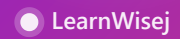


Production Validation UX, Custom Error Providers and Capstone

Lesson Guide · Lab / Exam Guide



Lesson Guide

Putting every layer together: a boring, repeatable Save pipeline, a custom `ILoggerProvider` that feeds a summary panel, icon and summary providers side by side, ten validation UX rules, and the `ValidationClinic` capstone that proves where each check belongs. 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

Putting every layer together: a boring, repeatable Save pipeline, a custom `ILoggerProvider` that feeds a summary panel, icon and summary providers side by side, ten validation UX rules, and the `ValidationClinic` capstone that proves where each check belongs.

1. The production problem

A user filling in the `ValidationClinic` intake form never asks which API found a mistake; they only notice whether the screen tells them what is wrong and lets them finish. After six modules the app has field handlers, extender rules, a validator service, annotated models and grid rules, but each was added on its own, and nothing yet guarantees they all run, in the right order, every time Save is clicked. This module joins the layers into one predictable Save pipeline and one visible summary, then asks you to prove the design in a capstone.

2. Key concepts

Concept	Why it matters
Save pipeline	A fixed sequence inside <code>btnSave_Click</code> that commits pending edits, runs each validation layer in turn and returns at the first layer that fails, so nothing reaches the repository unchecked.
Early return	Each stage ends the handler with <code>return</code> when it finds errors, so later stages never run against data an earlier stage already rejected.
<code>IErrorProvider</code>	The small interface the Wisej.NET validation system sends messages to: <code>SetError(Control, string)</code> to set or clear a message and <code>GetError(Control)</code> to read it back.
<code>SummaryErrorProvider</code>	Your own <code>IErrorProvider</code> that keeps a dictionary of control-to-message pairs and rewrites a summary label from it every time an error is set or cleared.
Dual providers	A single <code>SetError</code> helper that writes the same message to the built-in <code>ErrorProvider</code> (the icon beside the field) and to the summary provider (the list at the top).
User error vs system failure	A mistake the user can fix, such as a duplicate name, becomes a field message; an unexpected exception is logged for developers and shown as a calm, generic message.
Boundary test	A test value sitting exactly on a rule's edge, like age 0 or 120, which catches the off-by-one mistakes that ordinary valid and invalid values miss.

3. Lesson narrative

A Save button that is deliberately boring

A good Save handler does the same steps in the same order on every click and stops as soon as something is wrong. In ValidationClinic the order is: clear the summary, call `bindingSource.EndEdit()` so the value still being typed is pushed into the model, call `ValidateChildren()` so every control's field rules run, run the DataAnnotations check with `Validator.TryValidateObject`, run `ContactValidator.Validate` for business rules, check that no grid row still carries an error, and only then save. `EndEdit` comes first because every model check after it would otherwise inspect stale data.

```

ResetSummary();
bindingSource.EndEdit();
if (!this.ValidateChildren()) return;

var model = (ContactEditModel)bindingSource.Current;
if (!ApplyAnnotationErrors(model)) return;

var businessErrors = contactValidator.Validate(model);
if (businessErrors.Count > 0) { ApplyBusinessErrors(businessErrors); return; }

if (GridHasErrors()) return;

try { SaveContact(model); MessageBox.Show("Saved.", MessageBoxIcon.Information); }
catch (DuplicateNameException) { SetError(txtName, "A contact with this name already exists."); }

```

The helper names here (`ResetSummary` , `ApplyAnnotationErrors` , `GridHasErrors`) are your own small methods, not framework members. The save itself sits inside `try` / `catch` , and the final `catch` (`Exception ex`) (not shown) writes the exception with `System.Diagnostics.Debug.WriteLine` and shows a friendly `MessageBox.Show` text. The pipeline is boring on purpose: when a tester reports "Save let a bad email through", you can point to the exact stage that should have stopped it.

A summary panel through IErrorProvider

The Wisej.NET validation system does not care what displays its messages, only that the target implements `IErrorProvider` . The built-in `ErrorProvider` is one implementation and a `Label` can be another. Your `SummaryErrorProvider` is a third: its constructor receives the summary label, `SetError` removes the control's entry when the message is blank or stores it otherwise, then joins the distinct messages into the label's `Text` and sets `Visible` only while at least one error exists. `GetError` returns the stored message or an empty string.

You connect it with two lines: create the provider around `validationSummaryLabel` , then assign `validation.ErrorProvider = summaryProvider` . From that moment every rule the Validation extender runs, including your Module 4 `MinimumAgeValidationRule` , reports into the panel without any extra code in the rule itself.

Field, grid and model errors in one place

Errors now arrive from three directions. Field errors come from `Validating` handlers and extender rules. Model errors come from `ValidationResult` member names and from `ContactValidator` messages, which you map back to controls with the field map from Module 5. Grid errors live on cells and rows as `ErrorText` , set in `CellValidating` , `ValidateRow` and `DataError` . The user should not have to know which of these produced a message.

The way to achieve that is to route form and model errors through one `SetError(Control, string)` helper in the form, which calls `errorProvider.SetError` for the icon and `summaryProvider.SetError` for the list. Grid errors stay on the grid, where the user can see the red cell, and the pipeline's grid stage adds one summary line such as "Row 3: Closed contacts require a closed date." so the problem is also visible at

the top. Clearing works the same way in reverse: a blank message removes the icon and the summary line together.

Ten rules for validation people can use

Most of good validation UX is discipline, not code. The ten rules below, grouped into six lines, are the checklist the capstone is reviewed against:

- Put each message next to its field, and repeat it in a summary for the whole form.
- Write plain language that says what to do next, and never blame the user.
- Never show stack traces, SQL text or conversion exceptions.
- Never save invalid data, and never trap the user: do not block Cancel or Close without a real reason.
- Keep slow or remote checks out of per-keystroke validation; run them on Save.
- Run the same model rules from the form, the grid and any service, and test required, invalid, valid and boundary values.

A summary panel matters for accessibility too: an error that only appears when someone hovers an icon is invisible to keyboard users and easy to miss for everyone. A message a screen reader can reach as text is part of the design, not decoration.

The ValidationClinic capstone

The capstone is not a large application. It is proof that you know where each check belongs.

ValidationClinic must have a required name and email, an `ErrorProvider` with a visible summary, the Validation extender with at least three built-in rules, one custom `ValidationRule` class, a validator service used by Save, DataAnnotations or `IDataErrorInfo` model validation, email and age cell validation plus a `DataError` handler on the grid, the custom `IErrorProvider` summary and a short README explaining the design.

You then demonstrate it in a short, fixed script: leave an invalid field, click Save and watch

`ValidateChildren` stop it, show a model error that appears only after `EndEdit`, type a bad email into a grid cell, point at the summary panel listing everything, then correct each problem and save successfully. Write the README test table first, with required, invalid, valid and boundary values for every rule, and the demo becomes a matter of running your own tests out loud.



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:

- ✓ Build a full Save pipeline for a beginner Wisej.NET form.
- ✓ Create a custom IErrorProvider for a summary panel.
- ✓ Combine field errors, grid errors and model errors.
- ✓ Design validation UX that is clear, accessible and testable.
- ✓ Complete the course capstone application.

Lab / Exam Guide

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



Hands-on lab

Complete the ValidationClinic capstone. Write `btnSave_Click` as a fixed pipeline: reset the summary, call `bindingSource.EndEdit()`, stop if `ValidateChildren()` fails, stop if the `DataAnnotations` check fails, stop if `ContactValidator` returns business errors, stop if the contacts grid still has row errors, and only then save inside a `try/catch` that turns a duplicate name into a field error and any other exception into a friendly message. Create `SummaryErrorProvider` implementing `IErrorProvider`, assign it to `validation.ErrorProvider`, and route errors through a `SetError` helper that writes to both the icon `ErrorProvider` and the summary provider. Test required, invalid, valid and boundary values and write a short README explaining the validation design.

Starting point

ValidationClinic as you left it after Module 6: the intake form has field handlers, extender rules, `MinimumAgeValidationRule`, a `ContactValidator` service and a `ContactEditModel` bound through a `BindingSource`, and the contacts grid validates its email, age and closed-date cells. Every layer works on its own, but Save does not yet run them all in order, and errors appear in different places depending on which layer found them.

Suggested approach

- ✓ Write `SummaryErrorProvider` first: a class implementing `IErrorProvider` that takes the summary `Label` in its constructor, stores control-to-message pairs, removes an entry when the message is blank, and rewrites the label's text and visibility on every `SetError`. Implement `GetError` so it returns an empty string for a control with no error.
- ✓ Create the provider around `validationSummaryLabel`, assign `validation.ErrorProvider = summaryProvider`, and add a form-level `SetError(Control, string)` helper that calls `errorProvider.SetError` and `summaryProvider.SetError` with the same message. Replace direct `SetError` calls in your model and business mapping code with the helper.
- ✓ Rewrite `btnSave_Click` as the fixed pipeline: reset the summary, `bindingSource.EndEdit()`, `ValidateChildren()`, the `Validator.TryValidateObject` check, `ContactValidator.Validate`, then a grid check that skips the placeholder new row and reports any row still carrying `ErrorText`. Each stage returns as soon as it fails.
- ✓ Wrap only the save in `try / catch`: catch `DuplicateNameException` first and turn it into an error on `txtName` through your helper, then catch `Exception`, log it with `System.Diagnostics.Debug.WriteLine` and show a friendly `MessageBox.Show` text. On success call `AlertBox.Show("Saved.", MessageBoxIcon.Information)`.
- ✓ Write the README: one short paragraph per layer (field, model, save), a table of test cases with required, invalid, valid and boundary values for every rule, and the capstone checklist. Then rehearse the demo script until you can run it in five minutes.

The capstone is reviewed against ten required features: required customer name and email; an `ErrorProvider` with a visible summary; the Validation extender with at least three built-in rules; one custom `ValidationRule` class; one validator service used by Save; DataAnnotations or `IDataErrorInfo` model validation; email and age cell validation in the grid; a grid `DataError` handler; the custom `IErrorProvider` summary; and a short README. The demo follows a fixed order: show an invalid field, show Save stopped by `ValidateChildren`, show a model error that appears after `EndEdit`, show grid cell validation, show the summary provider listing everything, then correct the data and save successfully. Before you submit, check your screens against the ten UX rules from the lesson: messages near the field and in a summary, plain language that says what to do, no blame, no stack traces, no invalid saves, Cancel and Close never blocked without reason, no remote checks on every keystroke, the same model rules everywhere, and all four kinds of test value.

Required deliverables

- ✓ Save pipeline that runs EndEdit, ValidateChildren, model, business and grid checks before saving
- ✓ SummaryErrorProvider class implementing IErrorProvider SetError and GetError
- ✓ SetError helper writing to both the icon ErrorProvider and the summary provider
- ✓ try/catch around the save separating a duplicate-name error from system failures
- ✓ README with the validation design and test cases for required, invalid, valid and boundary values

Student review questions

- ✓ A tester types a new email into the form and clicks Save immediately; the old email is saved. Which pipeline stage is missing or misplaced, and why does its position matter?
- ✓ Your `SummaryErrorProvider` shows "Email is required." twice after the user fixes and breaks the field again. Which part of `SetError` should prevent that, and what does it depend on?
- ✓ The repository throws an exception with a database message when the network drops. Walk through what the user sees and what the developer can find afterwards, and explain why this differs from the duplicate-name case.



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.