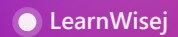


Validation Fundamentals in Wisej.NET

Lesson Guide · Lab / Exam Guide



Lesson Guide

What validation means in a server-side Wisej.NET UI: the Validating and Validated lifecycle, cancelling invalid input with `e.Cancel`, `AutoValidate` focus behavior, `CausesValidation` on Cancel buttons, and `ValidateChildren` as the guard on Save. 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

What validation means in a server-side Wisej.NET UI: the Validating and Validated lifecycle, cancelling invalid input with `e.Cancel`, `AutoValidate` focus behavior, `CausesValidation` on Cancel buttons, and `ValidateChildren` as the guard on Save.

1. The production problem

ValidationClinic is the sample application you build yourself as you work through this course — the Lab / Exam Guide for this module says exactly how to start it, and the AI Coding Exercise can have an assistant assemble it for you. Its customer intake form currently accepts anything: a blank name, an email with no @, or a Save click on a form nobody filled in. The first idea most beginners have is an `if` statement inside the Save button, but that gives the user no feedback while they work and turns Save into a wall of errors at the very end. This module puts validation where it belongs in a Wisej.NET form: on each editor when the user leaves it, and once more when the user clicks Save.

2. Key concepts

Concept	Why it matters
validation	The decision whether a value is acceptable before the application uses it, followed by visible feedback and, when the value is wrong, a refusal to continue.
<code>Validating</code>	The event a control raises when the user tries to leave it (or when code asks for validation); its handler reads the value and is the one place where accept or reject is decided.
<code>CancelEventArgs.Cancel</code>	The flag a <code>Validating</code> handler sets to <code>true</code> to reject the value: the control counts as invalid, focus can be kept in it, and <code>ValidateChildren()</code> reports failure.
<code>Validated</code>	The event raised only after <code>Validating</code> finished without cancelling; the place for tidy-up such as trimming text, never for saving.
<code>AutoValidate</code>	A property of a form, page or user control that decides what happens to focus when a child control fails validation.
<code>CausesValidation</code>	A per-control switch; set to <code>false</code> on a button such as Cancel, clicking it does not validate the field the user is leaving.
<code>ValidateChildren()</code>	A container method that runs validation for its child controls, including fields the user never visited, and returns <code>false</code> if any handler cancelled.

3. Lesson narrative

Validation in a server-side UI

In Wisej.NET you build a form the way you build any C# class: a `CustomerIntakeForm` with fields called `txtName`, `txtEmail` and `btnSave`. That form object and its controls live on the server, inside the user's session. The browser draws them and reports what the user does (typing, clicking, moving focus), and your event handlers run as ordinary C# on the server. So when the user presses Tab to leave `txtEmail`, the browser tells the server that focus is moving, the server raises the validation event for `txtEmail`, your handler reads `txtEmail.Text`, and the result, such as an error icon or focus kept in place, is sent back and drawn.

This has two practical consequences. The decision is made by your server code, so the browser cannot skip it. And every validating focus change is a round trip, so a handler should be quick and predictable: look at this one field, then set or clear a message. A lookup against a remote service on every Tab press makes the form feel slow; keep that kind of check for Save. Finally, a validation message is product text written for the user, such as "Enter the customer's name.", not an exception message.

Validating decides, Validated tidies

The lifecycle has a fixed order. The user edits a control, then tries to leave it, click a button that validates, or save the form. The control raises `Validating`. Your handler examines the value: if it is invalid, it sets a

message and cancels; if it is valid, it clears any old message. Only when `Validating` did not cancel does the control raise `Validated`.

Keep the roles apart. `Validating` is where the decision is made. `Validated` is for cleanup on a value you already accepted, such as trimming the spaces a user typed around a name. Never save from `Validated`: a user can validate the same field many times before clicking Save. You can attach both handlers from the events list in the Visual Studio Properties window, or in the constructor with `+=`.

```
private void txtName_Validating(object sender, CancelEventArgs e)
{
    if (string.IsNullOrEmpty(txtName.Text))
    {
        e.Cancel = true;
        errorProvider.SetError(txtName, "Enter the customer's name.");
        return;
    }
    errorProvider.SetError(txtName, "");
}

private void txtName_Validated(object sender, EventArgs e)
{
    txtName.Text = txtName.Text.Trim();
}
```

Rejecting a value with `e.Cancel`

A `Validating` handler receives a `CancelEventArgs` as its second parameter. Setting `e.Cancel = true` is the rejection; the `ErrorProvider` message is the explanation, and you need both. A message without `e.Cancel` is the most common beginner bug: the field looks invalid, yet the value is accepted, focus moves on, and `ValidateChildren()` later reports that everything is fine. Cancelling without a message is the opposite problem: the user is stuck and does not know why.

For the message, add one `ErrorProvider` to the form and set its `ContainerControl` to the form. `errorProvider.SetError(txtName, "Enter the customer's name.")` shows an error icon next to the control carrying that text, and `SetError(txtName, "")` removes it. The clearing branch matters: without it, a corrected field keeps showing its old error. Module 2 covers `ErrorProvider` in depth. Also, do not throw exceptions for ordinary mistakes like a missing `@`. Exceptions are for exceptional failures, such as a repository that cannot be reached.

Choosing an `AutoValidate` mode

What happens to focus when a handler cancels is decided by the `AutoValidate` property of the container: the form, page or user control that holds the editors. There are four choices:

- `EnablePreventFocusChange`: focus stays in the invalid field until it is corrected.
- `EnableAllowFocusChange`: focus may move and the error stays visible; Save still refuses.
- `Disable`: nothing validates on focus change; validation runs only when code calls `Validate()` or `ValidateChildren()`.
- `Inherit`: the container uses its parent's setting.

The lab form has two required fields, so `EnablePreventFocusChange` is a clear, beginner-friendly choice. It is also strict: if the name is blank and the user clicks Cancel, the invalid TextBox may refuse to give up focus and Cancel seems broken. The fix is `btnCancel.CausesValidation = false`. Clicking a control whose `CausesValidation` is `false` does not validate the field being left, so the Cancel handler runs and the user can always get out. Save keeps the default `true`. Your lab note records which mode you chose and why Cancel still works.

ValidateChildren() as the guard on Save

Field validation only runs for fields the user actually leaves. A user can type a name and click Save without ever entering `txtEmail`, so its `Validating` handler never ran. That is why the first statement of `btnSave_Click` is `if (!this.ValidateChildren()) return;`. `ValidateChildren()` raises validation for the form's child controls, visited or not, shows every message their handlers set, and returns `false` if any of them cancelled. An overload takes `ValidationConstraints` to limit which children are checked, for example only enabled controls, selectable controls, tab stops or immediate children, which helps when part of a form is hidden or disabled.

`Validate()` is the smaller tool: it validates the control that is losing focus, which is useful when `AutoValidate` is `Disable` and you want to check just the current field. After `ValidateChildren()` passes, put the actual save inside `try / catch`, so a failed write shows a friendly system message and is never confused with a correctable input mistake. Field validation protects editing; save validation protects saving. You need both.



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:

- ✓ Explain what validation means in a server-side Wisej.NET UI.
- ✓ Handle `Validating` and `Validated` on editor controls.
- ✓ Cancel invalid input by setting `CancelEventArgs.Cancel`.
- ✓ Choose an `AutoValidate` mode for a form or user control.
- ✓ Call `Validate` and `ValidateChildren` at the correct time.

Lab / Exam Guide

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



Hands-on lab

Create the ValidationClinic customer intake form with txtName and txtEmail TextBoxes, a Save button and a Cancel button. Attach Validating handlers that set e.Cancel = true and an ErrorProvider message when the name is blank or the email has no @, clear the message when the value is valid, and trim the name in its Validated handler. Set the form's AutoValidate mode, make sure Cancel does not cause validation so the user can always leave, and call ValidateChildren() at the top of btnSave_Click so Save stops when any field is invalid. Finish with a short lab note that explains the focus behavior you chose.

Starting point

ValidationClinic is the application you build yourself, module by module — nothing is supplied as a download and no starter solution is installed for you. Create a new Wisej.NET 4 project named `ValidationClinic` in Visual Studio, give it a main page, an editable contacts grid and a simple in-memory contact repository behind that grid, and validate nothing anywhere; every later module extends this same project. Two shortcuts if you would rather not assemble the shell by hand: this module's **AI Coding Exercise** section hands ChatGPT or Claude a ready-made prompt that builds the whole solution from the module specification, and our own finished build for every module of this course lives in the public samples repository at [LearnWisej-Samples](#), one folder per module — `Module 1` is that shell plus the answer to this lab. In this lab you add the customer intake form and give it its first two validation layers: field validation on each editor and save validation on the Save button.

Suggested approach

- ✓ Add a `CustomerIntakeForm` and use the designer to place `txtName`, `txtEmail`, `btnSave`, `btnCancel` and a `lblStatus` label. Drop one `ErrorProvider` on the form and set its `ContainerControl` to the form.
- ✓ Handle `txtName.Validating` and `txtEmail.Validating`. In each, test the value; when it is wrong, set `e.Cancel = true` and call `errorProvider.SetError` with a plain-language message, and when it is right, call `SetError` with an empty string.
- ✓ Handle `txtName.Validated` and trim the text there, so the value is tidied only after it has been accepted.
- ✓ Set the form's `AutoValidate`, then set `btnCancel.CausesValidation = false`. Run the form, leave the name blank and click Cancel to prove the user can still leave.
- ✓ Make `if (!this.ValidateChildren()) return;` the first line of `btnSave_Click`, then save inside `try / catch`. Test Save with a field you never clicked into, and write the lab note.

Required deliverables

- ✓ Customer intake form with txtName and txtEmail, a Save button and a Cancel button

- ✓ Validating handlers that set `e.Cancel = true` and an error message for a blank name and an invalid email
- ✓ Validated handler that trims the accepted name
- ✓ Save handler that calls `ValidateChildren()` and stops when it returns false
- ✓ Lab note naming the `AutoValidate` mode chosen and why `Cancel` still works on an invalid form

Student review questions

- ✓ You remove `e.Cancel = true` from `txtEmail_Validating` but keep the `SetError` call. Describe exactly what the user sees when they tab away from "maria.example.com" and then click Save.
- ✓ Which of your checks run while the user is editing, and which run only because Save called `ValidateChildren()` ? Give one user action that reaches Save with a field whose handler has never run.
- ✓ You switch the form from `EnablePreventFocusChange` to `EnableAllowFocusChange` . What changes for the user, what stays the same, and does `btnCancel.CausesValidation = false` still matter?



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.