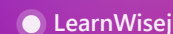


Control Library Mental Model

Lesson Guide · Lab / Exam Guide



Lesson Guide

The server control tree and browser widget model, the control families, common properties and naming conventions, and the Operations Console shell that grows across the course. What this means for a Wisej.NET developer — and how to read this module.



Production focus

The server control tree and browser widget model, the control families, common properties and naming conventions, and the Operations Console shell that grows across the course.

1. The production problem

Wisej.NET ships more than a hundred controls, and a team without a shared mental model of them ends up with twenty different search screens, five ways of reporting a validation error, and custom JavaScript where a native control would do. This module fixes the vocabulary, decision rules and conventions every later module builds on, and starts the Operations Console shell.

2. Key concepts

Concept	Why it matters
server control tree	The hierarchy of .NET control objects the server keeps for each session; your code sets properties and handles events here, never in markup.
browser widget	The JavaScript object Wisej.NET creates to render a server control, kept in sync by serialized state changes, not page reloads.
control family	A group of controls sharing a purpose (editors, containers, lists, grids, content, menus, notifications), where decisions are made.
extender	A component that adds properties or behaviour to existing controls at design time without subclassing them.
Widget and extension	The Widget control hosts a custom or third-party JavaScript widget inside the Wisej.NET event model; an extension is a packaged integration, usually from NuGet.
UserControl	A reusable server-side composition of controls with business meaning and a small public surface.
control selection guide	A living team document fixing standard controls, docking, validation and notification style and grid conventions per screen pattern.

3. Lesson narrative

Server objects first, browser widgets second

The mental model everything else depends on: application code manipulates .NET objects on the server, and Wisej.NET serializes the state changes those objects need to JavaScript widgets in the browser. When the Operations Console sets `statusLabel.Text`, no markup is written; the runtime sends the widget only the changed state. A `Button` is not a string of HTML but a server object with properties, events, a lifecycle, design-time support, theming and a client representation, and every browser session owns its own tree of them.

The first pitfall follows directly: treating controls as HTML snippets, reaching for markup and client scripts where a property, event or theme setting already exists. The browser is the second boundary: file-system access, printing, uploads, downloads, iframes and JavaScript libraries behave differently from desktop controls, which is why `Application.Download` and the `Upload` control exist instead of a file dialog.

Controls are theme participants: control documents list the widget class and the theme appearance keys it uses, which matters as soon as styling needs more than `BackColor`, `ForeColor` or `Font`. Know when a simple property is enough, when to use the theme, when an extender will do, and when a reusable `UserControl` or custom `Widget` is justified. Ignoring appearance keys until late in the project is a pitfall that ends in dozens of local colour overrides.

The control families and the decision rules

The library divides into families, and the family is where decisions are made. Editors (`TextBox`, `TypedTextBox`, `MaskedTextBox`, `DateTimePicker`, `NumericUpDown`, `ComboBox`) collect typed input. Buttons and menus (`Button`, `MenuBar`, `ToolBar`, `ContextMenu`) initiate commands. Containers (`Panel`, `SplitContainer`, `TabControl`, `Accordion`, layout panels) create screen structure. Lists, trees, repeaters and grids (`ListView`, `TreeView`, `DataRepeater`, `DataGridView`) display data. Content controls render text, images, HTML, PDFs, files and browser-hosted content. Notifications (`Toast`, `AlertBox`, `MessageBox`) communicate status. Extenders add behaviour to existing controls without subclassing, and extensions and widgets bring external web capability inside the component and event model.

The course treats the library as a design system, not a list to memorise: what matters is which family fits the task, how it is populated, how it behaves with large data, layout and theming, how it talks to the browser, and how it stays maintainable in a multi-user server application. A good decision starts with the user's task: an editor to edit a value, a `ListView` to choose from a collection with icon, tile or details presentation, a `TreeView` for hierarchical data with expand and collapse, a `DataGridView` for tabular records with sorting, editing, custom columns or large data, and a `UserControl` when a group of controls has reusable business meaning.

Three rules keep the decision honest. Choose the simplest control that expresses the task without custom code, preferring standard controls and extenders before any custom JavaScript widget; building a widget before exhausting the native library is the costliest pitfall in the course. Consider data size before choosing a population strategy: manual `Rows` and `Items` are fine for small sets, large sets need binding, virtual mode, caching or lazy loading. Use theme and appearance conventions for application-wide styling and individual properties only for local exceptions.

Common properties and conventions that hold every module together

Naming should identify the business role, not only the visual type: `customerGrid` says more than `dataGridView1`, and `StatusPanel`, `SearchTextBox` and `SaveButton` make event handlers readable without opening the designer. Keeping default designer names turns `button3_Click` into archaeology, and instructors deduct for it heavily.

Docking and anchoring should be intentional: `Dock` fills and stacks, `Anchor` follows the edges when the browser resizes, and absolute positioning belongs only where nothing else fits; mixing layout strategies at random in one screen is a pitfall that breaks on the first resize. Controls in the keyboard workflow need a deliberate `TabIndex`, validation errors need clear messages, images should follow an `ImageList` or resource convention, expensive operations need visible feedback, and accessibility labels should match the business role.

A production application captures all of this in a control selection guide: a living document that defines, per screen pattern, the standard controls, default docking, validation and notification style, menu placement, icon pack, grid conventions and accessibility expectations. It is not bureaucracy; it is how a team stays consistent. Your lab's `ControlSelection.md` is page one of that guide and answers these questions for every area:

- Does the selected control match the user's task, and is there a simpler native control?
- Is the data flat, hierarchical, tabular, repeated or visual, and how large will it get?
- Does the screen need a UserControl boundary, and is a custom Widget genuinely justified?

The Operations Console shell

The course project starts as a shell, and every module adds one section until you own a working reference application rather than isolated samples. The shell is a main `Page` (or `Form`) with four areas: navigation on the left, a content area that fills the rest, a command area for the current section's actions, and a status area that reports what the console is doing.

In the lab the navigation area holds one clearly named `Button` per placeholder page (Editors, Layouts, Lists and Trees, DataGridView, Dashboard, Widgets), the content area is a `Panel` docked to fill, and the status area is a docked panel with a `Label`. The one method worth writing swaps the content, docks the new page to fill and updates the status label, so every navigation change is reported from one place and a page that fails to load is not left blank.

```
private void Navigate(Control content, string title)
{
    try
    {
        contentPanel.Controls.Clear();
        content.Dock = DockStyle.Fill;
        contentPanel.Controls.Add(content);
        statusLabel.Text = $"Showing {title}";
    }
    catch (Exception ex)
    {
        statusLabel.Text = $"Could not open {title}";
        MessageBox.Show(ex.Message);
    }
}
```

Each placeholder page is its own class, later a `UserControl`, in its own folder so the capstone can reuse it, and names carry the role throughout: `navigationPanel`, `contentPanel`, `commandPanel`, `statusLabel`, `editorsButton`. The stretch goal is a small diagnostic panel showing the selected control name, record ID, active client profile and last refresh time.



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:

- ✓ Explain the Wisej.NET server control tree and browser widget relationship.
- ✓ Classify the control library by purpose: inputs, containers, lists, data grids, content, menus, notifications, extenders, and extensions.
- ✓ Apply common properties, events, naming, anchoring, docking, images, tab order, accessibility labels, and designer conventions consistently.
- ✓ Create a course project shell called Operations Console that will grow across the course.

Lab / Exam Guide

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



Hands-on lab

Create the Operations Console control catalog shell: a main Page with a navigation area, a content area, a command area and a status area; placeholder pages for Editors, Layouts, Lists and Trees, DataGridView, Dashboard and Widgets; a ControlSelection.md note inside the project listing which control family you will use for each area; clear control names throughout; and a status area that updates whenever navigation changes.

Starting point

Open Visual Studio and create a new Wisej.NET web application named `OperationsConsole`; this is the solution every later lab extends, so keep each module's work in its own folder or namespace. The template gives you an empty main `Page` and a `Program.cs` that opens it, and nothing else exists yet.

Suggested approach

- ✓ Sketch the four areas before touching the designer, then dock them in order: navigation left, command top, status bottom, and the content panel added last with Dock set to Fill.
- ✓ Rename every control by its business role as soon as you drop it (`navigationPanel` , `contentPanel` , `commandPanel` , `statusLabel` , `editorsButton`) and set a deliberate `TabIndex` on the navigation buttons.
- ✓ Create one placeholder UserControl per section (Editors, Layouts, Lists and Trees, DataGridView, Dashboard, Widgets), each with a title label, in its own folder so the capstone can reuse it.
- ✓ Write a single `Navigate(Control content, string title)` method that clears the content panel, docks the page to fill, adds it and updates the status label inside a try / catch; route every navigation button through it.
- ✓ Add `ControlSelection.md` to the project and record, for each area and placeholder page, the control family you chose and the simpler native alternative you rejected.

Required deliverables

- ✓ Main shell with navigation, content, command and status areas
- ✓ Placeholder pages for Editors, Layouts, Lists and Trees, DataGridView, Dashboard and Widgets
- ✓ ControlSelection.md listing the control family chosen for each area
- ✓ Clear control names and a status area that updates on navigation
- ✓ Screenshot and a short note explaining the control decisions

Student review questions

- ✓ When you set `statusLabel.Text` in the navigation method, what actually travels to the browser, and why is that different from writing HTML?
- ✓ Which shell area would you turn into a UserControl first, and what would its public surface be?

- ✓ If a later module needs a collapsible navigation area, which native container replaces the docked panel, and why is that preferable to a custom widget?



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.