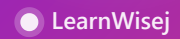


Programmatic Validation and ErrorProvider Feedback

Lesson Guide · Lab / Exam Guide



Lesson Guide

One ErrorProvider per form, SetError and clearing errors per control, InvalidMessage for simple local feedback, a small FieldValidation result type that keeps rules out of the UI, and a visible validation summary so no error depends on hovering an icon. 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

One ErrorProvider per form, SetError and clearing errors per control, InvalidMessage for simple local feedback, a small FieldValidation result type that keeps rules out of the UI, and a visible validation summary so no error depends on hovering an icon.

1. The production problem

Module 1 gave the ValidationClinic intake form two **Validating** handlers, and that works for two fields. A real contact form has a name, an email, a phone number, a customer code and more, and if each field gets its own hand-written handler with its own way of showing a message, the form quickly becomes inconsistent: one field shows an icon, another shows nothing, a third leaves an old error on screen after the user fixed it. This module replaces that drift with one pattern: small helper methods decide what is wrong, one method shows the result, and a visible summary tells every user what to fix without making them hover over anything.

2. Key concepts

Concept	Why it matters
<code>ErrorProvider</code>	An extender component that draws an error icon next to any control you give a message to and shows that message as a tooltip; one instance serves the whole form.
<code>ContainerControl</code>	The form or user control the provider belongs to; setting it to <code>this</code> tells the provider which surface its icons are drawn on.
<code>setError(control, message)</code>	Shows <code>message</code> beside <code>control</code> ; calling it with an empty string or <code>null</code> removes the error for that one control only.
<code>GetError</code> , <code>HasErrors</code> , <code>Clear()</code>	Read one control's current message, ask whether any error remains on the form, or remove every error at once when the whole form is reset.
<code>InvalidMessage</code>	A property on editor controls that support the <code>IValidation</code> interface; the control shows the message itself as an invalid tooltip, with no separate provider involved.
<code>FieldValidation</code> record	A tiny result type holding the control, a field name and a message, where an empty message means valid; it lets rules return answers instead of changing the screen.
<code>validationSummaryLabel</code>	A plain Label that lists every current problem in text, visible only while errors exist, so feedback never depends on seeing or hovering an icon.

3. Lesson narrative

Showing and clearing field errors with ErrorProvider

Create one `ErrorProvider` for the form, either by dropping it from the Toolbox onto the designer or as a field created in code, and set its `ContainerControl` to the form. Two optional settings make it calmer for users: `BlinkStyle = ErrorBlinkStyle.NeverBlink` stops the icon from flashing, and `SetIconAlignment(txtEmail, ErrorIconAlignment.MiddleRight)` with `SetIconPadding(txtEmail, 4)` places the icon just to the right of the field, where the eye expects it.

Showing an error is one call: `errorProvider.SetError(txtName, "Name is required.")` . Remember that Wisej.NET runs this on the server; the browser simply receives the updated icon and tooltip. Clearing is the same call with an empty string. That detail matters more than it looks. When the user fixes the name, you clear the error on `txtName` and nothing else. `Clear()` removes every error on the form, so keep it for the moments when the whole form really starts over: loading a different contact, resetting the form, or starting a full validation pass that immediately re-applies every current error. `GetError(control)` tells you what a single control currently shows, and `HasErrors` is a quick final check before Save continues.

InvalidMessage or SetError?

Wisej.NET gives you a second, lighter way to give feedback. Editor controls that implement `IValidation` have an `InvalidMessage` property. Set it inside the control's `Validating` handler when the value is

wrong, set it back to an empty string when the value is right, and the control shows the message as its own invalid tooltip. No provider, no icon placement, nothing shared.

That makes `InvalidMessage` a good fit for a very simple form or for one field whose feedback should stay local. `SetError` is the better fit when a form has several fields, because all errors then look the same, can be read back with `GetError`, and can be counted with `HasErrors`. In the lab you use both on purpose: the customer code field uses `InvalidMessage`, every other field goes through the `ErrorProvider`, and you write a short note comparing what the user sees.

A summary label everyone can read

An error icon is small, it is only a colour and a shape, and its text appears only when the pointer rests on it. A keyboard user, a user on a touch screen, or a user with low vision can miss it completely. So every form in this course also shows its errors as plain text. The simplest version is a Label called `validationSummaryLabel` placed above the Save button, with one line per problem, such as *Email: Enter a valid email address*.

Two rules keep the summary honest. It is visible only while at least one error exists, so an empty red box never lingers. And it is rebuilt from the same results that set the icons, so the icons and the summary can never disagree. A visible summary also makes the form easier to test: you can check the label text instead of hovering over icons.

One helper method for several fields

The key step in this module is to separate deciding from displaying. Each rule becomes a small method, `ValidateName`, `ValidateEmail` or `ValidatePhone`, that reads a value and returns a `FieldValidation` record. It never calls `SetError` and never touches a Label. A single `Apply` method then takes all the results and does the display work in one place.

```
public sealed record FieldValidation(Control Control, string FieldName, string Message)
{
    public bool IsValid => string.IsNullOrEmpty(Message);
}

private bool Apply(params FieldValidation[] results)
{
    foreach (var result in results)
        errorProvider.SetError(result.Control, result.Message);
    validationSummaryLabel.Text = string.Join("\n", results
        .Where(r => !r.IsValid).Select(r => $"{r.FieldName}: {r.Message}"));
    validationSummaryLabel.Visible = results.Any(r => !r.IsValid);
    return results.All(r => r.IsValid);
}
```

Notice that `Apply` calls `SetError` for every result, valid or not. A valid result has an empty message, so passing it clears that control's icon. That is exactly how fixing one field clears only that field's error: the other results still carry their messages, so their icons stay. The Save button calls `Apply(ValidateName(), ValidateEmail(), ValidatePhone())` and returns early if it gets `false`.

Field feedback versus business-rule feedback

Not every message belongs beside a field. *Field feedback* is about the value in one control: it is empty, it has no @, the phone number is too short. The user can fix it without leaving that control, so it goes beside the control and in the summary. *Business-rule feedback* is about what the data means: a contact with this name already exists, or two dates do not fit together. Those rules often need other fields or stored data, and they usually run on Save rather than when the user tabs out of a field.

Keep the two apart in your code and in your wording. Field helpers stay small and fast. A business rule can still point at the most relevant control with `SetError`, but its message explains the rule in plain language and appears in the summary so the user understands why Save stopped. Neither of them is an exception: a bad email is an expected mistake with a normal product message, while a lost database connection is a system failure handled separately. Later modules move business rules into a reusable service; this module sets up the habit.



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:

- ✓ Use `ErrorProvider` to show and clear field-level errors.
- ✓ Understand `InvalidMessage` versus `ErrorProvider.SetError`.
- ✓ Create a validation summary label for accessibility.
- ✓ Validate several fields from a single helper method.
- ✓ Separate field feedback from business-rule feedback.

Lab / Exam Guide

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



Hands-on lab

Turn the ValidationClinic contact form into a programmatic validation form. Create one ErrorProvider for the form with ContainerControl set, NeverBlink and a right-aligned icon, write a FieldValidation record plus ValidateName, ValidateEmail and ValidatePhone helpers that return a message instead of touching the UI, and an Apply method that calls SetError for each result and fills a validationSummaryLabel that is visible only while errors exist. Use InvalidMessage on the customer code field to compare the two feedback styles, and make sure fixing one field clears only that field's error.

Starting point

You open the `ValidationClinic` project you built in Module 1 — or, if you are joining here, the Module 1 build from the `public samples repository`. The customer intake form already has `txtName`, `txtEmail`, a Save button and a Cancel button, with the `Validating` handlers and the `ValidateChildren()` guard you wrote last time; in this lab you add `txtPhone`, `txtCustomerCode` and a summary Label, and move the checks into helper methods.

Suggested approach

- ✓ Add an `ErrorProvider` to the form and, in a `SetupValidation` method called from the constructor, set `ContainerControl = this`, `BlinkStyle = ErrorBlinkStyle.NeverBlink`, and `SetIconAlignment(..., ErrorIconAlignment.MiddleRight)` with a small `SetIconPadding` for each field.
- ✓ Declare the `FieldValidation` record, then write `ValidateName`, `ValidateEmail` and `ValidatePhone` so each one trims the value, checks it and returns a record with a plain-language message, or an empty message when the value is fine. None of them may call `SetError`.
- ✓ Place a Label named `validationSummaryLabel` above the buttons with `Visible = false`, and write `Apply(params FieldValidation[] results)` so it sets or clears each control's error, fills the label with one line per problem, shows the label only while problems exist, and returns whether everything is valid.
- ✓ Call `Apply(ValidateName(), ValidateEmail(), ValidatePhone())` at the top of `btnSave_Click` and return when it is `false`. Have each field's `Validating` handler set or clear only its own control's error from its helper's result and, while the summary is visible after a failed Save, run the full `Apply` again so the list shrinks as fields are fixed.
- ✓ In `txtCustomerCode_Validating`, use `InvalidMessage` instead of the provider, then run the form with several invalid values, fix one field at a time, and write a short note on how the two feedback styles differ for the user.

Required deliverables

- ✓ Form-level ErrorProvider with ContainerControl, BlinkStyle and icon alignment configured

- ✓ FieldValidation record and ValidateName, ValidateEmail and ValidatePhone helper methods
- ✓ Apply method that sets or clears each control's error and returns whether all fields are valid
- ✓ validationSummaryLabel listing every current problem and hidden when there are none
- ✓ One field using InvalidMessage instead of ErrorProvider, with a note comparing the two

Student review questions

- ✓ A teammate's `txtName_Validating` handler calls `errorProvider.Clear()` as soon as the name becomes valid. What goes wrong for a user who also typed an invalid email, and what should the handler call instead?
- ✓ The customer code field shows its message through `InvalidMessage`. What would the summary label need to know about that field to list it too, and what does that tell you about when a shared provider is the better choice?
- ✓ A user types a well-formed email that already belongs to another contact. Is that field feedback or business-rule feedback, when should the check run, and where should the message appear?



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.