ASP.NET Core Basics: Dealing with Backend Validations

When developing web applications, it is common to come across situations where taking care of the processed data is essential for the success of a project.

Imagine a registration system that allows you to register a user’s phone number with a text value like “John” for example. This isn’t terribly uncommon. Data such as telephone and email are used for basic things like authentication. In scenarios like this, backend validations play a crucial role: helping safeguard the integrity of the data processed by applications.

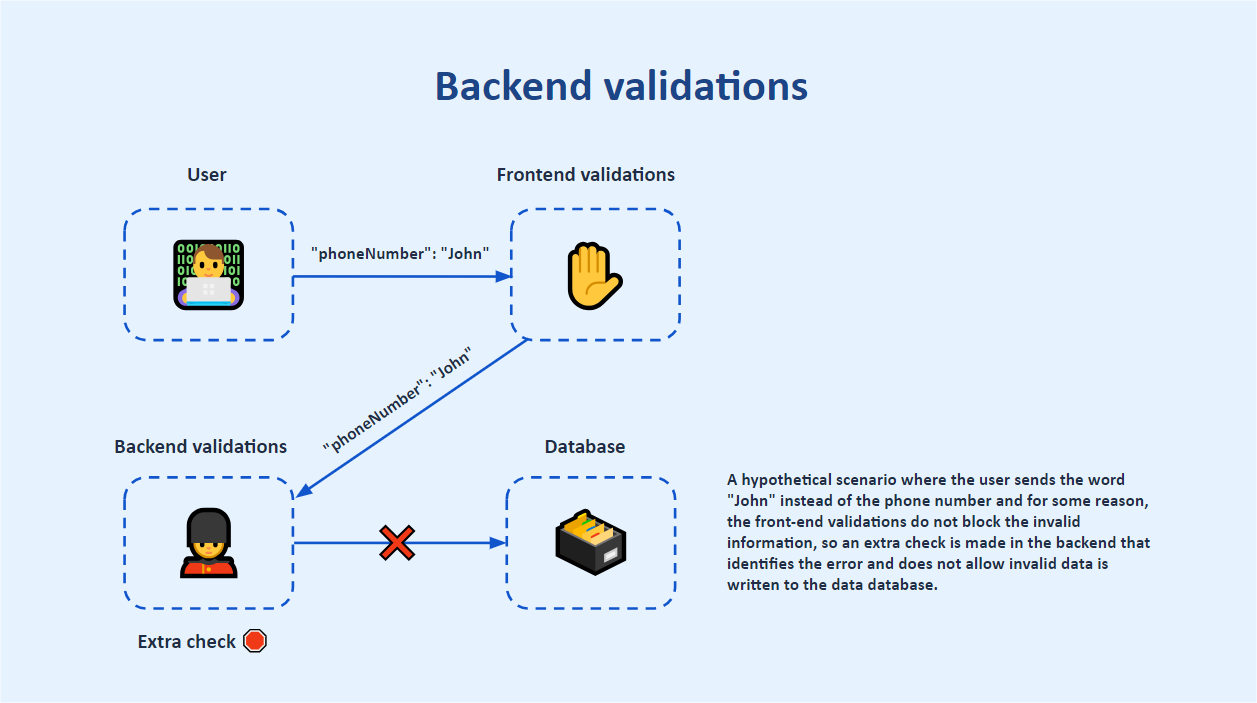
In this post, we’ll explore how to implement backend validations using ASP.NET Core. Through practical examples and step-by-step guidance, you’ll learn how to validate data models, validate input data in controllers, use third-party libraries and more.

What Are Backend Validations? And What Kinds Are Available in ASP.NET Core?

Backend validations are an additional layer of security implemented on the server to complement client-side validations.

While client-side validations offer a more fluid and responsive user experience by providing immediate feedback on entered data, backend validations only allow valid and secure data to be processed and stored on the server.

To handle validation on the backend, ASP.NET Core has some techniques and libraries that can be used to facilitate development and better protect the integrity and security of data stored on the server side. These validations include validation in the model, custom validation in the controller, custom validation attributes in addition to third-party libraries, and others.



Implementing Backend Validations in ASP.NET Core

Next, we will create applications in ASP.NET Core and implement validations through practical examples using native .NET resources in addition to third-party libraries such as Fluent Validation.

You can access the source code of the examples here: [Student Management Service source code](https://github.com/zangassis/student-management-service).

1. Model Validation with Data Annotations

Data annotations are attributes that can be applied to class properties in ASP.NET Core to provide metadata about how those properties should be handled.

When performing model validations, we use these attributes to define validation rules that ASP.NET Core uses to validate model data during model binding. Several rules can be included in the model class. To see in practice how these validations work, let’s create a simple API in ASP.NET Core and make some requests to test it.

To create the example application, you can use the commands below. Remember that you must have the latest version of [.NET](https://dotnet.microsoft.com/en-us/download) installed locally.

dotnet new web -o StudentManagementService

Then open the application with your favorite IDE and in it execute the following command to download the Swashbuckle library to use the Swagger interface:

dotnet add package Swashbuckle.AspNetCore

Now, inside the project create a new folder called “Models” and inside it create the class below:

using System.ComponentModel.DataAnnotations;

namespace StudentManagementService.Models;

public class Student

{

public Guid Id { get; set; }

[Required(ErrorMessage = "The name is required.")]

[StringLength(50)]

public string Name { get; set; }

[Required(ErrorMessage = "The LastName is required.")]

[StringLength(50)]

public string LastName { get; set; }

[Required(ErrorMessage = "The email is required.")]

[EmailAddress(ErrorMessage = "Invalid email format.")]

public string Email { get; set; }

[Required(ErrorMessage = "The phone number is required.")]

[Phone(ErrorMessage = "Invalid phone number format.")]

public string PhoneNumber { get; set; }

[MaxLength(15)]

[Required(ErrorMessage = "The street address is required.")]

public string StreetAddress { get; set; }

[Required(ErrorMessage = "The city is required.")]

public string City { get; set; }

[Required(ErrorMessage = "The state is required.")]

[StringLength(2, MinimumLength = 2, ErrorMessage = "The state abbreviation must be 2 characters.")]

public string State { get; set; }

}

C#

Note that the Student class has several properties and each of them has a data annotation determining one more rule for the model class.

The Name property, for example, has the data annotation [Required(ErrorMessage = "The name is required.")], which requires that a value for this property be informed when instantiating the Student class. It also has the data annotation [StringLength (50)] which determines that the value entered does not exceed 50 characters. There are several other data annotations available such as [EmailAddress] for email fields and [Phone] for phone number fields, among others.

To validate these fields, let’s create an endpoint and make some requests through Swagger. In the Program.cs file, replace the existing code with the code below:

using System.ComponentModel.DataAnnotations;

using StudentManagementService.Models;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen();

var app = builder.Build();

app.UseSwagger();

app.UseSwaggerUI(c =>

{

c.SwaggerEndpoint("/swagger/v1/swagger.json", "Student Management Service V1");

});

app.MapPost("/student/create", (Student student) =>

{

if (!IsValid(student, out var validationResult))

return Results.BadRequest(validationResult);

return Results.Ok();

});

static bool IsValid<T>(T obj, out ICollection<ValidationResult> results) where T : class

{

var validationContext = new ValidationContext(obj);

results = new List<ValidationResult>();

return Validator.TryValidateObject(obj, validationContext, results, true);

}

app.Run();

C#

Note that the code above defines the Swagger settings. In addition, there is an endpoint to simulate the creation of a new student. When requested, this endpoint calls the IsValid method, which receives a class, and makes the necessary checks according to the rules established in the model class—in this case, the data annotations. If one or more rules were not met in the request, an object of type ValidationResult is returned, with the error messages configured in the Student class. If all rules were respected, an ok status is returned.

Now, let’s run the application and check if the validations are working as expected. To run the application, open a terminal in the application and execute the command below.

dotnet run

Then, in your browser, access the address shown in the terminal, plus /swagger/index.html, which takes you to the Swagger interface.

In the Swagger interface, expand the request tab, click on Try it out and in the editable field place the following request:

{

"id": "3fa85f64-5717-4562-b3fc-2c963f66afa6",

"name": "John",

"lastName": null,

"email": "johnsmith",

"phoneNumber": null,

"streetAddress": "097 Main St, Apt 101",

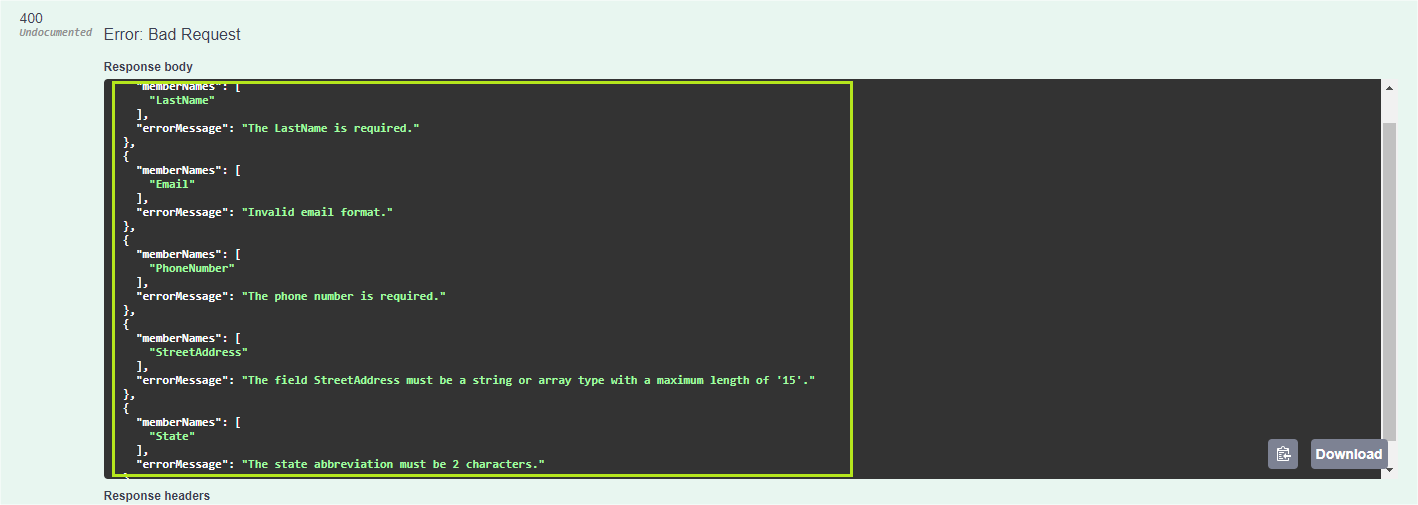
"city": "Freeport",

"state": "New York"

}

JSON

Then click on Execute. The following results will be displayed:



Note that the API response returned an HTTP status 400 - Bad Request, and in the body of the response, there is a JSON that informs which fields have a problem (memberNames) and the message with error details (errorMessage).

For example, in the StreetAddress field, note that the message informs that the value sent in the request must be a string or array with a maximum of 15 characters. This happened because the value sent was “097 Main St, Apt 101” which has 20 characters (white spaces are also considered in validation).

Now make a new API request, but this time pass the following body request:

{

"Name": "John",

"LastName": "Smith",

"Email": "john.Smith@example.com",

"PhoneNumber": "1234567890",

"StreetAddress": "097 Main St",

"City": "Freeport",

"State": "NY"

}

JSON

The following results will be displayed:



The result of the request was now an HTTP status of 200 - Success because all data sent in the request body is valid.

Data annotations provide a simple way to implement validations directly on data models in ASP.NET Core. With automatic generation of error responses, it simplifies the validation process and improves code consistency.

Visualize your APIs with Fiddler Everywhere

For robust debugging options, check out [Fiddler Everywhere](https://www.telerik.com/fiddler/fiddler-everywhere) from Progress Telerik.

Another advantage of using data annotations is the ability to centralize validation rules in data models, which helps keep the code organized and facilitates maintenance. With data annotations, validation rules are directly associated with relevant model properties, avoiding code duplication and providing a more cohesive approach to data validation.

Creating Custom Data Annotations

In addition to using the data annotations provided by the System.ComponentModel.DataAnnotations namespace, it is also possible to create our custom data annotations.

So, in the project create a new folder called “Attributes” and then within it create the following class:

* CityValidationAttribute

using System.ComponentModel.DataAnnotations;

namespace StudentManagementService.Attributes;

public class CityValidationAttribute : ValidationAttribute

{

protected override ValidationResult IsValid(object value, ValidationContext validationContext)

{

var cityName = value as string;

var newYorkCities = new List<string>

{

"New York City",

"Buffalo",

"Rochester",

"Yonkers",

"Syracuse",

"Albany",

"New Rochelle",

"Mount Vernon",

"Schenectady",

"Utica"

};

if (!newYorkCities.Contains(cityName))

return new ValidationResult("The city provided is not a valid city in New York.", new[] { validationContext.MemberName });

return ValidationResult.Success;

}

}

C#

The CityValidationAttribute class implements inheritance from the ValidationAttribute class, which is responsible for providing the ASP.NET Core validation mechanism through attributes. The CityValidationAttribute class also replaces the ValidationResult method, which validates whether the city entered is part of one of the cities configured in the list—in this case, some random cities in New York.

If the city is valid, it returns success; otherwise, it returns an error message stating that the city is not valid.

The next step is to add the custom attribute to the City property of the Student class. In the Student class, above the City property, add the CityValidation attribute, which is the name of the custom class, without the Attribute. So the City property will look like this:

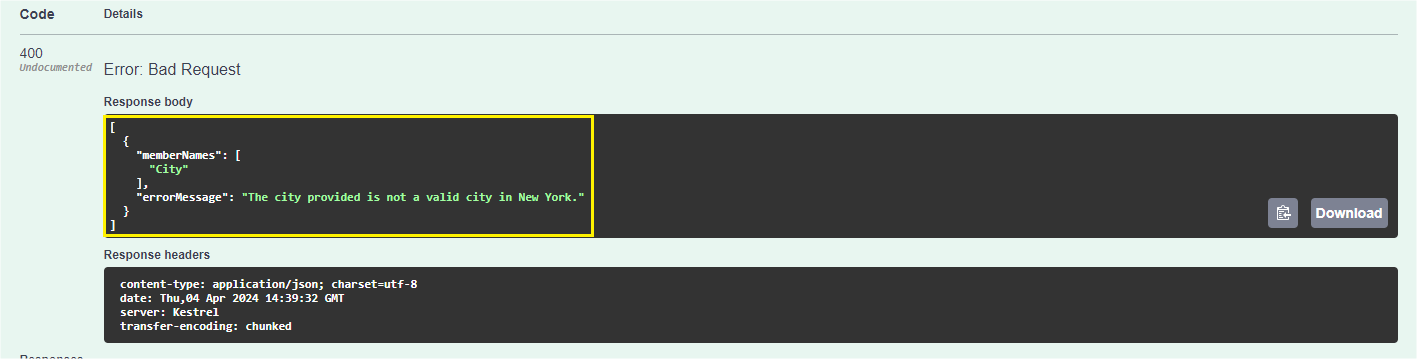
[CityValidation]

[Required(ErrorMessage = "The city is required.")]

public string City { get; set; }

C#

Now run the application again and, in the body of the request, in the city field, send a value that is not one of the cities configured in the validator. Here I will use “Chicago” and the result of the request will be the following:



Note that the custom validation worked in the same way as the native validations, which allow any custom validation to be created, depending only on the input and output requirements.

2. Endpoint Filters

[Endpoint filters](https://learn.microsoft.com/en-us/aspnet/core/fundamentals/minimal-apis/min-api-filters?view=aspnetcore-7.0) are a feature available as of .NET 7 and are used to validate the parameters and body sent in a request to an endpoint, and record information about the request and the response.

To add a filter endpoint to the API, first create a new folder called “Helpers” and inside it add the class below:

using StudentManagementService.Models;

namespace StudentManagementService.Helpers;

public static class Utilities

{

public static string IsValid(Student student)

{

string errorMessage = string.Empty;

if (string.IsNullOrEmpty(student.Name))

errorMessage = "Student name is required";

return errorMessage;

}

}

C#

Now, on the endpoint to create a student, add the code below:

.AddEndpointFilter(async (efiContext, next) =>

{

var tdparam = efiContext.GetArgument<Student>(0);

var validationError = Utilities.IsValid(tdparam);

if (!string.IsNullOrEmpty(validationError))

{

app.Logger.LogInformation($"Error when creating a new student: {validationError}");

return Results.Problem(validationError);

}

return await next(efiContext);

});

C#

The complete code will look like this:

app.MapPost("/student/create", (Student student) =>

{

if (!IsValid(student, out var validationResult))

return Results.BadRequest(validationResult);

return Results.Ok();

}).AddEndpointFilter(async (efiContext, next) =>

{

var tdparam = efiContext.GetArgument<Student>(0);

var validationError = Utilities.IsValid(tdparam);

if (!string.IsNullOrEmpty(validationError))

{

app.Logger.LogInformation($"Error when creating a new student: {validationError}");

return Results.Problem(validationError);

}

return await next(efiContext);

});

C#

In the code above, we added the AddEndpointFilter extension method that validates whether the student name is null or empty. If yes, it returns an error message; otherwise the error message returns no value.

Now, run the application again, and in the body of the request use the JSON below:

{

"Name": "",

"LastName": "Smith",

"Email": "john.Smith@example.com",

"PhoneNumber": "1234567890",

"StreetAddress": "097 Main St",

"City": "Freeport",

"State": "NY"

}

C#

When executing the request, you will have the following result:



Note that the result of the request was an HTTP status 500 - Internal Server Error and the detail contains the error message configured in the endpoint validation (“Student name is required”). One of the advantages of using endpoint filters is that validation occurs before any type of logic or even another validation that may exist within the endpoint.

3. Validations with Third-Party Libraries

In addition to the validations with native ASP.NET Core resources seen previously, there is also the possibility of using external resources to create validations through the use of NuGet packages developed by third parties.

The advantages of using third-party libraries are that they bring agility to development, as they already have several implementations, requiring only the configuration and customization of validations.

On the other hand, it is worth considering that when adding third-party libraries to the project, there may be an increase in complexity in the development and maintenance of the program, in addition to the fact that some libraries may have some cost or copyright involved.

In this post, we will check out Fluent Validation, which is an open-source library developed for ASP.NET Core and offers a more fluent and expressive data validation experience.

Fluent Validation simplifies the data entry validation process in ASP.NET Core applications. It offers a more fluent and declarative way to define validation rules for your application’s data models. With Fluent Validation, we can easily define complex validation rules clearly and concisely, which makes the code more readable and maintainable.

To download the Fluent Validation dependency to the project, use the command below:

dotnet add package FluentValidation.AspNetCore

Now let’s implement the validations. To do this, we will create a new entity and a new endpoint. So, inside the “Models” folder, create the following class:

namespace StudentManagementService.Models;

public class University

{

public Guid Id { get; set; }

public string Name { get; set; }

public string Location { get; set; }

}

C#

Then create a new folder called “Validators” and inside it create the class below:

using FluentValidation;

using StudentManagementService.Models;

namespace StudentManagementService.Validators

{

public class UniversityValidator : AbstractValidator<University>

{

public UniversityValidator()

{

RuleFor(university => university.Name)

.NotEmpty()

.WithMessage("The name is required.")

.MaximumLength(50);

RuleFor(university => university.Location)

.NotEmpty()

.WithMessage("The Location is required.")

.MaximumLength(50);

}

}

}

C#

Note that the class above is inheriting from the AbstractValidator class, which receives the University class as a parameter. Some validations are implemented within the constructor, checking that the property is not null or empty and that it has a maximum of 50 characters. If any of these rules are broken, an error message is configured using the “WithMessage” method.

It is possible to highlight that through FluentValidation, validations can be implemented in a fluid way that is easy to understand and maintain—after all, the code used to create validations is written using extension methods, making it possible to create validation structures as needed.

Now, let’s create the FluentValidator configuration and create an endpoint to test the validations.

Then in the Program.cs file add the code below:

builder.Services.AddValidatorsFromAssemblyContaining<UniversityValidator>();

C#

Then, add the new endpoint:

app.MapPost("/university/create", (University university, [FromServices] IValidator<University> validator) =>

{

var validationResult = validator.Validate(university);

if (validationResult.IsValid)

return Results.Ok("University object is valid.");

else

return Results.BadRequest(validationResult.Errors);

});

C#

Note that the code above is validating the object sent in the request through the Validate() method of the FluentValidation interface (Ivalidator). If the object is valid, it returns a success message; if not, it returns an error with the messages configured in the validation class.

You also need to add the following “using” to the Program class:

using System.ComponentModel.DataAnnotations;

using StudentManagementService.Models;

using StudentManagementService.Helpers;

using FluentValidation;

using Microsoft.AspNetCore.Mvc;

using StudentManagementService.Validators;

C#

Now let’s run the application and test the validations using FluentValidation.

In the Swagger interface, run the /university/create endpoint using the following JSON in the request:

{

"id": "3fa85f64-5717-4562-b3fc-2c963f66afa6",

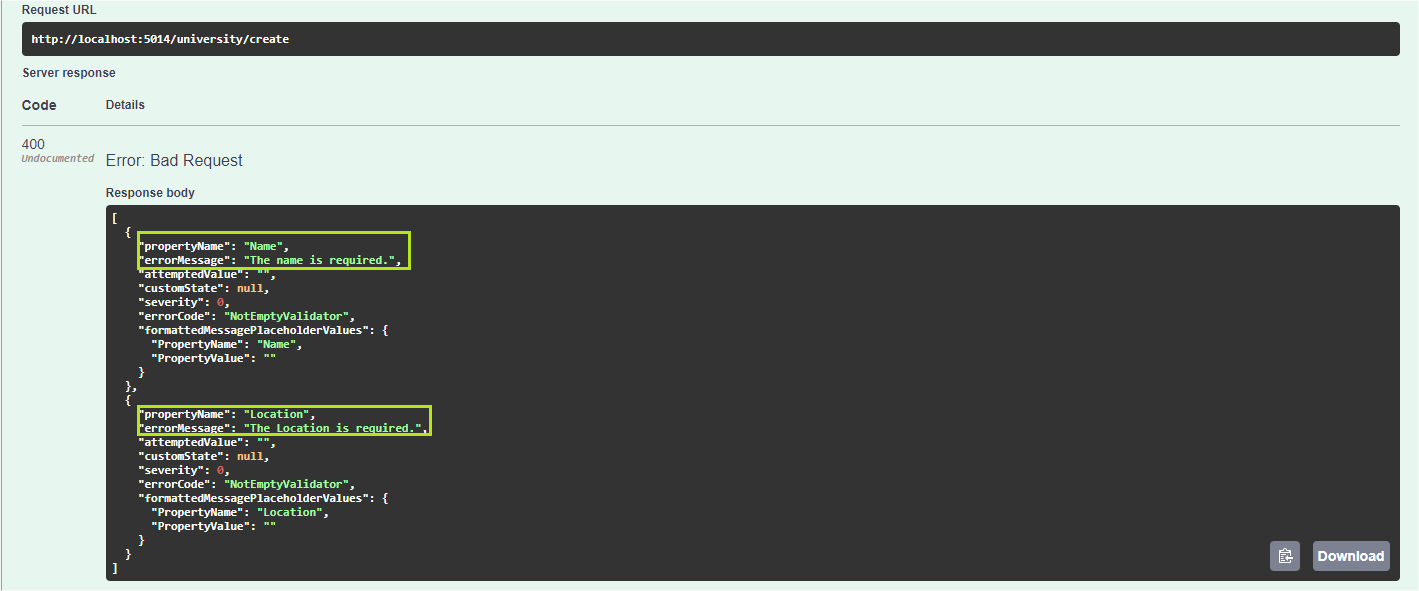
"name": "",

"location": ""

}

JSON

The response to the request will be something similar to the image below:



Note that the API returned an HTTP status 400 - Bad Request, informing which fields resulted in an error and what the errors were. In addition, there is other information such as the type of validation, in this case, NotEmptyValidator.

There are several other validation methods native to FluentValidation, you can check the complete list in the [FluentValidation documentation](https://docs.fluentvalidation.net/en/latest/built-in-validators.html" \t "_blank). In addition, you can create your own [custom validations](https://docs.fluentvalidation.net/en/latest/custom-validators.html).

Conclusion

In this post, we checked out three examples of backend validations in ASP.NET Core using native .NET resources in addition to external libraries such as FluentValidation. We provide details on each of them and how to implement them in practice.

When developing a new application or even creating a new resource, it is important to remember that each type of validation has its advantages and disadvantages, so the use or not of each one must be based on the needs of each project.