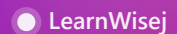


# DataGridView Mastery

Lesson Guide · Lab / Exam Guide



## Lesson Guide

BindingSource and explicit columns, custom editors and command cells, virtual mode with CellValueNeeded and a server-side cache, and cell HTML, cell controls and cell painting used responsibly. What this means for a Wisej.NET developer — and how to read this module.



### Production focus

BindingSource and explicit columns, custom editors and command cells, virtual mode with CellValueNeeded and a server-side cache, and cell HTML, cell controls and cell painting used responsibly.

## 1. The production problem

`DataGridView` is the control most likely to decide whether a business application feels mature. Orders, invoices, deadlines, logs, users and documents all end up in a grid, and users judge the app by how that grid loads, scrolls, edits and reports status. Wisej.NET offers everything from manual population to virtual mode and cell painting; mastery is choosing the right technique for the data size and the user's task.

## 2. Key concepts

| Concept                   | Why it matters                                                                                                                                                                                            |
|---------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BindingSource             | A server-side intermediary between view-model rows and the grid, giving it a stable data source, currency and sorting without coupling the screen to the database.                                        |
| AutoGenerateColumns       | Creates a column for every public property of the bound type; handy for a prototype, dangerous in production because a model change silently rearranges the screen.                                       |
| explicit column           | A column you define with field binding, header text, width, alignment, format, read-only state, sort mode and editor type, so the UI stays stable when the model changes.                                 |
| virtual mode              | With <code>VirtualMode</code> on, the grid knows only its row count and asks for each visible value through <code>CellValueNeeded</code> , so nothing is created for rows nobody sees.                    |
| server-side cache         | A page-sized buffer that feeds <code>CellValueNeeded</code> so scrolling does not hit the database per cell; the <code>DataRead</code> event is the documented hook for building and maintaining it.      |
| custom cell editor        | A control assigned to a column's <code>Editor</code> property; it must push its value back into the cell, via <code>CellBeginEdit</code> and <code>CellEndEdit</code> if it has no suitable change event. |
| cell HTML vs cell control | HTML produced in <code>CellFormatting</code> with <code>AllowHtml</code> is display-only markup and nearly free; an embedded control is a real server control and browser widget with a cost per cell.    |

## 3. Lesson narrative

### Three ways to populate, one for production

The grid can display data with or without an underlying data source. Direct `Rows` and `Columns` access is the simplest: you add columns and rows and own every value; it suits a small static table such as a status lookup. `DataSource` and `DataMember` binding populates the grid from a list, a `BindingSource` or a data set, and is the mode for business data. The third mode is dynamic column generation: with `AutoGenerateColumns` true, the grid rebuilds its columns whenever the data source changes.

Generated columns suit a ten-minute prototype and trap a screen that ships: add a property to `OrderRow`, rename another, and the grid follows the model rather than the design. The Operations Console Orders grid therefore defines every column explicitly: `DataPropertyName`, `HeaderText`, `Width`, alignment and format in `DefaultCellStyle`, `ReadOnly` where the value is informational, `SortMode` and the column type that carries the right editor. The binding itself stays small:

```
private readonly BindingSource ordersSource = new BindingSource();

private void BindOrders(IEnumerable<OrderRow> rows)
{
    this.ordersGrid.AutoGenerateColumns = false;    // columns are designed, not discovered
    this.ordersSource.DataSource = rows.ToList();
    this.ordersGrid.DataSource = this.ordersSource;
}

private void ordersGrid_CellFormatting(object sender, DataGridViewCellFormattingEventArgs e)
{
    if (ordersGrid.Columns[e.ColumnIndex].Name != "StatusColumn") return;
    var status = System.Net.WebUtility.HtmlEncode(Convert.ToString(e.Value));
    e.Value = $"<span class='status-badge'>{status}</span>";
    e.FormattingApplied = true;
}
```

Turn `AutoGenerateColumns` off the moment the screen is stable, and bind a view-model collection rather than an entity: a view model is shaped for display and carries status text, a formatted total and an ID without leaking persistence details.

### Built-in columns, custom editors and command cells

Typed values deserve typed columns. Text, check-box, combo-box, button, link and image column types cover most of the Orders grid, and Wisej.NET adds columns built on the date, numeric and masked editors from Module 2. A command cell is a button or link column: the lab's *Ship* column raises `CellContentClick`, the handler reads the row's ID from the bound item and calls the service. Nothing about the order lives in the cell.

When no built-in editor fits, assign a control to the column's `Editor` property; the classic example is a `MonthCalendar` on a due-date cell. The important warning: the editor must update the cell value correctly. The grid expects `TextChanged` or an equivalent change event; if your control does not raise one, handle `CellBeginEdit` to copy the cell value into the editor and `CellEndEdit` to copy the edited value back. Test the round trip with Tab, Enter and a click elsewhere: an editor that never commits is the classic custom-editor bug.

### Virtual mode, CellValueNeeded and the cache

Large data needs a different approach. Loading 100,000 orders because the database can return them is not a UI strategy: nobody can inspect that many rows, and each costs server memory and browser rendering. Filter first, offer server-side search and paging, and reserve virtual mode for the path where the user genuinely must scroll a very large result. There you set `VirtualMode` to true, give the grid its `RowCount` and columns, and handle `CellValueNeeded` to supply each value on request.

The event alone is not enough. A handler that queries the database inside `CellValueNeeded` runs once per visible cell, so one scroll can produce hundreds of queries. The lab's `OrderCache` keeps the current page on the server: `CellValueNeeded` asks it for row and column, it serves a hit from memory and fetches a whole page on a miss. Wisej.NET documentation points to the `DataRead` event as the place to build and

maintain that cache, because it tells you which block of rows the grid is about to read. Profile with realistic volumes, not the demo service's twenty rows, and ask what breaks first at ten times the data.

### Cell HTML, cell controls, cell painting and the composed grid

Cell HTML is the best intermediate option for visual richness. If a status needs a badge, an icon, a progress bar or several text fragments, set `AllowHtml` and produce the markup in `CellFormatting`, as in the snippet above; the raw status stays in the model and the HTML is display only. Treat it that way: encode user text before it goes into markup so a customer name never becomes executable HTML, and keep business validation out of the formatting handler.

Cell controls are more powerful and more dangerous. Wisej.NET can place a real control inside a cell, even spanning several cells: a button in a row, a track bar across columns, a mini preview. Every one is a server control and a browser widget, so a control in every cell of a 10,000-row grid is 10,000 controls. Use them for the selected row, summary rows, a command area or a small grid. Cell painting sits at the far end of the scale: enable user painting on the cell and handle `CellPaint` when the display needs custom drawing or reuses existing .NET drawing code, not for styling that themes, cell styles or HTML already cover.

Finally, the composable `DataGridView` behaves as a container, hosting connected controls such as a filter panel, a status strip, a navigator or a tree, docked inside or around it and ordered with `BringToFront` and `SendToBack`. That keeps the full grid API visible instead of hiding it behind a UserControl; the lab's filter and status strip are built this way.

- Explicit columns and a BindingSource or view model for every production screen; `AutoGenerateColumns` off once the screen is stable.
- Filter before displaying; virtual mode plus a cache for genuinely large sets.
- Cell HTML before embedded controls; embedded controls sparingly; cell painting only when styling, HTML and normal controls are not enough.
- Profile with realistic data, not sample data.



#### Common pitfalls

`AutoGenerateColumns` left on in production. Loading all records at startup. Querying the database from `CellValueNeeded` for every cell with no cache. Embedding controls in every row of a huge grid. Mixing business validation with visual formatting. Treating a grid as a database table instead of a task-oriented view.



#### 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:

- ✓ Bind DataGridView to a BindingSource and control AutoGenerateColumns intentionally.
- ✓ Use built-in columns and custom editors for typed values and command cells.
- ✓ Implement virtual mode, CellValueNeeded, DataRead/caching concepts, and large-data performance boundaries.
- ✓ Use cell HTML, cell controls, cell painting, and composed grids responsibly.

## Lab / Exam Guide

---

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



### Hands-on lab

Build the Operations Console Orders grid: a DataGridView bound to a BindingSource with explicit columns and AutoGenerateColumns under control, a status cell rendered with AllowHtml and CellFormatting, one custom editor or command column, a virtual-mode or cache-backed version for the large-data path, and a composed filter and status strip docked inside or around the grid.

## Starting point

Open the `OperationsConsole` solution from Module 4 and select the DataGridView placeholder page in the shell. The document explorer, the `CustomerEditor` and the navigation shell already exist; the Orders page is still an empty `TabPage`, and a small in-memory `OrderService` that returns `OrderRow` view models is enough to drive every step of this lab.

## Suggested approach

- ✓ Add `ordersGrid` and an `ordersSource` BindingSource, set `AutoGenerateColumns` to false, and define every column explicitly: order number, customer, due date, total, status, with header text, width, alignment, format and read-only state chosen on purpose.
- ✓ Mark the status column `AllowHtml` and handle `CellFormatting` to turn the raw status into an encoded badge; keep the model value untouched and set `FormattingApplied`.
- ✓ Add either a MonthCalendar editor on the due-date column through the column's `Editor` property (with `CellBeginEdit` / `CellEndEdit` if the value does not round-trip) or a button column whose `CellContentClick` calls the service by order ID.
- ✓ Build the large-data path: `VirtualMode` true, `RowCount` from the service, an `OrderCache` that serves `CellValueNeeded` from a page in memory and fetches a new page on a miss, using `DataRead` to prefetch the block the grid is about to show.
- ✓ Compose the grid: dock a filter strip (search box, status ComboBox, Apply button) above or inside the grid and a status strip below it that shows row count, selected order and last refresh; order them with `BringToFront` and `SendToBack`.

## Required deliverables

- ✓ Orders DataGridView with explicit columns bound to a BindingSource
- ✓ Status cell rendered with AllowHtml and CellFormatting
- ✓ One custom editor or command column
- ✓ Virtual-mode or cache-backed large-data version of the grid
- ✓ Composed filter and status strip docked inside or around the grid

## Student review questions

- ✓ If `OrderRow` gains two new properties next sprint, what changes on the screen with generated columns, and what changes with your explicit columns?
- ✓ Your `CellValueNeeded` handler works with 200 orders. What happens to the database when a user drags the scrollbar through 200,000, and which part of your design absorbs that?
- ✓ Why does the status badge belong in `CellFormatting` rather than in the `OrderRow` model, and what would go wrong if a customer name were placed in that HTML without encoding?



### 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.