Blog Fluentvalidation In Aspnet Core

12 min read Updated on May 6, 2024

# How to use FluentValidation in ASP.NET Core - Super Powerful Validations

Mukesh Murugan
@iammukeshm

#dotnet

When it comes to Validating Models and incoming HTTP Requests, aren't we all leaning toward Data Annotations? Although it can be quick to set up Data Annotations on your models, there are a few drawbacks to this approach.

In this article, we will try to understand the problems with Data Annotations for larger projects

The end game of this series is to build scalable .NET applications with Clean Architecture! I am building the content one by one to reach to the finale - *Practical Clean Architecture with ASP.NET Core!* Join Now to BOOST your .NET Skills 🚀

## The Need for Validation

Data validation is a critical aspect of developing robust and secure web applications. It ensures that the data entered by users is accurate, reliable, and safe for processing. Without proper validation, applications are vulnerable to various issues, including:

Data Integrity: Validation ensures that the data conforms to the expected format and rules, preventing incorrect or incomplete data from being saved in the database. This helps maintain the integrity of the data and ensures its consistency.

Security: Proper validation helps prevent security vulnerabilities such as SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF) by ensuring that user input is sanitized and safe for processing.

User Experience: Validation improves the user experience by providing immediate feedback to users about the correctness of their input. This helps users quickly correct any mistakes, reducing frustration and improving usability.

Business Logic: Validation enforces business rules and logic, ensuring that data meets the requirements of the application.

## Why are Data Annotations not suitable always?

The GO-TO Approach for Model validation in any .NET Application is Data Annotations, where you have to declare attributes over the properties of respective classes. Worked with it before?

```
public class UserRegistrationRequest
{
    [Required(ErrorMessage = "First name is required.")]
    public string? FirstName { get; set; }

    [Required(ErrorMessage = "Last name is required.")]
    public string? LastName { get; set; }

    [Required(ErrorMessage = "Email is required.")]
    [EmailAddress(ErrorMessage = "Invalid email address.")]
```

```
        public string? Email { get; set; }

        [Required(ErrorMessage = "Password is required.")]
        [MinLength(6, ErrorMessage = "Password must be at least 6 characters.")]
        public string? Password { get; set; }

        [Required(ErrorMessage = "Please confirm your password.")]
        [Compare("Password", ErrorMessage = "Passwords do not match.")]
        public string? ConfirmPassword { get; set; }
    }
```

In the above example, we have a `UserRegistraionRequest` with fields like Email, Name, and Password. To ensure that the client has given us valid input, we introduced `Data Annotations` to validate the request. For instance, the FirstName field is a required property, the ComparePassword field should match the Password field, and so on.

It is fine for small projects and POCs. But once you start learning clean code, or begin to understand the SOLID principles of application design, you would just never be as happy with Data Annotations as you were before. It is not a good approach to combine your models and validation logic.

Using data annotations ties your validation rules to your model classes. This can violate the separation of concerns principle, as validation logic is mixed with your domain logic.

So, what's the solution?

## Introducing FluentValidation - The Solution

FluentValidation is a powerful open-source .NET validation library that helps you make your validations clean, easy to create, and maintain. It helps you build strongly typed validation rules. It even works on external models that you don't have access to. With this library, you can separate the model classes from the validation logic as it is supposed to be. It doesn't make your classes cluttered like Data Annotations do. Also, better control of validation is something that makes the developers prefer FluentValidation.

> *FluentValidation provides a fluent interface for defining validation rules, allowing you to express complex validation logic clearly and concisely. This makes it easy to create custom validation rules and handle complex validation scenarios.*

For smaller projects, I would recommend just using Data Annotations because they're so easy to set up. For larger, more complex systems, FluentValidation is the way to go, due to the above-mentioned benefits.

## Using FluentValidation in ASP.NET Core Applications

For this demonstration, I will be using an ASP.NET Core 8 Web API project, and my IDE as Visual Studio 2022 Community. We will do the API testing using Swagger.

Open up Visual Studio and create a new .NET 8 Web API project. Ensure that you have enabled Swagger, as we will be testing our API with this.

## Installing FluentValidation Package

Let's get started by installing the required FluentValidation NuGet packages to your project via the Package Manage Console.

```
Install-Package FluentValidation
Install-Package FluentValidation.DependencyInjectionExtensions
```

> *Note that the `FluentValidation.AspNetCore` package is now deprecated.*

## User Registration Request

For the demo, we will take the scenario where our API is responsible for registering new users into the system by exposing a POST API endpoint. Here is what the C# class would look like.

```
public class UserRegistrationRequest
{
    public string? FirstName { get; set; }
    public string? LastName { get; set; }
    public string? Email { get; set; }
```

```
        public string? Password { get; set; }
        public string? ConfirmPassword { get; set; }
    }
```

Our end goal is to validate this request class whenever the registration API is hit. For example, we need to ensure that the user has entered a valid `FirstName` , `Email` and so on. We will build up on this by introducing various features offered by FluentValidation.

## Adding Validator Rules

In this step, we will define the validation rules of our `UserRegistrationRequest` . Create a new folder named Validators and add a new class named `UserRegistrationValidator`

To begin with, we will add a single validation rule that validates if the passed Email property is an Email Address.

```
    public class UserRegistrationValidator : AbstractValidator<UserRegistrationRequest>
    {
        public UserRegistrationValidator()
        {
            RuleFor(x => x.Email).EmailAddress();
        }
    }
```

Note that the `UserRegistrationValidator` inherits from `AbstractValidator<T>` where T is the actual class to be validated.

## Registering Validators in ASP.NET Core

We will have to add FluentValidation to our application. Open up `Program.cs` .

Now there are several ways to register the `UserRegistrationValidator` into the application's DI Container.

If you want to register just a single validator, you can use the following.

```
builder.Services.AddScoped<IValidator<UserRegistrationRequest>, UserRegistrationValidator
```

This ensures that you can inject `IValidator<UserRegistrationRequest>` into your constructors and validate as required.

In case your project has multiple validators, and you don't want to register them manually one by one, you can use the following.

```
builder.Services.AddValidatorsFromAssemblyContaining<UserRegistrationValidator>();
```

In the above code, FluentValidation registers all the validator instances that are found in the assembly containing `UserRegistrationValidator`.

## Approaches for Validation

`FluentValidation` validates incoming models in different ways, like

- Manual Validation - This is the recommended approach.
- Automatics Validation using ASP.NET Core Pipeline - This is deprecated and no longer supported in the latest release of FluentValidation, version `11.9.1`. Read more about the breaking change from this discussion.
- Validation using Filters - requires external packages.

In this demo, we will learn about Manual Validation using the FluentValidation package.

## API Endpoint

Now, we will create a Minimal API endpoint that allows one to register a new user into the system. This will be a POST endpoint and will accept a body of type `UserRegistrationRequest`.

```
app.MapPost("/register", async (UserRegistrationRequest request, IValidator<UserRegistrat
{
    var validationResult = await validator.ValidateAsync(request);
```

```
        if (!validationResult.IsValid)
        {
            return Results.ValidationProblem(validationResult.ToDictionary());
        }
        // perform actual service call to register the user to the system
        // _service.RegisterUser(request);
        return Results.Accepted();
    });
```

Note that we have injected an instance of `IValidator<UserRegistrationRequest>` to the endpoint which we will use for validating the incoming object.

We first use the validator object to asynchronously validate the `UserRegistrationRequest` object. If the result is valid, the endpoint returns an Accepted status code. Otherwise, Problem Details of Type Validation in returned. Let's see this in action.

Build and run your .NET 8 Web API, and open up Swagger.

```
{
  "firstName": "string",
  "lastName": "string",
  "email": "string",
  "password": "string",
  "confirmPassword": "string"
}
```

I am going to invoke the API by passing the above default values. As you see, we have specified an invalid email id. If you execute the request, you will be able to see the following response.

```
{
  "type": "https://tools.ietf.org/html/rfc9110#section-15.5.1",
  "title": "One or more validation errors occurred.",
  "status": 400,
  "errors": {
    "Email": ["'Email' is not a valid email address."]
  }
}
```

As you see, we got a 400 Bad Request response with validation errors that include the `'Email' is not a valid email address.` error message. If you pass a valid email address to the request, you will be able to see that this validation error goes away. Note that this is capable of returning an array of errors if required.

## Extending the Validation Rules

In the next steps, we will further explore the capabilities of `FluentValidation` and how it can help build powerful validations in our system.

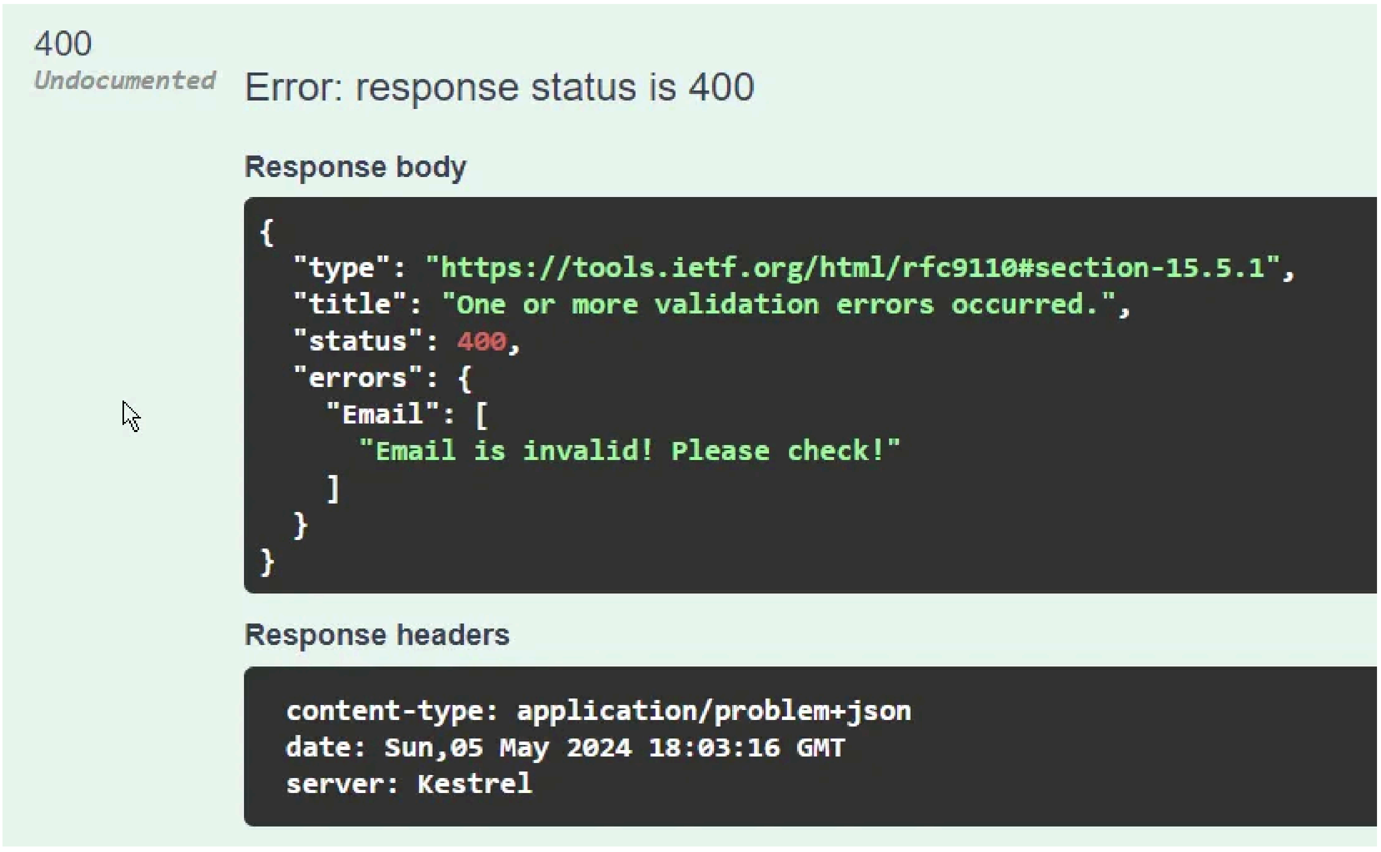
### Custom Validation Messages

It's highly practical to show custom validation messages based on the property. It's quite simple with FluentValidation. Go to the validator that we created and add an extension `WithMessage()` to the rule associated with the Email property.

```
RuleFor(x => x.Email).EmailAddress().WithMessage("{PropertyName} is invalid! Please check
```

> *Use `{PropertyName}` to get the corresponding Property Name. These are placeholder variables of FluentValidation Library. Read more about them here.*

Let's check out the response now.



400
*Undocumented*

Error: response status is 400

Response body

```
{
  "type": "https://tools.ietf.org/html/rfc9110#section-15.5.1",
  "title": "One or more validation errors occurred.",
  "status": 400,
  "errors": {
    "Email": [
      "Email is invalid! Please check!"
    ]
  }
}
```

Response headers

```
content-type: application/problem+json
date: Sun,05 May 2024 18:03:16 GMT
server: Kestrel
```

## Built-In Validators

There are several validators you get by default with FluentValidation. We will use some of them to write additional validations to the other properties of the class.

```
public class UserRegistrationValidator : AbstractValidator<UserRegistrationRequest>
{
    public UserRegistrationValidator()
    {
        RuleFor(x => x.FirstName).NotEmpty().MinimumLength(4);
        RuleFor(x => x.LastName).NotEmpty().MaximumLength(10);
        RuleFor(x => x.Email).EmailAddress().WithMessage("{PropertyName} is invalid! Plea
        RuleFor(x => x.Password).Equal(z => z.ConfirmPassword).WithMessage("Passwords do
    }
}
```

Let me explain what each rule validates.

1. `FirstName` should not be empty and have a minimum string length of 4.
2. `LastName` also should not be empty and should have a length with a maximum of 10 characters.

3. Email validation as we saw earlier.
4. This ensures that the `Password` and `ConfirmPassword` fields match in the incoming request.

## Overriding Property Name

You also get the provision to change the name of the property returned to the validation messages. For instance, if you want to change the property name of `Email` to `MailID`, you can simply add the `WithName` extension.

```
RuleFor(x => x.Email).EmailAddress().WithName("MailID").WithMessage("{PropertyName} is in
```

And here is the response from our ASP.NET Core Web API:

```
{
  "type": "https://tools.ietf.org/html/rfc9110#section-15.5.1",
  "title": "One or more validation errors occurred.",
  "status": 400,
  "errors": {
    "Email": ["MailID is invalid! Please check!"]
  }
}
```

## Chaining Validators

There can be cases where you want to add more than validation to a particular property. The below rule has 2 such validations in the `FirstName` property. It should not be empty, and at the same time the minimum length of the property has to be at least 4 characters.

```
RuleFor(x => x.FirstName).NotEmpty().MinimumLength(4);
```

## Stop on First Failure

You can also customize how FluentValidation executes the validation checks. For instance, if you consider the above code, where we have 2 validations bound to the same rule, and you pass an empty string as the value for `FirstName`, you will be seeing 2 validation messages in the response.

```
{
  "type": "https://tools.ietf.org/html/rfc9110#section-15.5.1",
  "title": "One or more validation errors occurred.",
  "status": 400,
  "errors": {
    "FirstName": [
      "'First Name' must not be empty.",
      "The length of 'First Name' must be at least 4 characters. You entered 0 characters
    ]
  }
}
```

In certain cases, you want the validation checks to stop as soon as the first validation breaks for a property. In our case, we might need to stop the validation check right at the `NotEmpty` validator.

To achieve this, we have the Cascade property which allows 2 modes,

- Continue - This is the default behavior, which continues to invoke the validations even though a previous validation has failed.
- Stop - Stops executing validation checks as soon as the first validation breaks.

You can modify your code as below,

```
RuleFor(x => x.FirstName).Cascade(CascadeMode.Stop).NotEmpty().MinimumLength(4);
```

## Custom Validations

As an example, theoretically, you could pass a number as the first name and the API would just say ‘Cool, that’s fine’. Let’s see how to build a validation rule with Fluent Validation for such a scenario.

First, create a simple helper method that would take in our `FirstName` property and return a boolean based on its content. We will place this method in our `UserRegistrationValidator` class itself.

```
private bool IsValidName(string name)
{
    return name.All(Char.IsLetter);
}
```

Now, let's add this requirement to the rule and wire it up with our helper method.

```
RuleFor(x => x.FirstName)
    .Cascade(CascadeMode.Stop)
    .NotEmpty()
    .MinimumLength(4)
    .Must(IsValidName).WithMessage("{PropertyName} should be all letters.");
```

Line #5 suggests that the property with return a true when passed to our helper.

Now, if I pass a number to the `FirstName` property as part of the request, I will get the following response from my API.

```
{
  "type": "https://tools.ietf.org/html/rfc9110#section-15.5.1",
  "title": "One or more validation errors occurred.",
  "status": 400,
  "errors": {
    "FirstName": ["First Name should be all letters."]
  }
}
```

So, that is taken care of as well.

## Enhancing with MediatR Pipeline Behavior

You might have already seen a problem with our implementation. In our API endpoint code we are always trying to validate the request first, before proceeding to the actual application logic. Is there a way to remove the below code and let you app handle validation by itself?

```
var validationResult = await validator.ValidateAsync(request);
if (!validationResult.IsValid)
{
    return Results.ValidationProblem(validationResult.ToDictionary());
}
```

To solve this, we can use MediatR Pipeline Behavior, a middleware that can automatically execute the validation mechanism as soon as the request enters into the application pipeline. This way you no longer need to write the validation calls to each and every API endpoint. Ideally, this is how you would have to build your applications!

You can read the article here : Validation with MediatR Pipeline Behavior and FluentValidation - Handling Validation Exceptions in ASP.NET Core

## Summary

I hope you have enjoyed this simple-to-follow guide about FluentValidation in ASP.NET Core. I am sure you will be switching to this powerful library from today onwards! What are your opinions about this library? Some devs still prefer Data Annotations over this. What about you? Let me know in the comments below.

### Source Code ✌️

Grab the source code of the entire implementation by clicking here. Do Follow me on GitHub .

### Support ❤️

If you have enjoyed my content and code, do support me by buying a couple of coffees. This will enable me to dedicate more time to research and create new content. Cheers!

## Share this Article

Share this article with your network to help others!

## What's your Feedback?

Do let me know your thoughts around this article.

1 reaction

👍 1

3 comments – *powered by giscus*

Oldest Newest

Write Preview Aa

Sign in to comment

Sign in with GitHub

nileshgr May 10

Thanks for the nice intro to FluentValidation!

↑ 1 ❤️ 1 0 replies

abelmiraval May 29

Hello, how to validate a decimal type field that does not send in the request?

↑ 1 0 replies

zabronm Jun 10

This is another excellent read from you, thank you.

↑ 1 0 replies

## Mukesh's .NET Newsletter 🚀

Join **5,000+ Engineers** to Boost your .NET Skills. I have started a .NET Zero to Hero Series that covers everything from the basics to advanced topics to help you with your .NET Journey! You will receive 1 Awesome Email every week.

Subscribe

codewith**mukesh**

web development simplified!

Home Blog Contact Sponsorship About