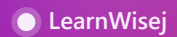


Validation, ErrorProvider, and User Feedback

Lesson Guide · Lab / Exam Guide



Lesson Guide

Three layers of validation, DataAnnotations on edit models, a TicketValidator service returning field and summary messages, ErrorProvider and summary feedback in Wisej.NET, and DbUpdateException as the final safety layer. What this means for a Wisej.NET developer working with real databases — and how to read this module.



Production focus

Three layers of validation, DataAnnotations on edit models, a TicketValidator service returning field and summary messages, ErrorProvider and summary feedback in Wisej.NET, and DbUpdateException as the final safety layer.

1. The production problem

The Module 4 ticket editor saves whatever the user typed. An empty title, a ticket with no customer, or a closed ticket due next week all travel through `SaveChangesAsync` and are stopped, if at all, by the database, which answers with a provider exception that a support agent cannot read. Users need immediate, friendly messages before the database rejects a record, and the application needs a last line of defence that stays in place when the UI is bypassed by an import, a second screen or a bug.

2. Key concepts

Concept	Why it matters
three validation layers	UI validation gives immediate feedback, domain validation enforces rules that must hold everywhere, and database constraints guarantee final consistency; each layer catches what the one above it missed.
DataAnnotations on the edit model	Attributes such as <code>[Required]</code> and <code>[StringLength]</code> on <code>TicketEditModel</code> describe user-input rules in one declarative place, separate from the EF Core entity.
explicit validation	<code>Validator.TryValidateObject</code> runs the annotations on demand; EF Core does not run them for you before <code>SaveChangesAsync</code> , so the form must call it.
TicketValidator and ValidationMessage	A service that returns <code>ValidationMessage(FieldName, Message)</code> items, so the rule set is testable and the form only maps messages to controls.
ErrorProvider	The Wisej.NET extender that shows an icon and tooltip beside the invalid control through <code>SetError</code> , and can also be bound to data errors on the current item.
validation summary	A label or <code>AlertBox</code> that lists cross-field errors which belong to no single control, such as a closed ticket with a future due date.
DbUpdateException	The exception EF Core raises when the database rejects the change; it is the final safety layer and must be translated into a friendly message, never shown raw.

3. Lesson narrative

Three layers, one final authority

A production Wisej.NET and EF Core application validates in three places. The UI layer catches obvious mistakes while the user is still on the screen: a missing title, a due date in the wrong range, a customer that was never selected. The domain layer holds business rules that must be true no matter which screen, import or background job creates the record. The database layer, with its NOT NULL columns, unique indexes, foreign keys and the `RowVersion` token from Module 2, is the final authority for consistency.

The layers are not alternatives. UI feedback is a courtesy that makes the common case fast; the database remains the only place where uniqueness or referential integrity can be guaranteed under concurrent users. The Support Desk editor therefore validates before it saves and still expects the database to say no occasionally.

Annotate the edit model, then validate it explicitly

Module 4 introduced `TicketEditModel` as the object the editor binds to. This module decorates it with `System.ComponentModel.DataAnnotations`: `[Required, StringLength(180)]` on `Title`, `[StringLength(4000)]` on `Description`, `[Required]` on `Status`, `Priority`, `CustomerId` and `CategoryId`. The foreign keys are nullable `int?` on purpose: an unselected `ComboBox` yields null, and null is what `[Required]` rejects, whereas a non-nullable `int` would silently pass as zero.

Annotations describe rules; something has to run them. `Validator.TryValidateObject` with `validateAllProperties: true` evaluates every attribute and fills a list of `ValidationResult` items. The lab wraps that call in a `TicketValidator` service that flattens the results into `ValidationMessage` records and appends the rules annotations cannot express. The cross-field rule is the canonical example: a closed ticket cannot have a future due date.

```
public IReadOnlyList<ValidationMessage> Validate(TicketEditModel model)
{
    var results = new List<ValidationResult>();
    var context = new ValidationContext(model);
    Validator.TryValidateObject(model, context, results, validateAllProperties: true);

    var messages = results
        .SelectMany(r => r.MemberNames.DefaultIfEmpty(null),
            (r, member) => new ValidationMessage(member, r.ErrorMessage ?? "Invalid
value."))
        .ToList();

    if (model.Status == "Closed" && model.DueDate?.Date > DateTime.Today)
        messages.Add(new ValidationMessage(nameof(model.DueDate), "Closed tickets cannot
have a future due date."));
    return messages;
}
```

Because the validator is a plain service with no reference to `Wisej.Web`, the five negative test cases the lab asks for run in the Tests project without a form, a session or a database.

Why mapping attributes are not a validation pipeline

EF Core also understands `[Required]` and `[MaxLength]`, but on an entity they are mapping instructions: they decide whether a column is nullable and how wide it is. EF Core does not run a full DataAnnotations validation pass before `SaveChangesAsync`. A title of 300 characters on an entity mapped to 180 is not caught in .NET; it is sent to the database, which truncates it or throws, depending on the provider. Keep persistence shape in the Fluent API on `SupportDeskContext`, keep user-input rules on the edit model, and call the validator yourself.

The practical test is simple: if a rule needs a message a human should read, it belongs on the edit model and in `TicketValidator`. If it is a fact about the column, it belongs in the model configuration and the migration.

ErrorProvider for fields, a summary for cross-field rules

Wisej.NET's `ErrorProvider` attaches a red icon with a tooltip to any control. The editor keeps one instance, created against the form, and follows a fixed sequence in `SaveAsync`: call `Clear()` so the previous run's icons disappear, call `EndEdit` on the `BindingSource` so the model holds what the user typed, run `TicketValidator.Validate`, and then walk the messages. A message whose `FieldName` is `Title`, `CustomerId`, `CategoryId` or `DueDate` becomes `errorProvider.SetError(control, message)` on the matching `TextBox`, `ComboBox` or `DateTimePicker`. The provider can also be bound to the current

item of a data source so a model that reports its own errors lights up controls automatically, and the Validation extender covers simple per-control rules without code.

Cross-field rules have no single home, so every message is also joined into a summary label, or shown in an `AlertBox` when the form is crowded. Save stays disabled while the model is invalid and is re-enabled in `finally`, and the method returns before a `DbContext` is ever created. If validation fails, EF Core is not called.

DbUpdateException as the final safety layer

Even a fully validated model can be rejected. The lab's example is a duplicate ticket number: two agents create tickets in the same second, the unique index on `Number` fires for the second save, and EF Core throws `DbUpdateException`. The same exception covers foreign-key violations when a category was deleted while the editor was open, and required-column errors that the UI never knew about.

The handler does three things. It logs the full exception, with the inner provider error, on the server through the application's logger. It shows the user one plain sentence, such as "The ticket could not be saved because the database rejected the change. Please verify required fields and lookup values." And when a lookup could be stale, it reloads the lookup lists so the user can pick a valid value. It never displays the SQL statement, the constraint name or anything that could carry a connection string.

`DbUpdateConcurrencyException` derives from `DbUpdateException`, so catch it first; Module 7 gives it its own dialog.



Common mistakes

Assuming EF Core will run all `DataAnnotations` validation automatically before `SaveChanges`; it only maps them. Showing raw SQL or provider errors to users, which leaks schema details and sometimes connection information. Putting reusable business rules only in button handlers, where they cannot be tested and are silently missing from the next screen that saves a ticket.



Data-binding habit

Keep the two lifetimes apart. Wisej.NET forms, `BindingSources` and the selected row live for the session; a `DbContext` lives for one operation. Create it from the factory, query or save, dispose it, and never let it become session state.



Learning objectives

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

- ✓ Use `System.ComponentModel.DataAnnotations` or custom validators on edit models.
- ✓ Display validation errors with `ErrorProvider`, Validation extender, labels, and summary messages.
- ✓ Understand that EF Core mapping attributes are not a full automatic UI validation system.
- ✓ Handle database constraints and `DbUpdateException` as the final safety layer.

Lab / Exam Guide

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



Hands-on lab

Add validation to the Support Desk ticket editor: DataAnnotations on TicketEditModel, a TicketValidator that returns field-level and summary errors, an ErrorProvider cleared before each run and set for Title, Customer, Category and DueDate, a summary label for cross-field rules, Save disabled while the model is invalid, a DbUpdateException handler that shows a friendly message for a duplicate ticket number, and at least five negative test cases.

Starting point

Open the `SupportDesk` solution from Module 4: the ticket browser, the Add Ticket and Edit Ticket modal forms, `TicketEditModel` and a `TicketEditorForm` with a `BindingSource` and an `ErrorProvider` already exist, and `SaveAsync` still has the validation placeholder that lets any model through to `SaveChangesAsync`. The Services project has no validator yet and the Tests project is empty.

Suggested approach

- ✓ Decorate `TicketEditModel` in the Services project with `[Required, StringLength(180)]` on `Title`, `[StringLength(4000)]` on `Description` and `[Required]` on `Status`, `Priority`, `CustomerId` and `CategoryId`; keep the foreign keys nullable so an unselected lookup is caught as null.
- ✓ Add a `ValidationMessage(FieldName, Message)` record and a `TicketValidator` service whose `Validate` method runs `Validator.TryValidateObject` with `validateAllProperties: true`, flattens the results by member name, and appends the cross-field rule that a closed ticket cannot have a future due date with a null `FieldName` or `DueDate` as the field; register it in Microsoft DI next to the context factory.
- ✓ In `SaveAsync`, call `errorProvider.Clear()` and `editBindingSource.EndEdit()`, then call the validator; write a `ShowValidation` method that maps `Title`, `CustomerId`, `CategoryId` and `DueDate` to `SetError` on the matching `TextBox`, `ComboBox` and `DateTimePicker`, joins every message into the summary label, and returns before any `DbContext` is created.
- ✓ Keep Save disabled while the model is invalid and while a save is running, restoring it in `finally`; then add a `catch (DbUpdateException)` below the concurrency catch that logs the full exception server-side, shows one friendly sentence through `AlertBox`, and reloads the lookup lists when a foreign key may be stale.
- ✓ In the Tests project write at least five negative cases against `TicketValidator` alone (empty title, title over 180 characters, missing customer, missing category, closed ticket with a future due date), then reproduce the duplicate ticket number against the unique index and confirm the message contains no SQL.

Required deliverables

- ✓ DataAnnotations on TicketEditModel and a TicketValidator returning field-level and summary errors
- ✓ ErrorProvider cleared before each run and set for Title, Customer, Category and DueDate
- ✓ Summary label for cross-field rules and Save disabled while invalid
- ✓ DbUpdateException handled with a friendly message for a duplicate ticket number
- ✓ At least five negative test cases

Student review questions

- ✓ If you removed every DataAnnotation from `TicketEditModel` but kept them on the `Ticket` entity, which of your five negative tests would still fail before `SaveChangesAsync`, and why?
- ✓ The closed-ticket rule reports `DueDate` as its field. What would change for the user if it reported no field at all, and when would that be the better choice?
- ✓ Two agents create a ticket in the same second and the second save throws `DbUpdateException`. What does the second agent see, what does the server log, and which of the two would you show to a customer?



Data-binding habit

Keep the two lifetimes apart. Wisej.NET forms, BindingSources and the selected row live for the session; a DbContext lives for one operation. Create it from the factory, query or save, dispose it, and never let it become session state.