Common.Validation

An interoperable, multi-layer validation framework for .NET 10 and TypeScript. Define validation rules once -- in C# or JSON -- and enforce them across API models, DTOs, database entities, and frontend forms with per-layer severity control.

Table of Contents

Key Concepts

Common.Validation is built around three ideas that distinguish it from other validation libraries:

Severity is not binary. Validation failures are not just "valid" or "invalid". Each failure carries a `Severity`:

Severity	Meaning	Typical Action

`Forbidden`	The value is invalid. The operation must not proceed.	Block the request.
`AtOwnRisk`	The value is risky. The caller accepts responsibility.	Warn the user, log it, proceed.
`NotRecommended`	The value is technically valid but not ideal.	Show an informational hint.

Layers change severity. The same validation rule can produce different severities depending on where it runs. A phone number might be `Forbidden` to omit at the API layer but merely `NotRecommended` in a database entity.

One definition, many runtimes. Rules defined in a shared JSON schema are consumed by both the C# backend and TypeScript frontend, ensuring client-server parity without duplicating logic.

Getting Started

Installation

Add the NuGet package to your project:

```
dotnet add package Common.Validation
```

For Blazor components:

```
dotnet add package Common.Validation.Blazor
```

For TypeScript/JavaScript:

```
npm install common-validation
```

Your First Validator

Create a model and a validator:

```
using Common.Validation.Core;
using Common.Validation.Extensions;

public class CreateOrderRequest
{
    public string CustomerName { get; set; } = string.Empty;
    public string Email { get; set; } = string.Empty;
    public decimal Total { get; set; }
    public string? PromoCode { get; set; }
}

public class CreateOrderValidator : AbstractValidator<CreateOrderRequest>
{
    public CreateOrderValidator()
    {
        RuleFor(x => x.CustomerName)
            .NotEmpty().WithMessage("Customer name is required.")
            .MaxLength(200).WithMessage("Name is too long.");

        RuleFor(x => x.Email)
```

```
            .NotEmpty().WithMessage("Email is required.")
            .EmailAddress().WithMessage("Not a valid email address.");

        RuleFor(x => x.Total)
            .GreaterThan(0m).WithMessage("Order total must be positive.");

        RuleFor(x => x.PromoCode)
            .MaxLength(20).WithMessage("Promo code is too long.")
            .WithSeverity(Severity.NotRecommended);
    }
}
```

Use it:

```
var validator = new CreateOrderValidator();
var result = validator.Validate(new CreateOrderRequest
{
    CustomerName = "",
    Email = "bad",
    Total = -5m
});

if (result.HasForbidden)
{
    // Block: mandatory fields are missing or invalid
    foreach (var error in result.BySeverity(Severity.Forbidden))
        Console.WriteLine($"  ERROR: {error.PropertyName} - {error.ErrorMessage}");
}
```

Interpreting Results

`ValidationResult` provides several ways to inspect failures:

```
result.IsValid              // true if zero failures
result.HasForbidden         // any blocking errors?
result.HasAtOwnRisk         // any risk warnings?
result.HasNotRecommended    // any soft hints?
result.Errors               // all failures as IReadOnlyList<ValidationFailure>
result.BySeverity(severity) // filter by severity level
```

You can combine results from multiple validators:

```
var combined = ValidationResult.Combine(
    addressValidator.Validate(order.Address),
    paymentValidator.Validate(order.Payment)
);
```

Fluent API Reference

Built-in Rules

Common rules (any property type):

Method	Description

`.NotNull()`	Value must not be null
`.Null()`	Value must be null
`.NotEmpty()`	String not null/whitespace, collection not empty
`.Empty()`	Inverse of NotEmpty
`.Equal(value)`	Must equal the given value
`.NotEqual(value)`	Must not equal the given value
`.Must(predicate, msg)`	Custom predicate

String rules:

Method	Description
`.MinLength(n)`	At least n characters
`.MaxLength(n)`	At most n characters
`.Length(min, max)`	Between min and max characters
`.Matches(pattern)`	Matches a regex pattern
`.EmailAddress()`	Valid email format
`.PhoneNumber()`	Valid phone format

Comparison rules (for `IComparable<T>`):

Method	Description
`.GreaterThan(n)`	Strictly greater than n
`.GreaterThanOrEqual(n)`	Greater than or equal to n
`.LessThan(n)`	Strictly less than n
`.LessThanOrEqual(n)`	Less than or equal to n
`.InclusiveBetween(a, b)`	Between a and b (inclusive)

Modifiers (chain after any rule):

Method	Description
`.WithMessage("...")`	Custom error message
`.WithErrorCode("...")`	Programmatic error code
`.WithSeverity(Severity.X)`	Set default severity
`.WithLayerSeverity("api", Severity.X)`	Layer-specific severity
`.Cascade(CascadeMode.StopOnFirstFailure)`	Stop on first failure for this property

Custom Rules

Use `.Must()` for inline predicates:

```
RuleFor(x => x.StartDate)
    .Must(date => date >= DateTime.Today, "Start date must be in the future.");
```

Access the parent object for cross-property validation:

```
RuleFor(x => x.EndDate)
    .Must((order, endDate) => endDate > order.StartDate,
          "End date must be after start date.");
```

Conditions

Apply rules conditionally:

```
RuleFor(x => x.CompanyName)
    .When(x => x.CustomerType == CustomerType.Business)
    .NotEmpty().WithMessage("Company name is required for business customers.");

RuleFor(x => x.PersonalId)
    .Unless(x => x.CustomerType == CustomerType.Business)
    .NotEmpty().WithMessage("Personal ID is required for individual customers.");
```

Cascade Mode

Control whether validation continues after a failure:

```
// Validator level: stop after the first rule with failures
public class StrictValidator : AbstractValidator<Order>
{
    public StrictValidator()
    {
        CascadeMode = CascadeMode.StopOnFirstFailure;
        // ...
    }
}

// Property level: stop checking this property after first failure
RuleFor(x => x.Email)
    .Cascade(CascadeMode.StopOnFirstFailure)
    .NotEmpty().WithMessage("Required.")
    .EmailAddress().WithMessage("Invalid format.");
    // If NotEmpty fails, EmailAddress is skipped
```

Multi-Layer Severity

The Problem

In a typical enterprise application, the same data passes through multiple layers:

```
Browser Form  -->  API Controller  -->  Service Layer  -->  Database Entity
```

Each layer has different constraints. An email address might be:

- **Forbidden** to omit at the API (you can't create a user without one)
- **AtOwnRisk** at the DTO level (partial updates may skip it)
- **NotRecommended** at the entity level (legacy records might lack one)

Common.Validation solves this with per-layer severity overrides.

Layer Attributes

Mark your models with the layer they belong to:

```
using Common.Validation.Layers;

[ValidationLayer("api")]
public class UserApiModel
{
    public string Email { get; set; } = string.Empty;
    public string Phone { get; set; } = string.Empty;
}

[ValidationLayer("dto")]
public class UserDto
{
    public string Email { get; set; } = string.Empty;
    public string Phone { get; set; } = string.Empty;
}

[ValidationLayer("entity")]
public class UserEntity
{
    public string Email { get; set; } = string.Empty;
    public string Phone { get; set; } = string.Empty;
}
```

The attribute is inherited, so base class attributes flow to subclasses:

```
[ValidationLayer("api")]
public abstract class ApiModelBase { }

public class UserApiModel : ApiModelBase { }
// UserApiModel automatically resolves to the "api" layer
```

Layer-Aware Rules

Define rules once with per-layer severity overrides:

```
public class UserValidator : AbstractValidator<UserApiModel>
{
    public UserValidator()
    {
        RuleFor(x => x.Email)
            .NotEmpty().WithMessage("Email is required.")
            .WithSeverity(Severity.Forbidden)               // default
            .WithLayerSeverity("api", Severity.Forbidden)    // mandatory at API
            .WithLayerSeverity("dto", Severity.AtOwnRisk)    // risky to skip in DTO
            .WithLayerSeverity("entity", Severity.NotRecommended); // nice-to-have in DB

        RuleFor(x => x.Phone)
            .NotEmpty().WithMessage("Phone is recommended.")
            .WithSeverity(Severity.AtOwnRisk)
            .WithLayerSeverity("api", Severity.AtOwnRisk)
            .WithLayerSeverity("entity", Severity.NotRecommended);
    }
}
```

When you call `validator.Validate(instance)`, the layer is automatically resolved from the `[ValidationLayer]` attribute on the model type. No additional configuration needed.

Explicit Context

Override the layer at runtime:

```
var context = ValidationContext.ForLayer("entity");
var result = validator.Validate(instance, context);
// Uses "entity" layer severities regardless of the type's attribute
```

JSON-Based Validation

Schema Overview

Define rules in a JSON file that both C# and TypeScript can consume:

```
{
  "$schema": "https://common-validation/schema/v1.json",
  "type": "Invoice",
  "properties": {
    "invoiceNumber": {
      "rules": [
        {
          "validator": "notEmpty",
          "message": "Invoice number is required.",
          "severity": "forbidden"
        },
        {
          "validator": "matches",
          "params": { "pattern": "^INV-\\d{6}$" },
          "message": "Must match format INV-XXXXXX.",
          "severity": "forbidden"
        }
      ]
    },
    "amount": {
      "rules": [
        {
          "validator": "greaterThan",
          "params": { "value": 0 },
          "message": "Amount must be positive.",
          "severity": "forbidden",
          "layers": {
            "api": "forbidden",
            "entity": "atOwnRisk"
          }
        }
      ]
    }
  }
}
```

Available validator types in JSON: `notNull`, `null`, `notEmpty`, `empty`, `maxLength`, `minLength`, `length`, `email`, `phone`, `matches`, `equal`, `notEqual`, `greaterThan`, `greaterThanOrEqual`, `lessThan`, `lessThanOrEqual`, `inclusiveBetween`.

Loading Definitions

```
using Common.Validation.Json;
```

```
var loader = new JsonValidationDefinitionLoader();

// From a file
var definition = loader.LoadFromFile("Invoice.validation.json");

// From a JSON string (e.g., fetched from an API or embedded resource)
var definition = loader.Load(jsonString);

// From all *.validation.json files in a directory
var definitions = loader.LoadFromDirectory("Validations/");

// Create a validator from the definition
var validator = new JsonValidator<Invoice>(definition);
var result = validator.Validate(invoice);
```

Custom Validator Types

Extend the registry with domain-specific validators:

```
using Common.Validation.Json.Registry;

var registry = new ValidatorTypeRegistry();

// Register a custom "iban" validator
registry.Register("iban", parameters =>
{
    return new DelegatePropertyCheck(value =>
        value is string s && s.Length >= 15 && s.Length <= 34 && s.StartsWith("PL"));
});

// Use it
var validator = new JsonValidator<BankAccount>(definition, registry);
```

To implement `IPropertyCheck`:

```
public class IbanCheck : IPropertyCheck
{
    public bool IsValid(object? value)
    {
        return value is string iban
            && iban.Length is >= 15 and <= 34
            && char.IsLetter(iban[0])
            && char.IsLetter(iban[1]);
    }
}
```

Dependency Injection

Basic Setup

```
using Common.Validation.DependencyInjection;

var builder = WebApplication.CreateBuilder(args);

builder.Services.AddCommonValidation(options =>
{
```

```
    options.DefaultCascadeMode = CascadeMode.Continue;
    options.DefaultLayer = "api";
    options.JsonDefinitionPaths.Add("Validations/");
});
```

Assembly Scanning

Automatically discover and register all validators in an assembly:

```
// Scan a specific assembly
builder.Services.AddValidatorsFromAssembly(typeof(Program).Assembly);

// Or use a marker type
builder.Services.AddValidatorsFromAssemblyContaining<CreateOrderValidator>();

// Control lifetime (default: Transient)
builder.Services.AddValidatorsFromAssembly(
    typeof(Program).Assembly,
    ServiceLifetime.Scoped);
```

Validator Factory

Resolve validators at runtime when you don't know the type at compile time:

```
public class ValidationMiddleware
{
    private readonly IValidatorFactory _factory;

    public ValidationMiddleware(IValidatorFactory factory)
    {
        _factory = factory;
    }

    public ValidationResult ValidateModel<T>(T model) where T : class
    {
        var validator = _factory.GetValidator<T>();
        if (validator is null)
            return new ValidationResult(); // no validator registered, pass through

        return validator.Validate(model);
    }
}
```

Property-Level Validation

You can validate a single property instead of the entire object. This is useful for:

- **Real-time field validation** (e.g., on blur) without re-running rules for untouched fields
- **Partial updates** where only certain fields changed
- **Performance** when the object has many properties but you care about one

Usage

Use the `ValidateProperty` extension method on any `IValidator<T>`:

```
using Common.Validation.Extensions;

var validator = new CreateOrderValidator();
var order = new CreateOrderRequest { CustomerName = "", Email = "bad", Total = -5 };

// Validate only the Email field
var emailResult = validator.ValidateProperty(order, x => x.Email);

if (emailResult.IsValid)
    Console.WriteLine("Email is valid");
else
    foreach (var e in emailResult.Errors)
        Console.WriteLine($"{e.PropertyName}: {e.ErrorMessage}");
```

With Validation Context

Property-level validation respects the same layer and context as full validation:

```
var context = ValidationContext.ForLayer("entity");
var result = validator.ValidateProperty(model, x => x.FirstName, context);
```

Supported Validators

- **AbstractValidator<T>** – Validates only the rules defined for the specified property.
- **JsonValidator<T>** – Validates only the rules from the JSON definition for that property.
- **Other IValidator<T>** – Falls back to full validation and filters the result by property name (less efficient but works).

JSON-based property validation

`JsonValidator<T>` supports property-level validation with the same rules defined in your JSON definition. This is useful when rules come from configuration or are shared with a TypeScript frontend:

```
using Common.Validation.Extensions;
using Common.Validation.Json;

// Load definition (e.g. from PersonalData.validation.json)
var definition = "PersonalData.validation.json".LoadFromFile();
var validator = new JsonValidator<PersonalData>(definition);

var model = new PersonalData
{
    FirstName = "Jan",
    LastName = "Nowak",
    Email = "invalid-email",  // Invalid
    Citizenship = "PL",
    TaxResidency = ""        // Not recommended
};

// Validate only the Email field
var emailResult = validator.ValidateProperty(model, x => x.Email);

if (!emailResult.IsValid)
    foreach (var e in emailResult.Errors)
        Console.WriteLine($"{e.PropertyName}: {e.ErrorMessage}");
// Output: Email: Invalid email format.
```

Layer context is respected when validating a single property:

```
var context = ValidationContext.ForLayer("entity");
var result = validator.ValidateProperty(model, x => x.FirstName, context);
// Uses "entity" layer severities from the JSON definition
```

When to Use

Scenario	Use
Form field blur / `onChange`	`ValidateProperty`
Submit button clicked	`Validate`
API receives partial PATCH	`ValidateProperty` per changed field
Full model save	`Validate`

Blazor Integration

Components

Validation summary -- displays all failures grouped by severity:

```
@using Common.Validation.Blazor

<CvValidationSummary Result="@_validationResult" />
```

Per-field messages -- display failures for a specific property:

```
<InputText @bind-Value="_model.Email" class="form-control" />
<CvValidationMessage Result="@_result" PropertyName="Email" />
```

Both components use CSS classes for severity-based styling:

- `.cv-forbidden` -- red
- `.cv-at-own-risk` -- amber
- `.cv-not-recommended` -- gray

EditContext Integration

Hook into Blazor's built-in form validation:

```
@using Common.Validation.Blazor

<EditForm EditContext="@_editContext" OnSubmit="HandleSubmit">
    <InputText @bind-Value="_model.Name" />
    <ValidationMessage For="@(() => _model.Name)" />

    <button type="submit">Save</button>
</EditForm>

@code {
    private EditContext _editContext = default!;
    private MyModel _model = new();
```

```
    protected override void OnInitialized()
    {
        _editContext = new EditContext(_model);
        _editContext.AddCommonValidation(new MyValidator());
    }
}
```

TypeScript Client

The TypeScript package consumes the same JSON definitions as the C# backend:

```
import { Validator } from 'common-validation';
import type { ValidationDefinition } from 'common-validation';

// Load the same JSON definition used by the backend
const definition: ValidationDefinition = await fetch('/api/validations/invoice')
  .then(r => r.json());

const validator = new Validator(definition);

const result = validator.validate({
  invoiceNumber: '',
  amount: -5
});

if (result.hasForbidden) {
  result.errors
    .filter(e => e.severity === 'forbidden')
    .forEach(e => showFieldError(e.propertyName, e.errorMessage));
}
```

Layer-aware validation works the same way:

```
// Validate with a specific layer
const result = validator.validate(formData, 'api');
```

Register custom validators on the client side:

```
import { Validator } from 'common-validation';

const validator = new Validator(definition, {
  iban: () => (value) =>
    typeof value === 'string' && value.length >= 15 && value.length <= 34
});
```

Architecture

```
common.validation/
  src/
    Common.Validation/                  # Core NuGet package
      Core/                             #   IValidator, AbstractValidator, Severity,
                                        #   ValidationResult, CascadeMode, ValidationContext
      Rules/                            #   IRuleBuilder, IValidationRule, PropertyRule
```

```
      Extensions/                          #   Fluent API (NotEmpty, MaxLength, WithSeverity, e
      Layers/                              #   ValidationLayerAttribute
      Json/                                #   JsonValidator, definition models, loader
        Registry/                          #   IValidatorTypeRegistry, built-in checks
      DependencyInjection/                 #   AddCommonValidation, IValidatorFactory
    Common.Validation.Blazor/              # Blazor NuGet package (Razor Class Library)
                                           #   CvValidationSummary, CvValidationMessage,
                                           #   EditContext extensions, CSS isolation
  client/
    common-validation/                     # TypeScript npm package
      src/                                 #   Validator, types, built-in rules
      schema/                              #   JSON Schema for IDE autocompletion
  tests/
    Common.Validation.Tests/               # xUnit tests (117 tests)
  demo/
    Common.Validation.Demo/                # Console demo (fluent, layers, JSON, DI)
    Common.Validation.Demo.Blazor/         # Blazor interactive demo
```

Use Cases

REST API Input Validation

Validate incoming requests in a controller or middleware, rejecting `Forbidden` failures and passing through `AtOwnRisk` / `NotRecommended` as response headers or metadata.

```
[HttpPost]
public IActionResult CreateUser([FromBody] CreateUserRequest request)
{
    var result = _validator.Validate(request);

    if (result.HasForbidden)
        return BadRequest(result.BySeverity(Severity.Forbidden)
            .Select(e => new { e.PropertyName, e.ErrorMessage }));

    if (result.HasAtOwnRisk)
        Response.Headers.Append("X-Validation-Warnings",
            string.Join("; ", result.BySeverity(Severity.AtOwnRisk).Select(e => e.ErrorMessag

    return Ok(_service.CreateUser(request));
}
```

Multi-Tenant Configurable Validation

Load validation rules from a database or configuration API per tenant, using JSON definitions:

```
builder.Services.AddCommonValidation(options =>
{
    options.JsonDefinitionPaths.Add($"Validations/{tenantId}/");
});
```

Form Wizard with Progressive Strictness

In a multi-step form, earlier steps use relaxed severity; the final step enforces everything:

```
// Step 1: only show hints
var step1Context = ValidationContext.ForLayer("draft");
```

```
// Step 2: warn about risks
var step2Context = ValidationContext.ForLayer("review");

// Final submit: block on forbidden
var submitContext = ValidationContext.ForLayer("api");
```

Shared Client-Server Validation

Define rules in JSON, serve them from the API, validate on both sides:

```
Backend:  JsonValidator<T>(definition)          --> ValidationResult
Frontend: new Validator(definition).validate(t) --> ValidationResult
```

Both produce the same failures for the same input, ensuring UI error messages match server-side enforcement.

PATCH API with per-field JSON validation

When handling partial updates (PATCH), validate only the fields that changed using rules from a JSON definition:

```
[HttpPatch("{id}")]
public IActionResult UpdateUser(Guid id, [FromBody] Dictionary<string, object?> changedFields
{
    var target = _repository.Get(id);
    var update = MapToUpdate(target, changedFields);

    var definition = "User.validation.json".LoadFromFile();
    var validator = new JsonValidator<UserUpdate>(definition);

    foreach (var (key, _) in changedFields)
    {
        var result = key switch
        {
            "email" => validator.ValidateProperty(update, u => u.Email),
            "firstName" => validator.ValidateProperty(update, u => u.FirstName),
            _ => new ValidationResult()
        };
        if (result.HasForbidden)
            return BadRequest(result.Errors.Select(e => new { e.PropertyName, e.ErrorMessage
    }

    return Ok(_service.Update(target, update));
}
```

This avoids validating untouched fields and keeps rules consistent with the shared JSON definition.

Domain Entity Invariants

Validate invariants within domain entities themselves, using the non-generic `IValidator` interface for polymorphic dispatch:

```
public abstract class Entity
{
    public ValidationResult Validate(IValidatorFactory factory)
    {
        var validator = factory.GetValidator(GetType());
        return validator?.Validate(this) ?? new ValidationResult();
```

```
    }
}
```

Composite Validation Across Aggregates

Validate an entire aggregate root by merging results from child validators:

```
var orderResult = _orderValidator.Validate(order);
var itemResults = order.Items.Select(item => _itemValidator.Validate(item));

var combined = ValidationResult.Combine(
    new[] { orderResult }.Concat(itemResults).ToArray());

if (combined.HasForbidden) { /* ... */ }
```

Roadmap

The project is under active development. Planned directions include:

- **Async validation** -- `ValidateAsync` for rules that need I/O (e.g., checking uniqueness against a database). The `IValidator<T>` interface is designed to support this.
- **Localization** -- message templates with placeholders and resource-file integration for multi-language support.
- **Source generators** -- compile-time validator discovery and AOT-compatible JSON serialization, eliminating reflection overhead.
- **Validation profiles** -- named rule sets within a single validator, selectable at validation time (e.g., "create" vs. "update" profiles).
- **OpenAPI integration** -- auto-generate `x-validation` metadata in Swagger/OpenAPI specs from JSON definitions.
- **React / Vue adapters** -- framework-specific wrappers around the TypeScript client for seamless form integration.
- **Rule composition** -- `Include()` and `InheritRulesFrom()` for composing validators from reusable fragments.
- **Diagnostic analyzers** -- Roslyn analyzers to catch common mistakes at compile time (e.g., forgetting `.WithMessage()` after a rule).

License