Articles Tutorials

Filter...

latest (v7.1)

ABP Framework

Overall

- Introduction
- Tutorials & Articles
- NLayer Architecture
- Module System
- Startup Configuration
- Multi-Tenancy
- OWIN Integration
- Debugging
- API Reference

Common Structures

- Dependency Injection
- Session
- Caching
- Logging
- Setting Management
- Timing
- Object To Object Mapping (and AutoMapper Integration)
- Email Sending (and MailKit Integration)

Domain Layer

- Entities
- Multi-Lingual Entities
- Value Objects
- Repositories
- Domain Services
- Specifications
- Unit Of Work
- Domain Events (EventBus)
- Data Filters
- Dynamic Parameter System
- Object Comparators

Application Layer

- Application Services
- Data Transfer Objects
- Validating Data Transfer Objects

Edit on GitHub

In this document

- Introduction
- IDomainService Interface and DomainService Class
- Example
  - Creating an Interface
  - Service Implementation
  - Using the Domain Service from an Application Service
- Some Discussions
  - Why not use only the Application Services?
  - How do we force to use of the Domain Service?

Introduction

Domain Services (or just Services in DDD) is used to perform domain operations and business rules. In his DDD book, Eric Evans describes a good Service in three characteristics:

1. The **operation** relates to a **domain concept** that is not a natural part of an Entity or Value Object.
2. The **interface** is defined in terms of other elements of the **domain model**.
3. The operation is **stateless**.

Unlike Application Services which get/return Data Transfer Objects, a Domain Service gets/returns **domain objects** (like entities or value types).

A Domain Service can be used by Application Services and other Domain Services, but not directly by the presentation layer (application services are for that).

IDomainService Interface and DomainService Class

ASP.NET Boilerplate defines the **IDomainService interface** that is implemented by all domain services conventionally. When it's implemented, the domain service is **automatically registered** to the Dependency Injection system as **transient**.

A domain service can optionally inherit from the **DomainService class**. With it, you can use the power of some inherited properties for logging, localization and so on...

Even if you do not inherit, it can be injected if you need it.

Example

Assume that we have a task management system and we have some business rules while assigning a task to a person.

Creating an Interface

First, we define an interface for the service (not required, but good practice):

Copy

Authorization
Feature Management
Audit Logging
Entity History
Distributed Service Layer
ASP.NET Web API
Web API Controllers
Dynamic Web API Layer
OData Integration

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

As you can see, the **TaskManager** service works with domain objects: a **Task** and a **Person**. There are some conventions when naming domain services. It can be TaskManager, TaskService or TaskDomainService...

Service Implementation

Let's see the implementation:

Copy

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

    private readonly ITaskRepository _taskRepository;

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

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

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

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

        task.AssignedPersonId = person.Id;
    }

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

We have two business rules here:

- A task should be in an **Active state** in order for it to be assigned to a new Person.
- A person can have a **maximum of 3** active tasks.

Wondering why we throw an **ApplicationException** for the first check and **UserFriendlyException** for the second check (see exception handling)? This is not related to domain services at all. This is just an example, it's completely up to you. The user interface must check a task's state and should not allow us to assign it to a person. This is an application-level error and we may want to hide it from user.

The second exception is harder to check by the UI so we will show a readable error message to the user.

For example:

Using the Domain Service from an Application Service

Now, let's see how to use TaskManager from an application service:

Copy

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

    public TaskAppService(IRepository<Task, long> taskRepository, IRepository<Person>
personRepository, ITaskManager taskManager)
    {
        _taskRepository = taskRepository;
        _personRepository = personRepository;
        _taskManager = taskManager;
    }

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

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

The Task **Application Service** uses a given **DTO** (input), then uses **repositories** to retrieve that related **task** and **person**. Finally, it passes them to the **Task Manager** (the domain service).

Some Discussions

Based on the example above, you may have some questions.

Why not use only the Application Services?

You may wonder why the application service itself does not implenent the logic contained in the domain service.

We can simply say that it's not an application service task. Because it's not a **use-case**, instead, it's a **business operation**, we may end up using the same 'assign a task to a user' domain logic in a different use-case. Say that we have **another screen** to somehow update the task. This updating can include assigning the task to another person. We can use the same domain logic there. We may also have **2 different UIs** (one mobile application and one web application) that share the same domain or we may have a web API for remote clients that includes a task-assigning operation.

If your domain is simple, will only have one UI, and assigning a task to a person can be done at just a single point, then you may consider skipping domain services and implementing the logic in your application service. This is not the best practice for DDD, but ASP.NET Boilerplate does not force you to use such a design.

How do we force to use of the Domain Service?

You can see that the application service could simply do the following:

Copy

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

The developer writing the application service may not know there is a **TaskManager** and can directly set a given **PersonId** to a task's **AssignedPersonId**. So, how do we **prevent** this? There are many discussions in DDD based on this and there are some commonly used patterns. We will not delve into this too deeply, but we will provide a simple way of doing it.

We can change **Task** entity as shown below:

Copy

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

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

    public void AssignToPerson(Person person, ITaskPolicy taskPolicy)
    {
        taskPolicy.CheckIfCanAssignTaskToPerson(this, person);
        AssignedPersonId = person.Id;
    }
}
```

We changed the setter of **AssignedPersonId** as **protected**. It can not be changed outside of this Task entity class. We added an **AssignToPerson** method that takes a person and a task policy. The **CheckIfCanAssignTaskToPerson** method checks if it's a valid assignment and throws a proper exception if not (it's implementation is not important here). The application service method will look like this:

Copy

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

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

We injected ITaskPolicy as _taskPolicy and passed it to the AssignToPerson method. Now there is no second way of assigning a task to a person. We will have to use AssignToPerson and we can therefore not skip the business rules.