**Fluent API Validation in ASP.NET Core Web API**

Validation is a crucial aspect of any web application to ensure that the data received from the user is valid, consistent, and meets business rules. In ASP.NET Core Web API, validation can be implemented in multiple ways, including:

* **Data Annotations:** This involves decorating model properties with attributes such as [Required], [StringLength], etc. It is attribute-based validation directly on model properties.
* **Fluent Validation:** Writing validation rules in a separate class with a fluent interface. This is separating validation logic into dedicated classes.
* **Manual Validation:** Writing custom logic to validate data as needed in controllers or services.

**What is Fluent API Validation in ASP.NET Core?**

Fluent API Validation (commonly implemented using the **FluentValidation** library) is a programmatic (fluent) approach to defining validation rules for data models. Unlike Data Annotations, which use attributes directly on model properties (e.g., [Required] or [StringLength]), Fluent API Validation separates validation logic into a dedicated validator class (a class derived from **AbstractValidator<T>**), providing flexibility, maintainability, and a more organized codebase.

**When Should We Use Fluent API Validation in ASP.NET Core Web API?**

Fluent API Validation is particularly useful in the following scenarios:

* **Complex Validation Rules:** If a model’s validation depends on other properties or external conditions, Fluent Validation makes these complex scenarios more manageable. Suppose you have a product that belongs to a luxury category. For such products, you want to enforce that the Price must be above $500, and if a Discount is applied, it must not exceed 10% of the price. This multi-property and conditional logic can be implemented neatly using Fluent Validation.
* **Separation of Concerns:** Imagine you have a Product model used in various parts of our application (e.g., creation, update, listing). Instead of adding validation logic directly into the model using Data Annotation Attributes, we can create separate validator classes. This separation keeps our models clean and makes maintaining or testing the validation logic easier.
* **Dynamic or Conditional Validation:** Validation rules can change based on runtime conditions or business logic. Fluent Validation allows adding conditions that trigger specific validation rules at runtime. For example, an order’s shipping date is validated only if its status is Shipped.
* **Reusable Validation Rules:** When the same validation logic applies across multiple models or controllers, encapsulating the rules in a validator class makes them easily reusable. Suppose multiple models (User, Admin, Vendor) share an Email property with the same validation rules. You can create a reusable validator rule snippet or a separate EmailValidator and invoke that on each model’s validator rather than repeating the same EmailAddress() checks everywhere.

**Basic Validation Fluent API Methods:**

* **NotEmpty():** Validates that a property is not null, empty, or whitespace.
* **NotNull():** Validates that a property is not null.
* **Length(min, max):** Ensures that a string property’s length falls between the specified minimum and maximum values.
* **InclusiveBetween(min, max):** Confirms that a property’s value is within a defined range, including the boundary values.
* **ExclusiveBetween(min, max):** Confirms that a property’s value falls within a defined range while excluding the specified boundary values.
* **GreaterThan(value):** Validates that the property’s value is strictly greater than the given value.
* **GreaterThanOrEqualTo(value):** Ensures the property’s value is greater than or equal to the specified limit.
* **LessThan(value):** Checks that the property’s value is strictly less than the given value.
* **LessThanOrEqualTo(value):** Verifies that the property’s value is less than or equal to the specified limit.

**String Validation Fluent API Methods:**

* **Matches(regex):** Validates that the string property conforms to the pattern defined by the regular expression.
* **EmailAddress():** Ensures that the string property is in a valid email address format.
* **MaximumLength(max):** Validates that the string property does not exceed the specified maximum length.
* **MinimumLength(min):** Checks that the string property meets the minimum length requirement.
* **NotEqual(value):** Ensures the property’s value does not equal the specified disallowed value.
* **StartsWith(value):** Validates that the string property begins with the specified substring.
* **EndsWith(value):** Validates that the string property ends with the specified substring.

**Custom Validation Fluent API Methods:**

* **Must(condition):** Applies a custom synchronous condition to validate a property.
* **MustAsync(asyncCondition):** This applies a custom asynchronous condition to validate a property.
* **WithMessage(message):** Specifies a custom error message to be returned when a validation rule fails.
* **Custom():** Enables the creation of complex validations that may involve multiple properties without asynchronous operations.
* **CustomAsync():** Enables the creation of complex validations that require asynchronous operations like IO tasks.

**Conditional Validation Fluent API Methods:**

* **When():** Applies validation rules conditionally when a specified condition is evaluated as true.
* **Unless():** Applies validation rules conditionally when a specified condition evaluates to false.

**Chaining Rules:**

* Allows combining multiple validation rules in a fluent, readable sequence for a single property, e.g., **NotEmpty().WithMessage(“Required”).Length(5, 50).WithMessage(“The length must be between 5 and 50 characters”).**

**Validation for Complex Types:**

* **SetValidator():** Delegates the validation of a nested object or child properties to another dedicated validator.

**Collection Validation:**

* **ForEach():**Applies specified validation rules to every individual element within a collection property.

**Implementing Fluent API Validation in ASP.NET Core Web API**

Let’s Understand How to Implement Fluent API Validation in ASP.NET Core Web API. We will create a simple application to validate a Product model using the EF Core Database First approach.

**Create a New ASP.NET Core Web API Project and Install Required NuGet Packages**

First, create a new ASP.NET Core Web API Project named **FluentAPIValidationDemo** and install the following Packages required for Fluent API validation and Entity Framework core. You can install the packages using the Package Manager Console by executing the following commands:

* **Install-Package Microsoft.EntityFrameworkCore.SqlServer**
* **Install-Package Microsoft.EntityFrameworkCore.Tools**
* **Install-Package FluentValidation.AspNetCore**

**Create the Product Model**

First, create a folder named **Models** in the project root directory where we will create our Models. This folder contains entity classes representing our application’s data. Keeping these classes in one folder helps maintain a clean project structure.

**Product Model**

Create a class file named **Product.cs** within the Models folder as below,   This model now uses many‑to‑many relationships with the Tag entity.

using System.ComponentModel.DataAnnotations.Schema;

namespace FluentAPIValidationDemo.Models

{

public class Product

{

public int ProductId { get; set; }

// Stock Keeping Unit following a specific pattern (e.g., 8 uppercase letters/digits)

public string SKU { get; set; }

public string Name { get; set; }

[Column(TypeName = "decimal(8,2)")]

public decimal Price { get; set; }

public int Stock { get; set; }

public int CategoryId { get; set; }

public string? Description { get; set; }

// Discount percentage (0-100)

[Column(TypeName ="decimal(18,2)")]

public decimal Discount { get; set; }

// Manufacturing date (should not be in the future)

public DateTime ManufacturingDate { get; set; }

// Expiry date (must be after the manufacturing date)

public DateTime ExpiryDate { get; set; }

// Many-to-many relationship with Tag

public ICollection<Tag> Tags { get; set; } = new List<Tag>();

}

}

**Tag Model**

Each tag is stored as a separate entity. Products and Tags are connected via many‑to‑many relationships. So, create a class file named **Tag.cs** within the Models folder and copy and paste the following code.

**namespace** *FluentAPIValidationDemo.Models*

**{**

**public** **class** Tag

**{**

**public** **int** TagId **{** **get**; **set**; **}**

**public** **string** Name **{** **get**; **set**; **}**

// Many-to-many relationship with Product

**public** ICollection**<**Product**>** Products **{** **get**; **set**; **}** = new List**<**Product**>()**;

**}**

**}**

**Real-time Usage of Tags:**

Tags are metadata that can be applied to products to enhance searchability, categorization, and user experience in real time. The following are a few real-world scenarios where tags prove valuable:

* **Enhanced Search and Filtering:** Tags allow users to quickly filter products by attributes or keywords (e.g., “gaming,” “wireless,” “eco-friendly”). When users search on an e-commerce site, tags help refine results and display relevant products.
* **Dynamic Categorization:** Instead of relying solely on category structures, tags offer a flexible way to group products. For example, a laptop might be tagged as both “ultrabook” and “gaming,” enabling it to appear in multiple sections based on user interests.
* **Personalized Recommendations:** Analyzing tags can help create personalized product recommendations. If users often interact with products tagged “organic” or “sustainable,” the platform can highlight similar products.

**DbContext Class:**

First, create a folder named **Data** in the project root directory. Then, add a class file named **ECommerceDbContext.cs** within the Data folder and copy and paste the following code. Entity Framework Core uses the DbContext class to interact with the database. It maps your domain models to database tables, configures relationships, and handles migrations. Using the Fluent API, this context class configures the many‑to‑many relationships between **Product** and **Tag**.

**using** *FluentAPIValidationDemo.Models;*

**using** *Microsoft.EntityFrameworkCore;*

**namespace** *FluentAPIValidationDemo.Data*

**{**

**public** **class** ECommerceDbContext : DbContext

**{**

**public** ECommerceDbContext**(**DbContextOptions**<**ECommerceDbContext**>** options**)** : **base(**options**)** **{** **}**

**public** DbSet**<**Product**>** Products **{** **get**; **set**; **}**

**public** DbSet**<**Tag**>** Tags **{** **get**; **set**; **}**

**protected** **override** **void** OnModelCreating**(**ModelBuilder modelBuilder**)**

**{**

// Configure many-to-many relationship between Product and Tag using an implicit join table "ProductTag"

modelBuilder.Entity**<**Product**>()**

.HasMany**(**p =**>** p.Tags**)**

.WithMany**(**t =**>** t.Products**)**

//Tells EFCore, create a table named ProductTag with TagId and ProductId as columns,

//but don’t bother me with an explicit class.

.UsingEntity**<**Dictionary**<string**, **object>>(**

"ProductTag",

pt =**>** pt.HasOne**<**Tag**>()**.WithMany**()**.HasForeignKey**(**"TagId"**)**,

pt =**>** pt.HasOne**<**Product**>()**.WithMany**()**.HasForeignKey**(**"ProductId"**))**;

// Seed Tags (one record per line)

modelBuilder.Entity**<**Tag**>()**.HasData**(**

new Tag **{** TagId = 1, Name = "electronics" **}**,

new Tag **{** TagId = 2, Name = "gaming" **}**,

new Tag **{** TagId = 3, Name = "office" **}**,

new Tag **{** TagId = 4, Name = "accessories" **}**,

new Tag **{** TagId = 5, Name = "home" **}**

**)**;

// Seed Products (one record per line)

modelBuilder.Entity**<**Product**>()**.HasData**(**

new Product **{** ProductId = 1, SKU = "GAM12345", Name = "Gaming Laptop", Price = 1500.00m, Stock = 10, CategoryId = 1, Description = "High performance gaming laptop.", Discount = 10, ManufacturingDate = new DateTime**(**2023, 1, 1**)**, ExpiryDate = new DateTime**(**2024, 1, 1**)** **}**,

new Product **{** ProductId = 2, SKU = "OFF12345", Name = "Office Desktop", Price = 800.00m, Stock = 20, CategoryId = 1, Description = "Efficient desktop for office work.", Discount = 5, ManufacturingDate = new DateTime**(**2023, 1, 1**)**, ExpiryDate = new DateTime**(**2024, 1, 1**)** **}**,

new Product **{** ProductId = 3, SKU = "SMA12345", Name = "Smartphone", Price = 700.00m, Stock = 50, CategoryId = 2, Description = "Latest model smartphone.", Discount = 0, ManufacturingDate = new DateTime**(**2023, 1, 1**)**, ExpiryDate = new DateTime**(**2024, 1, 1**)** **}**,

new Product **{** ProductId = 4, SKU = "WIR12345", Name = "Wireless Mouse", Price = 50.00m, Stock = 100, CategoryId = 3, Description = "Ergonomic wireless mouse.", Discount = 15, ManufacturingDate = new DateTime**(**2023, 1, 1**)**, ExpiryDate = new DateTime**(**2024, 1, 1**)** **}**,

new Product **{** ProductId = 5, SKU = "MEC12345", Name = "Mechanical Keyboard", Price = 120.00m, Stock = 75, CategoryId = 3, Description = "RGB mechanical keyboard.", Discount = 20, ManufacturingDate = new DateTime**(**2023, 1, 1**)**, ExpiryDate = new DateTime**(**2024, 1, 1**)** **}**,

new Product **{** ProductId = 6, SKU = "4KMON12", Name = "4K Monitor", Price = 400.00m, Stock = 30, CategoryId = 4, Description = "Ultra HD 4K monitor.", Discount = 5, ManufacturingDate = new DateTime**(**2023, 1, 1**)**, ExpiryDate = new DateTime**(**2024, 1, 1**)** **}**,

new Product **{** ProductId = 7, SKU = "GAMCHAIR", Name = "Gaming Chair", Price = 300.00m, Stock = 15, CategoryId = 4, Description = "Ergonomic gaming chair.", Discount = 10, ManufacturingDate = new DateTime**(**2023, 1, 1**)**, ExpiryDate = new DateTime**(**2024, 1, 1**)** **}**,

new Product **{** ProductId = 8, SKU = "BLU12345", Name = "Bluetooth Speaker", Price = 150.00m, Stock = 40, CategoryId = 5, Description = "Portable Bluetooth speaker.", Discount = 0, ManufacturingDate = new DateTime**(**2023, 1, 1**)**, ExpiryDate = new DateTime**(**2024, 1, 1**)** **}**,

new Product **{** ProductId = 9, SKU = "SMAW1234", Name = "Smartwatch", Price = 250.00m, Stock = 25, CategoryId = 2, Description = "Feature-packed smartwatch.", Discount = 5, ManufacturingDate = new DateTime**(**2023, 1, 1**)**, ExpiryDate = new DateTime**(**2024, 1, 1**)** **}**,

new Product **{** ProductId = 10, SKU = "HOMECAM1", Name = "Home Security Camera", Price = 100.00m, Stock = 60, CategoryId = 5, Description = "HD home security camera.", Discount = 10, ManufacturingDate = new DateTime**(**2023, 1, 1**)**, ExpiryDate = new DateTime**(**2024, 1, 1**)** **}**

**)**;

// Seed join table for ProductTag (each record provided in the same line)

modelBuilder.Entity**(**"ProductTag"**)**.HasData**(**

new **{** ProductId = 1, TagId = 1 **}**, new **{** ProductId = 1, TagId = 2 **}**,

new **{** ProductId = 2, TagId = 1 **}**, new **{** ProductId = 2, TagId = 3 **}**,

new **{** ProductId = 3, TagId = 1 **}**,

new **{** ProductId = 4, TagId = 4 **}**,

new **{** ProductId = 5, TagId = 4 **}**, new **{** ProductId = 5, TagId = 3 **}**,

new **{** ProductId = 6, TagId = 1 **}**, new **{** ProductId = 6, TagId = 3 **}**,

new **{** ProductId = 7, TagId = 2 **}**, new **{** ProductId = 7, TagId = 3 **}**,

new **{** ProductId = 8, TagId = 1 **}**, new **{** ProductId = 8, TagId = 4 **}**,

new **{** ProductId = 9, TagId = 1 **}**, new **{** ProductId = 9, TagId = 4 **}**,

new **{** ProductId = 10, TagId = 1 **}**, new **{** ProductId = 10, TagId = 5 **}**

**)**;

**}**

**}**

**}**

**Why Use Dictionary<string, object> while Configuring the Joining table?**

We use Dictionary<string, object> here to define a shadow entity for the join table without creating an explicit CLR class for it. By specifying Dictionary<string, object>, we indicate that the join table (named “ProductTag”) doesn’t have its own dedicated entity class. It’s just a table with the foreign keys (ProductId and TagId).

* **No Explicit Entity Needed:** It allows us to configure the join table without writing a separate ProductTag class.
* **Shadow Properties:** The keys (e.g., “ProductId”, “TagId”) become shadow properties that EF manages internally.

This approach can be used for simple many-to-many relationships where extra fields on the join table are not needed.

**Creating DTOs**

Create a folder named DTOs in the Project root directory. A DTO (Data Transfer Object) transfers data between the client and server. It often contains a subset of the entity’s properties or additional properties for validation that should not be persisted directly. In our example, we will define:

* **ProductCreateDTO**for creation,
* **ProductUpdateDTO** for updating, and
* **ProductResponseDTO** for sending data back to clients.

**ProductCreateDTO**

So, create a class file named **ProductCreateDTO.cs** within the DTOs folder and copy and paste the following code. We will use this DTO to create a new Product.

**namespace** *FluentAPIValidationDemo.DTOs*

**{**

**public** **class** ProductCreateDTO

**{**

**public** **string** SKU **{** **get**; **set**; **}**

**public** **string** Name **{** **get**; **set**; **}**

**public** **decimal** Price **{** **get**; **set**; **}**

**public** **int** Stock **{** **get**; **set**; **}**

**public** **int** CategoryId **{** **get**; **set**; **}**

**public** **string**? Description **{** **get**; **set**; **}**

**public** **decimal** Discount **{** **get**; **set**; **}**

**public** DateTime ManufacturingDate **{** **get**; **set**; **}**

**public** DateTime ExpiryDate **{** **get**; **set**; **}**

// List of tag names (will be normalized and stored in the Tags table)

**public** List**<string>**? Tags **{** **get**; **set**; **}**

**}**

**}**

**ProductUpdateDTO**

So, create a class file named **ProductUpdateDTO.cs** within the DTOs folder and copy and paste the following code. We will use this DTO to update an existing Product.

**namespace** *FluentAPIValidationDemo.DTOs*

**{**

**public** **class** ProductUpdateDTO

**{**

**public** **int** ProductId **{** **get**; **set**; **}**

**public** **string** SKU **{** **get**; **set**; **}**

**public** **string** Name **{** **get**; **set**; **}**

**public** **decimal** Price **{** **get**; **set**; **}**

**public** **int** Stock **{** **get**; **set**; **}**

**public** **int** CategoryId **{** **get**; **set**; **}**

**public** **string**? Description **{** **get**; **set**; **}**

**public** **decimal** Discount **{** **get**; **set**; **}**

**public** DateTime ManufacturingDate **{** **get**; **set**; **}**

**public** DateTime ExpiryDate **{** **get**; **set**; **}**

**public** List**<string>**? Tags **{** **get**; **set**; **}**

**}**

**}**

**ProductResponseDTO**

So, create a class file named **ProductResponseDTO.cs** within the DTOs folder and copy and paste the following code. We will use this DTO to return Product details.

**using** *System;*

**using** *System.Collections.Generic;*

**namespace** *FluentAPIValidationDemo.DTOs*

**{**

**public** **class** ProductResponseDTO

**{**

**public** **int** ProductId **{** **get**; **set**; **}**

**public** **string** SKU **{** **get**; **set**; **}**

**public** **string** Name **{** **get**; **set**; **}**

**public** **decimal** Price **{** **get**; **set**; **}**

**public** **int** Stock **{** **get**; **set**; **}**

**public** **int** CategoryId **{** **get**; **set**; **}**

**public** **string**? Description **{** **get**; **set**; **}**

**public** **decimal** Discount **{** **get**; **set**; **}**

**public** DateTime ManufacturingDate **{** **get**; **set**; **}**

**public** DateTime ExpiryDate **{** **get**; **set**; **}**

**public** List**<string>**? Tags **{** **get**; **set**; **}**

**}**

**}**

**Add Fluent Validation for the Product Model:**

First, create a new folder named **Validators** in the project root directory. This folder contains classes inherited from AbstractValidator<T> to define validation rules separate from the model classes.

**ProductCreateDTOValidator:**

Add a class file named **ProductCreateDTOValidator.cs** within the **Validators** folder and copy and paste the following. The following class is inherited from the **AbstractValidator<T>** class where T is the model being validated and, in our example, it is the ProductCreateDTO. Inside the parameterless constructor of this class, we need to write the ProductCreateDTO model validation rules.

**using** *FluentAPIValidationDemo.DTOs;*

**using** *FluentValidation;*

**namespace** *FluentAPIValidationDemo.Validators*

**{**

// Validator for ProductCreateDTO, which contains the rules for creating a product.

**public** **class** ProductCreateDTOValidator : AbstractValidator**<**ProductCreateDTO**>**

**{**

**public** ProductCreateDTOValidator**()**

**{**

// SKU validation: must not be empty and must match the specified regex pattern.

RuleFor**(**p =**>** p.SKU**)**

.NotEmpty**()**.WithMessage**(**"SKU is required."**)** // Ensures SKU is provided.

.Matches**(**"^[A-Z0-9]{8}$"**)**.WithMessage**(**"SKU must be 8 characters long and contain only uppercase letters and digits."**)**; // Must be exactly 8 uppercase letters or digits.

// Name validation: must not be empty and length must be between 3 and 50 characters.

RuleFor**(**p =**>** p.Name**)**

.NotEmpty**()**.WithMessage**(**"Product name is required."**)** // Ensures product name is provided.

.Length**(**3, 50**)**.WithMessage**(**"Product name must be between 3 and 50 characters."**)**; // Validates the length of the product name.

// Price validation: must be greater than 0 and conform to precision and scale.

RuleFor**(**p =**>** p.Price**)**

.GreaterThan**(**0**)**.WithMessage**(**"Price must be greater than 0."**)** // Price must be positive.

.PrecisionScale**(**8, 2, **true)**.WithMessage**(**"Price must have at most 8 digits in total and 2 decimals."**)**; // Validates total digits and decimals (ignoring trailing zeros).

// Stock validation: must be zero or positive.

RuleFor**(**p =**>** p.Stock**)**

.GreaterThanOrEqualTo**(**0**)**.WithMessage**(**"Stock cannot be negative."**)**; // Stock can't be negative.

// CategoryId validation: must be a positive number.

RuleFor**(**p =**>** p.CategoryId**)**

.GreaterThan**(**0**)**.WithMessage**(**"Category ID is required."**)**; // Ensures a valid category is selected.

// Description validation: if provided, must not exceed 500 characters.

RuleFor**(**p =**>** p.Description**)**

.MaximumLength**(**500**)**.WithMessage**(**"Description cannot exceed 500 characters."**)** // Limits description length.

.When**(**p =**>** !**string**.IsNullOrEmpty**(**p.Description**))**; // Only applies rule if Description is provided.

// Discount validation: must be between 0 and 100.

RuleFor**(**p =**>** p.Discount**)**

.InclusiveBetween**(**0, 100**)**.WithMessage**(**"Discount must be between 0 and 100."**)**; // Ensures discount is a valid percentage.

// Manufacturing date validation: must not be a future date.

RuleFor**(**p =**>** p.ManufacturingDate**)**

.LessThanOrEqualTo**(**DateTime.Now**)**.WithMessage**(**"Manufacturing date cannot be in the future."**)**; // Date must be in the past or today.

// Expiry date validation: must be later than the manufacturing date.

RuleFor**(**p =**>** p.ExpiryDate**)**

.GreaterThan**(**p =**>** p.ManufacturingDate**)**.WithMessage**(**"Expiry date must be after the manufacturing date."**)**; // Validates date order.

// Tags validation: for each tag in the list, ensure it is not empty and doesn't exceed 20 characters.

RuleForEach**(**p =**>** p.Tags**)**.ChildRules**(**tag =**>**

**{**

tag.RuleFor**(**t =**>** t**)**

.NotEmpty**()**.WithMessage**(**"Tag cannot be empty."**)** // Ensures each tag is provided.

.MaximumLength**(**20**)**.WithMessage**(**"Tag cannot exceed 20 characters."**)**; // Limits tag length.

**})**;

**}**

**}**

**}**

**ProductUpdateDTOValidator:**

Add a class file named **ProductUpdateDTOValidator.cs** within the **Validators** folder and copy and paste the following. The following class is inherited from the **AbstractValidator<T>** class where T is the model being validated and, in our example, it is the ProductUpdateDTO. Inside the parameterless constructor of this class, we need to write the ProductUpdateDTO model validation rules.

**using** *FluentAPIValidationDemo.DTOs;*

**using** *FluentValidation;*

**namespace** *FluentAPIValidationDemo.Validators*

**{**

// Validator for ProductCreateDTO, which contains the rules for creating a product.

**public** **class** ProductUpdateDTOValidator : AbstractValidator**<**ProductUpdateDTO**>**

**{**

**public** ProductUpdateDTOValidator**()**

**{**

// ProductId validation: must be greater than 0.

RuleFor**(**p =**>** p.ProductId**)**

.GreaterThan**(**0**)**.WithMessage**(**"ProductId Must be Greater than 0"**)**; // Validate ProductId.

// SKU validation: must not be empty and must match the specified regex pattern.

RuleFor**(**p =**>** p.SKU**)**

.NotEmpty**()**.WithMessage**(**"SKU is required."**)** // Ensures SKU is provided.

.Matches**(**"^[A-Z0-9]{8}$"**)**.WithMessage**(**"SKU must be 8 characters long and contain only uppercase letters and digits."**)**; // Must be exactly 8 uppercase letters or digits.

// Name validation: must not be empty and length must be between 3 and 50 characters.

RuleFor**(**p =**>** p.Name**)**

.NotEmpty**()**.WithMessage**(**"Product name is required."**)** // Ensures product name is provided.

.Length**(**3, 50**)**.WithMessage**(**"Product name must be between 3 and 50 characters."**)**; // Validates the length of the product name.

// Price validation: must be greater than 0 and conform to precision and scale.

RuleFor**(**p =**>** p.Price**)**

.GreaterThan**(**0**)**.WithMessage**(**"Price must be greater than 0."**)** // Price must be positive.

.PrecisionScale**(**8, 2, **true)**.WithMessage**(**"Price must have at most 8 digits in total and 2 decimals."**)**; // Validates total digits and decimals (ignoring trailing zeros).

// Stock validation: must be zero or positive.

RuleFor**(**p =**>** p.Stock**)**

.GreaterThanOrEqualTo**(**0**)**.WithMessage**(**"Stock cannot be negative."**)**; // Stock can't be negative.

// CategoryId validation: must be a positive number.

RuleFor**(**p =**>** p.CategoryId**)**

.GreaterThan**(**0**)**.WithMessage**(**"Category ID is required."**)**; // Ensures a valid category is selected.

// Description validation: if provided, must not exceed 500 characters.

RuleFor**(**p =**>** p.Description**)**

.MaximumLength**(**500**)**.WithMessage**(**"Description cannot exceed 500 characters."**)** // Limits description length.

.When**(**p =**>** !**string**.IsNullOrEmpty**(**p.Description**))**; // Only applies rule if Description is provided.

// Discount validation: must be between 0 and 100.

RuleFor**(**p =**>** p.Discount**)**

.InclusiveBetween**(**0, 100**)**.WithMessage**(**"Discount must be between 0 and 100."**)**; // Ensures discount is a valid percentage.

// Manufacturing date validation: must not be a future date.

RuleFor**(**p =**>** p.ManufacturingDate**)**

.LessThanOrEqualTo**(**DateTime.Now**)**.WithMessage**(**"Manufacturing date cannot be in the future."**)**; // Date must be in the past or today.

// Expiry date validation: must be later than the manufacturing date.

RuleFor**(**p =**>** p.ExpiryDate**)**

.GreaterThan**(**p =**>** p.ManufacturingDate**)**.WithMessage**(**"Expiry date must be after the manufacturing date."**)**; // Validates date order.

// Tags validation: for each tag in the list, ensure it is not empty and doesn't exceed 20 characters.

RuleForEach**(**p =**>** p.Tags**)**.ChildRules**(**tag =**>**

**{**

tag.RuleFor**(**t =**>** t**)**

.NotEmpty**()**.WithMessage**(**"Tag cannot be empty."**)** // Ensures each tag is provided.

.MaximumLength**(**20**)**.WithMessage**(**"Tag cannot exceed 20 characters."**)**; // Limits tag length.

**})**;

**}**

**}**

**}**

**Create the Products Controller**

The ProductsController handles HTTP requests related to product operations (such as creating, retrieving, and updating products). It is the intermediary between client requests and the underlying business logic and data access layer. The controller demonstrates:

* **Create and Update** endpoints that use DTOs and FluentValidation.
* A **GET** endpoint that filters products based on a comma‑separated list of tags.
* Proper mapping between DTOs and domain models with tag processing.

So, create an Empty API Controller named **ProductsController** within the Controllers folder and copy and paste the following code.

**using** *FluentAPIValidationDemo.Data;*

**using** *FluentAPIValidationDemo.DTOs;*

**using** *FluentAPIValidationDemo.Models;*

**using** *Microsoft.AspNetCore.Mvc;*

**using** *Microsoft.EntityFrameworkCore;*

**namespace** *FluentAPIValidationDemo.Controllers*

**{**

// The ProductsController handles all product-related API endpoints.

**[**Route**(**"api/[controller]"**)]**

**[**ApiController**]**

**public** **class** ProductsController : ControllerBase

**{**

// The database context used to interact with the underlying database.

**private** **readonly** ECommerceDbContext \_context;

// Constructor injection of the database context.

**public** ProductsController**(**ECommerceDbContext context**)**

**{**

\_context = context;

**}**

// GET: api/products?tags=tag1,tag2

// Retrieves all products, optionally filtered by any matching tags.

**[**HttpGet**]**

**public** **async** Task**<**ActionResult**<**ProductResponseDTO**>>** GetProducts**([**FromQuery**]** **string**? tags**)**

**{**

// Build the query and include the related Tags collection for each product.

IQueryable**<**Product**>** query = \_context.Products

.AsNoTracking**()**

.Include**(**p =**>** p.Tags**)**;

**if** **(**!**string**.IsNullOrEmpty**(**tags**))**

**{**

// Split the comma-separated tags string into a list.

// Trim spaces and convert each tag to lower case for case-insensitive comparison.

var tagList = tags.Split**(**',', StringSplitOptions.RemoveEmptyEntries**)**

.Select**(**t =**>** t.Trim**()**.ToLower**())**

.ToList**()**;

// Filter products that contain ANY of the specified tags.

// If a product has at least one matching tag, it is included in the result.

query = query.Where**(**p =**>** p.Tags.Any**(**t =**>** tagList.Contains**(**t.Name.ToLower**())))**;

**}**

// Execute the query and retrieve the list of products from the database.

var products = **await** query.ToListAsync**()**;

// Map each Product entity to a ProductResponseDTO.

var result = products.Select**(**p =**>** new ProductResponseDTO

**{**

ProductId = p.ProductId,

SKU = p.SKU,

Name = p.Name,

Price = p.Price,

Stock = p.Stock,

CategoryId = p.CategoryId,

Description = p.Description,

Discount = p.Discount,

ManufacturingDate = p.ManufacturingDate,

ExpiryDate = p.ExpiryDate,

// Map the Tags collection to a list of tag names.

Tags = p.Tags.Select**(**t =**>** t.Name**)**.ToList**()**

**})**.ToList**()**;

// Return the filtered list of products with an HTTP 200 OK status.

**return** Ok**(**result**)**;

**}**

// GET: api/products/{id}

// Retrieves a single product by its ID.

**[**HttpGet**(**"{id}"**)]**

**public** **async** Task**<**ActionResult**<**ProductResponseDTO**>>** GetProduct**(int** id**)**

**{**

// Retrieve the product with the given ID, including its Tags.

// AsNoTracking() is used here since no update is needed (improves performance).

var product = **await** \_context.Products

.AsNoTracking**()**

.Include**(**p =**>** p.Tags**)**

.FirstOrDefaultAsync**(**p =**>** p.ProductId == id**)**;

// If the product is not found, return a 404 Not Found response.

**if** **(**product == **null)**

**return** NotFound**()**;

// Map the Product entity to a ProductResponseDTO.

var response = new ProductResponseDTO

**{**

ProductId = product.ProductId,

SKU = product.SKU,

Name = product.Name,

Price = product.Price,

Stock = product.Stock,

CategoryId = product.CategoryId,

Description = product.Description,

Discount = product.Discount,

ManufacturingDate = product.ManufacturingDate,

ExpiryDate = product.ExpiryDate,

Tags = product.Tags.Select**(**t =**>** t.Name**)**.ToList**()**

**}**;

// Return the product details with an HTTP 200 OK status.

**return** Ok**(**response**)**;

**}**

// POST: api/products

// Creates a new product based on the data provided in ProductCreateDTO.

**[**HttpPost**]**

**public** **async** Task**<**ActionResult**<**ProductResponseDTO**>>** CreateProduct**([**FromBody**]** ProductCreateDTO productDto**)**

**{**

// Validate the incoming DTO; if invalid, return a 400 Bad Request with the validation errors.

**if** **(**!ModelState.IsValid**)**

**return** BadRequest**(**ModelState**)**;

// Map the incoming DTO to a new Product entity.

var product = new Product

**{**

SKU = productDto.SKU,

Name = productDto.Name,

Price = productDto.Price,

Stock = productDto.Stock,

CategoryId = productDto.CategoryId,

Description = productDto.Description,

Discount = productDto.Discount,

ManufacturingDate = productDto.ManufacturingDate,

ExpiryDate = productDto.ExpiryDate

**}**;

// Process Tags: For each tag provided in the DTO, check if it exists.

// If the tag exists, use the existing Tag; otherwise, create a new Tag.

**if** **(**productDto.Tags != **null** && productDto.Tags.Any**())**

**{**

**foreach** **(**var tagName in productDto.Tags**)**

**{**

// Normalize the tag by trimming whitespace and converting to lower case.

var normalizedTagName = tagName.Trim**()**.ToLower**()**;

var existingTag = **await** \_context.Tags.FirstOrDefaultAsync**(**t =**>** t.Name.ToLower**()** == normalizedTagName**)**;

**if** **(**existingTag != **null)**

product.Tags.Add**(**existingTag**)**; // Associate the existing tag with the product.

**else**

product.Tags.Add**(**new Tag **{** Name = normalizedTagName **})**; // Create a new tag and associate it.

**}**

**}**

// Add the new product to the database context.

\_context.Products.Add**(**product**)**;

// Save changes to persist the new product (and associated tags) to the database.

**await** \_context.SaveChangesAsync**()**;

// Map the newly created Product entity to a ProductResponseDTO.

var response = new ProductResponseDTO

**{**

ProductId = product.ProductId,

SKU = product.SKU,

Name = product.Name,

Price = product.Price,

Stock = product.Stock,

CategoryId = product.CategoryId,

Description = product.Description,

Discount = product.Discount,

ManufacturingDate = product.ManufacturingDate,

ExpiryDate = product.ExpiryDate,

Tags = product.Tags.Select**(**t =**>** t.Name**)**.ToList**()**

**}**;

// Return the created product with an HTTP 200 OK (or 201 Created) status.

**return** Ok**(**response**)**;

**}**

// PUT: api/products/{id}

// Updates an existing product based on the data provided in ProductUpdateDTO.

**[**HttpPut**(**"{id}"**)]**

**public** **async** Task**<**ActionResult**<**ProductResponseDTO**>>** UpdateProduct**(int** id, **[**FromBody**]** ProductUpdateDTO productDto**)**

**{**

// Verify that the ID provided in the URL matches the ID in the request body.

**if** **(**id != productDto.ProductId**)**

**return** BadRequest**(**new **{** error = "Product ID in URL and body do not match." **})**;

// Validate the incoming DTO; if invalid, return a 400 Bad Request with the validation errors.

**if** **(**!ModelState.IsValid**)**

**return** BadRequest**(**ModelState**)**;

// Retrieve the existing product from the database, including its associated Tags.

var product = **await** \_context.Products

.Include**(**p =**>** p.Tags**)**

.FirstOrDefaultAsync**(**p =**>** p.ProductId == id**)**;

// If the product does not exist, return a 404 Not Found response.

**if** **(**product == **null)**

**return** NotFound**()**;

// Update the product properties with the new values from the DTO.

product.SKU = productDto.SKU;

product.Name = productDto.Name;

product.Price = productDto.Price;

product.Stock = productDto.Stock;

product.CategoryId = productDto.CategoryId;

product.Description = productDto.Description;

product.Discount = productDto.Discount;

product.ManufacturingDate = productDto.ManufacturingDate;

product.ExpiryDate = productDto.ExpiryDate;

// Clear the existing Tags associated with the product.

product.Tags.Clear**()**;

// Process Tags: For each tag provided in the DTO, normalize and add it.

**if** **(**productDto.Tags != **null** && productDto.Tags.Any**())**

**{**

**foreach** **(**var tagName in productDto.Tags**)**

**{**

// Normalize the tag by trimming whitespace and converting to lower case.

var normalizedTagName = tagName.Trim**()**.ToLower**()**;

var existingTag = **await** \_context.Tags.FirstOrDefaultAsync**(**t =**>** t.Name.ToLower**()** == normalizedTagName**)**;

**if** **(**existingTag != **null)**

product.Tags.Add**(**existingTag**)**; // Associate the existing tag with the product.

**else**

product.Tags.Add**(**new Tag **{** Name = normalizedTagName **})**; // Create and add a new tag.

**}**

**}**

// Mark the product entity as updated in the database context.

\_context.Products.Update**(**product**)**;

// Save changes to persist the updated product (and tag associations) to the database.

**await** \_context.SaveChangesAsync**()**;

// Map the updated product entity to a ProductResponseDTO.

var response = new ProductResponseDTO

**{**

ProductId = product.ProductId,

SKU = product.SKU,

Name = product.Name,

Price = product.Price,

Stock = product.Stock,

CategoryId = product.CategoryId,

Description = product.Description,

Discount = product.Discount,

ManufacturingDate = product.ManufacturingDate,

ExpiryDate = product.ExpiryDate,

Tags = product.Tags.Select**(**t =**>** t.Name**)**.ToList**()**

**}**;

// Return the updated product with an HTTP 200 OK status.

**return** Ok**(**response**)**;

**}**

**}**

**}**

**Configure the Database Connection String in the appsettings.json file**

To connect our DbContext to a database, we need to add a connection string in our appsettings.json file. So, please modify the appsettings.json file as follows:

**{**

"Logging": **{**

"LogLevel": **{**

"Default": "Information"**,**

"Microsoft.AspNetCore": "Warning"

**}**

**},**

"AllowedHosts": "\*"**,**

"ConnectionStrings": **{**

"ECommerceDBConnection": "Server=LAPTOP-6P5NK25R\\SQLSERVER2022DEV;Database=ECommerceDB;Trusted\_Connection=True;TrustServerCertificate=True;"

**}**

**}**

**Register FluentValidation and DbContext Services in Program Class**

Next, we need to register the DbConext and FluentValidation services to the Dependency Injection container. Please modify the Program class as follows. In the configuration below, we are enabling automatic validation of Fluent API.

**using** *FluentAPIValidationDemo.Data;*

**using** *FluentValidation.AspNetCore;*

**using** *FluentValidation;*

**using** *Microsoft.EntityFrameworkCore;*

**namespace** *FluentAPIValidationDemo*

**{**

**public** **class** Program

**{**

**public** **static** **void** Main**(string[]** args**)**

**{**

var builder = WebApplication.CreateBuilder**(**args**)**;

// Add services to the container.

// Add services to the container.

builder.Services.AddControllers**()**

.AddJsonOptions**(**options =**>**

**{**

// Keep original property names during serialization/deserialization.

options.JsonSerializerOptions.PropertyNamingPolicy = **null**;

**})**;

// Register the ECommerceDbContext with dependency injection

builder.Services.AddDbContext**<**ECommerceDbContext**>(**options =**>**

options.UseSqlServer**(**builder.Configuration.GetConnectionString**(**"ECommerceDBConnection"**)))**;

// Learn more about configuring Swagger/OpenAPI at https://aka.ms/aspnetcore/swashbuckle

builder.Services.AddEndpointsApiExplorer**()**;

builder.Services.AddSwaggerGen**()**;

// Enable automatic FluentValidation integration and scan for validators in the assembly

builder.Services.AddFluentValidationAutoValidation**()**;

builder.Services.AddValidatorsFromAssemblyContaining**<**Program**>()**;

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

// Configure the HTTP request pipeline.

**if** **(**app.Environment.IsDevelopment**())**

**{**

app.UseSwagger**()**;

app.UseSwaggerUI**()**;

**}**

app.UseHttpsRedirection**()**;

app.UseAuthorization**()**;

app.MapControllers**()**;

app.Run**()**;

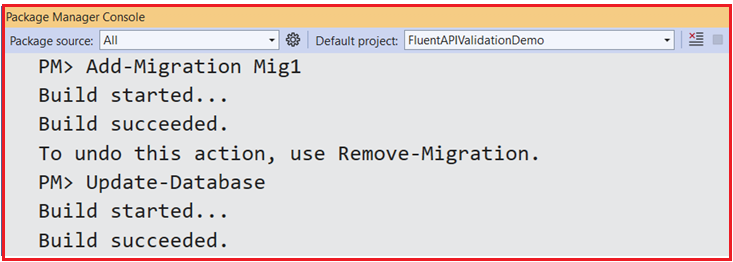
**}**

**}**

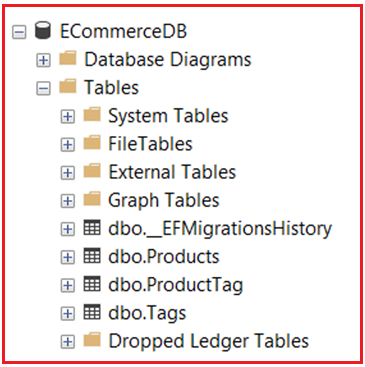
**}**

**Generate and Apply Database Migration:**

Now, open the Package Manager Console and execute the **Add-Migration** command to create a new Migration file. Then, execute the **Update-Database** command to apply the migration and update and sync the database with our model, as shown in the image below.



Once you execute the above commands, it should have created the **ECommerceDB**database with the Required Products table, as shown in the below image:

****

**Testing Get All Products (with Optional Tag Filtering)**

**Method: GET**  
**URL: https://localhost:5001/api/products**

**Query Parameters (for filtering by tags):**

You can add a query parameter named tags. For example, to filter products that are tagged with “electronics” and “gaming”, update the URL to:  
**URL: https://localhost:5001/api/products?tags=electronics,gaming**

**Test the POST Endpoint (Create Product)**

Create a New Product  
Method: POST  
URL: https://localhost:5001/api/products  
Headers: Content-Type: application/json  
Body: Use a sample JSON like the one below (adjust values as needed)

**{**

"SKU": "ELEC1234"**,**

"Name": "Ultra HD TV"**,**

"Price": 1200.00**,**

"Stock": 15**,**

"CategoryId": 2**,**

"Description": "55-inch Ultra HD Smart TV"**,**

"Discount": 10**,**

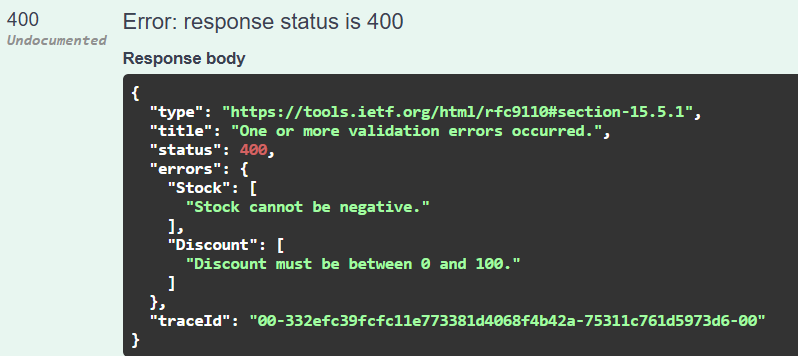
"ManufacturingDate": "2023-01-15T00:00:00"**,**

"ExpiryDate": "2024-01-15T00:00:00"**,**

"Tags": **[** "electronics"**,** "home" **]**

**}**

This should create a new Product in the database. Now, test with invalid data, like Discount as 105 and Stock as -15. You should get the following error message.



**Test the PUT Endpoint (Update Product)**

Update an Existing Product  
**Method: PUT**  
**URL: https://localhost:5001/api/products/{id}**  
Replace {id} with the actual product ID you want to update (e.g., 1).  
Headers: Content-Type: application/json  
Body: Use a JSON body similar to the update DTO. For example:

**{**

"ProductId": 1**,**

"SKU": "ELEC1234"**,**

"Name": "Ultra HD TV - Updated"**,**

"Price": 1150.00**,**

"Stock": 12**,**

"CategoryId": 2**,**

"Description": "55-inch Ultra HD Smart TV with updated features"**,**

"Discount": 12**,**

"ManufacturingDate": "2023-01-15T00:00:00"**,**

"ExpiryDate": "2024-01-15T00:00:00"**,**

"Tags": **[** "electronics"**,** "home"**,** "updated" **]**

**}**

Now, test the above endpoint with invalid data, such as Discount as 105 and Stock as -15, and see whether it works as expected.

**How to Implement Manual Fluent API Validation in ASP.NET Core Web API:**

By default, calling AddFluentValidationAutoValidation() integrates FluentValidation into the ASP.NET Core model binding pipeline. However, disabling this automatic integration and manually validating models in your controllers is also possible.

First, let us disable the Fluent API automatic validation, which validates the model during the model binding process. Please comment or remove the following line of code from the Program class to disable Fluent API auto validation.

// builder.Services.AddFluentValidationAutoValidation();

// builder.Services.AddValidatorsFromAssemblyContaining<Program>();

With the above changes in place, run the application and send a Post request with the same invalid data as follows:

**{**

"SKU": "ELEC1234"**,**

"Name": "Ultra HD TV"**,**

"Price": 1200.00**,**

"Stock": -15**,**

"CategoryId": 2**,**

"Description": "55-inch Ultra HD Smart TV"**,**

"Discount": 105**,**

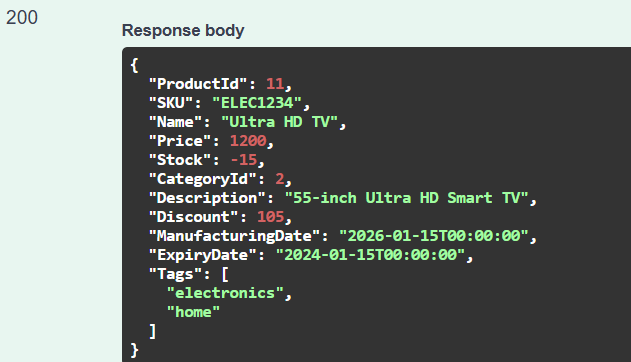
"ManufacturingDate": "2026-01-15T00:00:00"**,**

"ExpiryDate": "2024-01-15T00:00:00"**,**

"Tags": **[** "electronics"**,** "home" **]**

**}**

You will not get any validation error this time, and the server will process the request. A new product will be created with invalid data, and you will get the following response:



**How Do We Validate the Model Manually Using Fluent API Validators?**

We can check the validation manually by creating an instance of the Validator (i.e., **ProductCreateDTOValidator**) class and then invoking the **Validate** method by passing the **ProductCreateDTO**object we want to validate. For a better understanding, please modify the CreateProduct method of the Products Controller as follows:

**[**HttpPost**]**

**public** **async** Task**<**ActionResult**<**ProductResponseDTO**>>** CreateProduct**([**FromBody**]** ProductCreateDTO productDto**)**

**{**

// Validate the incoming DTO; if invalid, return a 400 Bad Request with the validation errors.

// Manually create an instance of ProductValidator

var validator = new ProductCreateDTOValidator**()**;

var validationResult = validator.Validate**(**productDto**)**;

// Approach 1 with Default Error Response

//if (!validationResult.IsValid)

//{

// // Return validation errors with 400 Bad Request

// return BadRequest(validationResult.Errors);

//}

// Approach 2 with Custom Error Response

**if** **(**!validationResult.IsValid**)**

**{**

var errorResponse = validationResult.Errors.Select**(**e =**>** new

**{**

Field = e.PropertyName,

Error = e.ErrorMessage

**})**;

**return** BadRequest**(**new **{** Errors = errorResponse **})**;

**}**

// Add product to DB if valid

// Map the incoming DTO to a new Product entity.

var product = new Product

**{**

SKU = productDto.SKU,

Name = productDto.Name,

Price = productDto.Price,

Stock = productDto.Stock,

CategoryId = productDto.CategoryId,

Description = productDto.Description,

Discount = productDto.Discount,

ManufacturingDate = productDto.ManufacturingDate,

ExpiryDate = productDto.ExpiryDate

**}**;

// Process Tags: For each tag provided in the DTO, check if it exists.

// If the tag exists, use the existing Tag; otherwise, create a new Tag.

**if** **(**productDto.Tags != **null** && productDto.Tags.Any**())**

**{**

**foreach** **(**var tagName in productDto.Tags**)**

**{**

// Normalize the tag by trimming whitespace and converting to lower case.

var normalizedTagName = tagName.Trim**()**.ToLower**()**;

var existingTag = **await** \_context.Tags.FirstOrDefaultAsync**(**t =**>** t.Name.ToLower**()** == normalizedTagName**)**;

**if** **(**existingTag != **null)**

product.Tags.Add**(**existingTag**)**; // Associate the existing tag with the product.

**else**

product.Tags.Add**(**new Tag **{** Name = normalizedTagName **})**; // Create a new tag and associate it.

**}**

**}**

// Add the new product to the database context.

\_context.Products.Add**(**product**)**;

// Save changes to persist the new product (and associated tags) to the database.

**await** \_context.SaveChangesAsync**()**;

// Map the newly created Product entity to a ProductResponseDTO.

var response = new ProductResponseDTO

**{**

ProductId = product.ProductId,

SKU = product.SKU,

Name = product.Name,

Price = product.Price,

Stock = product.Stock,

CategoryId = product.CategoryId,

Description = product.Description,

Discount = product.Discount,

ManufacturingDate = product.ManufacturingDate,

ExpiryDate = product.ExpiryDate,

Tags = product.Tags.Select**(**t =**>** t.Name**)**.ToList**()**

**}**;

// Return the created product with an HTTP 200 OK (or 201 Created) status.

**return** Ok**(**response**)**;

**}**

**Testing Manual Fluent API Validation:**

Now, reaccess the Post endpoint with the following data:

**{**

"SKU": "ELEC1234"**,**

"Name": "Ultra HD TV"**,**

"Price": 1200.00**,**

"Stock": -15**,**

"CategoryId": 2**,**

"Description": "55-inch Ultra HD Smart TV"**,**

"Discount": 105**,**

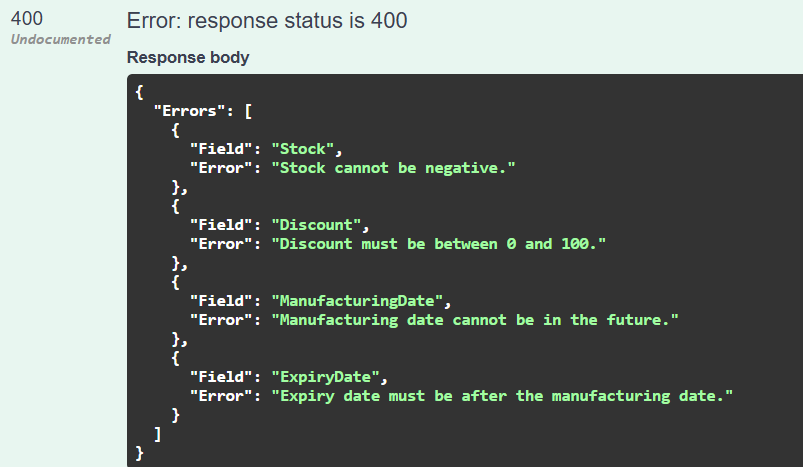
"ManufacturingDate": "2026-01-15T00:00:00"**,**

"ExpiryDate": "2024-01-15T00:00:00"**,**

"Tags": **[** "electronics"**,** "home" **]**

**}**

**This time, you will get the following error message:**

****

**When Should We Implement Manual Fluent API Validation in ASP.NET Core Web API?**

Manual Fluent API validation can be appropriate when:

* Validation needs to be conditionally applied based on business logic that isn’t known at startup.
* You have complex scenarios where different rules apply in different contexts.
* You want control over how and when validation is triggered.
* You need detailed validation error responses that might not align with the default model state handling.

**What is the Problem with the above Approach?**

Both your ProductCreateDTO and ProductUpdateDTO share nearly identical properties and validation rules. This leads to two sets of duplicated code:

* Duplicated DTO properties (everything except ProductId is the same).
* Duplicated validation rules (all the same checks except for ProductId).

We need to organize our DTOs and validators so that the common logic is written only once while still allowing small differences for Create vs. Update.

**Use a Shared Base DTO + Derived DTOs**

We can avoid code duplication by introducing a base DTO with all the common properties to create and update scenarios. Then, ProductCreateDTO and ProductUpdateDTO are derived from that base class (or interface).

**Create a Base DTO**

Define a base DTO that contains all the shared properties. Create a class file named **ProductBaseDTO.cs** within the DTOs folder and copy and paste the following code.

**namespace** *FluentAPIValidationDemo.DTOs*

**{**

**public** **class** ProductBaseDTO

**{**

**public** **string** SKU **{** **get**; **set**; **}**

**public** **string** Name **{** **get**; **set**; **}**

**public** **decimal** Price **{** **get**; **set**; **}**

**public** **int** Stock **{** **get**; **set**; **}**

**public** **int** CategoryId **{** **get**; **set**; **}**

**public** **string**? Description **{** **get**; **set**; **}**

**public** **decimal** Discount **{** **get**; **set**; **}**

**public** DateTime ManufacturingDate **{** **get**; **set**; **}**

**public** DateTime ExpiryDate **{** **get**; **set**; **}**

**public** List**<string>**? Tags **{** **get**; **set**; **}**

**}**

**}**

**Derive ProductCreateDTO and ProductUpdateDTO from the ProductBaseDTO**

Modify the **ProductCreateDTO** class as follows:

**namespace** *FluentAPIValidationDemo.DTOs*

**{**

**public** **class** ProductCreateDTO : ProductBaseDTO

**{**

// You can add create-specific properties here if needed.

**}**

**}**

Modify the **ProductUpdateDTO** class as follows:

**namespace** *FluentAPIValidationDemo.DTOs*

**{**

**public** **class** ProductUpdateDTO : ProductBaseDTO

**{**

**public** **int** ProductId **{** **get**; **set**; **}**

**}**

**}**

**Create a Base Validator for the Shared Rules**

Create a generic base validator for the shared properties using FluentValidation. So, add a class file named **ProductBaseDTOValidator.cs** within the **Validators** folder and then copy and paste the following.

**using** *FluentAPIValidationDemo.DTOs;*

**using** *FluentValidation;*

**namespace** *FluentAPIValidationDemo.Validators*

**{**

**public** **class** ProductBaseDTOValidator**<**T**>** : AbstractValidator**<**T**>** **where** T : ProductBaseDTO

**{**

**public** ProductBaseDTOValidator**()**

**{**

// SKU validation: must not be empty and must match the specified regex pattern.

RuleFor**(**p =**>** p.SKU**)**

.NotEmpty**()**.WithMessage**(**"SKU is required."**)** // Ensures SKU is provided.

.Matches**(**"^[A-Z0-9]{8}$"**)**.WithMessage**(**"SKU must be 8 characters long and contain only uppercase letters and digits."**)**; // Must be exactly 8 uppercase letters or digits.

// Name validation: must not be empty and length must be between 3 and 50 characters.

RuleFor**(**p =**>** p.Name**)**

.NotEmpty**()**.WithMessage**(**"Product name is required."**)** // Ensures product name is provided.

.Length**(**3, 50**)**.WithMessage**(**"Product name must be between 3 and 50 characters."**)**; // Validates the length of the product name.

// Price validation: must be greater than 0 and conform to precision and scale.

RuleFor**(**p =**>** p.Price**)**

.GreaterThan**(**0**)**.WithMessage**(**"Price must be greater than 0."**)** // Price must be positive.

.PrecisionScale**(**8, 2, **true)**.WithMessage**(**"Price must have at most 8 digits in total and 2 decimals."**)**; // Validates total digits and decimals (ignoring trailing zeros).

// Stock validation: must be zero or positive.

RuleFor**(**p =**>** p.Stock**)**

.GreaterThanOrEqualTo**(**0**)**.WithMessage**(**"Stock cannot be negative."**)**; // Stock can't be negative.

// CategoryId validation: must be a positive number.

RuleFor**(**p =**>** p.CategoryId**)**

.GreaterThan**(**0**)**.WithMessage**(**"Category ID is required."**)**; // Ensures a valid category is selected.

// Description validation: if provided, must not exceed 500 characters.

RuleFor**(**p =**>** p.Description**)**

.MaximumLength**(**500**)**.WithMessage**(**"Description cannot exceed 500 characters."**)** // Limits description length.

.When**(**p =**>** !**string**.IsNullOrEmpty**(**p.Description**))**; // Only applies rule if Description is provided.

// Discount validation: must be between 0 and 100.

RuleFor**(**p =**>** p.Discount**)**

.InclusiveBetween**(**0, 100**)**.WithMessage**(**"Discount must be between 0 and 100."**)**; // Ensures discount is a valid percentage.

// Manufacturing date validation: must not be a future date.

RuleFor**(**p =**>** p.ManufacturingDate**)**

.LessThanOrEqualTo**(**DateTime.Now**)**.WithMessage**(**"Manufacturing date cannot be in the future."**)**; // Date must be in the past or today.

// Expiry date validation: must be later than the manufacturing date.

RuleFor**(**p =**>** p.ExpiryDate**)**

.GreaterThan**(**p =**>** p.ManufacturingDate**)**.WithMessage**(**"Expiry date must be after the manufacturing date."**)**; // Validates date order.

// Tags validation: for each tag in the list, ensure it is not empty and doesn't exceed 20 characters.

RuleForEach**(**p =**>** p.Tags**)**.ChildRules**(**tag =**>**

**{**

tag.RuleFor**(**t =**>** t**)**

.NotEmpty**()**.WithMessage**(**"Tag cannot be empty."**)** // Ensures each tag is provided.

.MaximumLength**(**20**)**.WithMessage**(**"Tag cannot exceed 20 characters."**)**; // Limits tag length.

**})**;

**}**

**}**

**}**

**Create Specific Validators Using the Base Validator**

Now, we need to create specialized validators by including the base validator. FluentValidation’s Include method makes it easy.

**Modifying ProductCreateDTOValidator**

Please modify the ProductCreateDTOValidator as follows:

**using** *FluentAPIValidationDemo.DTOs;*

**using** *FluentValidation;*

**namespace** *FluentAPIValidationDemo.Validators*

**{**

// Validator for ProductCreateDTO, which contains the rules for creating a product.

**public** **class** ProductCreateDTOValidator : AbstractValidator**<**ProductCreateDTO**>**

**{**

**public** ProductCreateDTOValidator**()**

**{**

// Include all rules from the base validator

Include**(**new ProductBaseDTOValidator**<**ProductCreateDTO**>())**;

// If you have any create-specific rules, put them here

// (often there's none needed for create)

**}**

**}**

**}**

**Modifying ProductUpdateDTOValidator**

Please modify the ProductUpdateDTOValidator as follows:

**using** *FluentAPIValidationDemo.DTOs;*

**using** *FluentValidation;*

**namespace** *FluentAPIValidationDemo.Validators*

**{**

**public** **class** ProductUpdateDTOValidator : AbstractValidator**<**ProductUpdateDTO**>**

**{**

**public** ProductUpdateDTOValidator**()**

**{**

// Include all rules from the base validator

Include**(**new ProductBaseDTOValidator**<**ProductUpdateDTO**>())**;

// Additional rule specific to updates

RuleFor**(**p =**>** p.ProductId**)**

.GreaterThan**(**0**)**.WithMessage**(**"ProductId must be greater than 0."**)**;

**}**

**}**

**}**

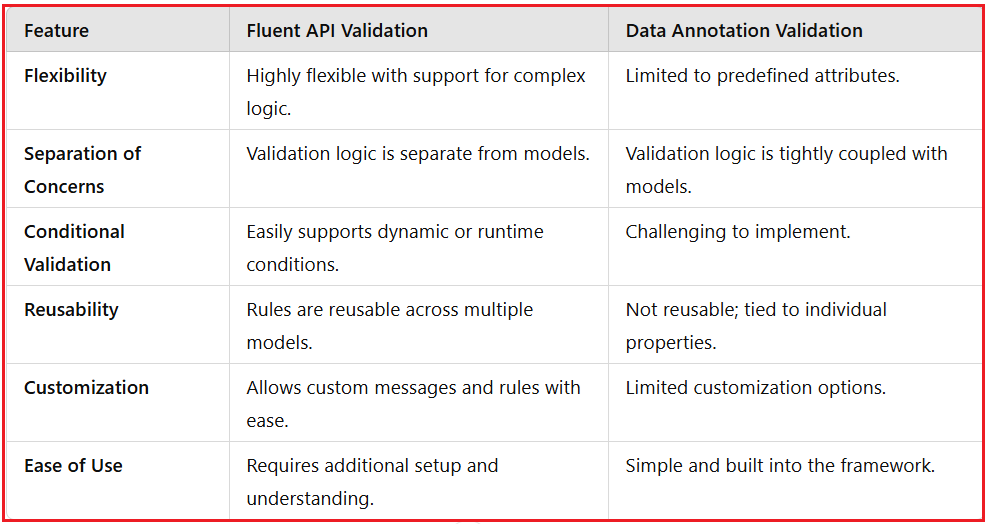
**Note:** There have been no changes in the Controller. Run the application and test the functionalities; it should work as expected.

**Benefits of This Approach**

This design keeps our codebase clean and minimizes duplication while allowing us to customize behavior to create and update operations. The following are the key benefits:

* **DRY Principle:** Common properties and validations are written only once.
* **Maintainability:** Changes to shared validation rules must be made in one place.
* **Clarity:** Each specialized DTO or validator only contains the differences, making your code clearer and more concise.

**Differences Between Fluent API and Data Annotation Validation in ASP.NET Core Web API:**



**When to Use Which Approach**

* Use **Data Annotations** for simple scenarios where validation rules are straightforward. There is no need to reuse them across different models. They are easy to implement and understand.
* Use **Fluent API** Validation for complex, reusable, or dynamic validation scenarios where separating the validation logic from the model helps maintain cleaner code. It’s also ideal when the same validation logic is needed across multiple models.

**Hybrid Approach:**

Using both Data Annotations and Fluent Validation can offer a balance. For example, you can apply simple and commonly understood rules (e.g., [Required]) via Data Annotations and reserve Fluent Validation for more complex, conditional, or reusable rules. This hybrid approach allows us to maintain straightforward validations within the model while still using the power and flexibility of fluent validation where needed.

**Note:** The **FluentValidation.AspNetCore** package is no longer maintained. Microsoft recommends to use the core FluentValidation package and adopting a manual validation approach. This change addresses limitations such as the non-asynchronous nature of ASP.NET Core’s validation pipeline and the complexities introduced by automatic validation. In our upcoming articles, I will show you how to use the FluentValidation package.

Fluent API Validation in ASP.NET Core Web API provides a flexible and powerful way to validate data models while maintaining a clean separation of concerns. Using FluentValidation, we can easily create reusable, complex, and dynamic validation rules.