In this document

Specifications

Specification Pattern is used to define **named, reusable, combinable and testable filters** for entities and other business objects.

> A Specification is a part of the Domain Layer.

Installation

> This package is **already installed** when you use the startup templates. So, most of the times you don't need to manually install it.

Install the Volo.Abp.Specifications package to your project. You can use the ABP CLI *add-package* command in a command line terminal when the current folder is the root folder of your project (`.csproj`):

```
abp add-package Volo.Abp.Specifications
```

Defining the Specifications

Assume that you've a Customer entity as defined below:

```
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; }
    }
}
```

You can create a new Specification class derived from the `Specification<Customer>` .

Example: A specification to select the customers with 18+ age:

Share on :

In this document

```
using System;
using System.Linq.Expressions;
using Volo.Abp.Specifications;

namespace MyProject
{
    public class Age18PlusCustomerSpecification : Speci
    {
        public override Expression<Func<Customer, bool>
        {
            return c => c.Age >= 18;
        }
    }
}
```

You simply define a lambda Expression to define a specification.

> Instead, you can directly implement the `ISpecification<T>` interface, but the `Specification<T>` base class much simplifies it.

Using the Specifications

There are two common use cases of the specifications.

IsSatisfiedBy

`IsSatisfiedBy` method can be used to check if a single object satisfies the specification.

Example: Throw exception if the customer doesn't satisfy the age specification

```
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().I
            {
                throw new Exception(
                    "This customer doesn't satisfy the
                );
            }

            //TODO...
        }
    }
}
```

Share on :

In this document

ToExpression & Repositories

`ToExpression()` method can be used to use the specification as Expression. In this way, you can use a specification to **filter entities while querying from the database**.

```
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, ITran
    {
        private readonly IRepository<Customer, Guid> _c

        public CustomerManager(IRepository<Customer, Gu
        {
            _customerRepository = customerRepository;
        }

        public async Task<List<Customer>> GetCustomersC
        {
            var query = _customerRepository.Where(
                new Age18PlusCustomerSpecification().To
            );

            return await AsyncExecuter.ToListAsync(quer
        }
    }
}
```

> Specifications are correctly translated to SQL/Database queries and executed efficiently in the DBMS side. While it is not related to the Specifications, see the Repositories document if you want to know more about the `AsyncExecuter`.

Actually, using the `ToExpression()` method is not necessary since the specifications are automatically casted to Expressions. This would also work:

```
var query = _customerRepository.Where(
    new Age18PlusCustomerSpecification()
);
```

Composing the Specifications

One powerful feature of the specifications is that they are composable with `And`, `Or`, `Not` and `AndNot` extension methods.

Assume that you have another specification as defined below:

```
using System;
using System.Linq.Expressions;
using Volo.Abp.Specifications;

namespace MyProject
{
    public class PremiumCustomerSpecification : Specifi
    {
        public override Expression<Func<Customer, bool>
        {
            return (customer) => (customer.Balance >= 1
        }
    }
}
```

You can combine the `PremiumCustomerSpecification` with the `Age18PlusCustomerSpecification` to query the count of premium adult customers as shown below:

```
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, ITran
    {
        private readonly IRepository<Customer, Guid> _c

        public CustomerManager(IRepository<Customer, Gu
        {
            _customerRepository = customerRepository;
        }

        public async Task<int> GetAdultPremiumCustomerC
        {
            return await _customerRepository.CountAsync
                new Age18PlusCustomerSpecification()
                .And(new PremiumCustomerSpecification()
            );
        }
    }
}
```

If you want to make this combination another reusable specification, you can create such a combination specification class deriving from the `AndSpecification`:

Share on :

In this document

```
using Volo.Abp.Specifications;

namespace MyProject
{
    public class AdultPremiumCustomerSpecification : An
    {
        public AdultPremiumCustomerSpecification()
            : base(new Age18PlusCustomerSpecification()
                   new PremiumCustomerSpecification())
        {
        }
    }
}
```

Now, you can re-write the `GetAdultPremiumCustomerCountAsync` method as shown below:

```
public async Task<int> GetAdultPremiumCustomerCountAsyn
{
    return await _customerRepository.CountAsync(
        new AdultPremiumCustomerSpecification()
    );
}
```

> You see the power of the specifications with these samples. If you change the `PremiumCustomerSpecification` later, say change the balance from `100.000` to `200.000`, all the queries and combined specifications will be effected by the change. This is a good way to reduce code duplication!

Discussions

While the specification pattern is older than C# lambda expressions, it's generally compared to expressions. Some developers may think it's not needed anymore and we can directly pass expressions to a repository or to a domain service as shown below:

```
var count = await _customerRepository.CountAsync(c => c
```

Since ABP's Repository supports Expressions, this is a completely valid use. You don't have to define or use any specification in your application and you can go with expressions.

So, what's the point of a specification? Why and when should we consider to use them?

When To Use?

Some benefits of using specifications:

- **Reusabe**: Imagine that you need the Premium Customer filter in many places in your code base. If you go with expressions and do not create a specification, what happens if you later change the

Share on :

In this document

"Premium Customer" definition? Say you want to change the minimum balance from $100,000 to $250,000 and add another condition to be a customer older than 3 years. If you'd used a specification, you just change a single class. If you repeated (copy/pasted) the same expression everywhere, you need to change all of them.
- **Composable**: You can combine multiple specifications to create new specifications. This is another type of reusability.
- **Named**: `PremiumCustomerSpecification` better explains the intent rather than a complex expression. So, if you have an expression that is meaningful in your business, consider using specifications.
- **Testable**: A specification is a separately (and easily) testable object.

When To Not Use?

- **Non business expressions**: Do not use specifications for non business-related expressions and operations.
- **Reporting**: If you are just creating a report, do not create specifications, but directly use `IQueryable` & LINQ expressions. You can even use plain SQL, views or another tool for reporting. DDD does not necessarily care about reporting, so the way you query the underlying data store can be important from a performance perspective.

Share on :

In this document