原文地址：https://docs.abp.io/zh-Hans/abp/latest/Domain-Driven-Design-Implementation-Guide

**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.

注：如果你认为你永远都不会离开 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()

**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.

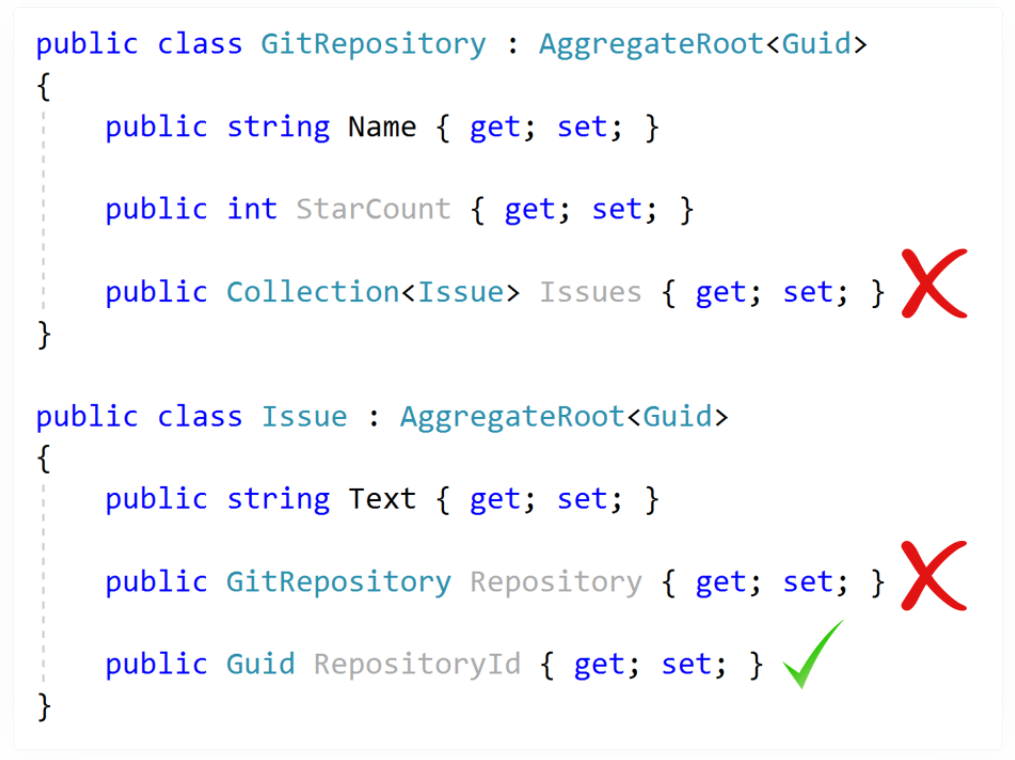
DDD忽略领域对象的数据展示,这并不意味着它们并不重要,如果应用程序没有精美的看板和报表,谁会愿意用呢?但是报表是另外一个讨论话题,你可以通过使用SQL Server报表功能或ElasticSearch来提供数据展示,又或者使用优化后的SQL查询语句,创建数据库索引或存储过程.唯一的原则是不要将这些内容混入领域的业务逻辑中.

**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

**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



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.

**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，UserRole这个值对象是分给User聚合还是Role聚合呢？



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，

注：聚合的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**

1. 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（永远不要使用 Guid.NewGuid() 创建Guid，因为他生成的Id是不连续的，在你插入的时候会存在性能问题）
2. 聚合中的实体(非聚合根)可以使用联合主键.

**聚合根 / 实体的构造函数**

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：

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 \*/ }

}}

**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.

我们建议:

* 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.

**业务逻辑与实体异常**

对实体进行验证，或执行业务逻辑时，如果出现逻辑错误（如用户的密码不符合规范啊，用户的地址填写错误啊），通常需要抛出异常

**实体中业务逻辑依赖外部服务时**

有时候,复杂的业务逻辑会查询数据库或使用依赖注入中的其它服务,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（注：会使实体变的复杂）
* 定义一个领域服务（推荐）

方式1示例

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;

}}

**仓储**

仓储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.

**规约（规则，规格）**

规约是一种非活动问题的业务规则，也就是判断实体是否满足某个规则，当出现这些业务规则时，我们应该使用规约

如下是一套规则示例

* 为关闭的
* 没有关联到任何人
* 创建时间大于30天
* 最近30天没有评论

如下我们可以新建一个规约类

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);

}

}

在实体中我们可以使用规约判断实体是否满足该规约

public class Issue : AggregateRoot<Guid>, IHasCreationTime{

//...

public bool IsInActive()

{

return new InActiveIssueSpecification().IsSatisfiedBy(this);

}

}

在仓库中我们可以使用该规约过滤实体

public async Task<List<Issue>> GetIssuesAsync(ISpecification<Issue> spec)

{

return await DbSet

.Where(spec.ToExpression())

.ToListAsync();

}

**领域服务**

领域服务主要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.

注：我们建议使用 Manager 后缀来命名领域服务.

**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.

**DTO通用原则和最佳实践**

* DTO应该是可被序列化的
* 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.

**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，不要在输入DTO中定义不使用的属性.

**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
* 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

注：这里本人具有不同的观点，和输入Dto一样，每个输出方法都具有一个输出Dto，这个Dto只能在该方法使用，而这个Dto可以引用其他通用的Dto

**对象映射到对象**

* 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.

**应用服务示例**

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)

    {

        // 创建发布实体

        var issue = new Issue(

            GuidGenerator.Create(),

            input.RepositoryId,

            input.Title,

            input.Text

        );

        // 调用领域层

        if (input.AssignedUserId.HasValue)

        {

            var user = await \_userRepository.GetAsync(input.AssignedUserId.Value);

            // 调用领域层方法，因为这里涉及2个聚合，所以调用的是领域服务

            await \_issueManager.AssignToAsync(issue, user);

        }

        // 更新聚合的改变

        await \_issueRepository.InsertAsync(issue);

        // 返回Dto

        return ObjectMapper.Map<Issue, IssueDto>(issue);

    }

}

**在创建实体时执行领域规则**

如果在创建实体时需要执行领域逻辑，那么创建过程应该放在领域层

// 发布聚合

public class Issue : AggregateRoot<Guid>

{

    // 将访问性改为 internal

    internal Issue(

        Guid id,

        Guid repositoryId,

        string title,

        string text = null

        ) : base(id)

    {

       // ...

    }

    //...

}

// 领域服务

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)

    {

        // 要做标题是否已存在

        if (await \_issueRepository.AnyAsync(i => i.Title == title))

        {

            throw new BusinessException("IssueTracking:IssueWithSameTitleExists");

        }

        return new Issue(

            GuidGenerator.Create(),

            repositoryId,

            title,

            text

        );

    }

}

**领域逻辑示例**

如下代码示例我们的领域逻辑代码划分是否正确

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,授权应该在应用层处理.
* 错误：它记录了日志,包括当前用户的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,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.

**应用服务逻辑示例**

如下代码示例我们的应用服务代码划分是否正确

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的工作单元系统可以使得此工作自动进行(甚至无需[UnitOfWork]注解).
* 正确: 授权应该在应用层处理.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?

* 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.