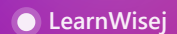


Editors, Buttons, Validation, and Feedback

Lesson Guide · Lab / Exam Guide



Lesson Guide

Choosing the editor that matches the value, a command model for buttons, ErrorProvider validation that helps rather than blocks, and non-blocking feedback with Toast and AlertBox. What this means for a Wisej.NET developer — and how to read this module.



Production focus

Choosing the editor that matches the value, a command model for buttons, ErrorProvider validation that helps rather than blocks, and non-blocking feedback with Toast and AlertBox.

1. The production problem

Editors are the first place where a control library turns into a business system. A form that looks correct but accepts ambiguous or invalid input creates support tickets and cleanup scripts long after it shipped.

Wisej.NET editors follow the familiar .NET component model but run in the browser, so one business rule can be enforced at several layers: editor properties, validation extenders, event handlers, data binding, service validation and database constraints. The UI layer's job is to prevent obvious mistakes and explain errors clearly; it must never be the only layer that protects the application.

2. Key concepts

Concept	Why it matters
value-matched editor	The editor whose type already constrains the input: <code>DateTimePicker</code> for dates, <code>NumericUpDown</code> for bounded numbers, <code>ComboBox</code> for list values, so the server never parses ambiguous text.
command model	A deliberate split between primary commands (Save, Approve, Search, Send, Export), secondary commands (ToolBar, MenuBar, SplitButton, context menu) and destructive commands that need confirmation or careful placement.
ErrorProvider	The extender that attaches a specific, actionable message to the control that is wrong, instead of a generic "Invalid value" elsewhere on the page.
blocking vs non-blocking feedback	<code>MessageBox</code> interrupts for a decision or critical confirmation; <code>AlertBox</code> and <code>Toast</code> report status without stopping the workflow.
named validators	Methods such as <code>ValidateEmail</code> or <code>ValidateCreditLimit</code> that hold one rule each, so the rule is testable and explainable rather than buried in a click handler.
busy state	Disabling Save and showing progress while an operation runs, which prevents duplicate submits and is part of reliability, not decoration.
editor UserControl	A composition of labels, editors, an <code>ErrorProvider</code> and a validate method that returns a result object, so the rest of the app never learns how the email field is checked.

3. Lesson narrative

Match the editor to the value

The first decision on any input screen is which editor represents each value: pick the control whose type already excludes impossible input. `TextBox` is for ordinary text. Set `Multiline` only when the user needs paragraphs, not because a field happens to be wide, and use `Watermark` as an example or prompt, never as a substitute for a label. When the shape of the input is predictable, such as a code, a phone-like pattern or a reference number, `MaskedTextBox` enforces that shape while the user types. When parsing and formatting a typed value is the heart of the field, `TypedTextBox` does the conversion for you.

Dates and numbers deserve their own editors. `DateTimePicker` removes parsing ambiguity and validates a date selection far more naturally than free text. `NumericUpDown`, `DomainUpDown` and `TimeUpDown` express constrained values with a minimum, a maximum and a step. `ComboBox` is the right choice when a value comes from a list; its `DropDownStyle` decides whether the user may also type a value that is not in the list. It is a poor choice when the user needs to search or browse a very large dataset: it exists for compact selection from a small or moderate list.

In the Operations Console the `CustomerEditor` applies exactly this mapping: name and email are `TextBox` controls, status and customer type are `ComboBox` controls with fixed items, the start date is a `DateTimePicker` and the credit limit is a `NumericUpDown` with a range. One habit matters with lists: keep the display text and the stored key distinct, so "Active" on screen never becomes the value you persist.

Buttons as commands, and the feedback controls

Buttons are not just clickable rectangles; a business screen needs a command model. Primary commands (Save, Approve, Search, Send, Export) are plain `Button` controls placed where the eye expects them. Secondary commands belong in a `ToolBar`, a `MenuBar`, a `SplitButton` or a context menu. A `SplitButton` is the right fit when one command has a default action and alternate actions, such as Save alongside Save and Close. Destructive commands need confirmation, a disabled state or clear placement. `CheckBox` and `RadioButton` are state controls, not commands: they belong where the user makes a choice, never where an action fires.

The feedback controls each have one role. `ToolTip` and `HelpTip` carry guidance about what a field expects; they are not error channels. `ErrorProvider` marks the control that has a validation error, and for data-bound controls it must be associated with the correct container or its icons appear in the wrong place. `MessageBox` is reserved for a blocking decision or a critical confirmation. `AlertBox` and `Toast` are for status messages that should not interrupt the workflow, and because a `Toast` can be reused and updated while visible, it doubles as progress feedback for a long-running save, alongside a `ProgressBar` when real progress is known.

Validation that helps without blocking

Validation should be visible, specific and recoverable, and it should arrive early enough to help without getting in the way of typing. The clean flow has six steps. First, set editor properties (`MaxLength`, `Minimum`, `Maximum`, fixed items) so impossible values cannot be entered at all. Second, use input-specific parsing for typed values. Third, on field validation, in the control's `Validating` handler, set or clear the `ErrorProvider` message for that one control. Fourth, on Save, run form-level validation and then service-level validation. Fifth, if anything fails, keep the user on the screen, focus the first invalid control and show specific errors. Sixth, when saving succeeds, show a non-blocking confirmation and update the status area.

The message itself should say what to fix: "Enter a credit limit between 0 and 250,000" rather than "Invalid value". Avoid mixing all validation into click handlers; it makes the screen hard to test and hard for a colleague or an AI assistant to explain. Put each rule near its control or in a named validator such as `ValidateRequiredName`, `ValidateEmail` and `ValidateCreditLimit`, and let Save call those methods. Keep the Save path explicit, validate, persist, refresh state, notify, and disable the commands while it runs so a double click cannot submit twice.

```
public EditorResult ValidateContent()
{
    var result = new EditorResult();
    this.errorProvider.SetError(this.txtEmail, "");
    this.errorProvider.SetError(this.numCreditLimit, "");
    if (!this.txtEmail.Text.Contains("@"))
        result.Add(this.txtEmail, "Enter an email address such as name@company.com.");
    if (this.numCreditLimit.Value < 0 || this.numCreditLimit.Value > 250000)
        result.Add(this.numCreditLimit, "Enter a credit limit between 0 and 250,000.");
    foreach (var error in result.Errors)
        this.errorProvider.SetError(error.Control, error.Message);
    result.FirstInvalid?.Focus();
    return result;
}
```

The CustomerEditor UserControl

This is the point where a collection of controls becomes an application-level component. The lab's `CustomerEditor` is a `UserControl` that owns its labels, its editors, one `ErrorProvider`, the Save, Reset and Validate buttons and a validate method that returns a clear result object: which controls are invalid and which messages were shown. The shell page that hosts it only asks "is this valid?" and "give me the customer"; it never learns how the email rule works. The method is named `ValidateContent` so it cannot be confused with the validation members a container control already exposes.

The design rules for input screens follow from that boundary and are the checklist the instructor grades against:

- Every editable field has a visible label or an accessible equivalent.
- The editor matches the value type; dates and numbers are not parsed from ordinary `TextBox` controls without a strong reason.
- Validation messages are actionable, `ErrorProvider` carries field errors and `Toast` or `AlertBox` carries workflow feedback.
- Commands are disabled while an operation is running, and the Save path is explicit: validate, persist, refresh state, notify.



Common pitfalls

Using `MessageBox` for every minor status message. Saving partial invalid state because validation only ran on the visible controls. Hiding validation inside anonymous event handlers. Reading a `ComboBox` value without distinguishing display text from the stored key. Forgetting that disabled buttons and busy feedback are part of reliability, not decoration.



Control-library habit

Choose the simplest native control that expresses the user's task before writing custom code. Native controls already give you design-time support, theming, events and server state; reach for an extender or a `UserControl` next, and build a JavaScript widget only when no native control fits.



Learning objectives

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

- ✓ Select the correct editor for text, numeric, date/time, masked, and selection input.
- ✓ Use Button, SplitButton, CheckBox, RadioButton, ToolTip, HelpTip, ErrorProvider, AlertBox, MessageBox, Toast, and ProgressBar appropriately.
- ✓ Implement validation that is immediate enough to help the user but not so aggressive that it blocks normal data entry.
- ✓ Create a reusable editor `UserControl` for customer/contact data.

Lab / Exam Guide

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



Hands-on lab

Build a CustomerEditor UserControl for the Operations Console with labelled fields for name, email, status, start date, credit limit and customer type, using the editor that matches each value instead of a TextBox for everything. Add ErrorProvider messages for required fields, email shape, credit-limit range and invalid dates; Save, Reset and Validate commands with disabled and busy behaviour while saving; and a Toast or MessageBox when validation or save completes.

Starting point

Open the `OperationsConsole` solution from Module 1: the shell page with its navigation, content, command and status areas already runs, and the Editors placeholder page is empty. Keep this module's work in its own `Editors` folder and namespace so the capstone can reuse the control unchanged.

Suggested approach

- ✓ Sketch the six fields and write down the editor for each one before opening the designer: `TextBox` for name and email, `ComboBox` for status and customer type, `DateTimePicker` for the start date, `NumericUpDown` for the credit limit.
- ✓ Add a `CustomerEditor` UserControl, drop the labels and editors with clear names (`txtName` , `cboStatus` , `dtpStartDate` , `numCreditLimit`), set `Minimum` , `Maximum` , `MaxLength` and fixed items so impossible values cannot be typed, and give each editor a `ToolTip` that describes what it expects.
- ✓ Add one `ErrorProvider` and named validators (`ValidateRequiredName` , `ValidateEmail` , `ValidateCreditLimit` , `ValidateStartDate`) that set or clear a specific message; call the matching validator from each editor's `Validating` event and all of them from a `ValidateContent` method that returns a result object.
- ✓ Add Save, Reset and Validate buttons; make Save run `ValidateContent` , then the service call, then a state refresh, and disable all three commands and show busy feedback while the save is in progress.
- ✓ Show a `Toast` or `MessageBox` when validation or save completes, update the shell's status area, and host the finished control on the Editors page.

Required deliverables

- ✓ CustomerEditor UserControl with labelled fields for name, email, status, start date, credit limit and customer type
- ✓ Matching editors: TextBox, DateTimePicker, NumericUpDown and ComboBox instead of free text everywhere
- ✓ ErrorProvider messages for required fields, email shape, credit-limit range and invalid dates
- ✓ Save, Reset and Validate commands with disabled and busy behaviour while saving

- ✓ Toast or MessageBox feedback when validation or save completes

Student review questions

- ✓ Which validation errors can no longer happen at all because of the editor you chose, and which still need an ErrorProvider message?
- ✓ If the user clicks Save twice in one second, what does your screen do, and where in the code is that guaranteed?
- ✓ What does the shell page know about how the email field is validated, and what would have to change if the rule moved to a service?



Control-library habit

Choose the simplest native control that expresses the user's task before writing custom code. Native controls already give you design-time support, theming, events and server state; reach for an extender or a UserControl next, and build a JavaScript widget only when no native control fits.