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

In this document

- Introduction to validation
- Using data annotations
- Custom Validation
- Fluent Validation
- Disabling Validation
- Normalization

Edit on GitHub

Introduction to validation

In an application, the inputs should be validated first. The input can be sent by a user or another application. In a web application, validation is usually implemented twice: on the client and server sides. Client-side validation is implemented mostly for user experience. It's better to check a form first in the client and show invalid fields to the user. However, server-side validation is unavoidable and more critical.

Server-side validation is generally implemented in application services or controllers (in general, all services get data from the presentation layer). An application service method should first check (validate) the input and then use it. ASP.NET Boilerplate provides the infrastructure to automatically validate inputs of an application for:

- All application service methods
- All ASP.NET Core MVC controller actions
- All ASP.NET MVC and Web API controller actions.

See the Disabling Validation section to disable validation if needed.

Using data annotations

ASP.NET Boilerplate supports data annotation attributes. Assume that we're developing a Task application service that is used to create a task by when it gets an input as shown below:

Copy

```
public class CreateTaskInput
{
    public int? AssignedPersonId { get; set; }

    [Required]
    public string Description { get; set; }
}
```

Here, the **Description** property is marked as **Required**. AssignedPersonId is optional. There are also many attributes (like MaxLength, MinLength, RegularExpression...) in the **System.ComponentModel.DataAnnotations** namespace. See the Task application service implementation:

Copy

```
public class TaskAppService : ITaskAppService
{
    private readonly ITaskRepository _taskRepository;
    private readonly IPersonRepository _personRepository;

    public TaskAppService(ITaskRepository taskRepository, IPersonRepository
personRepository)
    {
        _taskRepository = taskRepository;
        _personRepository = personRepository;
    }

    public void CreateTask(CreateTaskInput input)
    {
        var task = new Task { Description = input.Description };

        if (input.AssignedPersonId.HasValue)
        {
            task.AssignedPerson = _personRepository.Load(input.AssignedPersonId.Value);
        }

        _taskRepository.Insert(task);
    }
}
```

As you can see, there is no validation code written since ASP.NET Boilerplate does it automatically. ASP.NET Boilerplate also checks if input is **null** and throws an **AbpValidationException** if it is, so you don't have to write **null-check** code (guard clauses). It also throws an AbpValidationException if any of the input properties are invalid.

This mechanism is similar to ASP.NET MVC's validation but note that an application service class is not derived from a Controller, it's a plain class and can work even outside of a web application.

Custom Validation

If data annotations are not sufficient for your case, you can implement the **ICustomValidate** interface as shown below:

Copy

```
public class CreateTaskInput : ICustomValidate
{
    public int? AssignedPersonId { get; set; }

    public bool SendEmailToAssignedPerson { get; set; }

    [Required]
    public string Description { get; set; }

    public void AddValidationErrors(CustomValidationContext context)
    {
        if (SendEmailToAssignedPerson && (!AssignedPersonId.HasValue ||
AssignedPersonId.Value <= 0))
        {
            context.Results.Add(new ValidationResult("AssignedPersonId must be set if
SendEmailToAssignedPerson is true!"));
        }
    }
}
```

The ICustomValidate interface declares the **AddValidationErrors** method to be implemented. We must add the **ValidationResult** objects to the **context.Results** list if there are validation errors. You can also use the context.IocResolver to resolve dependencies if needed in the validation process.

In addition to ICustomValidate, ABP also supports .NET's standard IValidatableObject interface. You can also implement it to perform additional custom validations. If you implement both interfaces, both of them will be called.

Fluent Validation

In order to use FluentValidation, you need to install Abp.FluentValidation package first.

```
Install-Package Abp.FluentValidation
```

Then, You should set a dependency to AbpFluentValidationModule from your module. Example:

```
[DependsOn(typeof(AbpFluentValidationModule))]
public class MyProjectAppModule : AbpModule
{

}
```

After all, you can define your FluentValidation validators to validate matching input classes.

As an example, if you have an input class and a Controller which uses this class as it's input parameter;

```
public class MyCustomArgument1
{
        public int Value { get; set; }
}

public class MyTestController : AbpController {

        public JsonResult GetJsonValue([FromQuery] MyCustomArgument1 arg1)
        {
                return Json(new MyCustomArgument1
                {
                        Value = arg1.Value
                });
        }
}
```

If you want to limit the value of MyCustomArgument1's Value field between 1 and 99, you can define a validator like the one below;

```
public class MyCustomArgument1Validator : AbstractValidator<MyCustomArgument1>
{
        public MyCustomArgument1Validator()
        {
                RuleFor(x => x.Value).InclusiveBetween(1, 99);
        }
}
```

ABP will run MyCustomArgument1Validator to validate MyCustomArgument1 class automatically.

Disabling Validation

For automatically validated classes (see Introduction section), you can use these attributes to control validation:

- **DisableValidation** attribute can be used for classes, methods or properties of DTOs to disable validation.
- **EnableValidation** attribute can only be used to enable validation for a method, if it's disabled for the containing class.

Normalization

We may need to perform an extra operations to prepare DTO parameters after validation. ASP.NET Boilerplate defines an **IShouldNormalize** interface that has a **Normalize** method. If you implement this interface, the Normalize method is called just after validation (and just before the method call). Assume that our DTO gets a Sorting direction. If it's not supplied, we want to set a default sorting:

Copy

```
public class GetTasksInput : IShouldNormalize
{
    public string Sorting { get; set; }

    public void Normalize()
    {
        if (string.IsNullOrWhiteSpace(Sorting))
        {
            Sorting = "Name ASC";
        }
    }
}
```