Steps for Validations

In ASP.NET Core, there are various ways to perform validations on incoming data. Here's a brief explanation of the validation techniques available in ASP.NET Core:
Data annotations Data annotations are attributes that you can apply to properties in your model classes.
Model-Level Validation: through the IValidatableObject interface.
ModelState Validation: tracks the state of model binding and validation.
Fluent Validation: popular third-party library that provides a fluent and flexible way to define validation rules.
Custom Validation: ASP.NET Core allows you to implement custom validation logic by creating custom validation attributes or by using action filters.

Defining validation rules

Rules are defined through

- Data annotations (built-in or custom)
- IValiddatableObject

Focus on Input validation

Checking validation rules

Model State

- A dictionary containing both the state of the model binding validation
- Contains a collection of error messages for each property value submitted

Reporting validation errors

Response status should be 422 - Unprocessable entity

Response body should contain validation errors

- Problem details RFC

Data annotations

Data annotations are a set of attributes provided by the System.ComponentModel.DataAnnotations namespace in C#. They allow you to apply validation rules and constraints to your model properties in order to ensure data integrity and enforce business rules.

Some commonly used data annotations

Required: specifies that a property must have a non-null value.
StringLength: specifies the minimum and maximum length constraints for string properties.
Range: restricts numeric properties within a specific range. RegularExpression: applies a regular expression pattern to a string property.

EmailAddress: validates that a string has a valid email address format.
DataType: It helps ensure the property value is in the correct format.

Example of default error response

```
using System.ComponentModel.DataAnnotations;

public class User
{
    [Required(ErrorMessage = "The Name field is required.")]
    public string Name { get; set; }

    [StringLength(100, MinimumLength = 6, ErrorMessage = "The Password must be between 6 and 100 characters.")]
    public string Password { get; set; }

    [Range(18, 100, ErrorMessage = "The Age must be between 18 and 100.")]
    public int Age { get; set; }

    [RegularExpression(@"^[A-Za-z0-9]+(?:[._-][A-Za-z0-9]+)*@[A-Za-z0-9]+(?:[.-][A-Za-z0-9]+)*\.[A-Za-z]{2,}$",
                       ErrorMessage = "Invalid Email Address.")]
    public string Email { get; set; }

    [Compare("Password", ErrorMessage = "The Confirm Password does not match.")]
    public string ConfirmPassword { get; set; }

    [DataType(DataType.Currency)]
    public decimal Salary { get; set; }
}
```

Data annotations

Reporting validation errors

By default the APIController attribute will return a 400 Bad Request on validation errors, but there is a better approach to handle these type of errors by using 422 Unprocessable entity , the status code indicates that the server understands the request but cannot process it due to validation errors in the submitted data. It is a client error response code specifically used for reporting semantic validation errors. The response can report multiple validation errors if necessary, allowing clients to address all the issues at once.

```
.AddXmlDataContractSerializerFormatters()
.ConfigureApiBehaviorOptions(setupAction =>
{
    setupAction.InvalidModelStateResponseFactory = context =>
    {
        // create a validation problem details object
        var problemDetailsFactory = context.HttpContext.RequestServices
            .GetRequiredService<ProblemDetailsFactory>();

        var validationProblemDetails = problemDetailsFactory
            .CreateValidationProblemDetails(
                context.HttpContext,
                context.ModelState);
        validationProblemDetails.Detail =
            "See the errors field for details.";
        validationProblemDetails.Instance =
            context.HttpContext.Request.Path;

        // report invalid model state responses as validation issues
        validationProblemDetails.Type =
            "https://courselibrary.com/modelvalidationproblem";
        validationProblemDetails.Status =
            StatusCodes.Status422UnprocessableEntity;
        validationProblemDetails.Title =
            "One or more validation errors occurred.";

        return new UnprocessableEntityObjectResult(
            validationProblemDetails)
        {
            ContentTypes = { "application/problem+json" }
        };
    };
```

Setting up 422 as a validation error response type

```
Content-Type: application/problem+json

{
    "errors": {
        "title": [
            "The title shouldn't have more than 100 characters."
        ]   },
    "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
    "title": "One or more validation errors occurred.",
    "status": 400,
    "traceId": "00-f303f218c960a22c1d820b6478b4af5f-fd233fa633c4709e-00"
}
```

```
Content-Type: application/problem+json

{
   "errors": {
        "title": [
            "The title shouldn't have more than 100 characters."
        ]   },
    "type": "https://tools.ietf.org/html/rfc7231#section-6.5.1",
    "title": "One or more validation errors occurred.",
    "status": 422,
    "detail": "See the errors property for details.",
    "instance": "/api/authors/2902b665-1190-4c70-9915-b9c2d7680450/courses",
    "extensions": {
       "traceId": "0HLO3MNBSPFI2:00000001"
}}
```

422 vs 400 status code differences

Custom attributes

Allow you to define your own validation rules and apply them to properties or parameters in your code. They provide a way to extend the built-in data annotations and implement custom validation logic that aligns with your specific business requirements. To implement it you need to override the IsValid method to implement your custom validation logic, you can also override other methods like FormatErrorMessage to customize the error message displayed when the validation fails.

```
public class CourseTitleMustBeDifferentFromDescriptionAttribute
    : ValidationAttribute
{
    0 references
    public CourseTitleMustBeDifferentFromDescriptionAttribute()
    {
    }

    0 references
    protected override ValidationResult? IsValid(object? value,
        ValidationContext validationContext)
    {
        if (validationContext.ObjectInstance is not
            CourseForManipulationDto course)
        {
            throw new Exception($"Attribute " +
                $"{nameof(CourseTitleMustBeDifferentFromDescriptionAttribute)} " +
                $"must be applied to a " +
                $"{nameof(CourseForManipulationDto)} or derived type.");
        }

        if (course.Title == course.Description)
        {
            return new ValidationResult(
            "The provided description should be different from the title.",
                new[] { nameof(CourseForManipulationDto) });
        }

        return ValidationResult.Success;
    }
```

Creating a custom validation attribute

```
{
  "title" : "Same value",
  "description" : "Same value"
}
```

Body Cookies Headers (4) Test Results 422 Unprocessable Entity 74 ms 613 B Save Response

Pretty Raw Preview Visualize JSON

```
{
    "errors": {
        "courseForManipulationDto": [
            "The provided description should be different from the title."
        ]
    },
    "type": "https://courselibrary.com/modelvalidationproblem",
    "title": "One or more validation errors occurred.",
    "status": 422,
```

Custom attribute 422 error response

IValidatableObject

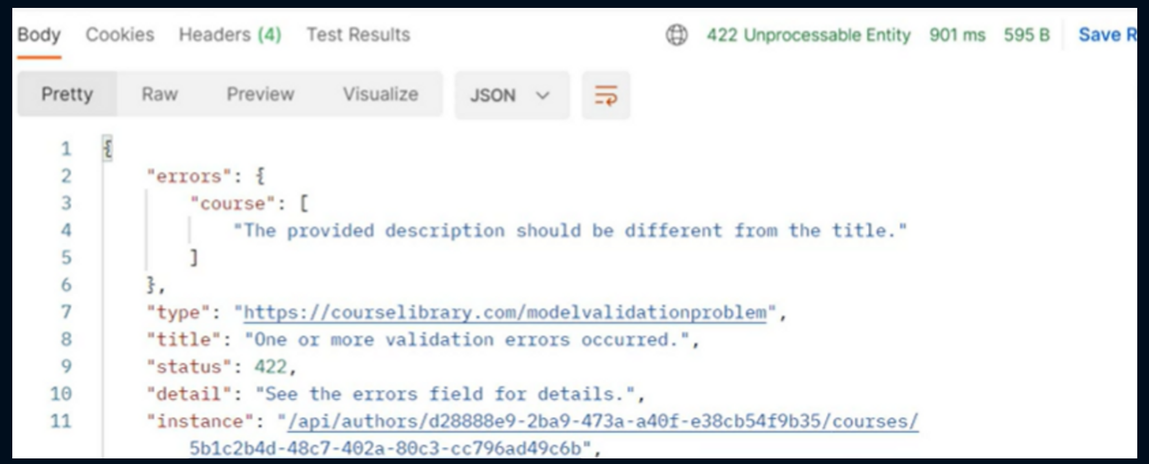
Is an interface in the .NET framework that allows you to implement custom validation logic on an object level. It provides a way to perform validation that involves multiple properties or complex validation rules that cannot be achieved using individual property-level data annotations.

```
public abstract class CourseForManipulationDto : IValidatableObject
{
    [Required(ErrorMessage = "You should fill out a title.")]
    [MaxLength(100, ErrorMessage = "The title shouldn't have more than 100 characte
    0 references
    public string Title { get; set; } = string.Empty;

    [MaxLength(1500, ErrorMessage = "The description shouldn't have more than 1500
    3 references
    public virtual string Description { get; set; } = string.Empty;

    0 references
    public IEnumerable<ValidationResult> Validate(
        ValidationContext validationContext)
    {
        if (Title == Description)
        {
            yield return new ValidationResult(
            "The provided description should be different from the title.",
            new[] { "Course" });
        }
    }
}
```

IValidatableObject implementation



422 error response from IValidatableObject validation

Fluent Validation overview

Is a popular third-party validation library for .NET applications. It provides a fluent and expressive way to define validation rules for your models, enabling you to perform robust and customizable data validation. Check more in the official docs

```
public class RegisterRequestValidator : AbstractValidator<RegisterRequest> {
    public RegisterRequestValidator() {
        RuleFor(x => x.Name).NotEmpty().Length(0, 200);
        RuleFor(x => x.Email).NotEmpty().Length(0, 150).EmailAddress();
        RuleFor(x => x.Address).NotNull().Length(0, 150);
    }
}
```

```
[HttpPut("{id}")]
public IActionResult EditPersonalInfo(long id, [FromBody] EditPersonalInfoRequest request)
{
    // Same validations here
    // Check that the student exists
    Student student = _studentRepository.GetById(id);

    var validator = new EditPersonalInfoRequestValidator();
    ValidationResult result = validator.Validate(request);

    if (result.IsValid == false)
    {
        return BadRequest(result.Errors[0].ErrorMessage);
    }

    student.EditPersonalInfo(request.Name, request.Address);
    _studentRepository.Save(student);

    return Ok();
}
```

Fluent validation example

Fluent Validation setup

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddMvc();

    // ... other configuration ...

    services.AddFluentValidationAutoValidation();

    services.AddScoped<IValidator<Person>, PersonValidator>();
}
```

This method must be called after `AddMvc` (or `AddControllers` / `AddControllersWithViews`). Make sure you add `using FluentValidation.AspNetCore` to your startup file so the appropriate extension methods are available.

```
public class Person
{
  public int Id { get; set; }
  public string Name { get; set; }
  public string Email { get; set; }
  public int Age { get; set; }
}

public class PersonValidator : AbstractValidator<Person>
{
  public PersonValidator()
  {
    RuleFor(x => x.Id).NotNull();
    RuleFor(x => x.Name).Length(0, 10);
    RuleFor(x => x.Email).EmailAddress();
    RuleFor(x => x.Age).InclusiveBetween(18, 60);
  }
}
```

```
[HttpPost]
public IActionResult Create(Person person)
{
  if(! ModelState.IsValid)
  {
    // re-render the view when validation failed.
    return View("Create", person);
  }

  Save(person); //Save the person to the database, or some other logic

  TempData["notice"] = "Person successfully created";
  return RedirectToAction("Index");
}
```

Setup from the official docs

Simple and complex validators

Fluent Validation offers a range of validators to handle different types of validations, including simple validators, collection validators, and complex validations.

- Simple validators such as NotEmpty, NotNull, GreaterThan, EmailAddress, StringLength, and many more.
- Collection validators are used to validate properties that are collections or arrays. (MustNotBeEmpty, Must Contain, etc)

```
public class RegisterRequestValidator : AbstractValidator<RegisterRequest>
{
    public RegisterRequestValidator()
    {
        RuleFor(x => x.Name).NotEmpty().Length(0, 200);
        RuleFor(x => x.Email).NotEmpty().Length(0, 150).EmailAddress();

        RuleFor(x => x.Address).NotNull().SetValidator(new AddressValidator());
    }
}

public class AddressValidator : AbstractValidator<AddressDto>
{
    public AddressValidator()
    {
        RuleFor(x => x.Street).NotEmpty().Length(0, 100);
        RuleFor(x => x.City).NotEmpty().Length(0, 40);
        RuleFor(x => x.State).NotEmpty().Length(0, 2);
        RuleFor(x => x.ZipCode).NotEmpty().Length(0, 5);
    }
}
```

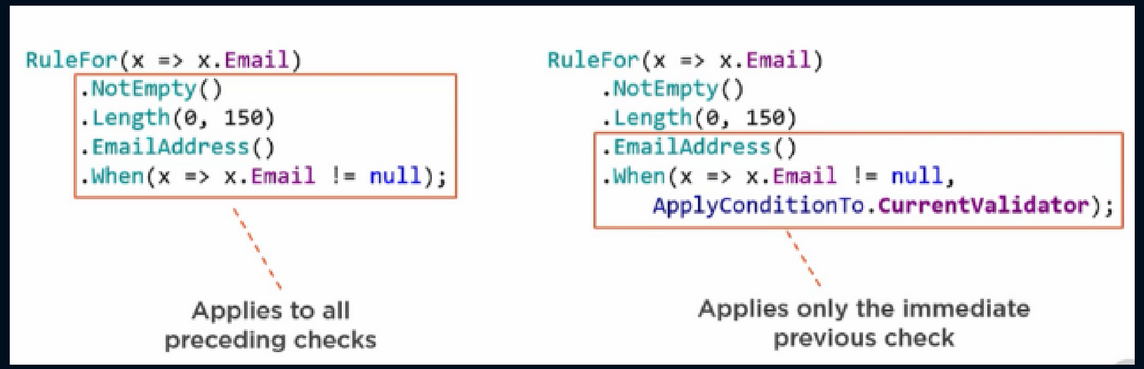
Simple validators and reuse a validator

```
RuleForEach(x => x.Addresses).ChildRules(address =>
{
    address.RuleFor(x => x.Street).NotEmpty().Length(0, 100);
    address.RuleFor(x => x.City).NotEmpty().Length(0, 40);
    address.RuleFor(x => x.State).NotEmpty().Length(0, 2);
    address.RuleFor(x => x.ZipCode).NotEmpty().Length(0, 5);
})
RuleFor(x => x.Addresses).NotNull()
    .Must(x => x?.Length >= 1 && x.Length <= 3)
    .WithMessage("The number of addresses must be between 1 and 3")
    .ForEach(x =>
    {
        x.NotNull();
        x.SetValidator(new AddressValidator());
    });
```

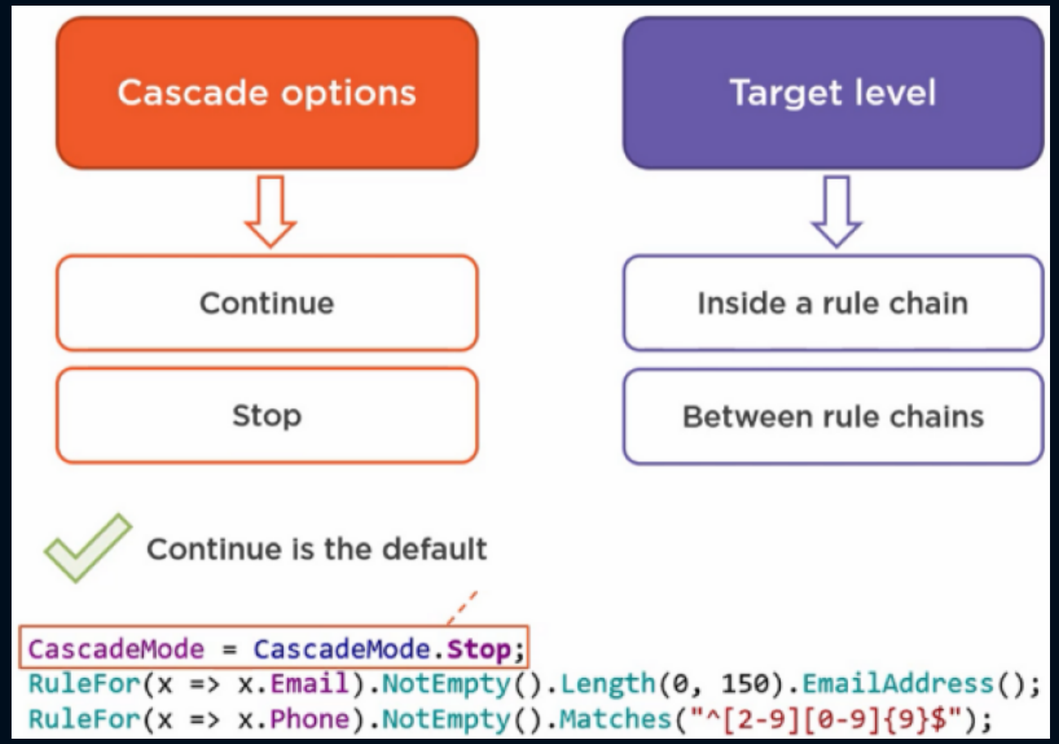
Collections validations

Conditional validation and cascade options

Fluent Validation provides conditional validation rules through the When method and cascade operators (CascadeMode) to control how validation rules are applied. This enables you to handle complex validation scenarios and ensure that validation rules are applied appropriately.



Using When for conditional validations



Cascade options

Versioning

ASP.NET Core Web API Versioning allows you to version your Web API endpoints to support different client versions and ensure backward compatibility as your API evolves over time. With versioning, you can make changes to your API without breaking existing client applications that rely on the previous version.

```
// URI Path
https://foo.org/api/v2/Customers
```

Versioning in the URI Path

Pros:

- Very clear to clients where the version is handled

Cons:

- Every version needs to change URIs, can be brittle

```
// Query String
https://foo.org/api/Customers?v=2.0
```

Versioning with Query String

Pros:

- Versioning is optionally included (can use default version)

Cons:

- Too easy for clients to miss needing the version

```
GET /api/camps HTTP/1.1
Host: localhost:44388
Content-Type: application/json
X-Version: 2.0
```

Versioning with Headers

Pros:

- Separates versioning from the rest of the API

Cons:

- Requires more sophisticated developer to manipulate headers

```
GET /api/camps HTTP/1.1
Host: localhost:44388
Content-Type: application/json
Accept: application/json;version=2.0
```

Versioning with Accept Header

Pros:

- No need to create your own custom header

Cons:

- Even less discoverable than query strings

```
GET /api/camps HTTP/1.1
Host: localhost:44388
Content-Type: application/vnd.yourapp.camp.v1+json
```

Versioning with Content Type

Pros:

- Can version the payload as well as the API call itself

Cons:

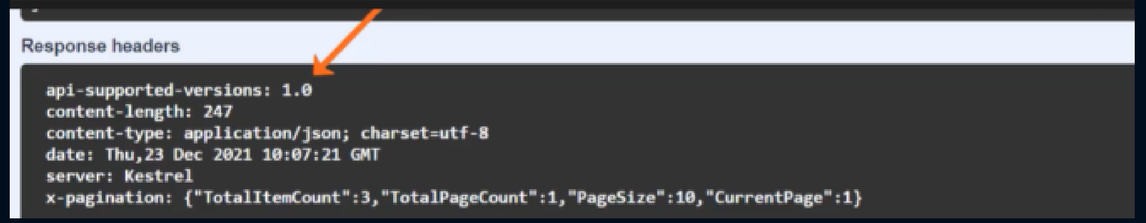
- Requires a lot more development maturity to create and maintain

Versioning

ASP.NET Core provides built-in support for versioning using the Microsoft.AspNetCore.Mvc.Versioning package. You can configure versioning in the Startup.cs file

```
builder.Services.AddApiVersioning(setupAction =>
{
    setupAction.AssumeDefaultVersionWhenUnspecified = true;
    setupAction.DefaultApiVersion = new Microsoft.AspNetCore.Mvc.ApiVersion(1, 0);
    setupAction.ReportApiVersions = true;
});

var app = builder.Build();
```



Setup

```
[Route("api/v{version:apiVersion}/cities/{cityId}/pointsofinterest")]
// [Authorize(Policy = "MustBeFromAntwerp")]
[ApiVersion("2.0")]
[ApiController]
```

```
[ApiController]
// [Authorize]
[ApiVersion("1.0")]
[Route("api/cities")]
1 reference
public class CitiesController : ControllerBase
{
```

Use

Demo

We will check the following:

Using Validations

Add validations to controllers

Overview of Fluent validation

You can check the following Repository for some examples:

C# fundamentals

Task

Exercise/ Homework

This is a continuation of the previous exercise. Add validations for the controllers implemented:

- Add validations for properties that are required or should be
- Add validations for the constraints of the properties, for example: the Name property should be of 200 characters
- Add validations to logic that is related to your application's business logic. For example: Goals can only live for one year, this means start and end date should be in that range

You can use your preferred way to handle validations but the recommendation is to go for **Fluent Validation**

Add versioning to your API