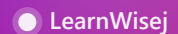


Dialogs, Validation & Workflows

Dialogs & Modal Workflows · Validation & the Save/Cancel Pattern



Dialogs & Modal Workflows

What a dialog is, when to use one instead of a page, the five-step modal workflow, and where message boxes help.



Lesson goal

Understand **dialogs** and the modal add/edit workflow that line-of-business apps rely on: open a focused window, let the user complete one task, confirm risky actions, and return a clear result.

1. What is a dialog?

A **dialog** is a smaller screen that appears on top of the main page so the user can complete one focused task. In this course the most common dialogs are **Create Ticket** and **Edit Ticket**. A dialog shouldn't try to be the whole application — it collects the needed values, validates them, and returns a clear result.



Key idea

A dialog is *modal*: the user completes or cancels it before returning to the page behind it. That focus is what makes add/edit workflows predictable.

2. Dialog vs page

Screen type	Best use	Example
Page / main content area	Large app screens where users spend time and navigate between features.	Tickets page, Customers page, Settings page.
Dialog / modal window	A small focused workflow that should be completed or canceled before going back.	Create Ticket dialog, Edit Ticket dialog, Confirm Delete message.

3. The common modal workflow pattern

Almost every add/edit feature follows the same five-step flow. Keeping to it makes the workflow readable and predictable.

Button click	→	user chooses New or Edit
Open dialog	→	show the modal form / window
Validate	→	check the required fields
Save	→	call the service layer
Refresh UI	→	update the grid and show feedback

The user clicks **New** or **Edit**, the app opens a dialog, the dialog validates the fields, the **Save** button calls a service, and the main page refreshes *after* the dialog closes successfully.

4. Message boxes and confirmations

Use message boxes when the user needs immediate feedback or must confirm a risky action.

Situation	Recommended feedback
Save completed	Show a short success message and refresh the ticket list.
Required field missing	Show a clear validation message near the field or at the top of the dialog.
Delete selected ticket	Ask for confirmation before deleting.
Cancel after changing values	Optional: warn that unsaved changes will be lost.



You're ready for the next reading

You can describe what a dialog is, when to use a dialog instead of a page, the five-step modal workflow, and where message boxes help. Next: validating input and the consistent Save/Cancel pattern.

Validation & the Save/Cancel Pattern

Validate close to the UI without trapping business rules there, write a readable Save handler, and apply a consistent Save/Cancel pattern.



Lesson goal

Validate input close to the UI without trapping business rules there, write a readable Save handler, and apply a consistent Save/Cancel pattern so users always know what happened.

1. Validation: close to the UI, but not trapped in it

Validation belongs close enough to the UI that users get instant feedback, but important business rules should still be testable outside the button click. The dialog checks simple field problems, then calls a service class for reusable rules.

Validation type	Example	Good place
Simple UI validation	Title textbox is empty; Priority is not selected.	A dialog method such as <code>ValidateForm()</code> .
Business rule	A closed ticket cannot be edited by a support agent.	<code>TicketService</code> or a validation service.
Persistence rule	Ticket ID already exists in storage.	The repository / service layer.

2. Safe event handling

Button click events should be readable. Don't hide slow database calls or complicated business logic inside a trivial UI event unless it's intentional and clearly named. A good Save event calls small helpers such as `ValidateForm()`, `BuildTicketFromFields()` and `ticketService.SaveTicket(ticket)`.

```
private void btnSave_Click(object sender, EventArgs e)
{
    if (!ValidateForm())
        return;

    Ticket ticket = BuildTicketFromFields();
    ticketService.SaveTicket(ticket);

    this.DialogResult = DialogResult.OK;
    this.Close();
}
```



Validate before you save

The handler returns early when `ValidateForm()` fails — nothing is saved and the dialog stays open. Only valid input reaches the service.

3. A consistent Save / Cancel pattern

- **Save** validates first, then calls the service, then closes the dialog *only after success*.
- **Cancel** closes the dialog without saving.
- Error messages should tell the user exactly what to fix.
- The parent page refreshes the grid only when the dialog returns `OK`.
- Keep button names clear: `btnSave`, `btnCancel`, `btnNew`, `btnEdit`.

4. Developer habits for this module

- Name controls clearly so event code is readable.
- Keep dialog layout simple and focused.
- Separate UI state, business rules and saved data.
- Use services for reusable ticket operations.
- Use message boxes and validation messages consistently.



You're ready for the lab

Next: watch the dialog walkthrough, pass the knowledge check, then build reusable Create and Edit Ticket dialogs — validate first, save through the service, return `DialogResult.OK`, and refresh the grid with clear feedback.