掩盖. 这是一个私人令牌,你必须从GitHub获取它. 请参阅 https://help.github.com/articles/creating-a-personal-access-token-for-the-command-line/

- MainWebsiteUrl: `/`

- LatestVersionBranchName: `dev`

对于 `SQL` 数据库,你可以使用下面的 `T-SQL` 命令将指定的示例插入到 `DocsProjects` 表中:

```mssql

INSERT [dbo].[DocsProjects] ([Id], [Name], [ShortName], [Format], [DefaultDocumentName], [NavigationDocumentName], [MinimumVersion], [DocumentStoreType], [ExtraProperties], [MainWebsiteUrl], [LatestVersionBranchName], [ParametersDocumentName]) VALUES (N'12f21123-e08e-4f15-bedb-ae0b2d939658', N'ABP framework (GitHub)', N'abp', N'md', N'Index', N'docs-nav.json', NULL, N'GitHub', N'{"GitHubRootUrl":"https://github.com/abpframework/abp/tree/{version}/docs","GitHubAccessToken":"\*\*\*","GitHubUserAgent":""}', N'/', N'dev', N'')

```

请注意,`GitHubAccessToken` 被屏蔽了.它是一个私人令牌,你必须获得自己的令牌并替换 `\*\*\*` 字符串.

现在你可以运行应用程序并导航到 `/Documents`.

#### "FileSystem" 项目的示例记录

你可以使用 [ABP Framework](https://github.com/abpframework/abp/) GitHub文档来配置你的文件系统存储.

- Name: `ABP framework (FileSystem)`

- ShortName: `abp`

- Format: `md`

- DefaultDocumentName: `Index`

- NavigationDocumentName: `docs-nav.json`

- MinimumVersion: `<NULL>` (no minimum version)

- DocumentStoreType: `FileSystem`

- ExtraProperties:

```json

{"Path":"C:\\Github\\abp\\docs"}

```

请注意 `Path` 必须使用本地docs目录替换. 你可以从https://github.com/abpframework/abp/tree/master/docs获取ABP Framework的文档并且复制到该目录 `C:\\Github\\abp\\docs` 使其正常工作.

- MainWebsiteUrl: `/`

- LatestVersionBranchName: `<NULL>`

对于 `SQL` 数据库,你可以使用下面的 `T-SQL` 命令将指定的示例插入到 `DocsProjects` 表中:

```mssql

INSERT [dbo].[DocsProjects] ([Id], [Name], [ShortName], [Format], [DefaultDocumentName], [NavigationDocumentName], [MinimumVersion], [DocumentStoreType], [ExtraProperties], [MainWebsiteUrl], [LatestVersionBranchName], [ParametersDocumentName]) VALUES (N'12f21123-e08e-4f15-bedb-ae0b2d939659', N'ABP framework (FileSystem)', N'abp', N'md', N'Index', N'docs-nav.json', NULL, N'FileSystem', N'{"Path":"C:\\Github\\abp\\docs"}', N'/', NULL, N'')

```

添加上面的一个示例项目后运行该应用程序. 在菜单中你会看到`文档` 链接,点击菜单链接打开文档页面.

到目前为止, 我们已经从abp.io网站创建了一个新的应用程序,并为Docs模块做好准备.

### 7- 添加一个新文档

在示例项目记录中, 你可以看到 `Format` 被指定为 `md` 指的是 [Mark Down](https://en.wikipedia.org/wiki/Markdown). 你可以打开下面的链接查看语法;

https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet

ABP文档模块可以把MarkDown渲染为HTML.

现在让我们看一下Markdown格式的示例文档.

~~~markdown

# This is a header

Welcome to Docs Module.

## This is a sub header

[This is a link](https://abp.io)

![This is an image](https://abp.io/assets/my-image.png)

## This is a code block

```csharp

public class Person

{

public string Name { get; set; }

public string Address { get; set; }

}

```

~~~

你可以使用 ABP Framework 的文档做为示例:

[https://github.com/abpframework/abp/blob/master/docs/zh-Hans/](https://github.com/abpframework/abp/blob/master/docs/zh-Hans/)

#### 有条件的部分功能(使用Scriban)

文档模块使用[Scriban](<https://github.com/lunet-io/scriban/tree/master/doc>)有条件的显示或隐藏文档的某些部分. 使用该功能你需要为每一种语言创建一个JSON文件做为**\*\*参数文档\*\***. 它包含所有键值以及它们的显示名称.

例如 [en/docs-params.json](https://github.com/abpio/abp-commercial-docs/blob/master/en/docs-params.json):

```json

{

"parameters": [{

"name": "UI",

"displayName": "UI",

"values": {

"MVC": "MVC / Razor Pages",

"NG": "Angular"

}

},

{

"name": "DB",

"displayName": "Database",

"values": {

"EF": "Entity Framework Core",

"Mongo": "MongoDB"

}

},

{

"name": "Tiered",

"displayName": "Tiered",

"values": {

"No": "Not Tiered",

"Yes": "Tiered"

}

}]

}

```

因为并不是项目中的每个文档都有章节或者不需要所有的参数,你必须声明哪些参数将用于对文档进行分段,在文档的任何地方都可以使用JSON块.

例如 [Getting-Started.md](https://github.com/abpio/abp-commercial-docs/blob/master/en/getting-started.md):

```

.....

​````json

//[doc-params]

{

"UI": ["MVC","NG"],

"DB": ["EF", "Mongo"],

"Tiered": ["Yes", "No"]

}

​````

........

```

这个部分会在渲染时自动删除.前提是这些键值必须与**\*\*参数文档\*\***中的键值匹配.

![Interface](../images/docs-section-ui.png)

现在你可以使用 **\*\*Scriban\*\*** 语法在文档中创建章节.

示例 :

````

{{ if UI == "NG" }}

\* `-u` argument specifies the UI framework, `angular` in this case.

{{ end }}

{{ if DB == "Mongo" }}

\* `-d` argument specifies the database provider, `mongodb` in this case.

{{ end }}

{{ if Tiered == "Yes" }}

\* `--tiered` argument is used to create N-tiered solution where authentication server, UI and API layers are physically separated.

{{ end }}

````

还可以在文本中使用变量,在其键中添加 **\*\*\_Value\*\*** 后缀:

````

This document assumes that you prefer to use \*\*{{ UI\_Value }}\*\* as the UI framework and \*\*{{ DB\_Value }}\*\* as the database provider.

````

如果你想要得到的当前文档的语言或版本,可以使用预定义的 **\*\*Document\_Language\_Code\*\*** 和 **\*\*DOCUMENT\_VERSION\*\*** 键(这对于创建重定向到另一个地区中另一个文档系统的链接很有用).

------

**\*\*重要提示\*\***: Scriban 的语法是 "{{" and "}}". 如果要在文档(如Angular文档)中使用转义,则必须使用转义块. 参阅 [Scriban文档](<https://github.com/lunet-io/scriban/blob/master/doc/language.md#13-escape-block> ) 了解更多信息.

### 8- 创建文档导航

导航文档是文档页面的主菜单. 它位于页面的左侧,是一个`JSON` 文件. 请查看以下示例导航文档以了解结构.

```json

{

"items":[

{

"text":"Sample Menu Item - 1",

"items":[

{

"text":"Sample Menu Item - 1.1",

"items":[

{

"text":"Sample Menu Item - 1.1.1",

"path":"SampleMenuItem\_1\_1\_1.md"

}

]

},

{

"text":"Sample Menu Item - 1.2",

"items":[

{

"text":"Sample Menu Item - 1.2.1",

"path":"SampleMenuItem\_1\_2\_1.md"

},

{

"text":"Sample Menu Item - 1.2.2",

"path":"SampleMenuItem\_1\_2\_2.md"

}

]

}

]

},

{

"text":"Sample Menu Item - 2",

"items":[

{

"text":"Sample Menu Item - 2.1",

"items":[

{

"text":"Sample Menu Item - 2.1.1",

"path":"SampleMenuItem\_2\_1\_1.md"

}

]

}

]

}

]

}

```

上面的示例 `JSON` 文件将下面的导航菜单呈现为 `HTML` .

![Navigation menu](../images/docs-module\_download-sample-navigation-menu.png)

最后,为你的项目添加了一个新的Docs模块, 该模块由GitHub提供.

## 全文搜索(Elastic Search)

文档模块支持使用Elastic Search对内容进行全文搜索. 默认没有启用, 你可以配置`DocsElasticSearchOptions`启用它.

```

Configure<DocsElasticSearchOptions>(options =>

{

options.Enable = true;

options.IndexName = "your\_index\_name"; //default IndexName is abp\_documents

});

```

应用程序启动后如果`Index`不存在则会自动创建`Index`.

`DefaultElasticClientProvider`负责创建`IElasticClient`, 默认情况下它会从`IConfiguration`中读取Elastic Search的`Url`.

如果你的 `IElasticClient` 需要其它配置请使用重写 `IElasticClientProvider` 服务并在[依赖注入](../Dependency-Injection.md)系统中替换它.

```

{

"ElasticSearch": {

"Url": "http://localhost:9200"

}

}

```

## 下一步

文档模块也可以做为独立的应用程序. 查看 [VoloDocs](../Apps/VoloDocs).

## 功能管理

# 功能管理模块

功能管理模块实现了 [功能系统](../Features.md) 定义的 `IFeatureManagementStore` 接口.

> 此文档仅介绍了将功能值持续保存到数据库的功能管理模块. 请参阅 [功能](../Features.md) 来了解功能系统.

## 如何安装

当你使用 ABP 框架 [创建一个新的解决方案](https://abp.io/get-started) 时, 此模块将被预安装(作为 NuGet/NPM 包). 你可以继续用其作为包并轻松地获取更新, 也可以将其源代码包含在解决方案中 (请参阅 `get-source` [CLI](../CLI.md) 命令) 以开发自定义模块.

### 源代码

可以 [在此处](https://github.com/abpframework/abp/tree/dev/modules/feature-management) 访问源代码。源代码使用 [MIT](https://choosealicense.com/licenses/mit/) 许可, 所以你可以免费使用和自定义它.

## 用户界面

### 功能管理对话框

功能管理模块提供了一个可重用的对话框来管理与对象相关的功能. 例如: [租户管理模块](Tenant-Management.md) 用它在租户管理页面管理租户的功能.

![features-module-opening](../images/features-module-opening.png)

当你点击租户的 *\*操作\** -> *\*功能\** 时, 将打开功能管理对话框. 下面是定义了两个功能的对话框示例截图:

![features-modal](../images/features-modal.png)

在对话框中, 你可以为租户启用, 禁用或设置功能值.

## IFeatureManager

`IFeatureManager` 是此模块提供的主要服务. 在多租户应用程序中, 它常用来读取和更改功能值. `IFeatureManager` 通常被 *\*租户管理对话框\** 使用. 但如果需要设置功能值, 你也可以注入它.

> 如果你只是想要读取或检查功能值, 请如 [功能文档](../Features.md) 中所述那样使用 `IFeatureChecker`.

**\*\*示例: 为租户获取或设置功能值\*\***

````csharp

using System;

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.FeatureManagement;

namespace Demo

{

public class MyService : ITransientDependency

{

private readonly IFeatureManager \_featureManager;

public MyService(IFeatureManager featureManager)

{

\_featureManager = featureManager;

}

public async Task SetFeatureDemoAsync(Guid tenantId, string value)

{

await \_featureManager

.SetForTenantAsync(tenantId, "Feature1", value);

var currentValue = await \_featureManager

.GetOrNullForTenantAsync("Feature1", tenantId);

}

}

}

````

## 功能管理提供程序

功能管理模块是可扩展的, 就像 [功能系统](../Features.md) 一样. 你可以通过定义功能管理提供程序来扩展它. 有3个预编译的功能管理提供程序已按下面的顺序注册:

\* `DefaultValueFeatureManagementProvider`: 从功能定义的默认值中获取功能值. 它无法设置默认值, 因为默认值是在功能定义中硬编码的.

\* `EditionFeatureManagementProvider`: 获取或设置版本的功能值. 版本是分配给租户的一个功能分组. 租户管理模块没有实现版本系统. 你可以自己实现它, 或者购买已实现它并且提供更多SaaS功能的 ABP 商业版 [SaaS 模块](https://commercial.abp.io/modules/Volo.Saas), 像订阅和支付.

\* `TenantFeatureManagementProvider`: 获取或设置租户的功能.

`IFeatureManager` 在获取和设置的方法中使用了这些提供程序. 通常, 在 `IFeatureManager` 服务中, 每个功能管理提供程序都定义了扩展方法 (像租户功能管理提供程序就定义了 `SetForTenantAsync`).

如果想要自定义提供程序, 你需要实现 `IFeatureManagementProvider` 接口或从 `FeatureManagementProvider` 基类继承:

````csharp

public class CustomFeatureProvider : FeatureManagementProvider

{

public override string Name => "Custom";

public CustomFeatureProvider(IFeatureManagementStore store)

: base(store)

{

}

}

````

`FeatureManagementProvider` 基类 (使用 `IFeatureManagementStore`) 为你做了默认的实现. 你可以按需重写基础方法. 在这个 `Custom` 的示例中, 每个提供程序必须有一个唯一的名称 (请保持简短, 因为每个功能值记录都会被保存到数据库中).

创建提供程序类后, 你应该用 `FeatureManagementOptions` [选项类](../Options.md) 注册它:

````csharp

Configure<FeatureManagementOptions>(options =>

{

options.Providers.Add<CustomFeatureProvider>();

});

````

提供程序的顺序很重要. 提供程序按相反的顺序被执行. 这意味着在此示例中 `CustomFeatureProvider` 是第一个被执行的. 你可以在 `Providers` 列表的任意位置插入你的提供程序.

## 另请参见

\* [功能](../Features.md)

## Identity

# 身份管理模块

身份模块基于 [Microsoft Identity 库](https://docs.microsoft.com/zh-cn/aspnet/core/security/authentication/identity) 用于管理角色,用户及其权限.

## 如何安装

当你使用 ABP 框架 [创建一个新的解决方案](https://abp.io/get-started) 时, 此模块将被预安装 (作为 NuGet/NPM 包). 你可以继续用其作为包并轻松地获取更新, 也可以将其源代码包含在解决方案中（请参阅 `get-source` [CLI](../CLI.md)）以开发自定义模块.

### 源代码

可以 [在此处](https://github.com/abpframework/abp/tree/dev/modules/identity) 访问源代码。源代码使用 [MIT](https://choosealicense.com/licenses/mit/) 许可, 所以你可以免费使用和自定义它.

## 用户界面

此模块提供了 [Blazor](../UI/Blazor/Overall.md), [Angular](../UI/Angular/Quick-Start.md) 和 [MVC / Razor Pages](../UI/AspNetCore/Overall.md) 的 UI 可选.

### 菜单项

此模块在 *\*管理\** 菜单下添加了一个 *\*身份管理\** 菜单项:

![身份模块菜单](../images/identity-module-menu.png)

这个菜单项和相关页面已获得授权. 这意味着当前用户必须拥有相关权限才能使其可见. `admin` 角色 (和拥有此角色的用户, 如 `admin` 用户) 已经拥有这些权限. 如果你想要使其他角色/用户也启用权限, 请打开 *\*角色\** 或 *\*用户\** 页面的 *\*权限\** 对话框, 并检查如下所示的权限:

![身分模块权限](../images/identity-module-permissions.png)

请参阅 [授权文档](../Authorization.md) 以了解权限系统.

### 页面

本节介绍此模块提供的主要页面.

#### 用户

此页用于查看用户列表. 您可以创建/编辑和删除用户, 将角色分配给用户.

![身分模块用户](../images/identity-module-users.png)

一个用户可以有零个或多个角色. 用户从其角色继承权限. 此外, 你可以给用户直接分配权限 (通过点击 *\*操作\** 按钮, 然后选择 *\*权限\**) .

#### 角色

角色用于按分组给用户分配权限.

![身份模块角色](../images/identity-module-roles.png)

角色除了角色名称之外还有2个属性:

\* `Default`: 如果某个角色被标记为 "default", 那么当新用户 (使用 [账户模块](Account.md)) 注册到应用程序时, 其将被作为默认角色分配给新用户.

\* `Public`: 用户的公共角色可以被应用程序中的其他用户看到. 这个功能在身份模块中没有用到, 但被提供作为你可能希望在你自己的应用程序中使用的功能.

## 其他功能

本节包括此模块提供的没有 UI 页面的其他功能.

### 组织单元

组织单元 (OU) 可被用于 **\*\*按层级对用户和实体进行分组\*\***.

#### 组织单元实体

组织单元由 **\*\*OrganizationUnit\*\*** 实体表示. 它的基本属性是:

- **\*\*TenantId\*\***: 组织单元的租户Id. 对于宿主可以是 null.

- **\*\*ParentId\*\***: 父级组织单元的Id. 如果这是一个根级组织单元，它可以是 null.

- **\*\*Code\*\***: 对于租户唯一的层级字符串编码.

- **\*\*DisplayName\*\***: 组织单元的显示名称.

#### 组织树

由于组织单元可以有父级, 因此租户的所有组织单元是一个 **\*\*树\*\*** 结构. 这个树有一些规则;

- 可以有多个根级 (`ParentId` 是 `null` 的) .

- 一个组织单元的第一级子项数量有限制 (因为下面解释的固定的组织单元编码单位长度) .

#### 组织单元编码

组织单元编码是通过 `OrganizationUnitManager` 服务自动生成和维护的. 它是一个字符串, 像这样:

"**\*\*00001.00042.00005\*\***"

通常用这种编码可以轻易地 (递归) 查询出数据库中组织单元的所有子项. 这个编码有一些规则 (当你使用 `OrganizationUnitManager` 时自动被应用的) :

- 对 [租户](../Multi-Tenancy.md) 是 **\*\*唯一的\*\***.

- 同一组织单元的全部子项都具有 **\*\*以父级组织单元编码开头的\*\*** 编码.

- 如示例中所示, 它是固定长度的且是基于树中组织单元的级别的.

- 尽管组织单元编码是唯一的, 但如果你移动了相关的组织单元, 它也可以被更改.

请注意, 你必须根据Id引用一个组织单元, 而不是编码, 因为编码后续是可以被更改的.

#### 组织单元管理

`OrganizationUnitManager` 类可以被 [注入](../Dependency-Injection.md) 并用来管理组织单元. 常见用例如下:

- 创建, 更新和删除组织单元.

- 在组织单元树中移动一个组织单元.

- 获取关于组织单元树及其项的信息.

### 身份安全日志

安全日志系统可以记录账户的一些重要的操作或者改动 (例如 *\*登录\** 和 *\*更改密码\**) . 如果需要, 你也可以保存安全日志.

你可以注入和使用 `IdentitySecurityLogManager` 或 `ISecurityLogManager` 来写入安全日志. 默认它会创建一个日志对象并填充常用的值, 如 `CreationTime`, `ClientIpAddress`, `BrowserInfo`, `current user/tenant` 等等. 当然你可以覆盖这些值.

```cs

await IdentitySecurityLogManager.SaveAsync(new IdentitySecurityLogContext()

{

Identity = "IdentityServer",

Action = "ChangePassword"

});

```

通过配置 `AbpSecurityLogOptions` 来为日志提供应用程序的名称 (如果你有多个应用程序并且想要在日志中区分应用程序) 或者禁用安全日志功能.

```cs

Configure<AbpSecurityLogOptions>(options =>

{

options.ApplicationName = "AbpSecurityTest";

});

```

## 选项

`IdentityOptions` 是由 Microsoft [Identity 库](https://docs.microsoft.com/zh-cn/aspnet/core/security/authentication/identity) 提供的标准 [选项类](../Options.md) . 所以, 你可以在 [模块](../Module-Development-Basics.md) 类的 `ConfigureServices` 方法中设置这些选项.

**\*\*例如: 设置所需的密码最小长度\*\***

````csharp

Configure<IdentityOptions>(options =>

{

options.Password.RequiredLength = 5;

});

````

ABP 允许你在运行时通过使用 [设置系统](../Settings.md) 更进一步地更改这些选项. 你可以 [注入](../Dependency-Injection.md) `ISettingManager` 并使用 `Set...` 方法中的一种来更改用户, 租户或全局所有用户的选项值.

**\*\*例如: 更改当前租户所需的密码最小长度\*\***

````csharp

public class MyService : ITransientDependency

{

private readonly ISettingManager \_settingManager;

public MyService(ISettingManager settingManager)

{

\_settingManager = settingManager;

}

public async Task ChangeMinPasswordLength(int minLength)

{

await \_settingManager.SetForCurrentTenantAsync(

IdentitySettingNames.Password.RequiredLength,

minLength.ToString()

);

}

}

````

`IdentitySettingNames` 类 (在 `Volo.Abp.Identity.Settings` 命名空间中) 为设置名称定义了一些常量.

## 分布式事件

此模块定义了如下 ETOs (事件传输对象) 以允许你去订阅模块中实体的改动.

\* `UserEto` 在 `IdentityUser` 实体更改完成时发布.

\* `IdentityRoleEto` 在 `IdentityRole` 实体更改完成时发布.

\* `IdentityClaimTypeEto` 在 `IdentityClaimType` 实体更改完成时发布.

\* `OrganizationUnitEto` 在 `OrganizationUnit` 实体更改完成时发布.

**\*\*例如: 当一个新用户被创建时接收通知\*\***

````csharp

public class MyHandler :

IDistributedEventHandler<EntityCreatedEto<UserEto>>,

ITransientDependency

{

public async Task HandleEventAsync(EntityCreatedEto<UserEto> eventData)

{

UserEto user = eventData.Entity;

// TODO: ...

}

}

````

`UserEto` 和 `IdentityRoleEto` 默认自动发布事件. 你应该自己配置其他的 Eto. 请参阅 [分布式事件总线文档](../Distributed-Event-Bus.md) 了解预定义事件的详细信息.

> 订阅分布式事件对于分布式场景 (如微服务架构) 特别有用. 如果你正在构建单体式应用程序, 或者在运行了身份模块的同一进程内监听事件, 那么订阅 [本地事件](../Local-Event-Bus.md) 更高效, 更简单.

## 内部构件

本节包括模块的一些内部详细信息, 您不是那么需要, 但在一些情况下可能需要使用.

### 领域层

#### 聚合

##### 用户

用户通常是一个登录并使用应用程序的人.

\* `IdentityUser` (聚合根): 表示系统中的一个用户.

\* `IdentityUserRole` (集合): 分配给用户的角色.

\* `IdentityUserClaim` (集合): 用户的自定义声明.

\* `IdentityUserLogin` (集合): 用户的外部登录.

\* `IdentityUserToken` (集合): 用户的令牌 (用于 Microsoft Identity 服务).

##### 角色

角色通常是分配给用户的一组权限.

\* `IdentityRole` (聚合根): 表示系统中的角色.

\* `IdentityRoleClaim` (集合): 角色的自定义声明.

##### 声明类型

声明类型是可以被分配给系统中其他实体 (如角色和用户) 的自定义声明的定义.

\* `IdentityClaimType` (聚合根): 表示声明类型的定义. 它包括一些用于定义声明类型和验证规则的属性 (例如, 必须的, 正则表达式, 描述, 值类型) .

##### 身份安全日志

`IdentitySecurityLog` 对象表示系统中与授权相关的操作 (如 *\*登录\**) .

\* `IdentitySecurityLog` (聚合根): 表示系统中的安全日志.

##### 组织单元

组织单元是一个有层级结构的实体.

\* ```OrganizationUnit``` (聚合根): 表示系统中的组织单元.

\* ```Roles``` (集合): 组织单元的角色.

#### 仓储

此模块定义了以下自定义仓储:

\* `IIdentityUserRepository`

\* `IIdentityRoleRepository`

\* `IIdentityClaimTypeRepository`

\* ```IIdentitySecurityLogRepository```

\* ```IOrganizationUnitRepository```

#### 领域服务

##### 用户管理

`IdentityUserManager` 常用于管理用户, 他们的权限, 声明, 密码, 电子邮件等等. 它派生自 Microsoft Identity 的 `UserManager<T>` 类, 其中 `T` 是 `IdentityUser`.

##### 角色管理

`IdentityRoleManager` 常用于管理角色和他们的声明. 它派生自 Microsoft Identity 的 `RoleManager<T>` 类, 其中 `T` 是 `IdentityRole`.

##### 声明类型管理

`IdenityClaimTypeManager` 常用于对聚合根 `IdentityClaimType` 执行某些操作.

##### 组织单元管理

```OrganizationUnitManager``` 常用于对聚合根 `OrganizationUnit` 执行某些操作.

##### 安全日志管理

```IdentitySecurityLogManager``` 常用于保存安全日志.

### 服务层

#### 应用服务

\* `IdentityUserAppService` (实现 `IIdentityUserAppService`): 实现了用户管理 UI 的用例.

\* `IdentityRoleAppService` (实现 `IIdentityRoleAppService`): 实现了角色管理 UI 的用例.

\* `IdentityClaimTypeAppService` (实现 `IIdentityClaimTypeAppService`): 实现了声明类型管理 UI 的用例.

\* `IdentitySettingsAppService` (实现 `IIdentitySettingsAppService`): 用于获取和更新身份模块的设置.

\* `IdentityUserLookupAppService` (实现 `IIdentityUserLookupAppService`): 用于根据 `id` 或 `userName` 获取用户信息. 它旨在由ABP内部使用.

\* `ProfileAppService` (实现 `IProfileAppService`): 用于更改用户的简介和密码.

\* ```IdentitySecurityLogAppService``` (实现 ```IIdentitySecurityLogAppService```): 实现了安全日志 UI 的用例.

\* ```OrganizationUnitAppService``` (实现 ```OrganizationUnitAppService```): 实现了组织单元管理 UI 的用例.

### 数据库提供程序

此模块为数据库提供 [Entity Framework Core](../Entity-Framework-Core.md) 和 [MongoDB](../MongoDB.md) 两种选择.

#### EF Core

NuGet 包 [Volo.Abp.Identity.EntityFrameworkCore](https://www.nuget.org/packages/Volo.Abp.Identity.EntityFrameworkCore) 实现了 EF Core 的集成.

##### 数据库表

\* **\*\*AbpRoles\*\***

\* AbpRoleClaims

\* **\*\*AbpUsers\*\***

\* AbpUserClaims

\* AbpUserLogins

\* AbpUserRoles

\* AbpUserTokens

\* **\*\*AbpClaimTypes\*\***

\* **\*\*AbpOrganizationUnits\*\***

\* AbpOrganizationUnitRoles

\* AbpUserOrganizationUnits

\* **\*\*AbpSecurityLogs\*\***

#### MongoDB

NuGet 包 [Volo.Abp.Identity.MongoDB](https://www.nuget.org/packages/Volo.Abp.Identity.MongoDB) 实现了 MongoDB 的集成.

##### 数据库集合

\* **\*\*AbpRoles\*\***

\* **\*\*AbpUsers\*\***

\* **\*\*AbpClaimTypes\*\***

\* **\*\*AbpOrganizationUnits\*\***

\* **\*\*AbpSecurityLogs\*\***

#### 常用数据库属性

你可以设置 `AbpIdentityDbProperties` 中的以下属性来更改数据库选项:

\* `DbTablePrefix` (`Abp` 作为默认值) 是表/集合名称的前缀.

\* `DbSchema` (`null` 作为默认值) 是数据库架构.

\* `ConnectionStringName` (`AbpIdentity` 作为默认值) 是此模块的 [连接字符串](../Connection-Strings.md) 名称.

它们是静态属性. 你需要在开始运行应用程序前设置它们 (通常在 `Program.cs` 中).

## IdentityServer

# 身份服务器模块

身份服务器模块提供了一个 [IdentityServer](https://github.com/IdentityServer/IdentityServer4) (IDS) 的完全集成, 该框架提供高级身份验证功能, 如单点登录和API访问控制.此模块将客户端,资源以及其他 IDS 相关的对象保存到数据库中.

## 如何安装

当你使用 ABP 框架 [创建一个新的解决方案](https://abp.io/get-started) 时, 此模块将被预安装(作为 NuGet/NPM 包).你可以继续用其作为包并轻松地获取更新, 也可以将其源代码包含在解决方案中(请参阅 `get-source` [CLI](../CLI.md))以开发自定义模块.

### 源代码

可以 [在此处](https://github.com/abpframework/abp/tree/dev/modules/identityserver) 访问源代码.源代码使用 [MIT](https://choosealicense.com/licenses/mit/) 许可, 所以你可以免费使用和自定义它.

## 用户界面

此模块使用了领域逻辑和数据库集成, 但没有提供任何 UI.如果你需要动态添加客户端和资源, 管理 UI 是非常有用的.在这种情况下, 你可以自己构建管理 UI, 或者考虑购买为此模块提供了管理 UI 的 [ABP 商业版](https://commercial.abp.io/).

## 与其他模块的关系

此模块基于 [身份模块](Identity.md) 并且[账户模块](Account.md) 有一个 [集成包](https://www.nuget.org/packages/Volo.Abp.Account.Web.IdentityServer).

## 选项

### AbpIdentityServerBuilderOptions

`AbpIdentityServerBuilderOptions` 在你的身份服务器 [模块](https://docs.abp.io/zh-Hans/abp/latest/Module-Development-Basics) 中的 `PreConfigureServices` 方法中配置.例如:

````csharp

public override void PreConfigureServices(ServiceConfigurationContext context)

{

PreConfigure<AbpIdentityServerBuilderOptions>(builder =>

{

//Set options here...

});

}

````

`AbpIdentityServerBuilderOptions` 属性:

\* `UpdateJwtSecurityTokenHandlerDefaultInboundClaimTypeMap` (默认值:true):更新 `JwtSecurityTokenHandler.DefaultInboundClaimTypeMap` 使其与身份服务器声明兼容.

\* `UpdateAbpClaimTypes` (默认值:true):更新 `AbpClaimTypes` 与身份服务器声明兼容.

\* `IntegrateToAspNetIdentity` (默认值:true):集成到 ASP.NET Identity.

\* `AddDeveloperSigningCredential` (默认值:true):设置为 false 禁止调用 IIdentityServerBuilder 中的 `AddDeveloperSigningCredential()`.

`IIdentityServerBuilder` 可以在你的身份服务器 [模块](https://docs.abp.io/zh-Hans/abp/latest/Module-Development-Basics) 中的 `PreConfigureServices` 方法中配置.例如:

````csharp

public override void PreConfigureServices(ServiceConfigurationContext context)

{

PreConfigure<IIdentityServerBuilder>(builder =>

{

builder.AddSigningCredential(...);

});

}

````

## 内部结构

### 领域层

#### 聚合

##### API 资源

需要 API 资源来允许客户端请求访问令牌.

\* `ApiResource` (聚合根):表示系统中的 API 资源.

\* `ApiSecret` (集合):API 资源的密钥.

\* `ApiScope` (集合):API 资源的作用域.

\* `ApiResourceClaim` (集合):API 资源的声明.

##### 客户端

客户端表示可以从你的身份服务器请求令牌的应用程序.

\* `Client` (聚合根):表示一个身份服务器的客户端应用程序.

\* `ClientScope` (集合):客户端的作用域.

\* `ClientSecret` (集合):客户端的密钥.

\* `ClientGrantType` (集合):客户端的授权类型.

\* `ClientCorsOrigin` (集合):客户端的 CORS 源.

\* `ClientRedirectUri` (集合):客户端的重定向 URIs.

\* `ClientPostLogoutRedirectUri` (集合):客户端的登出重定向 URIs.

\* `ClientIdPRestriction` (集合):客户端的提供程序约束.

\* `ClientClaim` (集合):客户端的声明.

\* `ClientProperty` (集合):客户端的自定义属性.

##### 持续化授权

持续化授权存储了授权码,刷新令牌和用户准许.

\* `PersistedGrant` (聚合根):表示为身份服务器持续化授权.

##### 身份资源

身份资源是用户的用户 ID ,名称或邮件地址等数据.

\* `IdentityResource` (聚合根):表示与身份服务器的身份资源.

\* `IdentityClaim` (集合):身份资源的声明.

#### 仓储

为此模块定义了以下自定义仓储:

\* `IApiResourceRepository`

\* `IClientRepository`

\* `IPersistentGrantRepository`

\* `IIdentityResourceRepository`

#### 领域服务

此模块不包含任何领域服务, 但重写了下面的服务;

\* `AbpProfileService` (当 `AbpIdentityServerBuilderOptions.IntegrateToAspNetIdentity` 为 true 时使用)

\* `AbpClaimsService`

\* `AbpCorsPolicyService`

### 设置

此模块未定义任何设置.

### 应用层

#### 应用服务

\* `ApiResourceAppService` (实现 `IApiResourceAppService`):实现了 API 资源管理 UI 的用例.

\* `IdentityServerClaimTypeAppService` (实现 `IIdentityServerClaimTypeAppService`):用于获取声明列表.

\* `ApiResourceAppService` (实现 `IApiResourceAppService`):实现了 API 管理资源 UI 的用例.

\* `IdentityResourceAppService` (实现 `IIdentityResourceAppService`):实现了身份资源管理 UI 的用例.

### 数据库提供程序

#### 公共

##### 表/集合 前缀 & 架构

所有表/集合都使用 `IdentityServer` 作为默认前缀.如果你需要改变表的前缀或设置一个架构名称(如果你的数据库提供程序支持), 请设置 `AbpIdentityServerDbProperties` 类的静态属性.

##### 连接字符串

此模块使用 `AbpIdentityServer` 作为连接字符串的名称.如果你没有用这个名称定义连接字符串, 它将回退到 `Default` 连接字符串.

有关详细信息, 请参阅 [连接字符串](https://docs.abp.io/zh-Hans/abp/latest/Connection-Strings) 文档.

#### EF Core

##### 表

\* **\*\*IdentityServerApiResources\*\***

\* IdentityServerApiSecrets

\* IdentityServerApiScopes

\* IdentityServerApiScopeClaims

\* IdentityServerApiClaims

\* **\*\*IdentityServerClients\*\***

\* IdentityServerClientScopes

\* IdentityServerClientSecrets

\* IdentityServerClientGrantTypes

\* IdentityServerClientCorsOrigins

\* IdentityServerClientRedirectUris

\* IdentityServerClientPostLogoutRedirectUris

\* IdentityServerClientIdPRestrictions

\* IdentityServerClientClaims

\* IdentityServerClientProperties

\* **\*\*IdentityServerPersistedGrants\*\***

\* **\*\*IdentityServerIdentityResources\*\***

\* IdentityServerIdentityClaims

#### MongoDB

##### 集合

\* **\*\*IdentityServerApiResources\*\***

\* **\*\*IdentityServerClients\*\***

\* **\*\*IdentityServerPersistedGrants\*\***

\* **\*\*IdentityServerIdentityResources\*\***

## 权限管理

# 权限管理模块

此模块实现了 `IPermissionStore` 接口, 并在数据库中管理权限值.

> 此文档仅介绍了将权限值持续保存到数据库的权限管理模块. 请参阅 [授权文档](../Authorization.md) 来了解授权和权限系统.

## 如何安装

当你使用 ABP 框架 [创建一个新的解决方案](https://abp.io/get-started) 时, 此模块将被预安装(作为 NuGet/NPM 包). 你可以继续用其作为包并轻松地获取更新, 也可以将其源代码包含在解决方案中 (请参阅 `get-source` [CLI](../CLI.md) 命令) 以开发自定义模块.

### 源代码

可以 [在此处](https://github.com/abpframework/abp/tree/dev/modules/permission-management) 访问源代码。源代码使用 [MIT](https://choosealicense.com/licenses/mit/) 许可, 所以你可以免费使用和自定义它.

## 用户界面

### 权限管理对话框

权限管理模块提供了一个可重用的对话框来管理与对象相关的权限. 例如: [身份模块](Identity.md) 用它去管理用户和角色的权限. 下图展示了身份模块的角色管理页面:

![permissions-module-open-dialog](../images/permissions-module-open-dialog.png)

当你点击角色的 *\*操作\** -> *\*权限\** 时, 将打开权限管理对话框. 下面是此对话框的一个示例截图:

![permissions-module-dialog](../images/permissions-module-dialog.png)

在对话框中, 你可以为所选角色授予权限. 左侧的选项卡表示主要权限组, 右侧包括了所选权限组中定义的权限.

## IPermissionManager

`IPermissionManager` 是此模块提供的主要服务. 它常用来读取和更改权限值. `IPermissionManager` 通常被 *\*权限管理对话框\** 使用. 但如果需要设置权限值, 你也可以注入它.

> 如果你只是想要读取或检查当前用户的权限值, 请如 [授权文档](../Authorization.md) 中所述那样使用 `IAuthorizationService` 或 `[Authorize]` 特性.

**\*\*示例: 使用** `IPermissionManager` **服务为角色和用户授权\*\***

````csharp

public class MyService : ITransientDependency

{

private readonly IPermissionManager \_permissionManager;

public MyService(IPermissionManager permissionManager)

{

\_permissionManager = permissionManager;

}

public async Task GrantRolePermissionDemoAsync(

string roleName, string permission)

{

await \_permissionManager

.SetForRoleAsync(roleName, permission, true);

}

public async Task GrantUserPermissionDemoAsync(

Guid userId, string roleName, string permission)

{

await \_permissionManager

.SetForUserAsync(userId, permission, true);

}

}

````

## 权限管理提供程序

权限管理模块是可扩展的, 就像 [权限系统](../Authorization.md) 一样. 你可以通过定义权限管理提供程序来扩展它.

[身份模块](Identity.md) 定义了如下权限管理提供程序.

\* `UserPermissionManagementProvider`: 管理用户基础权限.

\* `RolePermissionManagementProvider`: 管理角色基础权限.

`IPermissionManager` 使用这些提供程序获取或设置权限. 你可以通过实现 `IPermissionManagementProvider` 或从 `PermissionManagementProvider` 基类继承定义你自己的提供程序.

**\*\*示例:\*\***

````csharp

public class CustomPermissionManagementProvider : PermissionManagementProvider

{

public override string Name => "Custom";

public CustomPermissionManagementProvider(

IPermissionGrantRepository permissionGrantRepository,

IGuidGenerator guidGenerator,

ICurrentTenant currentTenant)

: base(

permissionGrantRepository,

guidGenerator,

currentTenant)

{

}

}

````

`PermissionManagementProvider` 基类 (使用 `IPermissionGrantRepository`) 为你做了默认的实现. 你可以按需重写基础方法. 在这个自定义的示例中, 每个提供程序必须有一个唯一的名称 (请保持简短, 因为每个权限值记录都会被保存到数据库中).

创建提供程序类后, 你应该用 `PermissionManagementOptions` [选项类](../Options.md) 注册它:

````csharp

Configure<PermissionManagementOptions>(options =>

{

options.ManagementProviders.Add<CustomPermissionManagementProvider>();

});

````

提供程序的顺序很重要. 提供程序按相反的顺序被执行. 这意味着在此示例中 `CustomPermissionManagementProvider` 是第一个被执行的. 你可以在 `Providers` 列表的任意位置插入你的提供程序.

## 另请参见

\* [授权](../Authorization.md)

## 设置管理

# 设置管理模块

设置管理模块实现了 `ISettingStore` (参阅 [设置系统](../Settings.md)) 将设置值存储在数据库中, 并提供 `ISettingManager` 管理 (更改) 数据库中设置值的功能.

> [启动模板](../Startup-Templates/Index.md)默认安装并配置了设置管理模块. 大部分情况下你不需要手动的添加该到模块到应用程序中.

## ISettingManager

`ISettingManager` 用于获取和设定设置值. 示例:

````csharp

using System;

using System.Threading.Tasks;

using Volo.Abp.DependencyInjection;

using Volo.Abp.SettingManagement;

namespace Demo

{

public class MyService : ITransientDependency

{

private readonly ISettingManager \_settingManager;

//Inject ISettingManager service

public MyService(ISettingManager settingManager)

{

\_settingManager = settingManager;

}

public async Task FooAsync()

{

Guid user1Id = ...;

Guid tenant1Id = ...;

//Get/set a setting value for the current user or the specified user

string layoutType1 =

await \_settingManager.GetOrNullForCurrentUserAsync("App.UI.LayoutType");

string layoutType2 =

await \_settingManager.GetOrNullForUserAsync("App.UI.LayoutType", user1Id);

await \_settingManager.SetForCurrentUserAsync("App.UI.LayoutType", "LeftMenu");

await \_settingManager.SetForUserAsync(user1Id, "App.UI.LayoutType", "LeftMenu");

//Get/set a setting value for the current tenant or the specified tenant

string layoutType3 =

await \_settingManager.GetOrNullForCurrentTenantAsync("App.UI.LayoutType");

string layoutType4 =

await \_settingManager.GetOrNullForTenantAsync("App.UI.LayoutType", tenant1Id);

await \_settingManager.SetForCurrentTenantAsync("App.UI.LayoutType", "LeftMenu");

await \_settingManager.SetForTenantAsync(tenant1Id, "App.UI.LayoutType", "LeftMenu");

//Get/set a global and default setting value

string layoutType5 =

await \_settingManager.GetOrNullGlobalAsync("App.UI.LayoutType");

string layoutType6 =

await \_settingManager.GetOrNullDefaultAsync("App.UI.LayoutType");

await \_settingManager.SetGlobalAsync("App.UI.LayoutType", "TopMenu");

}

}

}

````

你可以从不同的设置值提供程序中(默认,全局,用户,租户...等)中获取或设定设置值.

> 如果只需要读取设置值,建议使用 `ISettingProvider` 而不是`ISettingManager`,因为它实现了缓存并支持所有部署场景. 如果要创建设置管理UI,可以使用ISettingManager.

### Setting Cache

设置值缓存在 [分布式缓存](../Caching.md) 系统中. 建议始终使用 `ISettingManager` 更改设置值.

## Setting Management Providers

设置管理模块是可扩展的,像[设置系统](../Settings.md)一样. 你可以通过自定义设置管理提供程序进行扩展. 有5个预构建的设置管理程序程序按以下顺序注册:

\* `DefaultValueSettingManagementProvider`: 从设置定义的默认值中获取值,由于默认值是硬编码在设置定义上的,所以无法更改默认值.

\* `ConfigurationSettingManagementProvider`:从 [IConfiguration 服务](../Configuration.md)中获取值. 由于无法在运行时更改配置值,所以无法更改配置值.

\* `GlobalSettingManagementProvider`: 获取或设定设置的全局 (系统范围)值.

\* `TenantSettingManagementProvider`: 获取或设定租户的设置值.

\* `UserSettingManagementProvider`: 获取或设定用户的设置值.

`ISettingManager` 在 `get/set` 方法中使用设置管理提供程序. 通常每个设置程序提供程序都在 `ISettingManagement` 服务上定义了模块方法 (比如用户设置管理程序提供定义了 `SetForUserAsync` 方法).

## Setting Management UI.

设置管理模块默认提供了邮件设置页面.

![EmailSettingUi](../images/setting-management-email-ui.png)

> 你可以点击发送测试邮件按钮发送一封测试邮件来检查你的邮件设置.

设置UI是可扩展的; 你可以为你的应用程序设置添加设置标签到设置页面.

### MVC UI

#### 创建视图组件

在 `Components` 目录下创建 `MySettingGroup` 文件夹, 添加一个名为 `MySettingGroupViewComponent` 的视图组件:

![MySettingGroupViewComponent](../images/my-setting-group-view-component.png)

打开 `MySettingGroupViewComponent.cs` 替换为以下内容:

```csharp

public class MySettingGroupViewComponent : AbpViewComponent

{

public virtual IViewComponentResult Invoke()

{

return View("~/Components/MySettingGroup/Default.cshtml");

}

}

```

> 你还可以使用 `InvokeAsync` 方法,在这个示例中我们使用 `Invoke` 方法.

#### Default.cshtml

在 `MySettingGroup` 目录下创建 `Default.cshtml` 文件.

打开 `Default.cshtml` 替换为以下内容:

```html

<div>

<p>My setting group page</p>

</div>

```

#### BookStoreSettingPageContributor

在 `Settings` 目录下创建 `BookStoreSettingPageContributor.cs` 文件.

![BookStoreSettingPageContributor](../images/my-setting-group-page-contributor.png)

文件内容如下:

```csharp

public class BookStoreSettingPageContributor : ISettingPageContributor

{

public Task ConfigureAsync(SettingPageCreationContext context)

{

context.Groups.Add(

new SettingPageGroup(

"Volo.Abp.MySettingGroup",

"MySettingGroup",

typeof(MySettingGroupViewComponent)

)

);

return Task.CompletedTask;

}

public Task<bool> CheckPermissionsAsync(SettingPageCreationContext context)

{

// You can check the permissions here

return Task.FromResult(true);

}

}

```

打开 `BookStoreWebModule.cs` 文件添加以下代码:

```csharp

Configure<SettingManagementPageOptions>(options =>

{

options.Contributors.Add(new BookStoreSettingPageContributor());

});

```

#### 运行应用程序

导航到 `/SettingManagement` 路由查看更改:

![Custom Settings Tab](../images/my-setting-group-ui.png)

### Blazor UI

#### 创建 Razor 组件

在 `Pages` 目录下创建 `MySettingGroup` 文件夹, 添加一个名为 `MySettingGroupComponent` 的Razor组件:

![MySettingGroupComponent](../images/my-setting-group-component.png)

打开 `MySettingGroupComponent.razor` 替换为以下内容:

```csharp

<Row>

<p>my setting group</p>

</Row>

```

#### BookStoreSettingComponentContributor

在 `Settings` 目录下创建 `BookStoreSettingComponentContributor.cs` 文件.

![BookStoreSettingComponentContributor](../images/my-setting-group-component-contributor.png)

文件内容如下:

```csharp

public class BookStoreSettingComponentContributor : ISettingComponentContributor

{

public Task ConfigureAsync(SettingComponentCreationContext context)

{

context.Groups.Add(

new SettingComponentGroup(

"Volo.Abp.MySettingGroup",

"MySettingGroup",

typeof(MySettingGroupComponent)

)

);

return Task.CompletedTask;

}

public Task<bool> CheckPermissionsAsync(SettingComponentCreationContext context)

{

// You can check the permissions here

return Task.FromResult(true);

}

}

```

打开 `BookStoreBlazorModule.cs` 文件添加以下代码:

```csharp

Configure<SettingManagementComponentOptions>(options =>

{

options.Contributors.Add(new BookStoreSettingComponentContributor());

});

```

#### 运行应用程序

导航到 `/setting-management` 路由查看更改:

![Custom Settings Tab](../images/my-setting-group-blazor.png)

### Angular UI

不同的模块提供它们的设置选项卡. 你可以通过3个步骤在项目中自定义设置页面.

#### 创建组件

使用以下命令创建一个组件

```bash

yarn ng generate component my-settings

```

打开 `app.component.ts` 做以下修改:

```js

import { Component } from '@angular/core';

import { SettingTabsService } from '@abp/ng.setting-management/config'; // imported SettingTabsService

import { MySettingsComponent } from './my-settings/my-settings.component'; // imported MySettingsComponent

@Component(/\* component metadata \*/)

export class AppComponent {

constructor(private settingTabs: SettingTabsService) // injected MySettingsComponent

{

// added below

settingTabs.add([

{

name: 'MySettings',

order: 1,

requiredPolicy: 'policy key here',

component: MySettingsComponent,

},

]);

}

}

```

#### 运行应用程序

导航到 `/setting-management` 路由你会看到以下变化:

![Custom Settings Tab](../images/custom-settings.png)

## 租户管理

# 租户管理模块

[多租户](../Multi-Tenancy.md) 是 ABP 框架的核心功能之一. 它提供了构建 SaaS (软件及服务) 解决方案的基础架构. ABP 的多租户系统通过提供 `ITenantStore` 接口来抽象租户存储的位置. 您需要做的就是实现该接口.

**\*\*租户管理模块是** `ITenantStore` **接口的实现. 它将租户存储在数据库中. 它还提供了 UI 来管理租户及其 [**功能**](../Features.md).\*\***

> 请参阅 [多租户](../Multi-Tenancy.md) 文档，以了解 ABP 框架的多租户系统. 本文重点介绍租户管理模块.

### 关于商业版 SaaS 模块

[SaaS 模块](https://commercial.abp.io/modules/Volo.Saas) 是此模块的替代实现, 具有更多的功能和可能性. 它作为 [ABP 商业版](https://commercial.abp.io/) 订阅的一部分分发.

## 如何安装

当你使用 ABP 框架 [创建一个新的解决方案](https://abp.io/get-started) 时, 此模块将被预安装(作为 NuGet/NPM 包). 你可以继续用其作为包并轻松地获取更新, 也可以将其源代码包含在解决方案中 (请参阅 `get-source` [CLI](../CLI.md)) 以开发自定义模块.

### 源代码

可以 [在此处](https://github.com/abpframework/abp/tree/dev/modules/identity) 访问源代码。源代码使用 [MIT](https://choosealicense.com/licenses/mit/) 许可, 所以你可以免费使用和自定义它.

## 用户界面

此模块将 "管理 -> 租户管理 -> 租户" 菜单项添加到应用程序的主菜单, 这将打开如下页面:

![租户管理页面](../images/module-tenant-management-page.png)

在这个页面中, 你可以查看所有租户. 你可以按照如下所示创建一个新的租户:

![租户管理新增租户](../images/module-tenant-management-new-tenant.png)

在这个模态框中;

\* **\*\*租户名称\*\***: 租户唯一的名称. 如果为租户使用二级域名, 这将是二级域名名称.

\* **\*\*管理员电子邮件地址\*\***: 租户管理员的电子邮件地址.

\* **\*\*管理员密码\*\***: 租户管理员的密码.

当您点击租户附近的 *\*操作\** 按钮时, 您将看见可以执行的操作:

![租户管理操作](../images/module-tenant-management-actions.png)

### 管理租户功能

功能操作会打开一个模态框, 来对相关租户进行启用/禁用/设置 [功能](../Features.md). 这有一个模态框的例子:

![功能模态框](../images/features-modal.png)

### 管理主机功能

如果您使用应用程序的功能也在主机端, *\*管理主机功能\** 按钮则用于设置主机端的功能.

## 内部结构

如果要 [自定义](../Customizing-Application-Modules-Guide.md) 此模块而不更改 [其源代码](https://github.com/abpframework/abp/tree/dev/modules/tenant-management), 则可以将此节作为参考.

### 领域层

#### 聚合

\* `Tenant`

#### 仓储

\* `ITenantRepository`

#### 领域服务

\* `TenantManager`

### 应用层

#### 应用服务

\* `TenantAppService`

#### 权限

- `AbpTenantManagement.Tenants`: 租户管理.

- `AbpTenantManagement.Tenants.Create`: 创建一个新的租户.

- `AbpTenantManagement.Tenants.Update`: 编辑一个已存在的租户.

- `AbpTenantManagement.Tenants.Delete`: 删除一个已存在的租户.

- `AbpTenantManagement.Tenants.ManageFeatures`: 管理租户的功能.

### EF Core 集成

\* `TenantManagementDbContext` (实现了 `ITenantManagementDbContext`)

**\*\*数据库表:\*\***

\* `AbpTenants`

\* `AbpTenantConnectionStrings`

### MongoDB 集成

\* `TenantManagementMongoDbContext` (实现了 `ITenantManagementMongoDbContext`)

**\*\*数据库集合:\*\***

\* `AbpTenants` (也包括连接字符串)

## 注意事项

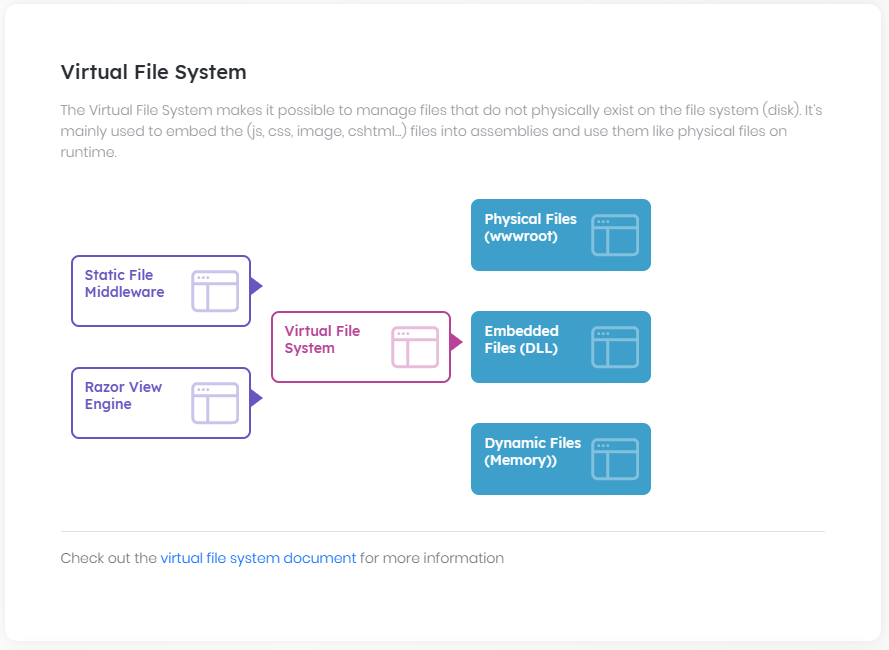
ABP 框架允许租户有专用的数据库, 以允许 *\*每个租户用一个数据库\**. 此模块具有使该实现成为可能 (查阅其源代码) 的基础架构, 但是它没有实现应用程序层和 UI 功能以使其开箱即用. 您可以自己实现这些功能, 也可以考虑使用已完全实现这些功能并提供了更多业务功能的 [ABP 商业版 Saas 模块](https://docs.abp.io/en/commercial/latest/modules/saas).

## 另请参见

\* [多租户](../Multi-Tenancy.md)

\* [ABP 商业版 SaaS 模块](https://docs.abp.io/en/commercial/latest/modules/saas)

## 虚拟文件浏览器



# 虚拟文件浏览器模块

## 什么是虚拟文件浏览器模块

虚拟文件浏览器模块提供了一个简单的UI来查看[虚拟文件系统](../Virtual-File-System.md)所有的文件.

> [启动模板](../Startup-Templates/Index.md)默认并没有安装这个模块,所以你需要手动安装到应用程序.

### 安装

#### 1- 使用ABP CLI

建议使用ABP CLI安装模块,在解决方案文件 (`.sln`) 目录打开 `CMD` 窗口,运行以下命令:

`abp add-module Volo.VirtualFileExplorer`

#### 2- 手动安装

或者你也可以手动安装nuget包到 `Acme.MyProject.Web` 项目:

\* 安装[Volo.Abp.VirtualFileExplorer.Web](https://www.nuget.org/packages/Volo.Abp.VirtualFileExplorer.Web/) nuget包到 `Acme.MyProject.Web` 项目.

`Install-Package Volo.Abp.VirtualFileExplorer.Web`

##### 2.1- 添加模块依赖

\* 打开 `MyProjectWebModule.cs` 并且添加 `typeof(AbpVirtualFileExplorerWebModule)` 如下所示;

```csharp

[DependsOn(

typeof(AbpVirtualFileExplorerWebModule),

typeof(MyProjectApplicationModule),

typeof(MyProjectEntityFrameworkCoreModule),

typeof(AbpAutofacModule),

typeof(AbpIdentityWebModule),

typeof(AbpAccountWebModule),

typeof(AbpAspNetCoreMvcUiBasicThemeModule)

)]

public class MyProjectWebModule : AbpModule

{

//...

}

```

##### 2.2- 添加NPM包

\* 打开 `package.json` 添加 `@abp/virtual-file-explorer": "^2.9.0` 如下所示:

```json

{

"version": "1.0.0",

"name": "my-app",

"private": true,

"dependencies": {

"@abp/aspnetcore.mvc.ui.theme.basic": "^2.9.0",

"@abp/virtual-file-explorer": "^2.9.0"

}

}

```

然后在 `Acme.MyProject.Web` 项目目录打开命令行终端运行以下命令:

````bash

abp install-libs

````

这就是全部,运行应用程序导航到 `/VirtualFileExplorer`. 你会看到虚拟文件浏览器页面:

![Virtual-File-Explorer](../images/virtual-file-explorer.png)

### 选项

你可以通过 `AbpVirtualFileExplorerOptions` 选项禁用虚拟文件浏览器模块:

```csharp

public override void PreConfigureServices(ServiceConfigurationContext context)

{

PreConfigure<AbpVirtualFileExplorerOptions>(options =>

{

options.IsEnabled = false;

});

}

```

# 发布信息

## 升级

# 升级 ABP 框架

本文档解释当ABP框架发布一个新版本时,怎样升级你已存在的解决方案.

## ABP UPDATE 命令

ABP 框架 & 模块生态系统由数百个 NuGet 和 NPM 包组成. 手动更新所有这些软件包以升级您的应用程序会很麻烦。

[ABP CLI](CLI.md) 提供了一个方便的命令，可以使用单个命令更新解决方案中所有与 ABP 相关的 NuGet 和 NPM 包：

````bash

abp update

````

在终端里,解决方案的根文件夹中运行此命令。

> 如果您的解决方案有 Angular UI，您的解决方案中可能有 `aspnet-core` 和 `angular` 文件夹。在这两个文件夹的父文件夹中运行此命令。

### 数据库迁移

> 警告：迁移数据库时要小心，因为在某些情况下可能会丢失数据。在执行之前仔细检查生成的迁移代码。建议备份您当前的数据库。

当您升级到新版本时，如果您的数据库提供者是 **\*\*Entity Framework Core\*\***，最好检查是否有数据库架构更改并升级您的数据库架构；

\* 在包管理器控制台 (PMC) 中使用 `Add-Migration "Upgraded\_To\_Abp\_4\_1"` 或类似命令创建新迁移（在 Visual Studio 中将 `EntityFrameworkCore` 设置为 PMC 中的默认项目，并将 `.DbMigrator` 在解决方案资源管理器中设置为启动项目）。

\* 运行 `.DbMigrator` 应用程序以升级数据库并播种初始数据。

如果 `Add-Migration` 生成一个空迁移，您可以在执行 `.DbMigrator` 之前使用 `Remove-Migration` 将其删除。

## 博客文章和指南

每当您升级您的解决方案时，强烈建议您查看 [ABP BLOG](https://blog.abp.io/) 以了解新版本的新功能和更改。我们会定期发布帖子并编写此类更改。

### 迁移指南

如果新版本为现有应用程序带来重大更改，我们会准备迁移指南。所有指南请参见 [迁移指南](Migration-Guides/Index.md) 页面。

### 升级 Startup 模板

有时我们会引入需要**\*\*在启动模板中进行更改\*\***的新功能/更改。我们已经在新应用程序的启动模板中实现了更改。但是，在某些情况下，您需要手动对现有解决方案进行一些小的更改。

逐行记录必要的更改是不切实际的。在这种情况下，我们建议您创建一个示例解决方案，一个是现有版本，一个是新版本，并使用差异工具进行比较。您可以 [查看本指南](Migration-Guides/Upgrading-Startup-Template.md) 了解如何使用 WinMerge 应用程序进行操作。

## 语义版本控制和重大更改

我们正在努力保持语义版本控制规则，因此您不会对 3.1、3.2、3.3 等次要（功能）版本进行重大更改......

但是，在某些情况下，我们也可能会在功能版本中引入重大更改；

\* ABP 有许多集成包，有时集成库/框架会发布主要版本并进行重大更改。在这种情况下，我们会仔细检查这些更改并决定是否升级集成包。如果变更的影响比较小，我们会更新集成包，并在发布博文中说明变更。在这种情况下，如果您使用过此集成包，则应按照博文中的说明进行操作。如果更改可能会破坏许多应用程序并且不容易修复，我们决定等到下一个主要的 ABP 框架版本发布时升级。

\* 有时我们必须进行重大更改以修复主要错误或使用问题。在这种情况下，我们认为开发人员已经无法正确使用该功能，因此通过重大更改修复它没有问题。在这种情况下，该功能通常是很少使用的功能。同样，我们尝试将影响保持在最低限度。

## 预览版和每日构建

预览版和夜间构建可以帮助您在新的稳定版本之前尝试新功能并调整您的解决方案。

\* [预览版](Previews.md) 通常在次要（功能）版本之前约 2 周发布（我们的次要版本开发周期约为 4 周）。

\* [每日构建](Nightly-Builds.md) 每天晚上（周末除外）从开发分支发布。这意味着您可以尝试前一天的开发。

请参阅他们的文档以了解有关此类版本的详细信息。

## 官方包

<https://abp.io/packages>

## 预览版本

# 预览版

预览版在发布ABP 框架的主要版本或功能版本之前约 4 周发布。它们是为开发人员发布的，以尝试提供反馈以获得更稳定的版本。

预览版的版本控制是类似这样的：

\* 3.1.0-rc.1

\* 4.0.0-rc.1

在稳定版本（如 3.1.0）之前，可能会发布多个预览版本（如 3.1.0-rc.2 和 3.1.0-rc.3）。

## 使用预览版

### 新解决方案

要创建用于测试预览版的项目，您可以选择[下载页面](https://abp.io/get-started)上的“预览”选项或带有 **\*\*--preview\*\*** 参数的[ABP CLI](CLI.md) **\*\*abp new\*\*** 命令：

````bash

abp new Acme.BookStore --preview

````

此命令将使用最新的预览版 NuGet 包、NPM 包和解决方案模板创建一个新项目。每当发布稳定版本时，您都可以使用解决方案abp switch-to-stable根文件夹中的命令切换到解决方案的稳定版本。

### 现有解决方案

如果您已有解决方案并希望使用/测试最新的预览版，请在解决方案的根文件夹中使用以下 [ABP CLI](CLI.md) 命令.

````bash

abp switch-to-preview

````

您可以稍后使用 `abp switch-to-stable ` 命令返回最新的稳定版.

````bash

abp switch-to-stable

````

## 提供反馈

如果您发现错误或想要提供任何类型的反馈，您可以在[GitHub 存储库](https://github.com/abpframework/abp/issues/new)上打开一个问题。

## 每日构建

# 每日构建

所有框架和模块包每晚都部署到MyGet. 因此你可以使用或测试最新的代码,而无需等待下一个版本.

## 安装和卸载每晚预览包

可以通过在应用程序的根文件夹中运行以下命令安装最新版本的夜间预览软件包:

```bash

abp switch-to-nightly

```

如果你正在使用ABP框架每晚预览包,你可以使用此命令切换回稳定版本:

```bash

abp switch-to-stable

```

参阅 [ABP CLI 文档](./CLI.md) 了解更多信息.

## 路线图

# ABP Framework 路线图

您可以随时在[GitHub仓库](https://github.com/abpframework/abp/milestones)上查看里程碑计划和优先的积压问题,获取详细的路线图.

虽然我们将**\*\*继续添加其他令人兴奋的功能\*\***,但我们将在`中期`中处理以下主要项目:

\* 为所有预构建模块实现 **\*\*Blazor UI\*\***.

\* **\*\*.NET 5.0\*\***! Microsoft宣布.NET 5.0将在2020年11月发布,我们将为此更改做准备并在Microsoft发布它之后立即迁移到.NET 5.0. 我们希望顺利过渡.虽然已经可以为您的应用程序创建或使用gRPC端点，但我们计划为标准应用程序模块创建端点

除了中期目录,还有一些[积压](https://github.com/abpframework/abp/milestone/2)的功能, 这里积压中重要功能的列表:

\* [#2882](https://github.com/abpframework/abp/issues/2882) / 提供 **\*\*gRPC集成\*\*** 基础设施 (虽然[已经可以](https://github.com/abpframework/abp-samples/tree/master/GrpcDemo)为你的应用程序创建和使用gRPC端点,但我们计划为所有[标准应用程序模块](https://docs.abp.io/en/abp/latest/Modules/Index)创建端点)

\* [#236](https://github.com/abpframework/abp/issues/236) 基于权限系统的资源

\* [#1754](https://github.com/abpframework/abp/issues/1754) / 多语言实体

\* [#347](https://github.com/abpframework/abp/issues/347) / 支持MongoDB ACID事务

\* [#633](https://github.com/abpframework/abp/issues/633) / 实时通知系统

\* [#57](https://github.com/abpframework/abp/issues/57) / 内置CQRS基础设施

\* [#4222](https://github.com/abpframework/abp/issues/4222) / 分布式事件总线Kafka集成

\* [#336](https://github.com/abpframework/abp/issues/336) / 健康检查抽象

\* [#2532](https://github.com/abpframework/abp/issues/2532), [#2564](https://github.com/abpframework/abp/issues/2465) / EF Core 与 MongoDB API 集成CosmosDB

\* [#1168](https://github.com/abpframework/abp/issues/1168) / Vue 启动模板

\* [#1638](https://github.com/abpframework/abp/issues/1638) React 启动模板

\* [#4223](https://github.com/abpframework/abp/issues/4223) / WebHook系统

\* [#162](https://github.com/abpframework/abp/issues/162) / 为多租户集成Azure ElasticDB

\* [#2296](https://github.com/abpframework/abp/issues/2296) / 功能切换基础架构

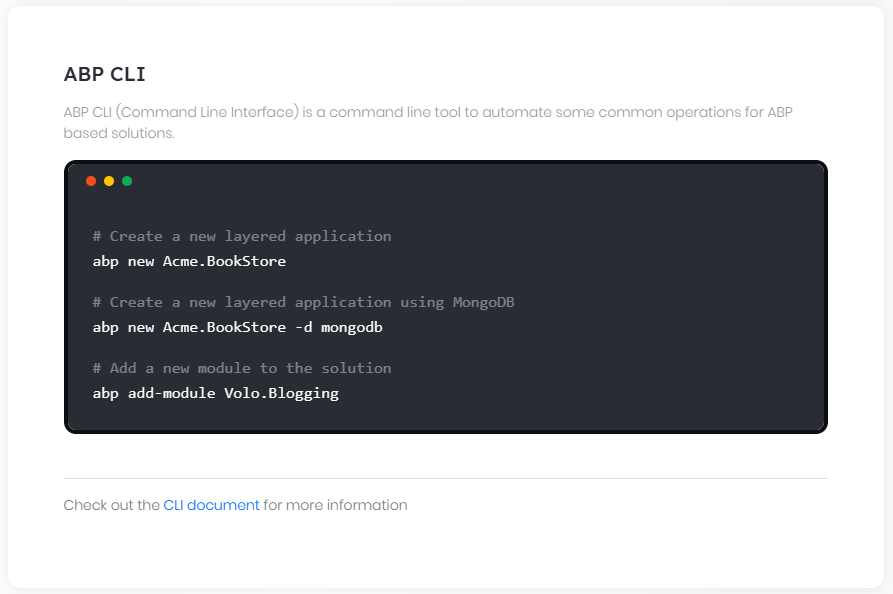
积压的项目有可能发生变化. 我们将根据社区反馈和项目目标添加新项目与更改优先级.

在Github相关issue为你感兴趣的功能投票(并写下你的想法)

. 你可以在[GitHub仓库](https://github.com/abpframework/abpork/abp/milestones)为你的功能请求创建issue,但在创建前请先搜索是否已存在类似的issues.

# 参考

## CLI



# ABP CLI

ABP CLI (命令行接口) 是一个命令行工具,用来执行基于ABP解决方案的一些常见操作.

## Installation

ABP CLI 是一个 [dotnet global tool](https://docs.microsoft.com/en-us/dotnet/core/tools/global-tools). 使用命令行窗口安装:

````bash

dotnet tool install -g Volo.Abp.Cli

````

更新最新版本:

````bash

dotnet tool update -g Volo.Abp.Cli

````

## 全局选项

虽然每个命令可能都有一组选项,但有些全局选项可以与任何命令一起使用:

\* `--skip-cli-version-check`: 跳过检查最新版本的ABP CLI. 如果没有指定,它会检查最新版本,如果检查到ABP CLI的新版本,会显示一条警告消息.

## Commands

这里是所有可用的命令列表:

\* **\*\***`help`**\*\***: 展示ABP CLI的用法帮助信息.

\* **\*\***`cli`**\*\***: 更新或删除ABP CLI.

\* **\*\***`new`**\*\***：生成基于ABP的[启动模板](Startup-Templates/Index.md).

\* **\*\***`update`**\*\***：自动更新的ABP解决方案ABP相关的NuGet和NPM包.

\* **\*\***`clean`**\*\***: 删除当前目录下所有的 `BIN` 和 `OBJ` 子目录.

\* **\*\***`add-package`**\*\***: 添加ABP包到项目.

\* **\*\***`add-module`**\*\***: 添加[应用模块](https://docs.abp.io/en/abp/latest/Modules/Index)到解决方案.

\* **\*\***`generate-proxy`**\*\***: 生成客户端代理以使用HTTP API端点.

\* **\*\***`remove-proxy`**\*\***: 移除以前生成的客户端代理.

\* **\*\***`switch-to-preview`**\*\***: 切换到ABP框架的最新预览版本。

\* **\*\***`switch-to-nightly`**\*\***: 切换解决方案所有ABP相关包为[夜间构建](Nightly-Builds.md)版本.

\* **\*\***`switch-to-stable`**\*\***: 切换解决方案所有ABP相关包为最新的稳定版本.

\* **\*\***`translate`**\*\***: 当源代码控制存储库中有多个JSON[本地化]（Localization.md文件时,可简化翻译本地化文件的过程.

\* **\*\***`login`**\*\***: 使用你在[abp.io](https://abp.io/)的用户名和密码在你的计算机上认证.

\* **\*\***`login-info`**\*\***: 展示当前登录用户信息.

\* **\*\***`logout`**\*\***: 在你的计算机注销认证.

\* **\*\***`bundle`**\*\***: 为 ABP Blazor 和 MAUI Blazor 项目生成引用的脚本和样式.

\* **\*\***`install-libs`**\*\***: 为 MVC / Razor Pages 和 Blazor Server UI 类型安装NPM包.

\* **\*\***`clear-download-cache`**\*\*** 删除下载的模版缓存.

### help

展示ABP CLI的基本用法:

用法:

````bash

abp help [command-name]

````

示例:

````bash

abp help # Shows a general help.

abp help new # Shows help about the "new" command.

````

### cli

更新或删除ABP CLI

用法:

````bash

abp cli [command-name]

````

示例:

````bash

abp cli update

abp cli update --preview

abp cli update --version 5.0.0

abp cli remove

````

### new

生成基于ABP[启动模板](Startup-Templates/Index.md)的新解决方案.

用法:

````bash

abp new <解决方案名称> [options]

````

示例:

````bash

abp new Acme.BookStore

````

\* Acme.BookStore是解决方案的名称.

\* 常见的命名方式类似于 *\*YourCompany.YourProject\**. 不过你可以使用自己喜欢的方式,如 *\*YourProject\** (单级命名空间) 或 *\*YourCompany.YourProduct.YourModule\** (三级命名空间).

参阅[ABP CLI 创建新解决方案示例](CLI-New-Command-Samples.md)查看更多示例.

#### Options

\* `--template` 或者 `-t`: 指定模板. 默认的模板是 `app`,会生成web项目.可用的模板有:

\* `app` (default): [应用程序模板](Startup-Templates/Application.md). 其他选项:

\* `--ui` 或者 `-u`: 指定ui框架.默认`mvc`框架.其他选项:

\* `mvc`: ASP.NET Core MVC.此模板的其他选项:

\* `--tiered`: 创建分层解决方案,Web和Http Api层在物理上是分开的.如果未指定会创建一个分层的解决方案,此解决方案没有那么复杂,适合大多数场景.

\* `angular`: Angular. 这个模板还有一些额外的选项:

\* `--separate-auth-server`: 将Auth Server应用程序与API host应用程序分开. 如果未指定,则服务器端将只有一个端点.

\* `blazor`: Blazor. 这个模板还有一些额外的选项:

\* `--separate-auth-server`: 将Auth Server应用程序与API host应用程序分开. 如果未指定,则服务器端将只有一个端点.

\* `none`: 无UI. 这个模板还有一些额外的选项:

\* `--separate-auth-server`: 将Auth Server应用程序与API host应用程序分开. 如果未指定,则服务器端将只有一个端点.

\* `--mobile` 或者 `-m`: 指定移动应用程序框架. 如果未指定,则不会创建任何移动应用程序,其他选项:

\* `none`: 不包含移动应用程序.

\* `react-native`: React Native.

\* `--database-provider` 或者 `-d`: 指定数据库提供程序.默认是 `ef`.其他选项:

\* `ef`: Entity Framework Core.

\* `mongodb`: MongoDB.

\* `module`: [Module template](Startup-Templates/Module.md). 其他选项:

\* `--no-ui`: 不包含UI.仅创建服务模块(也称为微服务 - 没有UI).

\* **\*\***`console`**\*\***: [Console template](Startup-Templates/Console.md).

\* **\*\***`maui`**\*\***: [Maui template](Startup-Templates/MAUI.md).

\* **\*\***`app-nolayers`**\*\***: 应用程序单层模板

\* `--ui` 或者 `-u`: 指定ui框架.默认`mvc`框架.其他选项:

\* `mvc`: ASP.NET Core MVC.

\* `angular`: Angular.

\* `blazor`: Blazor UI.

\* `blazor-server`: Blazor Server.

\* `none`: 不包含UI.

\* `--database-provider` 或 `-d`: 或者 `-d`: 指定数据库提供程序.默认是 `ef`.其他选项:

\* `ef`: Entity Framework Core.

\* `mongodb`: MongoDB.

\* `--output-folder` 或者 `-o`: 指定输出文件夹,默认是当前目录.

\* `--version` 或者 `-v`: 指定ABP和模板的版本.它可以是 [release tag](https://github.com/abpframework/abp/releases) 或者 [branch name](https://github.com/abpframework/abp/branches). 如果没有指定,则使用最新版本.大多数情况下,你会希望使用最新的版本.

\* `--preview`: 使用最新的预览版本.

\* `--template-source` 或者 `-ts`: 指定自定义模板源用于生成项目,可以使用本地源和网络源(例如 `D:\local-templat` 或 `https://.../my-template-file.zip`).

\* `--create-solution-folder` 或者 `-csf`: 指定项目是在输出文件夹中的新文件夹中还是直接在输出文件夹中.

\* `--connection-string` 或者 `-cs`: 重写所有 `appsettings.json` 文件的默认连接字符串. 默认连接字符串是 `Server=localhost;Database=MyProjectName;Trusted\_Connection=True`. 默认的数据库提供程序是 `SQL Server`. 如果你使用EF Core但需要更改DBMS,可以按[这里所述](Entity-Framework-Core-Other-DBMS.md)进行更改(创建解决方案之后).

\* `--local-framework-ref --abp-path`: 使用对项目的本地引用,而不是替换为NuGet包引用.

\* `--skip-cache` or `-sc`: 从服务器下载最新的模版并更新本地模版缓存.

### update

更新所有ABP相关的包可能会很繁琐,框架和模块都有很多包. 此命令自动将解决方案或项目中所有ABP相关的包更新到最新版本.

用法:

````bash

abp update [options]

````

\* 如果你的文件夹中有.sln文件,运行命令会将解决方案中所有项目ABP相关的包更新到最新版本.

\* 如果你的文件夹中有.csproj文件,运行命令会将项目中所有ABP相关的包更新到最新版本.

#### Options

\* `--npm`: 仅更新NPM包

\* `--nuget`: 仅更新的NuGet包

\* `--solution-path` 或 `-sp`: 指定解决方案路径/目录. 默认使用当前目录

\* `--solution-name` 或 `-sn`: 指定解决方案名称. 默认在目录中搜索`\*.sln`文件.

\* `--check-all`: 分别检查每个包的新版本. 默认是 `false`.

\* `--version` or `-v`: 指定用于升级的版本. 如果没有指定,则使用最新版本.

### clean

删除当前目录下所有的 `BIN` 和 `OBJ` 子目录.

用法:

````bash

abp clean

````

### add-package

通过以下方式将ABP包添加到项目中

\* 添加nuget包做为项目的依赖项目.

\* 添加 `[DependsOn(...)]` attribute到项目的模块类 (请参阅 [模块开发文档](Module-Development-Basics.md)).

> 需要注意的是添加的模块可能需要额外的配置,通常会在包的文档中指出.

用法:

````bash

abp add-package <包名> [options]

````

示例:

````

abp add-package Volo.Abp.AspNetCore.Mvc.UI.Theme.Basic

````

\* 示例中将`Volo.Abp.AspNetCore.Mvc.UI.Theme.Basic`包添加到项目中.

#### Options

\* `--project` 或 `-p`: 指定项目 (.csproj) 路径. 如果未指定,Cli会尝试在当前目录查找.csproj文件.

\* `--with-source-code`: 下载包的源码到你的解决方案文件夹，而不是NuGet/NPM软件包.

\* `--add-to-solution-file`: 添加下载/创建的包添加到解决方案文件中,你在IDE中打开解决方案时也会看到包的项目. (仅当 `--with-source-code` 为 `True` 时可用.)

> 当前只有基本主题包([MVC](https://docs.abp.io/zh-Hans/abp/latest/UI/AspNetCore/Basic-Theme) 和 [Blazor](https://docs.abp.io/zh-Hans/abp/latest/UI/Blazor/Basic-Theme)) 可以下载.

> - Volo.Abp.AspNetCore.Mvc.UI.Theme.Basic

> - Volo.Abp.AspNetCore.Components.WebAssembly.BasicTheme

> - Volo.Abp.AspNetCore.Components.Web.BasicTheme

> - Volo.Abp.AspNetCore.Components.Server.BasicTheme

### add-module

通过查找模块的所有包,查找解决方案中的相关项目,并将每个包添加到解决方案中的相应项目,从而将[多包应用程序模块](Modules/Index)添加到解决方案中.

> 由于分层,不同的数据库提供程序选项或其他原因,业务模块通常由多个包组成. 使用`add-module`命令可以大大简化向模块添加模块的过程. 但是每个模块可能需要一些其他配置,这些配置通常在相关模块的文档中指出.

用法:

````bash

abp add-module <模块名称> [options]

````

示例:

```bash

abp add-module Volo.Blogging

```

\* 示例中将Volo.Blogging模块添加到解决方案中.

#### Options

\* `--solution` 或 `-s`: 指定解决方案 (.sln) 路径. 如果未指定,CLI会尝试在当前目录中寻找.sln文件.

\* `--skip-db-migrations`: 对于EF Core 数据库提供程序,它会自动添加新代码的第一次迁移 (`Add-Migration`) 并且在需要时更新数据库 (`Update-Database`). 指定此选项可跳过此操作.

\* `-sp` 或 `--startup-project`: 启动项目的项目文件夹的相对路径. 默认值是当前文件夹.

\* `--with-source-code`: 添加模块的源代码,而不是NuGet/NPM软件包.

\* `--add-to-solution-file`: 添加下载/创建的模块添加到解决方案文件中,你在IDE中打开解决方案时也会看到模块的项目. (仅当 `--with-source-code` 为 `True` 时可用.)

### generate-proxy

为你的HTTP API生成Angular, C# 或 JavaScript服务代理,简化从客户端使用服务的成本. 在运行此命令之前,你的host必须启动正在运行.

用法:

````bash

abp generate-proxy -t <client-type> [options]

````

示例:

````bash

abp generate-proxy -t ng

abp generate-proxy -t js -url https://localhost:44302/

abp generate-proxy -t csharp -url https://localhost:44302/

````

#### Options

\* `--type` 或 `-t`: 客户端类型的名称. 可用的客户端有:

\* `csharp`: C#, 工作在 `\*.HttpApi.Client` 项目目录. 此客户端有一些可选选项:

\* `--without-contracts`: 取消生成应用程序服务接口,类,枚举和DTO.

\* `--folder`: 放置生成的 CSharp 代码的文件夹名称. 默认值: `ClientProxies`.

\* `ng`: Angular. 此客户端有一些可选选项:

\* `--api-name` 或 `-a`: 在 `/src/environments/environment.ts` 中定义的API端点名称。. 默认值: `default`.

\* `--source` 或 `-s`: 指定解析根名称空间和API定义URL的Angular项目名称. 默认值: `defaultProject`

\* `--target` 或 `-t`: 指定放置生成的代码的Angular项目名称. 默认值: `defaultProject`.

\* `--prompt` 或 `-p`: 在命令行提示符下询问选项(未指定的选项).

\* `js`: JavaScript. 工作在 `\*.Web` 项目目录. 此客户端有一些可选选项:

\* `--output` or `-o`: 放置生成的 JavaScript 代码的文件夹名称.

\* `--module` 或 `-m`: 指定要为其生成代理的后端模块的名称. 默认值: `app`.

\* `--working-directory` or `-wd`: 执行目录. 用于 `csharp` 和 `js` 客户端类型.

\* `--url` or `-u`: API定义的URL. 用于 `csharp` 和 `js` 客户端类型.

> 参阅 [Angular服务代理文档](UI/Angular/Service-Proxies.md) 了解更多.

### remove-proxy

从Angular, CSharp 或 JavaScript应用程序中删除以前生成的代理代码. 在运行此命令之前,你的host必须启动正在运行.

这在你之前为多个模块生成代理并且需要删除其中一个模块时特别有用.

用法:

````bash

abp remove-proxy -t <client-type> [options]

````

示例:

````bash

abp remove-proxy -t ng

abp remove-proxy -t js -m identity -o Pages/Identity/client-proxies.js

abp remove-proxy -t csharp --folder MyProxies/InnerFolder

````

#### Options

\* `--type` 或 `-t`: 客户端类型的名称. 可用的客户端有:

\* `csharp`: C#, 工作在 `\*.HttpApi.Client` 项目目录. 此客户端有一些可选选项:

\* `--folder`: 放置生成的 CSharp 代码的文件夹名称. 默认值: `ClientProxies`.

\* `ng`: Angular. 此客户端有一些可选选项:

\* `--api-name` 或 `-a`: 在 `/src/environments/environment.ts` 中定义的API端点名称。. 默认值: `default`.

\* `--source` 或 `-s`: 指定解析根名称空间和API定义URL的Angular项目名称. 默认值: `defaultProject`

\* `--target` 或 `-t`: 指定放置生成的代码的Angular项目名称. 默认值: `defaultProject`.

\* `--prompt` 或 `-p`: 在命令行提示符下询问选项(未指定的选项).

\* `js`: JavaScript. 工作在 `\*.Web` 项目目录. 此客户端有一些可选选项:

\* `--output` or `-o`: 放置生成的 JavaScript 代码的文件夹名称.

\* `--module` 或 `-m`: 指定要为其生成代理的后端模块的名称. 默认值: `app`.

\* `--working-directory` or `-wd`: 执行目录. 用于 `csharp` 和 `js` 客户端类型.

\* `--url` or `-u`: API定义的URL. 用于 `csharp` 和 `js` 客户端类型.

> 参阅 [Angular服务代理文档](UI/Angular/Service-Proxies.md) 了解更多.

### switch-to-preview

你可以使用此命令将项目切换到ABP框架的最新预览版本.

用法:

````bash

abp switch-to-preview [options]

````

#### Options

\* `--solution-directory` 或 `-sd`: 指定目录. 解决方案应该在该目录或其子目录中. 如果未指定默认为当前目录.

### switch-to-nightly

想要切换到ABP框架的最新[每晚构建](Nightly-Builds.md)预览版可以使用此命令.

用法:

````bash

abp switch-to-nightly [options]

````

#### Options

`--solution-directory` 或 `-sd`: 指定目录. 解决方案应该在该目录或其子目录中. 如果未指定默认为当前目录.

### switch-to-stable

如果你使用的是ABP框架预览包(包括每晚构建),可以使用此命令切换回最新的稳定版本.

用法:

````bash

abp switch-to-stable [options]

````

#### Options

`--solution-directory` 或 `-sd`: 指定目录. 解决方案应该在该目录或其子目录中. 如果未指定默认为当前目录.

### translate

源代码控制存储库中有多个JSON[本地化](Localization.md)文件时,用于简化翻译[本地化](Localization.md)文件的过程.

\* 该命令将基于参考文化创建一个统一的json文件

\* 它搜索当前目录和所有子目录中的所有本地化"JSON"文件(递归). 然后创建一个包含所有需要翻译的条目的文件(默认情况下名为 "abp-translation.json").

\* 翻译了此文件中的条目后,你就可以使用 `--apply` 命令将更改应用于原始本地化文件.

> 该命令的主要目的是翻译ABP框架本地化文件(因为[abp仓库](https://github.com/abpframework/abp)包括数十个要在不同目录中转换的本地化文件).

#### 创建翻译文件

第一步是创建统一的翻译文件:

````bash

abp translate -c <culture> [options]

````

示例:

````bash

abp translate -c de

````

该命令为 `de-DE` (德语)文化创建了统一的翻译文件.

##### 附加选项

\* `--reference-culture` 或 `-r`: 默认值 `en`. 指定参考文化.

\* `--output` 或 `-o`: 输出文件名. 默认值 `abp-translation.json`.

\* `--all-values` 或 `-all`: 包括所有要翻译的键. 默认情况下,统一翻译文件仅包含目标文化的缺失文本. 如果你可能需要修改之前已经翻译的值,请指定此参数.

#### 应用更改

翻译完统一翻译文件中的条目后,你可以使用 `--apply` 参数将更改应用于原始本地化文件:

````bash

abp translate --apply # apply all changes

abp translate -a # shortcut for --apply

````

然后,检查源代码控制系统上的更改,以确保它已更改了正确的文件. 如果你翻译了ABP框架资源, 请发送 "Pull Request". 提前感谢你的贡献.

##### 附加选项

\* `--file` 或 `-f`: 默认值: `abp-translation.json`. 翻译文件(仅在之前使用过 `--output` 选项时使用).

### login

CLI的一些功能需要登录到abp.io平台. 使用你的用户名登录

```bash

abp login <username> # Allows you to enter your password hidden

abp login <username> -p <password> # Specify the password as a parameter (password is visible)

abp login <username> --organization <organization> # If you have multiple organizations, you need set your active organization

abp login <username> -p <password> -o <organization> # You can enter both your password and organization in the same command

abp login <username> --device # Use device login flow

```

> 当使用-p参数,请注意,因为你的密码是可见的. 它对于CI / CD自动化管道很有用.

请注意,新的登录将终止先前的会话并创建一个新的会话.

### login-info

展示你的登录信息, 如 **\*\*名称\*\*** , **\*\*用户名\*\*** , **\*\*地址\*\*** 和 **\*\*组织\*\***.

```bash

abp login-info

```

### logout

通过从计算机中删除会话令牌来注销.

```

abp logout

```

### bundle

这个命令为ABP Blazor WebAssembly 和 MAUI Blazor 项目生成引用的脚本和样式并且更新 **\*\*index.html\*\*** 文件, 它帮助开发者轻松的管理ABP模块的依赖. 为了使 ```bundle``` 命令工作, 它的**\*\*执行目录\*\***或者传递 ```--working-directory``` 参数的目录必须包含Blazor或MAUI Blazor项目文件(\*.csproj)

用法:

````bash

abp bundle [options]

````

#### Options

\* ```--working-directory``` 或 ```-wd```: 指定当前执行目录, 这个命令在当前目录不包含项目文件时非常有用.

\* ```--force``` 或 ```-f```: 在生成引用之前强制构建项目.

\* ```--project-type``` 或 ```-t```: 指定项目类型. 默认类型是 `webassembly`. 可用的类型有:

\* `webassembly`

\* `maui-blazor`

`bundle` command reads the `appsettings.json` file inside the Blazor and MAUI Blazor project for bundling options. For more details about managing style and script references in Blazor or MAUI Blazor apps, see [Managing Global Scripts & Styles](UI/Blazor/Global-Scripts-Styles.md)

### install-libs

为 MVC / Razor Pages 和 Blazor Server UI 类型安装NPM包, 它的 **\*\*执行目录\*\*** 或者传递的 ```--working-directory``` 目录必须包含一个项目文件(\*.csproj).

`install-libs` 命令读取 `abp.resourcemapping.js` 来管理包. 参阅[客户端包管理](UI/AspNetCore/Client-Side-Package-Management.md)了解更多细节.

用法:

````bash

abp install-libs [options]

````

#### Options

\* ```--working-directory``` 或 ```-wd```: 指定工作目录, 当执行目录不包含项目文件时会很有用.

新命令示例

# ABP CLI - 新解决方案命令示例

`abp new`命令基于abp模板创建abp解决方案或其他组件. [ABP CLI](CLI.md)有一些参数可以用于创建新的ABP解决方案. 在本文档中, 我们将向你展示一些创建新的解决方案的命令示例. 所有的项目名称都是`Acme.BookStore`. 目前, 唯一可用的移动端项目是`React Native`移动端应用程序. 可用的数据库提供程序有`Entity Framework Core`和`MongoDB`. 所有命令都以`abp new`开头.

## Angular

以下命令用于创建Angular UI项目:

\* 在新文件夹中创建项目, **\*\*Entity Framework Core\*\***, 非移动端应用程序:

````bash

abp new Acme.BookStore -u angular --mobile none --database-provider ef -csf

````

\* 在新文件夹中创建项目, **\*\*Entity Framework Core\*\***, 默认应用程序模板, **\*\*拆分Auth Server\*\***:

```bash

abp new Acme.BookStore -t app -u angular -m none --separate-auth-server --database-provider ef -csf

```

\* 在新文件夹中创建项目, **\*\*Entity Framework Core\*\***, **\*\*自定义连接字符串\*\***:

```bash

abp new Acme.BookStore -u angular -csf --connection-string Server=localhost;Database=MyDatabase;Trusted\_Connection=True

```

\* 在`C:\MyProjects\Acme.BookStore`中创建解决方案, **\*\*MongoDB\*\***, 默认应用程序模板, 包含移动端项目:

```bash

abp new Acme.BookStore -u angular --database-provider mongodb --output-folder C:\MyProjects\Acme.BookStore

```

\* 在新文件夹中创建项目, **\*\*MongoDB\*\***, 默认应用程序模板, 不创建移动端应用程序, **\*\*拆分Auth Server\*\***:

```bash

abp new Acme.BookStore -t app -u angular -m none --separate-auth-server --database-provider mongodb -csf

```

## MVC

以下命令用于创建MVC UI项目:

\* 在新文件夹中创建项目, **\*\*Entity Framework Core\*\***, 不创建移动端应用程序:

```bash

abp new Acme.BookStore -t app -u mvc --mobile none --database-provider ef -csf

```

\* 在新文件夹中创建项目, **\*\*Entity Framework Core\*\***, **\*\*分层结构\*\*** (*\*Web和HTTP API层是分开的\**), 不创建移动端应用程序:

```bash

abp new Acme.BookStore -u mvc --mobile none --tiered --database-provider ef -csf

```

\* 在新文件夹中创建项目, **\*\*MongoDB\*\***, 不创建移动端应用程序:

```bash

abp new Acme.BookStore -t app -u mvc --mobile none --database-provider mongodb -csf

```

\* 在新文件夹中创建项目, **\*\*MongoDB\*\***, **\*\*分层结构\*\***:

```bash

abp new Acme.BookStore -u mvc --tiered --database-provider mongodb -csf

```

## Blazor

以下命令用于创建Blazor项目:

\* **\*\*Entity Framework Core\*\***, 不创建移动端应用程序:

```bash

abp new Acme.BookStore -t app -u blazor --mobile none

```

\* **\*\*Entity Framework Core\*\***, **\*\*拆分Auth Server\*\***, 包含移动端应用程序:

```bash

abp new Acme.BookStore -u blazor --separate-auth-server

```

\* 在新文件夹中创建项目, **\*\*MongoDB\*\***, 不创建移动端应用程序:

```bash

abp new Acme.BookStore -u blazor --database-provider mongodb --mobile none -csf

```

## Blazor Server

以下命令用于创建Blazor项目:

\* **\*\*Entity Framework Core\*\***, 不创建移动端应用程序:

```bash

abp new Acme.BookStore -t app -u blazor-server --mobile none

```

\* **\*\*Entity Framework Core\*\***, **\*\*拆分Auth Server\*\***, **\*\*拆分API Host\*\***, 包含移动端应用程序:

```bash

abp new Acme.BookStore -u blazor-server --tiered

```

\* 在新文件夹中创建项目, **\*\*MongoDB\*\***, 不创建移动端应用程序:

```bash

abp new Acme.BookStore -u blazor --database-provider mongodb --mobile none -csf

```

## 无UI

在默认应用程序模板中, 始终有一个前端项目. 在这个选项中没有前端项目. 它有一个`HttpApi.Host`项目为你的HTTP WebAPI提供服务. 这个选项适合在你想创建一个WebAPI服务时使用.

\* 在新文件夹中创建项目, **\*\*Entity Framework Core\*\***, 拆分Auth Server:

```bash

abp new Acme.BookStore -u none --separate-auth-server -csf

```

\* **\*\*MongoDB\*\***, 不创建移动端应用程序:

```bash

abp new Acme.BookStore -u none --mobile none --database-provider mongodb

```

## 控制台应用程序

这是一个基于.NET控制台应用程序的模板, 集成了ABP模块架构. 要创建控制台应用程序, 请使用以下命令:

\* 项目由以下文件组成: `Acme.BookStore.csproj`, `appsettings.json`, `BookStoreHostedService.cs`, `BookStoreModule.cs`, `HelloWorldService.cs` 和 `Program.cs`.

```bash

abp new Acme.BookStore -t console -csf

```

## 模块

模块是主项目使用的可重用子应用程序. 如果你正在构建微服务解决方案, 使用ABP模块是最佳方案. 由于模块不是最终的应用程序, 每个模块都有前端UI项目和数据库提供程序. 模块模板带有MVC UI, 可以在没有最终解决方案的情况下进行开发. 但是, 如果要在最终解决方案下开发模块, 可以添加`--no-ui`参数来去除MVC UI项目.

\* 包含前端: `MVC`, `Angular`, `Blazor`. 包含数据库提供程序: `Entity Framework Core`, `MongoDB`. 包含MVC启动项目.

```bash

abp new Acme.IssueManagement -t module

```

\* 与上面相同, 但不包括MVC启动项目.

```bash

abp new Acme.IssueManagement -t module --no-ui

```

\* 创建模块并将其添加到解决方案中

```bash

abp new Acme.IssueManagement -t module --add-to-solution-file

```

## 从特定版本创建解决方案

创建解决方案时, 它总是使用最新版本创建. 要从旧版本创建项目, 可以使用`--version`参数.

\* 使用v3.3.0版本创建解决方案, 包含Angular UI和Entity Framework Core.

```bash

abp new Acme.BookStore -t app -u angular -m none --database-provider ef -csf --version 3.3.0

```

要获取ABP版本列表, 请查看以下链接: https://www.nuget.org/packages/Volo.Abp.Core/

## 从自定义模板创建

ABP CLI使用默认的[应用程序模板](https://github.com/abpframework/abp/tree/dev/templates/app)创建项目. 如果要从自定义模板创建新的解决方案, 可以使用参数`--template-source`.

\* 在`c:\MyProjects\templates\app`目录中使用模板, MVC UI, Entity Framework Core, 不创建移动端应用程序.

```bash

abp new Acme.BookStore -t app -u mvc --mobile none --database-provider ef --template-source "c:\MyProjects\templates\app"

```

\* 除了此命令从URL `https://myabp.com/app-template.zip` 检索模板之外, 与上一个命令相同.

```bash

abp new Acme.BookStore -t app -u mvc --mobile none --database-provider ef --template-source https://myabp.com/app-template.zip

```

## 创建预览版本

ABP CLI始终使用最新版本. 要从预览(RC)版本创建解决方案, 请添加`--preview`参数.

\* 在新文件夹中创建项目, Blazor UI, Entity Framework Core, 不创建移动端应用程序, **\*\*使用最新版本\*\***:

```bash

abp new Acme.BookStore -t app -u blazor --mobile none -csf --preview

```

## 选择数据库管理系统

默认的数据库管理系统是 `Entity Framework Core` / ` SQL Server`. 你可以通过使用`--database-management-system`参数选择DBMS. [可用的值](https://github.com/abpframework/abp/blob/dev/framework/src/Volo.Abp.Cli.Core/Volo/Abp/Cli/ProjectBuilding/Building/DatabaseManagementSystem.cs) 包括 `SqlServer`, `MySQL`, `SQLite`, `Oracle`, `Oracle-Devart`, `PostgreSQL`. 默认值是 `SqlServer`.

\* 在新文件夹中创建项目, Angular UI, **\*\*PostgreSQL\*\*** 数据库:

```bash

abp new Acme.BookStore -u angular --database-management-system PostgreSQL -csf

```

## 使用静态HTTP端口

ABP CLI始终为项目分配随机端口. 如果需要保留默认端口并且创建解决方案始终使用相同的HTTP端口, 请添加参数`--no-random-port`.

\* 在新文件夹中创建项目, MVC UI, Entity Framework Core, **\*\*静态端口\*\***:

```bash

abp new Acme.BookStore --no-random-port -csf

```

## 引用本地ABP框架

在ABP解决方案中, 默认情况下从NuGet引用ABP库. 有时, 你需要在本地将ABP库引用到你的解决方案中. 这利于调试框架本身. 本地ABP框架的根目录必须有`Volo.Abp.sln`文件. 你可以将以下目录的内容复制到你的文件系统中

\* MVC UI, Entity Framework Core, **\*\*引用本地的ABP库\*\***:

本地路径必须是ABP存储库的根目录.

如果`C:\source\abp\framework\Volo.Abp.sln`是你的框架解决方案的路径, 那么你必须设置`--abp-path`参数值为`C:\source\abp`.

```bash

abp new Acme.BookStore --local-framework-ref --abp-path C:\source\abp

```

**\*\*输出\*\***:

如下所示, 引用本地ABP框架库项目.

```xml

<ItemGroup>

<ProjectReference Include="C:\source\abp\framework\src\Volo.Abp.Autofac\Volo.Abp.Autofac.csproj" />

<ProjectReference Include="C:\source\abp\framework\src\Volo.Abp.AspNetCore.Serilog\Volo.Abp.AspNetCore.Serilog.csproj" />

<ProjectReference Include="C:\source\abp\framework\src\Volo.Abp.AspNetCore.Authentication.JwtBearer\Volo.Abp.AspNetCore.Authentication.JwtBearer.csproj" />

<ProjectReference Include="..\Acme.BookStore.Application\Acme.BookStore.Application.csproj" />

<ProjectReference Include="..\Acme.BookStore.HttpApi\Acme.BookStore.HttpApi.csproj" />

<ProjectReference Include="..\Acme.BookStore.EntityFrameworkCore\Acme.BookStore.EntityFrameworkCore.csproj" />

</ItemGroup>

```

## 另请参阅

\* [ABP CLI文档](CLI.md)

## API文档

[https://docs.abp.io/api-docs/abp/{version}/api/index.html](https://docs.abp.io/api-docs/abp/%7bversion%7d/api/index.html)

https://docs.abp.io/api-docs/abp/7.3/api/index.html

# 贡献指南

# 贡献指南

ABP是[开源](https://github.com/abpframework)和社区驱动项目. 本指南旨在帮助任何想要为项目做出贡献的人.

## community.abp.io

如果你可编写文章或关于ASP框架和ASP.NET Core的 "如何" 指南,请提交你的文章到[community.abp.io](https://community.abp.io/)网站.

## 贡献代码

你可以将Pull request(拉取请求)发送到Github存储库.

- 从Github克隆[ABP存储库](https://github.com/abpframework/abp/).

- 进行必要的更改.

- 发送Pull request(拉取请求).

在进行任何更改之前,请在[Github问题](https://github.com/abpframework/abp/issues)上进行讨论. 通过这种方式, 其他开发人员将不会处理同一个问题, 你的PR将有更好的机会被接受.

### Bug修复 & 增强功能

你可能希望修复已知Bug或处理计划的增强功能. 请参阅Github上的[问题列表](https://github.com/abpframework/abp/issues).

### 功能请求

如果你对框架或模块有功能的想法, 请在Github上[创建一个问题](https://github.com/abpframework/abp/issues/new)或参加现有的讨论. 如果它被社区所接受你就可以实现它.

## 文档翻译

你可能希望将完整的[文档](https://abp.io/documents/)(包括本文)翻译成你的母语. 请按照下列步骤操作:

\* 从Github克隆[ABP存储库](https://github.com/abpframework/abp/).

\* 要添加新语言,请在[docs](https://github.com/abpframework/abp/tree/master/docs)文件夹中创建一个新文件夹. 文件夹名称可以是" en","es","fr","tr"等(参见[所有文化代码](https://msdn.microsoft.com/en-us/library/hh441729.aspx)).

\* 获取["en"文件夹](https://github.com/abpframework/abp/tree/master/docs/en)作为文件名和文件夹结构的参考. 如果要翻译相同的文档, 请保持相同的命名.

\* 翻译任何文档后发送拉取请求(PR). 请翻译文件后及时发送PR. 不要等到完成所有文件的翻译.

在[ABP文档网站](https://docs.abp.io)上新添加语言之前,需要翻译一些基本文档:

\* 入门文档

\* 教程

\* CLI

完成了这些基本的翻译后,将添加一种新的语言

## 资源本地化

ABP框架具有灵活的[本地化系统](../Localization.md). 你可以为自己的应用程序创建本地化用户界面.

除此之外,框架和预构建模块已经本地化了文本.请参阅[Volo.Abp.UI包的本地化文本](https://github.com/abpframework/abp/blob/master/framework/src/Volo.Abp.UI/Localization/Resources/AbpUi/en.json).

### 使用 "abp translate" 命令

这是推荐的方法,因为它会自动查找所有缺少的文本的特定文化,让你在一个地方翻译.

\* 从Github克隆[ABP存储库](https://github.com/abpframework/abp/).

\* 安装[ABP CLI](https://docs.abp.io/en/abp/latest/CLI).

\* 在abp仓储的根文件夹为你的语言运行`abp translate -c <culture-name>`命令. 例如对法语使用 `abp translate -c fr`, 检查[文档](https://docs.microsoft.com/en-us/bingmaps/rest-services/common-parameters-and-types/supported-culture-codes)找到你所用语言的文化代码.

\* 命令会在同一文件夹下创建 `abp-translation.json` 文件, 使用你喜欢的编辑器打开这个文件并填写缺少的文本值.

\* 一旦你完成了翻译,使用 `abp translate -a` 命令应用更改到相关的文件.

\* 在GitHub上发送PR.

### 手动翻译

如果你想更改特定的资源文件,你可以自己找到这个文件进行必要的更改(或为你的语言创建新文件),并在GitHub上发送PR.

## Bug 报告

如果你发现任何Bug, 请[在Github存储库上创建一个问题](https://github.com/abpframework/abp/issues/new).