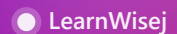


Data-Bound Validation and Model Errors

Lesson Guide · Lab / Exam Guide



Lesson Guide

Validating the model, not just the controls: BindingSource and EndEdit, DataAnnotations run with Validator.TryValidateObject, mapping MemberNames back to controls, IDataErrorInfo with a data-bound ErrorProvider, BindToDataAndErrors, and the safe save order. 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

Validating the model, not just the controls: BindingSource and EndEdit, DataAnnotations run with Validator.TryValidateObject, mapping MemberNames back to controls, IDataErrorInfo with a data-bound ErrorProvider, BindToDataAndErrors, and the safe save order.

1. The production problem

Everything ValidationClinic checks so far lives in the intake form: a `Validating` handler here, a rule on the `Validation` extender there, a `ContactValidator` call in `btnSave_Click`. But a contact does not only arrive through that form. It can be edited in the contacts grid, imported from a file or created by a service, and none of those paths run the form's handlers. This module moves the basic rules onto the `ContactEditModel` itself, so the data carries its own definition of "valid" and every screen can ask it the same question.

2. Key concepts

Concept	Why it matters
BindingSource	A component that sits between the controls and your object: its <code>DataSource</code> is the model, and each control binds to a property through it instead of being copied by hand.
EndEdit()	Pushes the value the user is still editing from the control into the model, so validation reads what is on screen rather than the last committed value.
DataAnnotations	Attributes such as <code>[Required]</code> , <code>[StringLength]</code> , <code>[EmailAddress]</code> and <code>[Range]</code> placed on model properties, each with an <code>ErrorMessage</code> written for the user.
Validator.TryValidateObject	The .NET method that runs those attributes against an object and fills a list of <code>ValidationResult</code> objects; it knows nothing about controls or Wisej.NET.
MemberNames	The property names a <code>ValidationResult</code> belongs to, which is the only link between an error message and the control that should display it.
IDataErrorInfo	An interface that lets a model answer "what is wrong with this property?" through an indexer, so the binding layer can ask for errors instead of your code pushing them.
Data-bound ErrorProvider	An <code>ErrorProvider</code> whose <code>DataSource</code> is the same binding source as the controls, so it shows the model's errors next to the bound controls without any <code>SetError</code> call.

3. Lesson narrative

Bind the form, then validate the model

Data binding connects a control property to a model property. Create a `BindingSource`, set its `DataSource` to a `ContactEditModel`, and call `txtName.DataBindings.Add("Text", contactBindingSource, nameof(ContactEditModel.Name))` for each field. From then on the form no longer copies text into an object when Save is clicked; the model is the form's data, and `contactBindingSource.Current` hands it back to you.

There is one gap a beginner almost always falls into. Binding does not update the model on every keystroke. If the user types an email and clicks Save straight away, the new text can still be sitting in the TextBox while `Email` on the model holds the old value. Validating the model at that moment checks data the user has already replaced. The fix is a single call: `contactBindingSource.EndEdit()` finishes the current edit and pushes the typed values into the model. Make it the first line of every save path, before any rule reads the model.

DataAnnotations and Validator.TryValidateObject

DataAnnotations put simple rules right beside the data they protect. On `ContactEditModel`, `Name` gets `[Required]` and `[StringLength(80)]`, `Email` gets `[Required]` and `[EmailAddress]`, and `Age` gets `[Range(0, 120)]`. Each attribute carries an `ErrorMessage` such as "Enter a valid email address." Treat those messages as product text the user will read, not as developer notes.

Attributes do nothing on their own; something has to run them. Create a `List<ValidationResult>` and a `ValidationContext` for the model, then call `Validator.TryValidateObject(model, context, results, validateAllProperties: true)`. It returns `false` when anything failed and adds one result per broken rule. Do not skip the last argument: without `validateAllProperties: true` only `[Required]` is checked, and a malformed email or an age of 300 passes silently. Note also that `[EmailAddress]` and `[Range]` treat a null value as valid, which is exactly why the required rule is a separate attribute.

IDataErrorInfo and the data-bound ErrorProvider

`IDataErrorInfo` turns the relationship around. Instead of your code running a validator and pushing messages onto controls, the model answers questions. It has two members: `Error`, a message about the object as a whole (often an empty string), and the indexer `this[string columnName]`, which returns the message for one property or an empty string when that property is fine. In the lab's second model, the indexer returns "Name is required." when `columnName` is `Name` and the name is blank.

The interface becomes useful when an `ErrorProvider` is connected to the same data. Set `errorProvider.ContainerControl` to the form and `errorProvider.DataSource` to `contactBindingSource`. The provider then asks the current item for the error of every bound property and shows each message beside the control bound to it. No `SetError` call appears anywhere in your code. .NET also has `INotifyDataErrorInfo`, which can report several errors per property and change them asynchronously, but for a beginner form `IDataErrorInfo` is the simpler, well-supported starting point.

BindToDataAndErrors at runtime

A data-bound provider needs two settings to agree: `DataSource`, the object or list, and `DataMember`, the path inside it. In the designer you set both once and never touch them again. At runtime, for example when `ValidationClinic` switches the editor from the annotated model to the `IDataErrorInfo` model, setting them one after the other leaves a moment where the new source is paired with the old member, and the provider may try to bind to something that does not exist.

`BindToDataAndErrors(source, member)` sets both in one call, so the provider only ever sees a matching pair. When the whole object is the data, pass `string.Empty` as the member: `errorProvider.BindToDataAndErrors(contactBindingSource, string.Empty)`. Use the two properties in the designer, and the method whenever the source changes while the form is open.

Map model errors to controls and the summary

`Validator.TryValidateObject` deliberately knows nothing about Wisej.NET, so the form must translate its results. Build a field map in the constructor, a `Dictionary<string, Control>` from `nameof(ContactEditModel.Email)` to `txtEmail` and so on, then loop over the results: for each name in `result.MemberNames`, look up the control and call `errorProvider.SetError(control, result.ErrorMessage)`. Finally join every message into `validationSummaryLabel` and show it, because an icon alone is easy to miss. A result whose member is not in the map still reaches the summary, so no error is ever lost. The map is the same shape as `ApplyModelValidation` from Module 4, which is why the two can share one summary.

With every layer in place, the save handler follows a fixed order: end the edit, validate the child controls, validate the model, run cross-field rules, save only if nothing failed, and catch system exceptions separately

so a database outage never looks like a typing mistake.

```
private void btnSave_Click(object sender, EventArgs e)
{
    contactBindingSource.EndEdit();           // 1. typed values into the model
    if (!ValidateChildren()) return;          // 2. field validation
    var model = (ContactEditModel)contactBindingSource.Current;
    if (!ApplyAnnotationErrors(model)) return; // 3. DataAnnotations
    if (!ApplyModelValidation(model)) return;  // 4. ContactValidator, cross-field
    try { _repository.Save(model); ShowSaved(); } // 5. only now
    catch (Exception ex) { ShowSystemError(ex); } // 6. not a validation message
}
```



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:

- ✓ Bind controls to a model and validate the model before save.
- ✓ Use DataAnnotations with Validator.TryValidateObject.
- ✓ Understand IDataErrorInfo and data-bound ErrorProvider behavior.
- ✓ Use BindToDataAndErrors when changing a provider data source at runtime.
- ✓ Map model errors to Wisej.NET controls and summaries.

Lab / Exam Guide

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



Hands-on lab

Bind the ValidationClinic intake form to a ContactEditModel through a contactBindingSource and add Required, StringLength, EmailAddress and Range annotations to its Name, Email and Age properties. In btnSave_Click call contactBindingSource.EndEdit() first, run Validator.TryValidateObject with validateAllProperties set to true, and map every ValidationResult member name to its control through a field map and the validation summary. Then implement IDataErrorInfo on a second model, point the ErrorProvider at the binding source with BindToDataAndErrors, and show the same errors appearing through data binding without a hand-written SetError call.

Starting point

ValidationClinic after Module 4: the intake form has field rules, a `ContactValidator` service, `ApplyModelValidation` and a `MinimumAgeValidationRule`, but its controls are not bound to anything, so `btnSave_Click` still reads values straight from `txtName`, `txtEmail` and `txtAge`.

Suggested approach

- ✓ Create `ContactEditModel` with `Name`, `Email` and a nullable `Age`, add the four annotations with plain-language `ErrorMessage` texts, and bind the three TextBoxes to it with `DataBindings.Add("Text", contactBindingSource, ...)`.
- ✓ Before writing any validation, prove the edit gap: type an email, click Save, and log `((ContactEditModel)contactBindingSource.Current).Email` with and without a `contactBindingSource.EndEdit()` call in front of it.
- ✓ Write `ValidateModel` around `Validator.TryValidateObject(model, context, results, validateAllProperties: true)`, then build `_fieldMap` in the constructor and an `ApplyAnnotationErrors` method that sets each control error and fills `validationSummaryLabel`.
- ✓ Rewrite `btnSave_Click` in the safe order: `EndEdit`, `ValidateChildren`, annotations, the Module 4 `ContactValidator`, save, and a separate `catch` that shows a friendly system error.
- ✓ Add a second model implementing `IDataErrorInfo`, switch the binding source to it at runtime, call `errorProvider.BindToDataAndErrors(contactBindingSource, string.Empty)`, and blank the name to watch the error appear without any `SetError`.

Required deliverables

- ✓ ContactEditModel with DataAnnotations bound to the form through a BindingSource
- ✓ Save handler that calls EndEdit before running Validator.TryValidateObject
- ✓ Field map that turns ValidationResult member names into control errors and summary lines
- ✓ IDataErrorInfo model whose errors appear through a data-bound ErrorProvider
- ✓ BindToDataAndErrors call used when the provider's data source changes at runtime

Student review questions

- ✓ A tester types a new email and presses Enter to save, and the old, valid email is what gets checked and stored. Which line is missing, and why does the form look correct on screen while the model is wrong?
- ✓ You remove `validateAllProperties: true` and every save still "works". Which of your four annotations stopped running, and what bad data can now reach the repository?
- ✓ The import service creates contacts without opening the form. Which of this module's rules still protect that path, and which ones only run because a user clicked Save?



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.