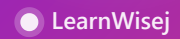


Custom Validation: Methods, Services and ValidationRule Classes

Lesson Guide · Lab / Exam Guide



Lesson Guide

Business rules the built-in rules cannot express: string-returning validation methods, a `ContactValidator` service that can be tested without the UI, a custom `ValidationRule` class with `OnValidating`, `OnValidated` and `OnControlCreated`, and cross-field rules that mark every control involved. What this means for a developer who knows basic C# and is beginning Wisej.NET form development — and how to read this module.



Production focus

Business rules the built-in rules cannot express: string-returning validation methods, a `ContactValidator` service that can be tested without the UI, a custom `ValidationRule` class with `OnValidating`, `OnValidated` and `OnControlCreated`, and cross-field rules that mark every control involved.

1. The production problem

After Module 3 the `ValidationClinic` intake form knows that an email needs an @ and an age must be a whole number. It still happily saves a contact whose start date comes after the end date, or a birth date that makes the person eleven years old. The built-in rules check the *shape* of one value; real applications also need rules about *meaning*, and some of those rules involve more than one control. This module gives those rules a proper home, so they are written once, shown clearly and never skipped on Save.

2. Key concepts

Concept	Why it matters
custom validation method	A method on the form that answers one question and returns an empty string when the data is valid, or a plain-language message when it is not.
validation service	A class such as <code>ContactValidator</code> that checks a model without touching any control, so the same rules run from the form, a grid or a unit test.
<code>ValidationMessage</code> record	A tiny result type holding a <code>Field</code> name and a <code>Message</code> ; the service returns a list of them instead of setting errors itself.
<code>ApplyModelValidation</code>	The form method that clears old errors, maps each field name back to its control with <code>errorProvider.SetError</code> , fills the summary and reports whether everything passed.
<code>ValidationRule</code>	The Wisej.NET base class behind every rule of the Validation extender; deriving from it turns your own check into a rule you can attach to any control.
<code>OnValidating</code> / <code>OnValidated</code> / <code>OnControlCreated</code>	The three overrides of a rule class: test the value and return true or false, act after a value has passed, and prepare once the control exists in the browser.
cross-field rule	A rule whose answer depends on two or more controls, such as start date before end date, so its error belongs on every control involved.

3. Lesson narrative

Start with a method that returns a message

The simplest custom rule is an ordinary C# method. It reads the values it needs, asks one question and returns a string: empty when everything is fine, a sentence the user can act on when it is not.

`ValidateStartEndDates` in the lab reads `dtpStart.Value` and `dtpEnd.Value`, returns "Both dates are required." when either is missing, "Start date must be before end date." when the order is wrong, and "" otherwise.

Notice what the method does *not* do: it does not touch the `ErrorProvider`. A second small method takes the returned text and passes it to `errorProvider.SetError` for each picker. Because an empty message clears an error, the same two lines both show the problem and remove it once it is fixed. Keeping the decision (the string) separate from the display (the `SetError` calls) is the habit that makes the next step easy.

Move repeated rules into a validation service

Once the same check is needed in a second place, for example the intake form and a quick-add dialog, a method on one form is no longer enough. The lab moves the name, email and birth-date rules into a `ContactValidator` class. Its `Validate` method receives a `ContactEditModel`, a plain object with `Name`, `Email` and `BirthDate` properties, and returns a list of `ValidationMessage(Field, Message)` records: "Name is required.", "Email must contain @.", "Birth date cannot be in the future."

The service knows nothing about text boxes, so you can call it from a test with a hand-made model and check the list it returns, with no browser and no running Wisej.NET session. The form then needs one adapter, `ApplyModelValidation`. It calls `errorProvider.Clear()`, runs the validator, uses a `switch` to turn `"Name"`, `"Email"` and `"BirthDate"` into `txtName`, `txtEmail` and `dtpBirthDate`, sets each error, joins every message into `validationSummaryLabel`, and returns true only when the list is empty. A message with no matching control still reaches the summary, so nothing is lost. Module 5 will build the model through data binding; here you fill it from the controls yourself.

Create a class derived from ValidationRule

Some rules belong to a single kind of control and should travel from form to form like the built-in ones. For those, derive from `ValidationRule`. `MinimumAgeValidationRule` has a `MinimumAge` property that defaults to 18, and its constructor calls `base("MinimumAge", "The person is too young.")`, which gives the rule its name and the invalid message the extender displays when the rule fails.

You attach it exactly like the Module 3 rules, by passing an instance to `validation.SetValidationRules` for `dtpBirthDate`. From then on the Validation extender runs it whenever the picker is validated, when the user leaves the field and when `ValidateChildren()` runs on Save, and routes the message to the same `ErrorProvider` as every other rule.

What OnValidating, OnValidated and OnControlCreated are for

`OnValidating(Control control)` is the heart of the rule. It returns `true` when the value is valid and `false` when it is not; it does not return a message, because the message was supplied to the constructor. The lab version checks that the control is a `DateTimePicker` whose `Value` is a `DateTime`, computes the age and subtracts one year if the birthday has not happened yet this year:

```
public override bool OnValidating(Control control)
{
    if (control is DateTimePicker picker && picker.Value is DateTime birthDate)
    {
        var today = DateTime.Today;
        var age = today.Year - birthDate.Year;
        if (birthDate.Date > today.AddYears(-age))
            age--;
        return age ≥ MinimumAge;
    }
    return true; // nothing to check: Required decides about empty values
}
```

The other two overrides are optional. `OnValidated` runs after a value has passed, which makes it the place for follow-up work such as formatting the accepted value. `OnControlCreated` runs once the control has been created on the client, so any setup that needs the browser widget to exist belongs there rather than in the constructor. For a beginner rule, overriding `OnValidating` alone is normal.

Cross-field rules that do not confuse users

A start date after an end date is not the fault of either picker alone, so do not blame one of them. Set the same message on `dtpStart` and `dtpEnd`, add one line to the summary, and clear both as soon as the

order is right. Avoid `e.Cancel = true` for this kind of rule: it would lock focus in the first picker while the fix may be in the second.

Then make Save the final gate. `btnSave_Click` first runs the field rules with `ValidateChildren()`, which includes `MinimumAgeValidationRule`. If that passes it builds the model and calls `ApplyModelValidation`, and after that runs the date check, because `errorProvider.Clear()` would otherwise wipe the date errors. Only when every layer passes does it call the repository. A failed rule is normal product feedback, not an exception, so the database is never opened for data you already know is wrong.



Validation layers habit

Name the layer before you write the check. Field validation protects editing and runs when the user leaves a control; model validation protects the data and runs the same rules from the form, the grid and any service; save validation protects the database and runs every layer again before anything is written. Show every error near its field and in a visible summary, in plain language that says what to do next, and keep user-correctable mistakes separate from system failures.



Learning objectives

By the end of this module you'll be able to:

- ✓ Write custom validation methods for beginner-level business rules.
- ✓ Move repeated rules into a reusable validation service.
- ✓ Create a custom class derived from `ValidationRule`.
- ✓ Understand `OnValidating`, `OnValidated` and `OnControlCreated`.
- ✓ Apply cross-field validation without confusing users.

Lab / Exam Guide

What you'll build in the hands-on lab, the suggested approach, and the required deliverables.



Hands-on lab

Add custom validation to ValidationClinic in three steps. Write `ValidateStartEndDates` as a method that returns an empty string when valid and a message when the start date is after the end date, and set that message on both `dtoStart` and `dtoEnd`. Move the name, email and birth-date rules into a `ContactValidator` service that returns a list of `ValidationMessage` records and map each field name back to its control in `ApplyModelValidation`. Then create `MinimumAgeValidationRule` deriving from `ValidationRule` with a `MinimumAge` property, override `OnValidating` to compute the age from the `DateTimePicker`, attach it to `dtoBirthDate` through the `Validation` extender, and run the service in `btnSave_Click` before anything is saved.

Starting point

Open ValidationClinic as you left it after Module 3: the intake form has a `Validation` extender and an `ErrorProvider` checking email, age, phone, credit limit and customer code, plus a `validationSummaryLabel`. The `dtoBirthDate`, `dtoStart` and `dtoEnd` pickers have no rules yet, and `btnSave_Click` saves whatever `ValidateChildren()` lets through.

Suggested approach

- ✓ Write `ValidateStartEndDates` so it returns "" or a message, then a small helper that passes that same string to `errorProvider.SetError` for both `dtoStart` and `dtoEnd`. Try a wrong order, then fix it, and check that both icons appear and both disappear.
- ✓ Create a `ValidationMessage(Field, Message)` record and a `ContactValidator` class whose `Validate` method checks a `ContactEditModel` for a blank name, a missing or @-less email and a birth date in the future. It must not reference any control.
- ✓ Add `ApplyModelValidation` to the form: clear the `ErrorProvider`, run the validator, map "Name", "Email" and "BirthDate" to their controls, fill the summary label and return whether the list was empty.
- ✓ Create `MinimumAgeValidationRule` deriving from `ValidationRule`, call `base("MinimumAge", "The person is too young.")`, override `OnValidating` to compute the age, and attach an instance to `dtoBirthDate` with `validation.SetValidationRules`.
- ✓ Rewrite `btnSave_Click` in layers: stop if `ValidateChildren()` fails, stop if `ApplyModelValidation` or the date check fails, and only then call the repository inside a `try/catch` for real system failures. Add a short note to your lab file naming each API you used.

Required deliverables

- ✓ `ValidateStartEndDates` method that sets the same error on both date pickers and clears both when fixed
- ✓ `ContactValidator` service returning `ValidationMessage` records for name, email and birth date

- ✓ ApplyModelValidation method mapping each field name to its control and the summary
- ✓ MinimumAgeValidationRule class overriding OnValidating, attached to dtpBirthDate
- ✓ Save handler that runs field rules, then the validator service, and only then saves

Student review questions

- ✓ The minimum-age check could live in `ContactValidator` or in `MinimumAgeValidationRule`. What does the user experience differently in each case, and when would you want both?
- ✓ Your Save handler calls `ApplyModelValidation` after setting the date errors, and the date icons never appear. Explain why, and what order fixes it.
- ✓ How would you prove that `ContactValidator` rejects an email without @ without starting the app or opening a browser?



Validation layers habit

Name the layer before you write the check. Field validation protects editing and runs when the user leaves a control; model validation protects the data and runs the same rules from the form, the grid and any service; save validation protects the database and runs every layer again before anything is written. Show every error near its field and in a visible summary, in plain language that says what to do next, and keep user-correctable mistakes separate from system failures.