**ALEXFW框架使用说明**

[1 框架配置 5](#_Toc5200113)

[1.1 获取框架 5](#_Toc5200114)

[1.2 打开解决方案 5](#_Toc5200115)

[1.3 启用种子数据 6](#_Toc5200116)

[1.4 编译运行，测试前台member登录功能 7](#_Toc5200117)

[2 自动化生产开发环境的使用 8](#_Toc5200118)

[2.1 在领域模型的Entity项目中添加演示实体Category 8](#_Toc5200120)

[2.2 参考注释说明，利用特性定义实体增删查改中的样式 9](#_Toc5200121)

[2.3 打开数据上下文类，添加EF的映射 9](#_Toc5200122)

[2.4 自动生成控制器 10](#_Toc5200123)

[2.5 实体的权限控制 13](#_Toc5200124)

[3 MVC示例 15](#_Toc5200125)

[4 框架详解 21](#_Toc5200127)

[4.1 基础结构 21](#_Toc5200129)

[4.2 建立实体 21](#_Toc5200130)

[4.2.1. IEntity接口 21](#_Toc5200137)

[4.2.2. IPassword接口 24](#_Toc5200138)

[4.2.3. EntityBase 24](#_Toc5200139)

[4.2.4. UserBase 25](#_Toc5200140)

[4.3 操作实体 25](#_Toc5200141)

[4.3.1. 实体上下文IEntityContextBuilder 26](#_Toc5200143)

[4.3.2. IEntityQueryable 26](#_Toc5200144)

[4.3.3. 创建实体 27](#_Toc5200145)

[4.3.4. 添加实体 27](#_Toc5200146)

[4.3.5. 编辑实体 28](#_Toc5200147)

[4.3.6. 删除实体 28](#_Toc5200148)

[4.3.7. 查询Query 28](#_Toc5200149)

[4.3.8. InParent 29](#_Toc5200150)

[4.3.9. Count与Contains 30](#_Toc5200151)

[4.3.10. 排序 30](#_Toc5200152)

[**4.4** 实体特性 31](#_Toc5200153)

[4.4.1. DisplayNameAttribute实体名称 31](#_Toc5200155)

[4.4.2. DisplayColumnAttribute实体显示 31](#_Toc5200156)

[4.4.3. EntityAuthenticationAttribute实体权限 32](#_Toc5200157)

[4.4.4. ParentAttribute实体父级 33](#_Toc5200158)

[**4.5** 属性特性 33](#_Toc5200159)

[4.5.1. ColumnAttribute数据库类型 33](#_Toc5200161)

[4.5.2. CustomDataTypeAttribute自定义类型 34](#_Toc5200162)

[4.5.3. DisplayAttribute显示信息 35](#_Toc5200163)

[4.5.4. DistinctAttribute不同内容 36](#_Toc5200164)

[4.5.5. HideAttribute隐藏字段 36](#_Toc5200165)

[4.5.6. KeyAttribute主键 37](#_Toc5200166)

[4.5.7. MaxLengthAttribute最大长度 37](#_Toc5200167)

[4.5.8. MinLengthAttribute最小长度 37](#_Toc5200168)

[4.5.9. PropertyAuthenticationAttribute权限 38](#_Toc5200169)

[4.5.10. RangeAttribute范围 38](#_Toc5200170)

[4.5.11. RegularExpressionAttribute正则表达式 39](#_Toc5200171)

[4.5.12. RequiredAttribute必填项 39](#_Toc5200172)

[4.5.13. SearchableAttribute可搜索 40](#_Toc5200173)

[4.5.14. ValueFilterAttribute值筛选（级联） 40](#_Toc5200174)

[**4.6** 重写 43](#_Toc5200175)

[4.6.1. 创建时 43](#_Toc5200177)

[4.6.2. 编辑时 44](#_Toc5200178)

[4.6.3. 是否可删除 45](#_Toc5200179)

[4.6.4. 是否可编辑 46](#_Toc5200180)

[4.6.5. 转换字符串 47](#_Toc5200181)

[4.7 使用上下文 48](#_Toc5200182)

[4.7.1. EntityContextBuilder 49](#_Toc5200184)

[4.7.2. EntityQueryable 49](#_Toc5200185)

[4.8 MVC 50](#_Toc5200186)

[4.8.1. 模板导入说明 50](#_Toc5200188)

[4.8.2. 无形参控制器 EntityController 51](#_Toc5200189)

[4.8.3. EntityBuilder属性 52](#_Toc5200190)

[4.8.4. 实体控制器 EntityController<TEntity> 52](#_Toc5200191)

[4.8.5. 列表 53](#_Toc5200192)

[4.8.6. 视图模型 54](#_Toc5200193)

[4.8.7. 重写 55](#_Toc5200194)

[4.8.8. 新增 56](#_Toc5200195)

[4.8.9. 编辑 57](#_Toc5200196)

[4.8.10. 更新 58](#_Toc5200197)

[4.8.11. 删除 59](#_Toc5200198)

[4.8.12. 详细 60](#_Toc5200199)

[4.8.13. 搜索 61](#_Toc5200200)

[4.8.14. 单项选择器 61](#_Toc5200201)

[4.8.15. 多项选择器 63](#_Toc5200202)

[4.8.16. 保存文件 66](#_Toc5200203)

[4.8.17. 显示图片 67](#_Toc5200204)

[4.9 编辑器 67](#_Toc5200205)

[4.9.1. 默认 68](#_Toc5200207)

[4.9.2. 自定义编辑器 68](#_Toc5200208)

[4.10 权限系统 71](#_Toc5200209)

[4.10.1. 权限实体 72](#_Toc5200211)

[4.10.2. 实体解释器 73](#_Toc5200212)

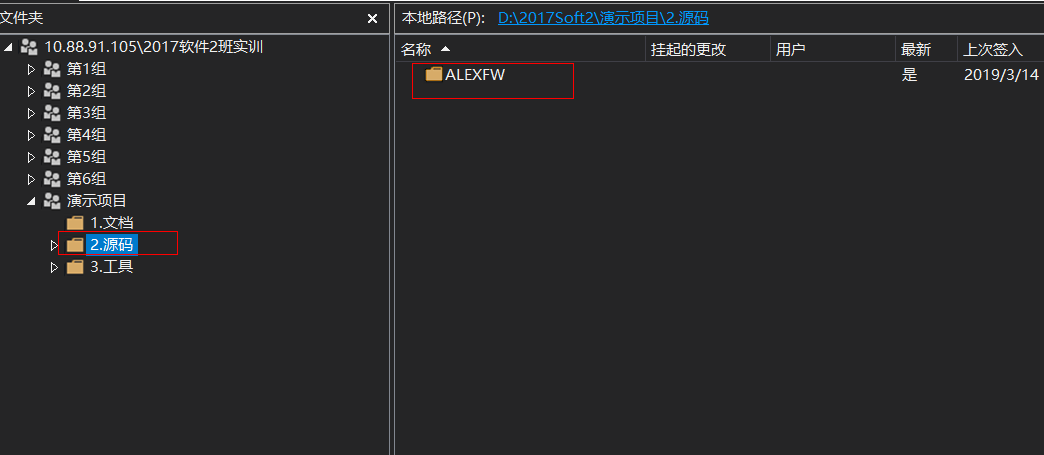
[4.10.3. 修改路由 73](#_Toc5200213)

[4.10.4. 权限的使用 75](#_Toc5200214)

[4.10.5. 登录与登出 77](#_Toc5200215)

# 框架配置

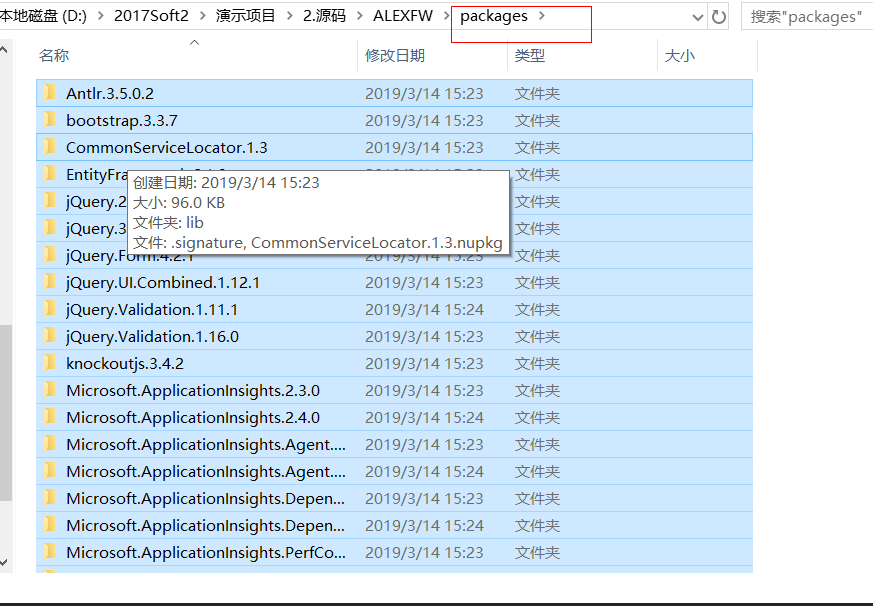
## 获取框架



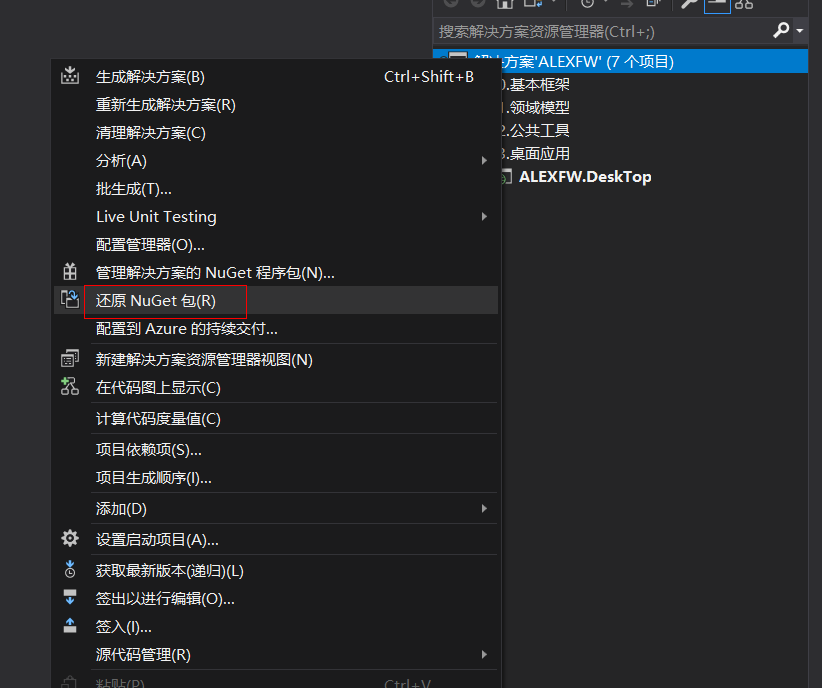
## 打开解决方案



删除Package中的内容。

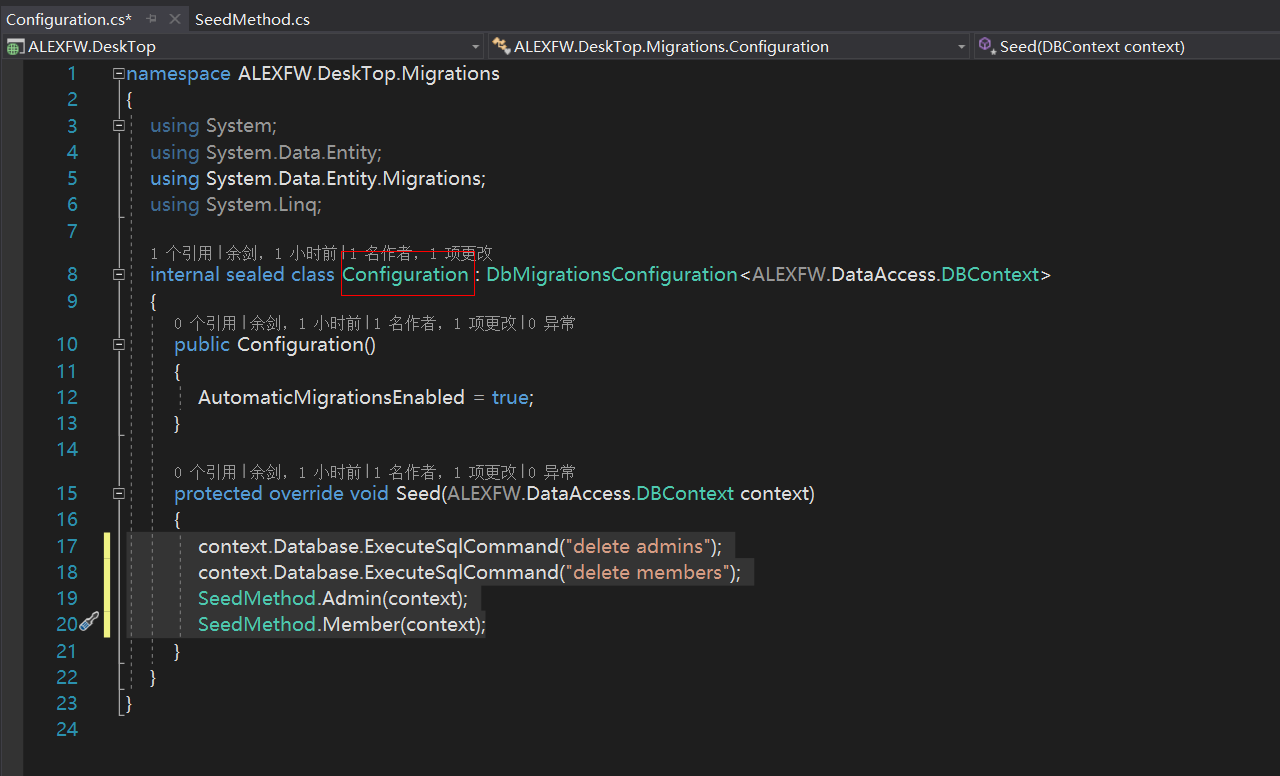


还原nuget包。

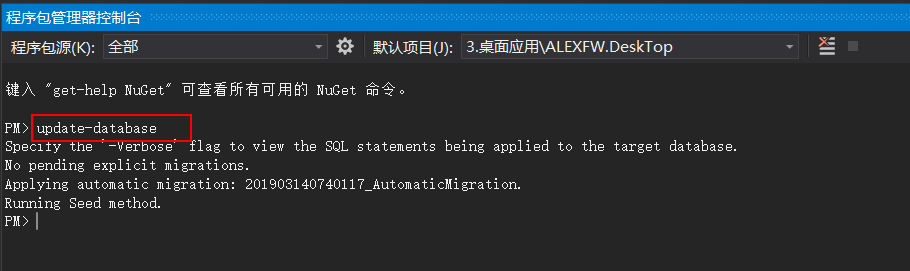


重新生成框架。

## 启用种子数据



更新到SQL Server。



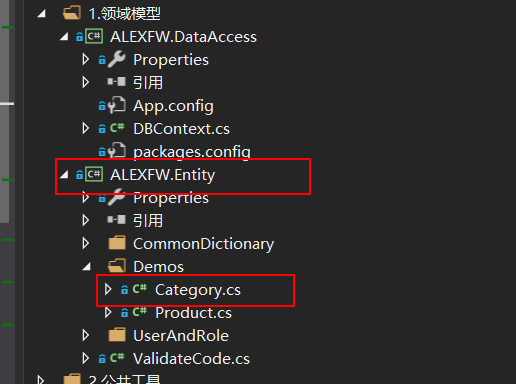
## 编译运行，测试前台member登录功能



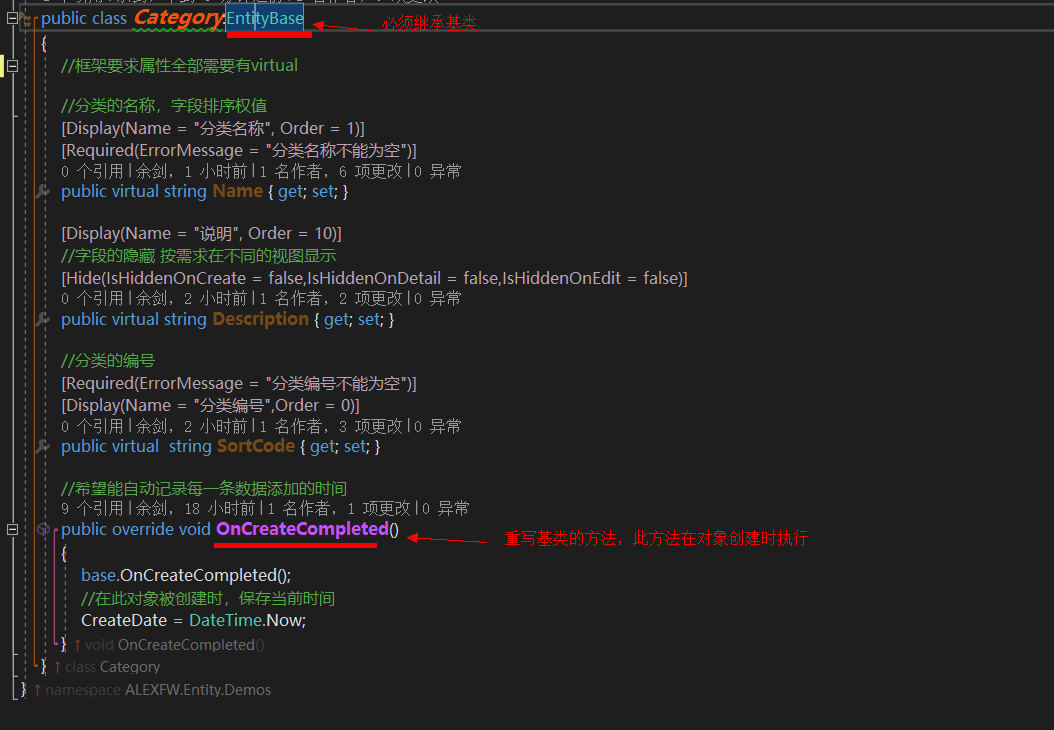
# 自动化生产开发环境的使用



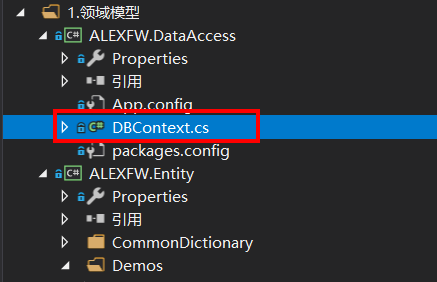
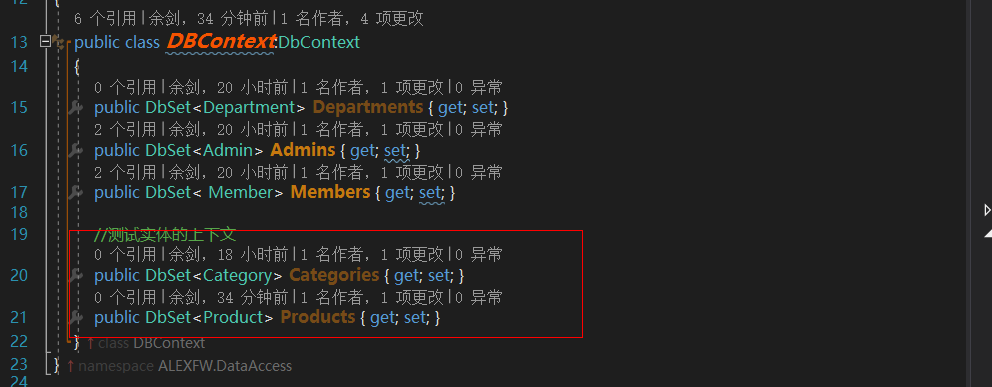
## 在领域模型的Entity项目中添加演示实体Category



## 参考注释说明，利用特性定义实体增删查改中的样式



## 打开数据上下文类，添加EF的映射

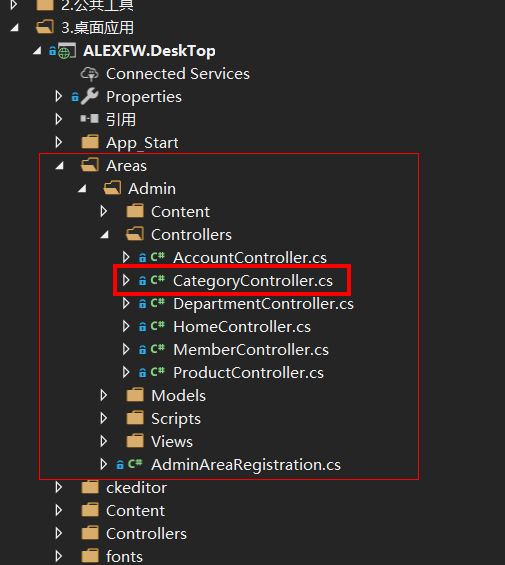
 

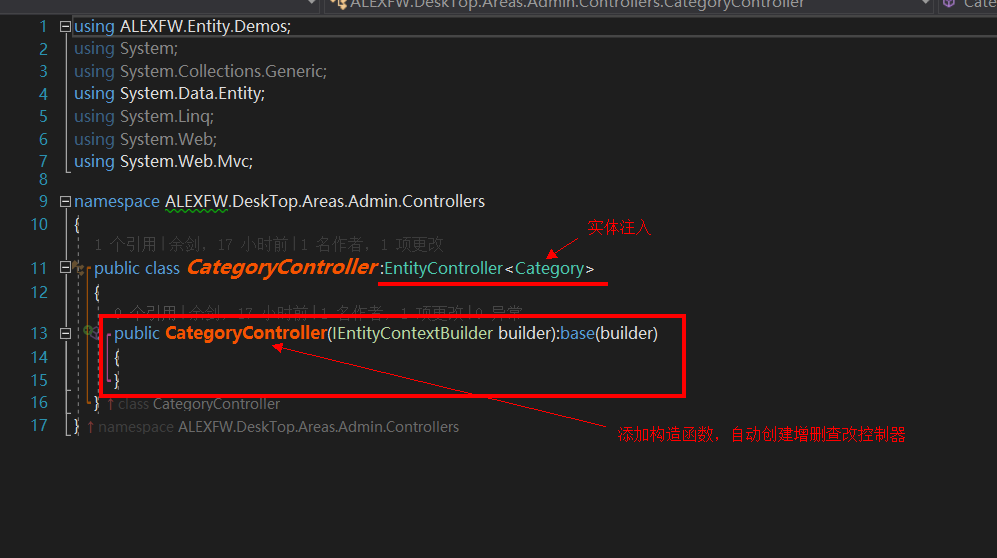
执行数据迁移，映射到SQL Server

## 自动生成控制器

框架为admin管理用户组单独使用管理域/admin做后台管理功能，并且在生产环境下自动生成CRUD控制器。

在对应位置创建与实体名称必须一致的空controller，如实体为Category，则控制器为CategoryController

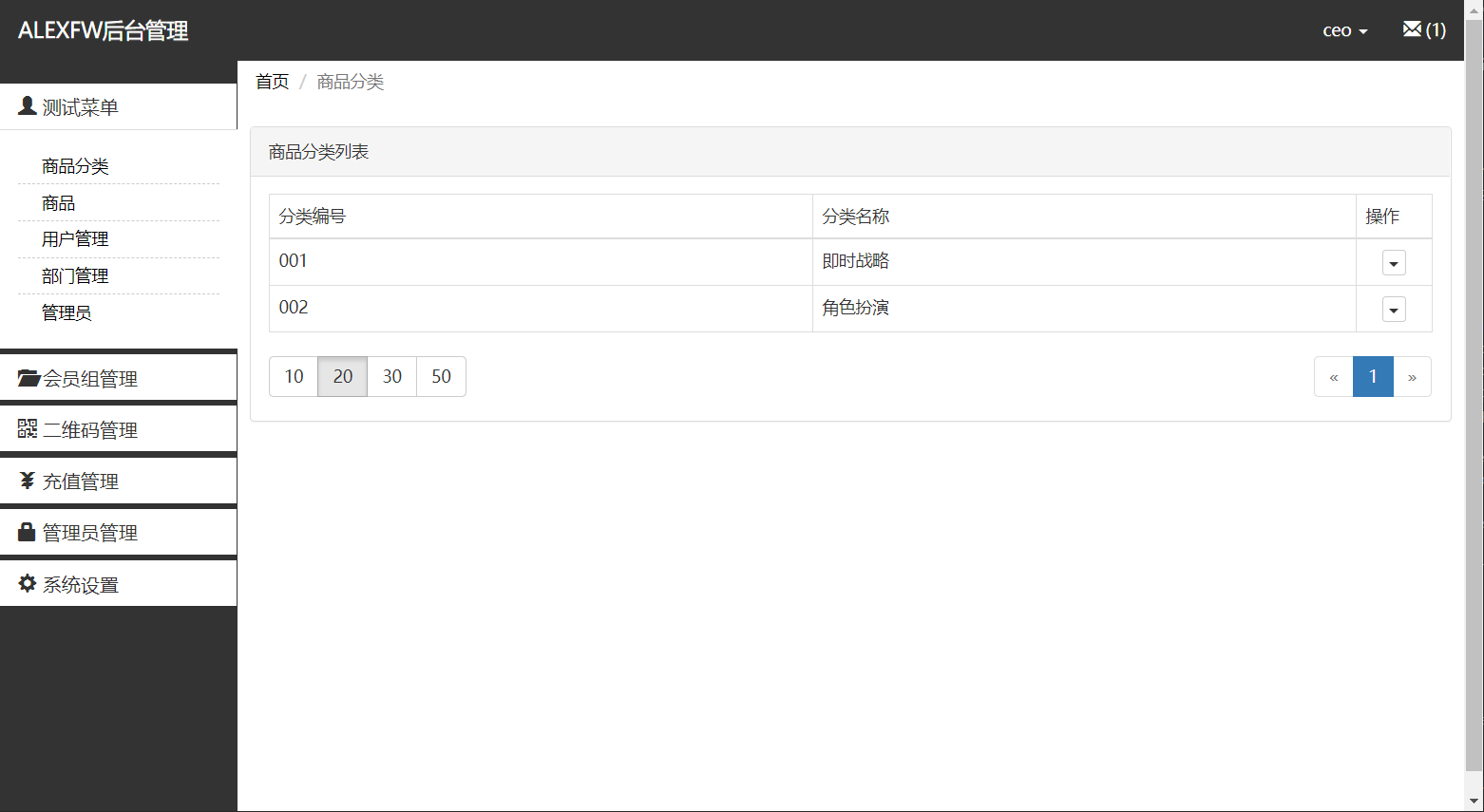




编译运行得到Category的实体维护功能



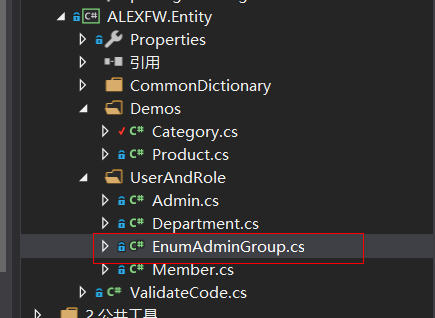
利用默认管理员登录



自动创建访问的控制器地址是http://localhost:XXXX/admin/category

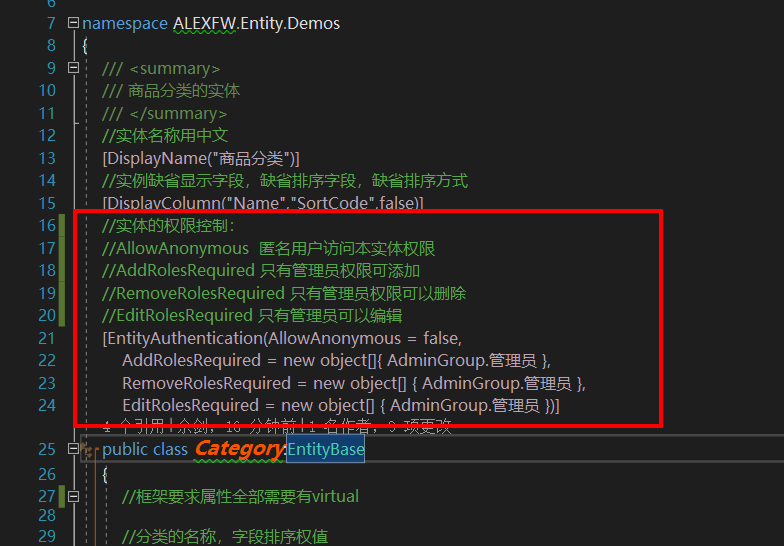
## 实体的权限控制

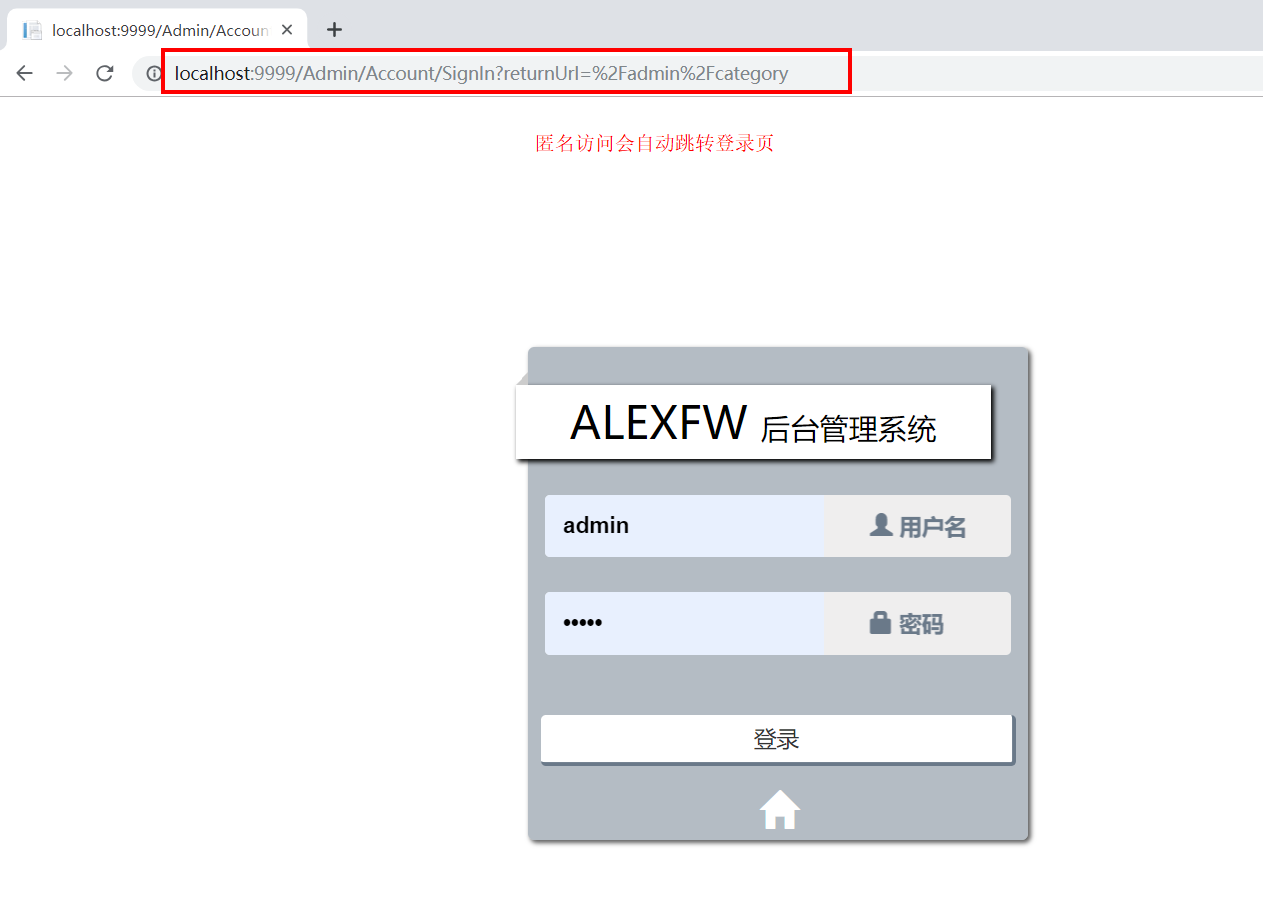
权限值在领域模型中利用Flags特性定义





实体中利用EntityAuthentication特性定义权限





用户不是“管理员”，受限访问



# MVC示例



在Models文件夹里添加EmployeeGroup员工组类。

EmployeeGroup继承EntityBase。

[复制代码](javascript:void(0);)

[DisplayName("员工组")]

[DisplayColumn("GroupName", "GroupName")]

public class EmployeeGroup : EntityBase

{

[Required]

[Display(Name = "员工组名称", Order = 0)]

public virtual string GroupName { get; set; }

[Hide]

public virtual ICollection<Employee> Employees { get; set; }

}

[复制代码](javascript:void(0);)

在Models文件夹里添加Employee员工组类。

Employee同样继承EntityBase。

[复制代码](javascript:void(0);)

[DisplayName("员工")]

[DisplayColumn("Name", "Name")]

public class Employee : EntityBase

{

[Display(Name = "员工名称", Order = 0)]

[Required]

public virtual string Name { get; set; }

[Display(Name = "性别", Order = 10)]

[CustomDataType(CustomDataType.Sex)]

public virtual bool Sex { get; set; }

[Required]

[Display(Name = "员工工号", Order = 20)]

public virtual string JobNumber { get; set; }

[Column(TypeName = "datetime2")]

[Display(Name = "出生日期", Order = 30)]

public virtual DateTime Birth { get; set; }

[Display(Name = "婚否", Order = 40)]

public virtual bool Marital { get; set; }

[Required]

[Display(Name = "部门", Order = 50)]

public virtual EmployeeGroup Group { get; set; }

[Display(Name = "联系电话", Order = 60)]

public virtual string Tel { get; set; }

[Display(Name = "电子邮件", Order = 70)]

public virtual string Email { get; set; }

[Display(Name = "QQ", Order = 80)]

public virtual string QQ { get; set; }

}

[复制代码](javascript:void(0);)

public class DataContext : DbContext

{

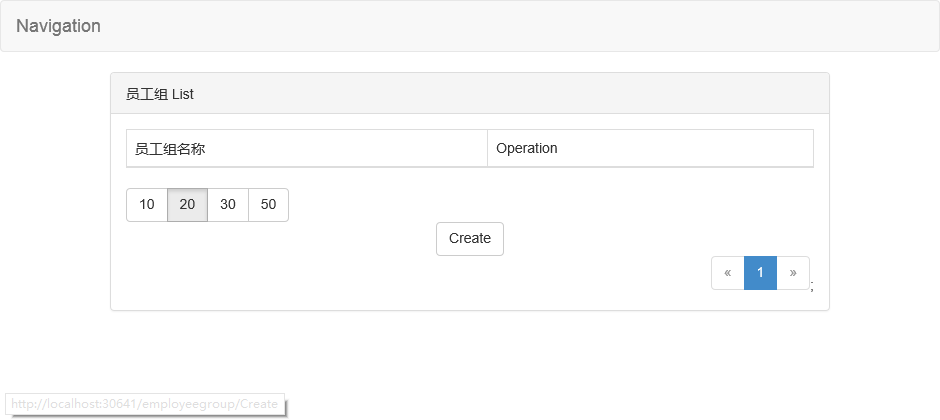
public DbSet<Employee> Employee { get; set; }

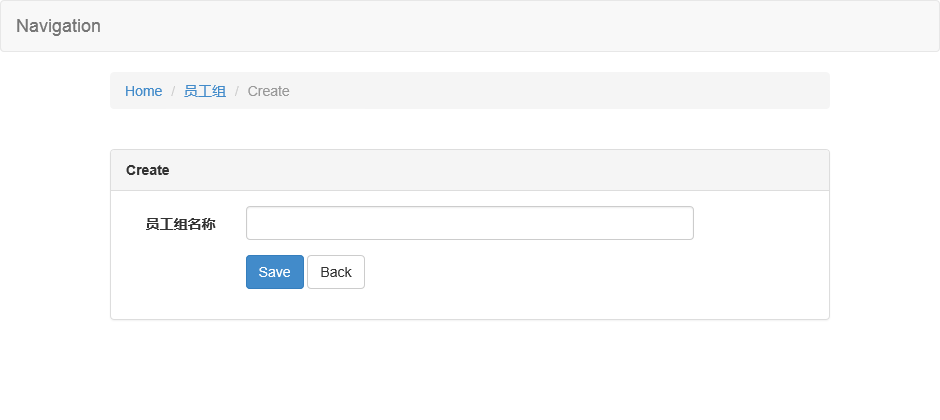
public DbSet<EmployeeGroup> EmployeeGroup { get; set; }

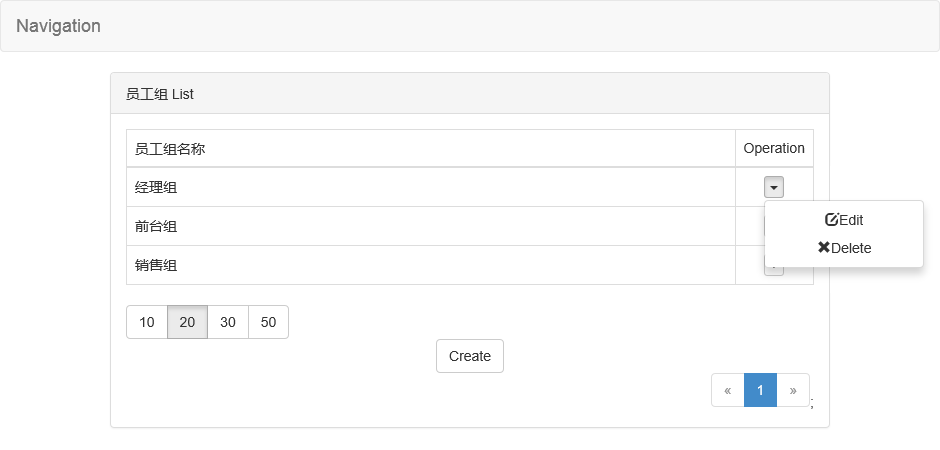
}

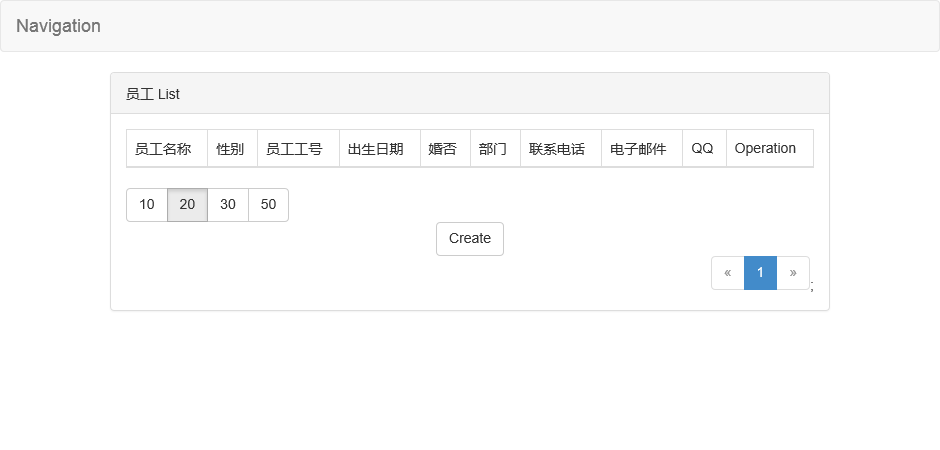
[复制代码](javascript:void(0);)

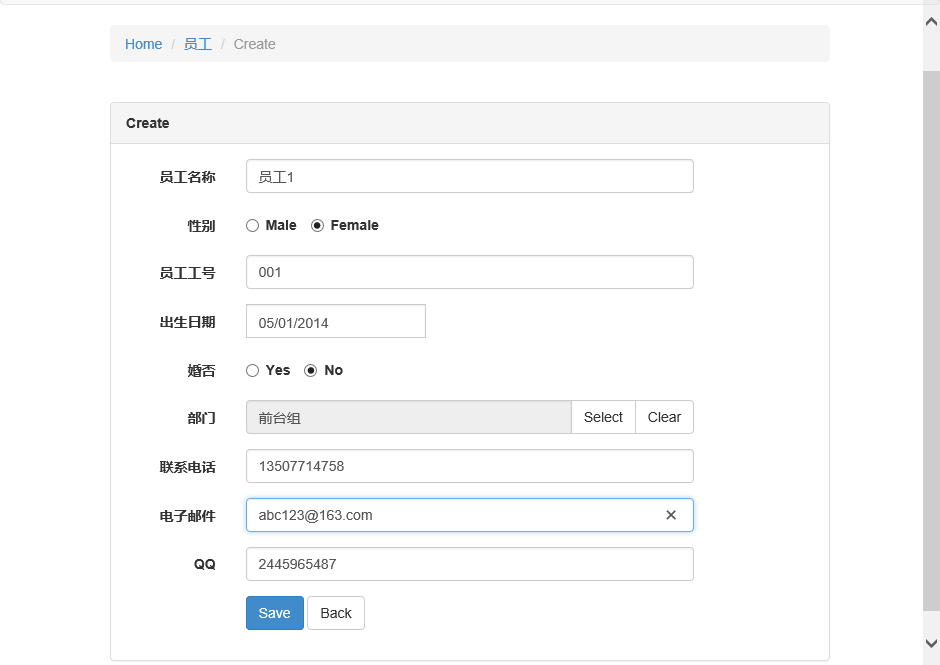
现在，您已经创建好项目了，您可以访问/Employee或/EmployeeGroup地址对实体进行编辑。

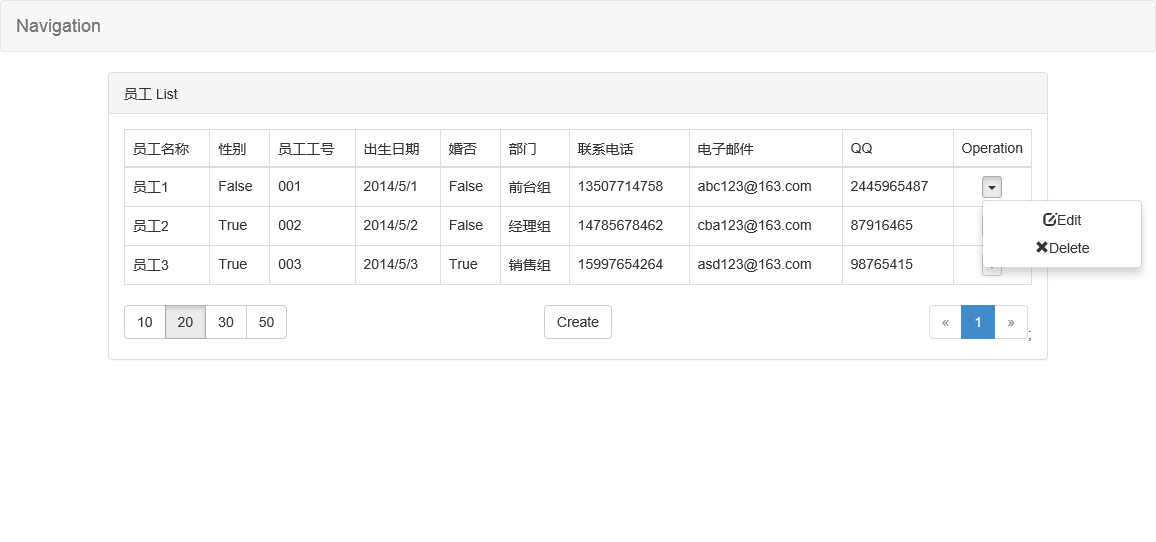












这是个最基础的示例，实际情况您需要对\_Layout.cshtml进行编辑，更有可能创建控制器实现其它功能。

# 框架详解

ALEXFW是一个Repository模式的实体框架，可以帮助你更快速的建立应用程序。



## 基础结构

* ALEXFW.Smart

整个架构基础部分

* ALEXFW.Smart.EntityFramework

用微软的EntityFramework建立起的Repository库。

* ALEXFW.Smart.Mvc网站部分

ALEXFW.Smart.Mvc专为微软Asp.Net Mvc架构设计的框架，免除增删查改系列代码工作。为适用EntityController而使用基于Unity的依赖注入。

## 建立实体

当包安装完毕后，您的实体项目会出现新的引用ALEXFW.Smart，这是建立实体需要的引用。



### IEntity接口

现在您可以开始创建您的实体。要求每个实体必须继承接口IEntity。

该接口要求该实体的主键属性名称为Index，且主键类型为Guid，该属性应该拥有KeyAttribute特性。

注意，实体必须拥有无参数构造函数，只有带参数的构造函数的实体将无法使用。

例如：

public class YourEntity : IEntity

{

public virtual DateTime CreateDate { get; set; }

[Key]

public virtual Guid Index { get; set; }

public virtual bool IsEditAllowed

{

get { return true; }

}

public virtual bool IsRemoveAllowed

{

get { return true; }

}

public void OnCreateCompleted()

{

}

public void OnEditCompleted()

{

}

private ReadOnlyCollection<ValidationResult> \_NoError = new ReadOnlyCollection<ValidationResult>(new List<ValidationResult>());

public IEnumerable<ValidationResult> Validate(ValidationContext validationContext)

{

return \_NoError;

}

}

IsEditAllowed属性通常被IEntityQueryable接口的Edit方法获取，如果为false则不允许编辑。同理，IsRemoveAllowed为false则不允许删除。OnCreateCompleted方法通常被IEntityQueryable接口的Create方法调用。OnEditCompleted方法通常被IEntityQueryable接口的Edit方法调用。您可以在Validate方法里验证您的属性，返回每一个错误。

建议，每个属性请使用virtual标签以确保其能被重写，某些Repository库可以通过重写属性实现更多功能。

### IPassword接口

该接口要求该实体拥有SetPassword与VerifyPassword方法。SetPassword方法应当实现设置密码的代码，通常该方法会加密密码然后储存到相应属性里。VerifyPassword方法应当实现验证密码的代码，通常该方法会加密密码然后对比相应储存密码的属性。

基于框架的某些功能在设置密码的时候必须要求该实体继承这个接口。

### EntityBase

为了节省劳动力，您不必为每个实体实现IEntity接口，框架提供了EntityBase基类，您的实体可以直接继承它来使用。

例如：

public class YourEntity : EntityBase

{

[Required]

public virtual string Name { get; set; }

}

EntityBase的ToString方法默认返回Index属性值。

您可以在您的实体里重写各个属性或方法。

### UserBase

该实体基类继承了EntityBase基类与IPassword接口。

该基类拥有Password与Salt属性用于存储密码。

实现的SetPassword把输入的密码进行SHA1加密，并串接随机生成6字节的Salt掩码再次进行SHA1加密，把结果保存到Password属性里。

您需要自定义其它用户信息属性。

public class Member : UserBase

{

[Required]

public virtual string Name { get; set; }

[Required]

public virtual MemberGroup Group { get; set; }

}

## 操作实体

当您的实体创建完毕后，您可以通过继承于IEntityQueryable<TEntity>接口的类进行使用。

框架已经帮您实现了基于微软EntityFramework的Repository库，[查看文档](http://comboost.wodsoft.com/zh-cn/Document/EntityFramework)。



### 实体上下文IEntityContextBuilder

一般来说，实现Repository的库通常拥有继承于IEntityContextBuilder接口的类，这个类允许你获取相应的IEntityQueryable<TEntity>实体上下文。

实例化了实体上下文构造器后，可以通过GetContext方法获取相关实体上下文。

IEntityQueryable<Member> context = builder.GetContext<Member>();

这个接口的EntityTypes属性包含了该实体上下文构造器所支持的实体类型，不包含在该数组里的实体类型一般不可从GetContext方法获取相关实体上下文。

该实体上下文构造器拥有一个Query方法，可以直接传入Sql语句与参数获取结果，但这个方法在某些Repository库可能不被支持。

IEnumerable<Member> result = builder.Query<Member>("select \* from members where name={0}", "admin");

### IEntityQueryable

该接口定义了应用于实体的操作方法。实现该接口的类并不一定只拥有该接口定义的方法，一些Repository库可能会提供更多操作实体的方法。当获取到实体的上下文后（IEntityQueryable）就可以对实体进行相关操作。

### 创建实体

创建实体请使用上下文中的Create方法，该方法会返回一个新实例化的实体，通常情况下Create方法内部在实例化实体后会调用实体的OnCreateCompleted方法。

Member item = context.Create();

item.Name = "admin";

我们推荐使用Create方法来创建实体，而不是使用new方法，Repository库需要在Create方法里执行相关代码。

### 添加实体

当使用Create方法创建实体后，该实体并没有添加到数据库。请使用Add方法来添加实体。该方法会返回布尔型结果以表示该实体是否添加成功。

bool success = context.Add(item);

如果该实体不是通过Create创建的，那么该方法可能会报错，这取决于Repository库。

如果该实体的Index属性值为默认值（00000000-0000-0000-0000-000000000000），那么该方法可能会为Index赋予新值Guid.New()，这也取决于Repository库。

您也可以使用AddRange方法批量添加实体。

一般来说，使用该方法的情况是为了效率，所以该方法通常不检测实体是否正确。请在使用该方法时确保您的实体数据正确，如果不正确可能会引发异常，这取决于Repository库。

### 编辑实体

修改实体后可以通过Edit方法保存实体到数据库。一般来说，该方法会执行实体的Validate方法，如果返回数量大于0，那么将不会添加到数据库。然后该方法会继续验证其它内容，如果验证出错，同样不会添加到数据库。该方法返回布尔型结果以表示该实体是否编辑成功。

item.Name = "admin2";

bool success = context.Edit(item);

### 删除实体

您可以通过Remove方法从数据库中删除该实体。

bool success = context.Remove(item);

### 查询Query

您可以使用Query方法获取IQueryable<TEntity>来查询数据。

Member item = context.Query().SingleOrDefault(t=> t.Name == "admin");

Member[] items = context.Query().Where(t=> t.Name.Contains("ad")).ToArray();

Query方法同样可以直接使用Sql语句与参数查询数据。

IEnumerable<Member> items = context.Query("select \* from members where name={0}, "admin");

### InParent

InParent方法可以构造一个Linq语句用来限定实体属于父级。

MemberGroup group = groupContext.Query().Single(t=> t.Name == "Admins");

Member[] items = context.InParent(context.Query(), new Guid[]{ group.Index });

或

MemberGroup group = groupContext.Query().Single(t=> t.Name == "Admins");

Member[] items = context.InParent(context.Query(), "Group", group.Index);

上下两个例子结果相同。

该例中Member需要使用ParentAttribute特性。

[Parent(typeof(MemberGroup), "Group")]

public class Member : UserBase

{

[Required]

public virtual string Name { get; set; }

[Required]

public virtual MemberGroup Group { get; set; }

}

### Count与Contains

Count方法会返回数据库中的实体数量。

Contains方法可以判断某个实体是否存在于数据库。

### 排序

OrderBy方法可以对实体进行排序。

IOrderedQueryable<Member> query1 = context.OrderBy(context.Where(t=> t.CreateDate < date));

IOrderedQueryable<Member> query2 = context.OrderBy().Where(t=> t.CreateDate < date);

排序依据DisplayColumnAttribute特性。

如果没有该特性，默认排序属性通常为CreateDate这将取决于Repository库。

[Parent(typeof(MemberGroup), "Group")]

[DisplayColumn("Name", "Name")]

public class Member : UserBase

{

[Required]

public virtual string Name { get; set; }

[Required]

public virtual MemberGroup Group { get; set; }

}

## 实体特性

这些特性用于实体。



### DisplayNameAttribute实体名称

该特性用于注明该实体的名称。

[DisplayName("会员信息")]

public class Member : UserBase

### DisplayColumnAttribute实体显示

该特性用于注明这个实体用于显示的属性、用于排序的属性。

[DisplayColumn("Name", "Name")]

public class Member : UserBase

{

[Required]

public virtual string Name { get; set; }

}

### EntityAuthenticationAttribute实体权限

该特性用于设置实体访问权限。

[EntityAuthentication(AllowAnonymous = false,

ViewRolesRequired = new string[]{ "Admin" }),

AddRolesRequired = new string[]{ "Admin" }),

EditRolesRequired = new string[]{ "Admin" }),

RemoveRolesRequired = new string[]{ "Admin" })]

public class Member : UserBase

{

[Required]

public virtual string Name { get; set; }

}

### ParentAttribute实体父级

该特性用于设置实体父级类型与属性。

[ParentAttribute(typeof(MemberGroup), "Group")]

public class Member : UserBase

{

[Required]

public virtual string Name { get; set; }

[Required]

public virtual MemberGroup Group { get; set; }

}

## 属性特性

这些特性用于属性。



### ColumnAttribute数据库类型

该特性一般用于手动设定数据库字段类型。

例如常用于设定DateTime类型的属性。

[Column(TypeName = "datetime2")]

public virtual DateTime CreateDate { get; set; }

### CustomDataTypeAttribute自定义类型

该特性用于为属性设定自定义类型，该类型不会影响数据库，只会用于Repository库。可以设定类型为CustomDataType枚举中的值，或者直接设定类型名称字符串。

[CustomDataType(CustomDataType.Email)]

public virtual string Email { get; set; }

[CustomDataType("Entity")]

publiv virtual Member Member { get; set; }

设定类型为自定义字符串后，通常需要自己实现Repository库要求的自定义类型相关代码。

如果不使用CustomDataTypeAttribute，那么框架将自动根据属性类型设置CustomDataType。

* DateTime类型默认为CustomDataType.Date。
* TimeSpan类型默认为CustomDataType.Time。
* bool类型默认为CustomDataType.Boolean。
* short、int、long类型默认为CustomDataType.Integer。
* float、double类型默认为CustomDataType.Number。
* decimal类型默认为CustomDataType.Currency。
* byte[]类型默认为CustomDataType.File。
* enum类型默认为"Enum"。
* ICollection<>类型默认为"Collection"。
* IEntity类型默认为"IEntity"。

### DisplayAttribute显示信息

该特性用于为属性设定显示信息，这几乎是每个属性必须设定的。

Name属性为名称，在列表、编辑等功能时显示。

Description属性为说明，通常用于该属性显示名称的弹出提示。

Order属性为排序，从小到大，越小显示越靠前。

Prompt属性为提示信息。

ShortName属性为简写名称，通常用于列表头显示。

[Display(Name = "姓名", Order = 0)]

public virtual string Name { get; set; }

[Display(Name = "手机号", Order = 10, ShortName = "手机")]

public virtual string Phone { get; set; }

推荐您为Order设置整十数，如果需要插入新属性，可以直接将Order设置为上下属性之间的值。

### DistinctAttribute不同内容

该特性用于设定属性值不能重复。

只能用于值类型属性。

### HideAttribute隐藏字段

该特性用于隐藏属性不被显示。

IsHiddenOnView该属性为true时，属性将不被显示于列表，默认为true。

IsHiddenOnCreate该属性为true时，属性在编辑时将不被显示，默认为true。

IsHiddenOnEdit该属性为true时，属性在编辑时将不被显示，默认为true。

IsHiddenOnDetail该属性为true时，属性在详情页将不被显示，默认为IsHiddenOnView与IsHiddenOnEdit的与运算。

[Display(Name = "手机号", Order = 10, ShortName = "手机")]

[Hide(IsHiddenOnEdit = false)]

public virtual string Phone { get; set; }

一般该特性用于隐藏属性过多的实体，防止列表太拥挤。

### KeyAttribute主键

该特性用于标识属性为主键，在属性中只用于Index。

[Key]

public virtual Guid Index { get; set; }

### MaxLengthAttribute最大长度

该特性一般用于string和byte[]类型的属性。

使用该特性后，数据库字段的长度将被设置。

[MaxLength(20)]

public virtual string Name { get; set; }

[MaxLength(100000)]

public virtual byte[] Image { get; set; }

### MinLengthAttribute最小长度

该特性一般用于string和byte[]类型的属性。

使用该特性后，在保存数据时将会验证属性值的长度是否达到要求。

[MaxLength(20)]

[MinLength(4)]

public virtual string Name { get; set; }

该特性不会影响数据库字段。

### PropertyAuthenticationAttribute权限

该特性用于设置实体访问权限。

[PropertyAuthentication(AllowAnonymous = false,

ViewRolesRequired = new string[]{ "Admin" }),

EditRolesRequired = new string[]{ "Admin" }))]

public virtual string Name { get; set; }

### RangeAttribute范围

该特性用于设定属性值的范围，一般用于数字类型属性。

[Range(0, 24)]

public virtual int Hours { get; set; }

[Range(0, 60)]

public virtual int Minutes { get; set; }

[Range(0, 60)]

public virtual int Seconds { get; set; }

该特性不会影响数据库字段。

### RegularExpressionAttribute正则表达式

该特性用于验证属性值，一般用于字符串类型属性。

通常用于验证电子邮箱、IP地址、URL等类型。

[RegularExpression(@"^([/w-/.]+)@((/[[0-9]{1,3}/.[0-9] {1,3}/.[0-9]{1,3}/.)|(([/w-]+/.)+))([a-zA-Z]{2,4}|[0-9]{1,3})(/)?]$")]

publiv virtual string Email { get; set; }

### RequiredAttribute必填项

该特性用于设定属性值不能为null。

通常用于名称、父级属性。

[Required]

public virtual string Name { get; set; }

### SearchableAttribute可搜索

该特性用于标注可以通过搜索该属性搜索到实体。如果该实体没有任何属性使用该特性，那么该实体将不可搜索，不过这取决于Repository库。

[Searchable]

public virtual string Name { get; set; }

[Searchable]

public virtual DateTime CreateDate { get; set; }

### ValueFilterAttribute值筛选（级联）

该特性用于标注属性的值只能从通过特定程序筛选出可选值。该构造函数的valueFilter参数用于传入继承于ValueFilter的类型，如果不继承则会抛出异常。dependencyProperty参数用于传入筛选依赖的属性名称，可以为空。

[ValueFilter(typeof(LocationProvider), null)]

public virtual string Province { get; set; }

[ValueFilter(typeof(LocationProvider), "Province")]

public virtual string City { get; set; }

您需要实现您自己的ValueFilter，用于返回筛选出的可选值。

示例：

public class LocationProvider : ValueFilter

{

public override NameValueCollection GetValues(string dependencyProperty, string dependencyValue)

{

NameValueCollection collection = new NameValueCollection();

switch (dependencyProperty)

{

case null:

collection.Add("北京", "北京");

collection.Add("广东", "广东");

collection.Add("广西", "广西");

break;

case "Province":

switch(dependencyValue)

{

case "北京":

collection.Add("北京市", "北京市");

break;

case "广东":

collection.Add("广州市", "广州市");

collection.Add("佛山市", "佛山市");

collection.Add("深圳市", "深圳市");

break;

case "广西":

collection.Add("南宁市", "南宁市");

collection.Add("柳州市", "柳州市");

collection.Add("桂林市", "桂林市");

break;

default:

throw new NotSupportedException();

}

break;

default:

throw new NotSupportedException();

}

return collection;

}

}

## 重写

通常实体都会继承于EntityBase而不会只继承IEntity接口而自己实现它。有些实体继承EntityBase后需要重写基类而实现某些功能。



### 创建时

OnCreateCompleted方法通常于实体创建时调用，这取决于Repository库。

您可以重写该方法来达到您需要的功能。

通常可以在该方法内初始化属性值。

public class Member : UserBase

{

[Required]

public virtual string Name { get; set; }

[Required]

public virtual decimal Discount { get; set; }

public override void OnCreateCompleted()

{

Discount = 1;

}

}

### 编辑时

OnEditCompleted方法通常于实体编辑时调用，这取决于Repository库。您可以重写该方法来达到您需要的功能。通常可以在该方法内计算隐藏属性的值。

public class Item : EntityBase

{

[Required]

public virtual decimal Amount { get; set; }

[Required]

public virtual decimal Price { get; set; }

[Hide]

[Required]

public virtual decimal TotalPrice { get; set; }

public override void OnEditCompleted()

{

TotalPrice = Amount \* Price;

}

}

### 是否可删除

IsRemoveAllowed属性通常于实体删除时获取，这取决于Repository库。您可以重写该属性来达到您需要的功能。通常可以在该方法内判断子项是否有值。

public class MemberGroup : EntityBase

{

[Required]

public virtual string Name { get; set; }

[Hide]

public virtual ICollection<Member> Members { get; set; }

public override bool IsRemoveAllowed

{

get { return Members.Count == 0; }

}

}

### 是否可编辑

IsEditAllowed属性通常于实体编辑时获取，这取决于Repository库。您可以重写该属性来达到您需要的功能。通常可以在该方法内判断实体是否能编辑。

public class Member : UserBase

{

[Required]

public virtual string Name { get; set; }

[Required]

public virtual bool Lock { get; set; }

public override bool IsEditAllowed

{

get { return !Lock; }

}

}

### 转换字符串

ToString方法用于很多个地方，例如实体显示于列表项中时。您可以重写该属性来达到您需要显示的值。

public class MemberGroup : EntityBase

{

[Required]

public virtual string Name { get; set; }

public override string ToString()

{

return Name;

}

}

基类里该方法会读取DisplayColumnAttribute特性的Name值，查找到Name值名称的属性并返回它的值的ToString方法。

所以通常情况下您只需要为实体设置DisplayColumnAttribute特性即可。

[DisplayColumn("Name")]

public class MemberGroup : EntityBase

{

[Required]

public virtual string Name { get; set; }

}

当然您可以根据需要自由组合返回需要显示的值。

## 使用上下文

当创建完DbContext子类后，即可使用Repository库。



### EntityContextBuilder

EntityContextBuilder的构造函数需要传入DbContext类型的数据库上下文。

实例化后即可通过GetContext方法获取实体操作上下文。

EntityContextBuilder提供DbContext属性给开发者获取该上下文构造器使用的数据库上下文。

EntityContextBuilder builder = new EntityContextBuilder(new DataContext());

### EntityQueryable

EntityQueryable可以通过EntityContextBuilder的GetContext方法获取。

EntityQueryable<Member> context = builder.GetContext<Member>();

也可以手动实例化。

EntityQueryable<Member> context = new EntityQueryable<Member>(new DataContext());

EntityQueryable除了标准IEntityQueryable接口功能外还额外多出一个Reload方法，该方法可以重新载入该实体的数据库数据。EntityQueryable内的方法全部是带有virtual标签，是可重写的。您可以基于EntityQueryable实现你自己的Repository库。EntityQueryable提供了protected级别的只读属性：DbContext、DbSet、EntityMetadata。

## MVC



### 模板导入说明

项目的~/Views/Shared文件夹公共视图。这些视图是框架需要的，这些视图的使用的CSS框架为Bootstrap。~/Scripts文件夹jstree脚本文件，用于显示实体导航。~/Views/Web.config文件，使得Razor视图直接支持命名空间ystem.ComponentModel。

<configuration>

<system.web.webPages.razor>

<pages>

<namespaces>

<add namespace="System.ComponentModel" />

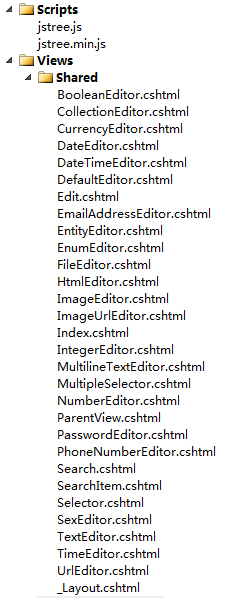
</namespaces>

</pages>

</system.web.webPages.razor>

</configuration>

以下是新增文件列表。



修改\_Layout.cshtml母版页视图文件。母版页视图带有导航栏需要您自己修改。

### 无形参控制器 EntityController

实体项目会出现新的引用：ALEXFW.Smart.Mvc。

如果您的控制器只需要用实体上下文而不需要增删查改，那么您可以把您的控制器继承EntityController。您需要满足基类EntityController的构造参数IEntityContextBuilder。

public YourController(IEntityContextBuilder builder) : base(builder)

{

}

由于构造函数需要参数，建议您使用依赖注入。框架已经为您实现了基于Unity的依赖注入，[点击查看](http://comboost.wodsoft.com/zh-cn/Document/MvcUnity)。

### EntityBuilder属性

继承EntityBuilder后您可以通过EntityBuilder属性获取实体上下文构造器。

获取实体上下文后就可以使用它访问数据库。

public ActionResult Index()

{

IEntityQueryable<MemberGroup> context = EntityBuilder.GetContext<MemberGroup>();

MemberGroup[] items = context.OrderBy().ToArray();

return View(items);

}

### 实体控制器 EntityController<TEntity>

实体控制器使用带形参的EntityController，使用方法和不带形参时一样需要调用基类的构造函数。

带形参的控制器已经帮您实现了增、删、查、改等必要方法，您只需要编写您需要的视图即可。

通常情况下，这些方法的视图文件都放置于Shared共享视图文件夹。已经为您制作了一套Boorstrap3风格的视图页面，[点击查看](http://comboost.wodsoft.com/zh-cn/Document/MvcBootstrap)。

### 列表

Index页面用于显示实体列表。

传入参数

* int: page

列表当前页数，默认为1。如果小于1返回400页面。

* int: size

列表显示实体个数，默认为20。如果小于1返回400页面。

* string: parentpath

导航路径，默认为null。如果路径错误返回400页面。

* Guid?: parentid

导航父级ID，默认为null。

* bool: search

是否为搜索，默认为false。

当search为true时，控制器会获取以Search.开头的客户端Query参数。获取后进行处理得到搜索结果。

### 视图模型

该方法返回的视图模型为IEntityViewModel<TEntity>，页面视图一般不需要知道泛型类型，只需要使用IEntityViewModel类型模型即可。

该视图模型包含属性：

* Buttons

页面自定义按钮。

* CurrentSize

当前显示实体个数。

* CurrentPage

当前页数。

* Headers

列表头。

* Items

当前页面实体数组。

* Metadata

实体元数据。

* Parent

实体导航模型。

* SearchItem

搜索内容项。

* TotalPage

列表总页数。

### 重写

您可以从基类获得ActionResult，并从Model属性获取IEntityViewModel<TEntity>视图模型。获取视图模型后您可以修改该模型达到您想要的功能。

public override ActionResult Index(int page = 1, int size = 20, string parentpath = null, Guid? parentid = null, bool search = false)

{

ViewResult result = (ViewResult)base.Index(page, size, parentpath, parentid, search);

EntityViewModel<Member> model = (EntityViewModel<Member>)result.Model;

MvcViewButton button = new MvcViewButton();

button.StaticLink = "";

model.Buttons = new EntityViewButton[] { button };

return result;

}

当然，您可以完全编写自己的代码，不必返回EntityViewModel<TEntity>视图模型，可以创建自己的视图。

### 新增

Create页面用于显示实体列表。

传入参数

* Guid?: parent

父级ID。如果该实体拥有ParentAttribute特性，传入此参数将设定新实体的父级为该ID的实体。

视图模型

该方法返回的视图模型为EntityEditModel<TEntity>，页面视图一般不需要知道泛型类型，只需要使用EntityEditModel类型模型即可。

该视图模型包含属性：

* Item

编辑的实体。

* Properties

可以编辑的属性。

* Metadata

实体元数据。

该方法使用Edit视图页面。

重写

一般情况下不会需要重写该方法。

### 编辑

Edit页面用于显示实体列表。

传入参数

* Guid: id

实体ID。如果该实体不存在，则返回404页面。

视图模型

该方法返回的视图模型为EntityEditModel<TEntity>，页面视图一般不需要知道泛型类型，只需要使用EntityEditModel类型模型即可。

该视图模型包含属性：

* Item

编辑的实体。

* Properties

可以编辑的属性。

* Metadata

实体元数据。

该方法于Create页面使用同样的视图，需要在视图页面区分实体Index是否为默认值以区别是新建还是编辑。

重写

一般情况下不会需要重写该方法。

### 更新

Update方法用于更新实体数据。

传入参数

* Guid: id

实体ID。如果该实体不存在，则返回404页面。

该方法会根据实体元数据搜索Request.Form内容，自动修改实体并保存。

视图

该方法为Ajax提交方法，只需要返回HttpStatusCodeResult即可。

返回200状态即成功，返回400状态即失败，返回内容为错误内容。

重写

该方法工作量繁多，如需重写请摒弃基类方法，根据实体内容重新编写。

### 删除

该方法为Ajax删除方法，只需要返回HttpStatusCodeResult即可。

传入参数

* Guid: id

实体ID。如果该实体不存在，则返回404页面。

视图

该方法返回的视图为HttpStatusCodeResult。

返回200状态即成功，返回400状态即失败，返回内容为错误内容。

重写

一般情况下需要在删除前处理数据时才会重写该方法。

### 详细

Detail页面用于显示实体完整内容。

传入参数

* Guid: id

实体ID。如果该实体不存在，则返回404页面。

视图模型

该方法返回的视图模型为EntityEditModel<TEntity>，页面视图一般不需要知道泛型类型，只需要使用EntityEditModel类型模型即可。

该视图模型包含属性：

* Item

编辑的实体。

* Properties

显示的属性。

* Metadata

实体元数据。

重写

一般情况下不会需要重写该方法。

### 搜索

Search页面用于填写实体搜索内容。

视图模型

该方法返回的视图模型为EntitySearchModel<TEntity>，页面视图一般不需要知道泛型类型，只需要使用EntitySearchModel类型模型即可。

该视图模型包含属性：

* Properties

可以搜索的属性。

* Metadata

实体元数据。

重写

一般情况下不会需要重写该方法。

### 单项选择器

Selector页面用于选择单一实体。

该页面通常用于实体编辑时的实体编辑器。

传入参数

* int: page

列表当前页数，默认为1。如果小于1返回400页面。

* string: parentpath

导航路径，默认为null。如果路径错误返回400页面。

* Guid?: parentid

导航父级ID，默认为null。

固定每一页只显示10个实体。

视图模型

该方法返回的视图模型为EntityViewModel<TEntity>，页面视图一般不需要知道泛型类型，只需要使用EntityViewModel类型模型即可。

该视图模型包含属性：

* Buttons

页面自定义按钮。

* CurrentSize

当前显示实体个数。

* CurrentPage

当前页数。

* Headers

列表头。

* Items

当前页面实体数组。

* Metadata

实体元数据。

* Parent

实体导航模型。

* SearchItem

搜索内容项。

* TotalPage

列表总页数。

重写

一般情况下不会需要重写该方法。

### 多项选择器

MultipleSelector页面用于选择多个实体。

该页面通常用于实体编辑时的实体编辑器。

传入参数

* int: page

列表当前页数，默认为1。如果小于1返回400页面。

* string: parentpath

导航路径，默认为null。如果路径错误返回400页面。

* Guid?: parentid

导航父级ID，默认为null。

固定每一页只显示10个实体。

视图模型

该方法返回的视图模型为EntityViewModel<TEntity>，页面视图一般不需要知道泛型类型，只需要使用EntityViewModel类型模型即可。

该视图模型包含属性：

* Buttons

页面自定义按钮。

* CurrentSize

当前显示实体个数。

* CurrentPage

当前页数。

* Headers

列表头。

* Items

当前页面实体数组。

* Metadata

实体元数据。

* Parent

实体导航模型。

* SearchItem

搜索内容项。

* TotalPage

列表总页数。

重写

一般情况下不会需要重写该方法。

### 保存文件

如果您的属性设置了CustomDataType为File，那么您必须继承IFileController接口。

该接口需要实现SaveFileToProperty方法。

实体控制器在Update方法里会调用该方法，传入要修改的实体与属性，并且提供用户上传的HttpPostedFileBase。

您需要做的就是把该文件保存到您需要的地方并且给该属性赋值。

public void SaveFileToProperty(TEntity entity, PropertyMetadata metadata, HttpPostedFileBase file)

{

if (metadata.Property.Name == "Image")

{

Guid id = Guid.NewGuid();

file.SaveAs(Server.MapPath("~/uploads/" + id.ToString()));

entity.Image = id.ToString();

}

}

### 显示图片

如果您的属性设置了CustomDataType为Image，那么您必须继承IImageController接口。

该接口需要实现ImageToProperty方法。

视图会在<img>标签的src属性使用实体控制器的ImageToProperty方法来显示图片。

方法传入实体的Id与要显示的图片的属性名称。

您需要做的就是从数据库获取该实体，并返回图片文件结果。

FileResult ImageToProperty(Guid id, string property)

{

var item = context.GetEntity(id);

return File(Server.MapPath("~/uploads/" + item.Image), "image/jpeg");

}

## 编辑器

每一个编辑器的视图模型都为MvcEditorModel。

这个模型拥有Metadata属性用于获取该属性类型元数据。

Value属性用于获取该实体的属性的值。

编辑器的命名规则为CustomDataType类型名称+Editor.cshtml。



### 默认

每一个CustomDataType枚举对应一个编辑器，它们都应存放于Shared共享视图文件夹里。

BooleanEditor.cshtml

CurrencyEditor.cshtml

DateEditor.cshtml

DateTimeEditor.cshtml

DefaultEditor.cshtml

...

框架已经为您制作了一套Boorstrap3风格的编辑器，[点击查看](http://comboost.wodsoft.com/zh-cn/Document/MvcBootstrap)。

### 自定义编辑器

当CustomDataTypeAttribute为自定义字符串时，则编辑器的名称也为该字符串+Editor.cshtml。

自定义名称不能与CustomDataType枚举名称相同。

自定义名称同时也不能为以下三个保留名称：

* Collection
* Entity
* Enum

您可以为您的属性设计自定义编辑器用于更好的UI体验。

例如下面的内容是一个自定义评分编辑器。

RateEditor.cshtml

@model MvcEditorModel

@{

byte value = (byte)@Model.Value;

}

<div id="@Model.Metadata.Property.Name">

<p class="form-control-static">

@for (int i = 1; i <= value; i++)

{

<i id="@(Model.Metadata.Property.Name + "\_" + i)" class="glyphicon glyphicon-star" style="cursor: pointer;" data-val="@i"></i>

}

@for (int i = value + 1; i < 6; i++)

{

<i id="@(Model.Metadata.Property.Name + "\_" + i)" class="glyphicon glyphicon-star-empty" style="cursor: pointer;" data-val="@i"></i>

}

</p>

<input type="hidden" id="@(Model.Metadata.Property.Name)\_Value" name="@Model.Metadata.Property.Name" value="@Model.Value" />

</div>

<script type="text/javascript">

$('#@Model.Metadata.Property.Name').find('i').click(function () {

var element = $(this);

var value = new Number(element.attr('data-val'));

$('#@(Model.Metadata.Property.Name)\_Value').val(value);

for (var i = 1 ; i <= value ; i++) {

$('#@(Model.Metadata.Property.Name)\_' + i).removeClass('glyphicon-star-empty');

$('#@(Model.Metadata.Property.Name)\_' + i).addClass('glyphicon-star');

}

for (var i = value + 1 ; i < 6 ; i++) {

$('#@(Model.Metadata.Property.Name)\_' + i).removeClass('glyphicon-star');

$('#@(Model.Metadata.Property.Name)\_' + i).addClass('glyphicon-star-empty');

}

});

</script>

在服务器启动时，需要为该自定义类型添加转换器。

我们将该自定义类型设置为byte类型时，需要添加ByteConverter。

EntityValueConverter.AddConverter("Rate", new ByteConverter());

http://comboost.wodsoft.com/Content/Images/RateEditor.png

编辑器需要提交到服务器的<input>的name属性必须为该属性的名称。

## 权限系统

您需要在web.config里以下节点配置框架权限模块。

<configuration>

<system.webServer>

<modules>

<remove name="FormsAuthentication" />

<add name="ALEXFWAuthentication" type="System.Web.Security.ALEXFWAuthenticationModule"/>

</modules>

</system.webServer>

</configuration>

移除FormsAuthentication模块以确保ALEXFWAuthentication的正确运行。



### 权限实体

在使用权限系统前，您需要定义一个带权限判断的实体。

该实体继承IRoleEntity接口。

该接口要求您的实体需要实现一个权限判断功能，传入一个string类型的权限，返回该实体是否具有权限。

public bool IsInRole(string role)

{

return true;

}

框架提供的UserBase实体基类已经继承IRoleEntity接口，并且永远返回true。

### 实体解释器

现在您需要实现一个实体解释委托。

在App\_Start文件夹里修改UnityControllerFactory.cs。

并在该类里添加如下方法。

private IRoleEntity EntityResolve(Type entityType, string username)

{

IEntityContextBuilder builder = \_container.Resolve<IEntityContextBuilder>();

dynamic context = builder.GetContext(entityType);

return context.GetEntity(new Guid(username));

}

然后在构造函数里添加以下代码。

System.Web.Security.ALEXFWPrincipal.Resolve = EntityResolve;

这样就把权限系统获取权限实体的委托实现了。

### 修改路由

现在您需要修改一下路由代码。

一般情况下，在App\_Start里的RouteConfig.cs里直接修改路由代码。

您只需要给您的路由代码MapRoute方法加上泛型参数即可指定该路由下的权限判断由某权限实体决定。

该泛型必须继承IRoleEntity接口。

routes.MapRoute<TRoleEntity>(

name: "Default",

url: "{controller}/{action}/{id}",

defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }

);

如果是区域的路由，那么修改您的区域AreaRegistration代码。

默认Admin区域代码。

public override void RegisterArea(AreaRegistrationContext context)

{

context.MapRoute(

"Admin\_default",

"Admin/{controller}/{action}/{id}",

new { action = "Index", id = UrlParameter.Optional }

);

}

修改后。

public override void RegisterArea(AreaRegistrationContext context)

{

context.Routes.MapRoute<TRoleEntity>(

"Admin\_default",

"Admin/{controller}/{action}/{id}",

new { action = "Index", id = UrlParameter.Optional }

).DataTokens["area"] = AreaName;

}

注意映射后要手动设置DataTokens的area值为区域名称。

### 权限的使用

在以上工作都完成后，您还需要自己编写用户登录。

一般在您的登录方法使用FormsAuthentication.SetAuthCookie方法设置权限实体Index。

[HttpPost]

public ActionResult Login(string username, string password)

{

if (string.IsNullOrEmpty(username))

{

Response.StatusCode = 400;

return Content("用户名不能为空！");

}

if (string.IsNullOrEmpty(password))

{

Response.StatusCode = 400;

return Content("密码不能为空！");

}

var context = EntityBuilder.GetContext<User>();

User user = context.Query().SingleOrDefault(c => c.Name == username);

if (user == null)

{

Response.StatusCode = 400;

return Content("该用户不存在！");

}

if (!user.VerifyPassword(password))

{

Response.StatusCode = 400;

return Content("密码错误！");

}

FormsAuthentication.SetAuthCookie(user.Index.ToString(), false);

return new HttpStatusCodeResult(200);

}

ALEXFWAuthentication.SignIn方法设置的用户名会被传入System.Web.Security.ALEXFWPrincipal.Resolve委托，一般情况下使用实体Index作为用户名。

登录后，所有的User.IsInRole权限判断方法将由该User实体进行判断。

### 登录与登出

如果您使用Forms验证，那么使用ALEXFWAuthentication.SignIn方法设置用户登录Cookie。

使用ALEXFWAuthentication.SignOut方法注销用户登录。

为您注册的路由Route添加Token数据可以区分用户登录区域。

Route myRoute = routes.MapRoute(

name: "Default",

url: "{controller}/{action}/{id}",

defaults: new { controller = "Home", action = "Index", id = UrlParameter.Optional }

);

myRoute.DataTokens.Add("authArea","LoginArea");

不同的authArea将不能互相登录。