

Data Binding & the DataGridView

Layout — SplitContainer, Dock & AutoSize

 LearnWisej

Lesson goal

Move from static screens to **data-driven** screens. By the end you'll model a business record, keep data logic in a service layer, bind a list to a `DataGridView`, and show the selected record in detail controls — refreshing the UI on purpose.

1. From static screens to data-driven screens

Module 4 builds a simple **ticket-management page**: a list of tickets appears in a `DataGridView`, and the selected ticket appears in detail controls on the side. The key habit is to keep each responsibility in its own layer instead of piling everything into button clicks.

Layer	What belongs there	Module 4 example
Model	The shape of the data object.	<code>Ticket.cs</code> with Id, Title, Customer, Status, Priority, AssignedTo, CreatedDate, Description.
Service / repository	Methods that get, save, add or delete data.	<code>TicketService.cs</code> with <code>GetTickets()</code> , <code>SaveTicket()</code> , <code>AddTicket()</code> , <code>DeleteTicket()</code> .
UI page	Controls, binding, selection and messages.	<code>DataGridView</code> , detail fields, a Save Ticket button, a Refresh button.
Persisted data	Where data eventually lives.	A database, file or API later. In this lab a simple in-memory list is enough.

2. A simple data model

The model is a plain C# class — just properties, no UI and no data access. It describes *what a ticket is*.

```
public class Ticket
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string Customer { get; set; }
    public string Status { get; set; }
    public string Priority { get; set; }
    public string AssignedTo { get; set; }
    public DateTime CreatedDate { get; set; }
    public string Description { get; set; }
}
```

3. A service (repository) layer

Put the get / save / add / delete methods in a service class instead of in button clicks. For this lab an in-memory `List<Ticket>` is enough; later the same methods could call a database.

```
public class TicketService
{
    private readonly List<Ticket> tickets = new List<Ticket>();

    public List<Ticket> GetTickets()
    {
        return tickets;
    }

    public void SaveTicket(Ticket ticket)
    {
        // Later this could save to a database.
        // For Lab 4, update the object in memory and refresh the UI.
    }
}
```



Why separate the layers?

You can explain why `Ticket.cs` and `TicketService.cs` live apart from the page: the model describes data, the service owns data logic, and the page only handles display and user actions.

4. Showing records in a DataGridView

A `DataGridView` is ideal when users need to see many records at once. In this module the grid shows the ticket list while the detail panel shows the selected ticket.

Grid setup	Recommended beginner setting	Reason
<code>AutoGenerateColumns</code>	False after the first basic test	You control which business columns users see.
<code>SelectionMode</code>	FullRowSelect	Clicking a row selects the whole ticket.
<code>MultiSelect</code>	False	Beginner labs are easier when one ticket is selected.
<code>ReadOnly</code>	True for the grid	Edits happen in the detail panel, not randomly inside grid cells.
Columns	ID, Title, Customer, Status, Priority, Assigned To	Shows useful fields without crowding the page.

The BindingSource pattern

```
private TicketService ticketService = new TicketService();
private BindingSource ticketsBindingSource = new BindingSource();

private void LoadTickets()
{
    ticketsBindingSource.DataSource = ticketService.GetTickets();
    dgvTickets.DataSource = ticketsBindingSource;
}

private Ticket SelectedTicket
{
    get { return ticketsBindingSource.Current as Ticket; }
}
```

5. Binding the detail controls

Bind each detail control to the same `BindingSource`. When the user selects a different row, the bound controls follow the current ticket automatically.

```
txtTitle.DataBindings.Add("Text", ticketsBindingSource, "Title", true,
    DataSourceUpdateMode.OnPropertyChanged);

cmbStatus.DataBindings.Add("Text", ticketsBindingSource, "Status", true,
    DataSourceUpdateMode.OnPropertyChanged);

cmbPriority.DataBindings.Add("Text", ticketsBindingSource, "Priority", true,
    DataSourceUpdateMode.OnPropertyChanged);
```

6. Bind once, refresh intentionally

Data binding connects controls to objects, but **you** still control when data loads, when edits save, and when the grid refreshes. Don't assume the screen magically knows every business change — call a refresh (for example `ResetBindings(false)`) after data changes.



Safe event handling

Don't hide expensive database calls inside trivial UI events such as `TextChanged` or every `SelectionChanged`. Keep small UI events simple; do the real work in Save buttons or clear service methods so behavior is predictable.

7. UI state vs business state vs persisted data

State type	Meaning	Example
UI state	What the screen is currently showing or selecting.	Selected row #1, Save button enabled, status message text.
Business state	The current object values used by the app.	Ticket #1 status changed from Open to In Progress.
Persisted data	The saved copy outside the UI.	Database row, JSON file or API record after Save.



You're ready for the next reading

You can model a ticket, keep data logic in a service, bind a list to a grid, show the selected record in detail controls, and refresh on purpose. Next: the layout tools that arrange the grid and detail panel cleanly.

Layout — SplitContainer, Dock & AutoSize

The layout tools that arrange a master-detail screen — SplitContainer, Dock and AutoSize — without fighting the page.



Lesson goal

Arrange the ticket-management screen with the three layout tools a data-binding page needs: a **SplitContainer** for the master-detail split, **Dock** for resizing, and **AutoSize** for tidy labels.

A data-binding page usually needs two areas: a large record list and a detail editor. The cleanest beginner layout puts the `DataGridView` on the left and the selected ticket's details on the right.

1. Using a SplitContainer

A `SplitContainer` creates two panels separated by a movable splitter — useful when the user compares a list and a detail form at the same time.

Step	What to do in the Designer	Why it matters
1	Drag a <code>SplitContainer</code> from the Toolbox onto the page.	Creates <code>Panel1</code> and <code>Panel2</code> for a master-detail screen.
2	Set <code>Dock = Fill</code> on the SplitContainer.	Makes it stretch with the page.
3	Set <code>Orientation = Vertical</code> .	Gives a left panel and a right panel.
4	Put the <code>DataGridView</code> in <code>Panel1</code> .	The ticket list belongs on the large side.
5	Put labels, text boxes, combo boxes, a date picker and the Save button in <code>Panel2</code> .	The detail editor belongs on the side panel.
6	Adjust <code>SplitterDistance</code> .	Controls how much space the grid gets versus the details panel.

Recommended Lab 4 layout

```
// The Designer can set these visually – this just shows the idea.
splitContainer1.Dock = DockStyle.Fill;
splitContainer1.Orientation = Orientation.Vertical;
splitContainer1.SplitterDistance = 900;

dgvTickets.Dock = DockStyle.Fill;           // ticket list fills the left panel
grpTicketDetails.Dock = DockStyle.Fill;     // or Top for fixed-height content
btnSaveTicket.Text = "Save Ticket";
```

2. Using Dock

Dock tells Wisej how a control should attach to its parent and resize when the browser window changes — different from just dragging controls into place.

Dock option	Use it for	Module 4 example
Fill	A control that should use all available space.	DataGridView inside the left SplitContainer panel.
Top	Header bars, titles or short toolbars.	A page title or header area.
Bottom	Status bars or command areas.	A status label such as "Selected ticket #1".
Left	Navigation panels or sidebars.	The Module 3 left nav, or a fixed filter panel.
None	Controls you position manually.	Individual labels and text boxes inside a details form.



Beginner rule

Don't set every control to **Dock = Fill**. Usually the container is docked, the grid is docked, and small input controls keep normal positions or use anchoring. Too much **Dock = Fill** makes controls cover each other.

3. Using AutoSize

AutoSize lets a control grow or shrink to fit its content. It's great for labels and short messages, but risky for large containers because the page can become unpredictable.

Control	Suggested AutoSize	Why
Label	True	Labels fit their text and stay readable.
Button	False or True by design	Fixed-width buttons look consistent; AutoSize helps if the text changes.
Panel / GroupBox	Usually False	The details area should stay stable while the user edits fields.
DataGridView	False — use Dock = Fill	The grid should fill available space, not resize to content height.
Columns	Use AutoSizeColumnsMode carefully	It can make columns readable, but too many columns crowd the grid.

Common DataGridView sizing settings

```
dgvTickets.AutoGenerateColumns = false;
dgvTickets.SelectionMode = DataGridViewSelectionMode.FullRowSelect;
dgvTickets.MultiSelect = false;
dgvTickets.ReadOnly = true;
dgvTickets.Dock = DockStyle.Fill;

// Optional: make columns fit the available width.
dgvTickets.AutoSizeColumnsMode = DataGridViewAutoSizeColumnsMode.Fill;
```

4. Layout tools are not data

SplitContainer, Dock and AutoSize make the screen usable — they don't replace your model, service layer or `BindingSource`. Keep each concern where it belongs:

Concern	Where it belongs
Screen layout	SplitContainer, Dock, AutoSize, panels, labels, grid, buttons.
Record data	The <code>Ticket.cs</code> model.
Get / save logic	The <code>TicketService.cs</code> service layer.
Current selection	<code>BindingSource.Current</code> and the DataGridView selected row.
Saving changes	The Save Ticket button calls the service, then refreshes intentionally.



You're ready for the lab

Next: watch the data-binding walkthrough, pass the knowledge check, then build the ticket screen — a SplitContainer with a bound `DataGridView`, detail controls, and a Save button that goes through the service layer and refreshes the grid.