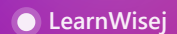


Containers, Layouts, Navigation, and Reuse

Lesson Guide · Lab / Exam Guide



Lesson Guide

Docking order, anchoring, flow and table layouts, responsive properties per client profile, navigation and command surfaces, and UserControls as the reuse boundary. What this means for a Wisej.NET developer — and how to read this module.



Production focus

Docking order, anchoring, flow and table layouts, responsive properties per client profile, navigation and command surfaces, and UserControls as the reuse boundary.

1. The production problem

Containers are the architecture of the user interface: they decide how controls are grouped, resized, navigated and reused. The Module 1 Operations Console still has placeholder panels, and a screen where every control is positioned by hand becomes fragile the moment the browser is resized. Layout problems are usually design problems, not pixel problems, and this module replaces pixel-pushing with a plan: docked regions, a SplitContainer, a TabControl, a real command model and UserControls that hide their internals.

2. Key concepts

Concept	Why it matters
docking order	Docking attaches a child to one side of its container or fills what is left, and it is applied in child order, so the same four controls produce different shells depending on the order in which they were added.
anchoring	Anchoring keeps a control's edges tied to its container's edges; anchored left and right it stretches with the width, anchored on neither axis it floats while keeping its relative placement.
SplitContainer	Two adjacent regions the user can resize, the right choice whenever the person, not the developer, should decide how much room the navigation tree or the detail panel gets.
layout panels	FlowLayoutPanel arranges children side by side and wraps, TableLayoutPanel aligns rows and columns, and FlexLayoutPanel distributes and wraps like a modern responsive layout.
responsive properties	A property that holds a different value per client profile, set at design time, so a desktop layout can change a handful of properties on narrow screens without a second screen.
command model	One consistent rule for where a command lives: MenuBar, ToolBar, RibbonBar, ContextMenu or StatusBar, each with its own job, all invoking application commands rather than containing business logic.
UserControl surface	The reuse boundary: a composed control that exposes a few meaningful properties, events and methods instead of every child control, so consumers never couple to its implementation.

3. Lesson narrative

Choose the container for the job

Wisej.NET gives you absolute positioning, docking, anchoring and richer layout panels, and the first decision is which surface hosts the screen at all. A **Page** is the natural entry surface for a web-style application such as the Operations Console: it fills the browser and has no window chrome. A **Form** suits dialog-like windows, modal workflows and MDI-style patterns, and a **Desktop** can host several forms when the application deliberately preserves a desktop metaphor. Inside that surface, **Panel** and **GroupBox** group related controls; the difference is only whether the group deserves a visible caption.

Three containers do most of the structural work. A **SplitContainer** is ideal when the user can resize two adjacent regions, such as a navigation tree beside a detail panel. A **TabControl** organizes peer pages so the user moves between sections without opening new windows. An **Accordion** stacks collapsible sections for navigation, settings and collapsible work areas. In the lab, the console's left region becomes a **SplitContainer** panel, the right region a **TabControl**, and the Module 1 placeholders become its tab pages.

Docking order, anchoring and the layout panels

Docking attaches a child control to a side of its container (**DockStyle.Top** , **Bottom** , **Left** , **Right**) or fills the remaining space (**DockStyle.Fill**). Because docking is applied in child order, the order in which controls are added matters: the ToolBar and StatusBar must claim their edges before the SplitContainer fills

the rest, otherwise the fill control is laid out first and the bars overlap it. When a shell looks wrong in the designer, check the child order before touching a single size.

Anchoring keeps a control's edges related to its container's edges. A control anchored left and right changes width when the container changes width; a control anchored neither horizontally nor vertically floats while preserving its relative placement, right for a centred sign-in card and wrong for a grid. Use docking for the application regions (navigation, header, footer or status, content) and anchoring inside those regions for editors and buttons.

For collections of controls, reach for a layout panel instead of coordinates. `FlowLayoutPanel` arranges children next to one another and can wrap, so a row of filter chips survives a narrow browser.

`TableLayoutPanel` arranges content in rows and columns, which is what a form with aligned labels and editors needs. `FlexLayoutPanel` distributes space and wraps the way modern responsive layouts do, the better choice when the interface needs flexible distribution rather than a fixed grid.

Responsive design in Wisej.NET is not limited to CSS media queries. Responsive properties hold multiple values for different client profiles and are set at design time in the property grid; client profiles identify device or browser situations, such as a phone up to 600 pixels wide, a tablet up to 1024 and a desktop above that. The practical rule is to build a good desktop layout first, then define the handful of properties that must change for tablets or narrow screens: visibility, size, docking, column display, side-panel behaviour and command placement. Do not create a completely separate screen unless the workflow truly changes. When a property alone is not enough, profile-aware logic in code does the same job, as the lab's narrow-profile method shows:

```
// OperationsConsole shell: add the regions in this order
this.toolBar.Dock = DockStyle.Top;
this.statusBar.Dock = DockStyle.Bottom;
this.splitMain.Dock = DockStyle.Fill; // navigation | TabControl detail

private void ApplyNarrowProfile(bool narrow)
{
    // narrow screens: hide the navigation panel, show the menu button
    this.splitMain.Panel1Collapsed = narrow;
    this.btnMobileMenu.Visible = narrow;
    this.pnlProfile.Text = narrow ? "Narrow profile" : "Desktop profile"; // a StatusBar
panel
}
```

Navigation and command surfaces

Navigation controls need a consistent command model, and the model is about where a command belongs, not how it looks. A `MenuBar` is appropriate for traditional application menus. A `ToolBar` holds the frequently used actions, which is why the Operations Console puts New, Refresh and Save there. A `RibbonBar` groups commands by task area when the application has many commands, overkill for a console with a dozen. A `ContextMenu` is for commands tied to a specific object: a row, a node or the current selection. In every case the control raises an event and a thin handler invokes an application command; navigation controls that contain business logic are one of the module's pitfalls.

The `StatusBar` is not a dumping ground. It should show relevant status such as the selected record count, the last refresh time, the validation state or the active client profile, and the lab asks for real status rather than the word "Ready". A `TabControl` deserves the same discipline. Tab pages are not a substitute for workflow: use them for peer sections such as Summary, Details, History, Documents and Notes, never as required wizard steps. Do not hide required fields on tabs without clear validation navigation, and if tabs can be moved, decide whether the new order is persisted and whether it affects one session or the user profile.

UserControl is the reuse boundary

When the same group of controls appears twice, the temptation is to copy it, and the cost arrives with the third copy and the first change. A `UserControl` is the main reuse boundary. A reusable control should have a small public surface: properties such as `CustomerId`, `ReadOnly` and `SelectedDate`, events such as `Saved` or `SelectionChanged`, and methods such as `LoadData()` or `ValidateContent()`. Avoid exposing all internal child controls unless this is an intentional advanced control; otherwise the consuming screens become coupled to the implementation, and renaming a `TextBox` breaks three pages. The lab moves the console's repeated status strip and record header into `UserControls`; the review question is whether a consumer could use them without knowing what is inside.

Write the decisions down. The lab's `LayoutNotes.md` is a short file you keep in the project recording which regions are docked and in what order, which controls are anchored and why, and which properties change on the narrow profile. It is what the next developer reads before opening the designer, and it should reflect these layout decision rules:

- Use docking for application regions and a `SplitContainer` when the user should control space allocation.
- Use `TabControl` for peer sections, `FlowLayoutPanel` for collections that wrap and `TableLayoutPanel` for forms with aligned labels and editors.
- Use responsive properties for profile-specific changes and encapsulate repeated layouts as `UserControls` rather than copying groups of controls.



Common pitfalls

Mixing absolute positioning, docking and flow without a plan. Setting `AutoSize` on large containers or grids and accidentally forcing huge layout calculations. Creating tabs that hide required fields and then showing vague validation errors. Exposing every child control of a `UserControl` publicly. Making navigation controls responsible for business logic instead of invoking application commands.



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:

- ✓ Use Form, Page, Desktop, Panel, GroupBox, SplitContainer, TabControl, Accordion, FlowLayoutPanel, TableLayoutPanel, and UserControl correctly.
- ✓ Combine docking, anchoring, flow, table, and responsive properties without fighting the layout engine.
- ✓ Design navigation and command surfaces using MenuBar, ToolBar, RibbonBar, StatusBar, ContextMenu, and TabControl.
- ✓ Package repeated UI patterns into UserControls with clean public properties and events.

Lab / Exam Guide

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



Hands-on lab

Replace the Module 1 placeholders with the real Operations Console shell: a SplitContainer with navigation on the left, a TabControl detail surface on the right, a ToolBar for frequent commands and a StatusBar that shows real status. Move repeated UI into UserControls that expose only meaningful public properties and events, implement a narrow-profile layout with responsive properties or profile-aware logic, and document your docking and anchoring choices in LayoutNotes.md.

Starting point

Open the `OperationsConsole` solution from Module 1 and its main `Page`: it still has the navigation, content, command and status placeholder panels, the placeholder pages for Editors, Layouts, Lists and Trees, DataGridView, Dashboard and Widgets, and the Module 2 `CustomerEditor` UserControl. Keep this module's work in its own folder or namespace so the capstone can reuse it.

Suggested approach

- ✓ Sketch the shell before opening the designer: ToolBar docked Top, StatusBar docked Bottom, a SplitContainer docked Fill, and write that child order into `LayoutNotes.md` first.
- ✓ Build the shell in that order, put the navigation control in the SplitContainer's left panel and a TabControl in the right panel, and turn each Module 1 placeholder page into a tab page for a peer section.
- ✓ Add the ToolBar buttons for New, Refresh and Save and a ContextMenu on the navigation list; every handler stays thin and calls a named command method on the page.
- ✓ Extract the record header and the status strip into `RecordHeader` and `StatusStrip` UserControls that expose a few properties (for example `Title`, `RecordCount`) and one event (for example `RefreshRequested`), never their child controls.
- ✓ Define a narrow client profile, give the responsive properties (navigation visibility, splitter panel collapse, ToolBar placement) a narrow value or implement `ApplyNarrowProfile`, update the StatusBar with the active profile, and capture screenshots at desktop and narrow widths.

Required deliverables

- ✓ Navigation shell built from SplitContainer, TabControl, ToolBar and StatusBar
- ✓ Repeated UI moved into UserControls with a small public surface
- ✓ Narrow-profile layout using responsive properties or profile-aware logic
- ✓ LayoutNotes.md documenting the docking and anchoring choices
- ✓ Screenshots at desktop and narrow widths

Student review questions

- ✓ If the SplitContainer were added to the page before the ToolBar and StatusBar, what would the user see and why does the order matter?
- ✓ Which of your UserControl's members could you rename without touching any consuming page, and which could you not? What does that tell you about its public surface?
- ✓ Which properties changed on the narrow profile, and why did you not build a separate narrow screen?



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.