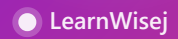


Custom Widgets, Extensions, Theming, and Capstone

Lesson Guide · Lab / Exam Guide



Lesson Guide

Native controls versus widgets, components, extender providers and icon packs; the Widget control's packages, InitScript, WidgetEvent and CallAsync bridge; theming and documentation; the Operations Console capstone. What this means for a Wisej.NET developer — and how to read this module.



Production focus

Native controls versus widgets, components, extender providers and icon packs; the Widget control's packages, InitScript, WidgetEvent and CallAsync bridge; theming and documentation; the Operations Console capstone.

1. The production problem

Sooner or later a product needs something the native library does not ship: a specialised browser capability, a third-party JavaScript library, a custom visualisation, or a reusable integration. The temptation is to drop into JavaScript and lose the Wisej.NET model: server state, theming, the designer and an event flow another developer can follow. This module integrates a client widget without giving any of that up, then finishes the Operations Console capstone.

2. Key concepts

Concept	Why it matters
native control	A control that implements both the server side and its browser widget, so it already has design-time support, theming, events and server state; always the first stop.
extension types	Control, Widget, Component, Extender Provider and Icon Pack: naming the type stops "reusable" from being confused with "new native control".
Widget control	A general-purpose client container that loads JavaScript and CSS packages, runs an initialisation script and exchanges events and data with server code.
Packages and InitScript	The package list declares the client files the widget depends on; the InitScript creates the JavaScript object, wires its client events and exposes only the functions the server needs.
WidgetEvent	The server event raised when client code calls <code>fireWidgetEvent</code> ; its type and data payload arrive from the browser and must be validated like any other user input.
CallAsync / Eval	The server-to-client half of the bridge: call a function exposed by the widget, or evaluate a script, without leaving C#.
widget contract	The documented properties, events, payload format, package dependencies and theming rules that let another developer, or a grounded AI assistant, maintain the widget.

3. Lesson narrative

Native first, then five kinds of extension

The control library is the first stop, every time. A native control implements the server side and the client side together, which is why it participates in the Visual Studio designer, in `Application.Theme` , in the event model and in per-session server state for free. Custom code is reserved for what the library cannot cover: a browser capability with no native equivalent, a third-party JavaScript library, a bespoke visualisation, or an integration you want to reuse.

When custom code is justified, name the kind of extension you are building. A **Control** is a full native control with a server class and a client widget. A **Widget** is the general-purpose client container for a third-party or custom JavaScript widget. A **Component** may live only on the server or also on the client. An **Extender Provider** adds properties and behaviour to existing controls, the way `ToolTip` or `ErrorProvider` do. An **Icon Pack** packages visual resources for the project and the designer. The taxonomy prevents design confusion: not every reusable thing is a `UserControl` , and not every JavaScript integration needs a new native control.

For the Operations Console the decision rules are short. Use native controls first. Use a `UserControl` for a reusable server-side composition such as the Module 2 customer editor. Use `Widget` for third-party JavaScript UI that can live inside a Wisej.NET container. Build a full extension only when the integration should be reused across applications and appear as a first-class component.

The Widget bridge in both directions

The `Widget` control does three things. Its `Packages` collection loads the JavaScript and CSS files the client library needs, in order. Its `InitScript` runs once the packages are present: `this.init` creates the client object inside `this.container`, wires the library's own events and exposes the few functions the server may call. Finally it carries messages: client code calls `this.fireWidgetEvent("ratingChanged", { value: 4 })` and the server receives a `WidgetEvent` whose `WidgetEventArgs` expose the `Type` and the `Data` payload; server code answers with `CallAsync` to invoke a client function, or `Eval` to run a script.

The lab's rating widget shows the round trip: a star click fires the event, the handler validates the value, saves it through a service and confirms back to the widget:

```
private async void ratingWidget_WidgetEvent(object sender, WidgetEventArgs e)
{
    if (e.Type != "ratingChanged") return;

    // The payload came from the browser: validate before trusting it.
    int value = Convert.ToInt32(e.Data.value);
    if (value < 1 || value > 5) return;

    await _ratings.SaveAsync(_customerId, value);
    await this.ratingWidget.CallAsync("setSaved", value);
    MessageBox.Show($"Rating {value} saved.");
}
```

Use the bridge carefully. JavaScript stays focused on browser-only work: rendering, gestures, animation. Business rules stay on the server, where they are secured, tested and shared with every screen; duplicate one in JavaScript only as a deliberate client-side optimisation, never as the only copy. One classic bug: inside a library callback `this` is the library's object, not the widget, so capture `var me = this` in `init` and call `me.fireWidgetEvent`.

A contract you can debug, theme and document

A robust wrapper has a contract. Public server-side properties correspond to meaningful configuration, not every option the library exposes. Events represent business-relevant user actions: `ratingChanged`, not `mouseup`. If payloads are more than a value or two, define a DTO or a clear JSON shape; the handler validates and normalises the data before using it, because a browser payload can be changed by anyone with developer tools.

Debugging widgets starts in the browser console, and the Wisej.NET documentation says so explicitly. When a widget renders blank or an event never arrives, check the console first, then package loading order, resource paths and initialisation timing, and whether the widget is re-created after a refresh or layout change, which means `init` must be safe to run again. Scripts dropped into the startup HTML can run too early for Wisej.NET widgets; embedded resources, `Packages`, `InitScript` and control-level calls are the safer patterns.

Theming and responsive behaviour are the final polish. If the widget must match the console, decide deliberately: expose style properties, map to theme appearances, use the `StyleSheet` or `Animation` extenders, or ship CSS in the widget package. Do not hard-code application colours in the JavaScript

unless the widget is intentionally independent. Apply responsive properties per client profile as you did for the shell, provide accessible labels and keyboard behaviour where practical, and write the AI grounding note: what the widget is for, why no native control fit and where the contract lives.

- Package dependencies and load order are documented.
- Properties, events and the payload format are written down next to the code.
- Business rules are enforced server-side and the browser console is clean.



Common pitfalls

Replacing standard controls with JavaScript because it seems modern. Putting business rules into client JavaScript where they are harder to secure and test. Failing to handle widget reinitialisation after a refresh or layout change. Forgetting to document the package dependencies and the event contract. Building a custom widget that no one else on the team can maintain.

Finishing the Operations Console

The capstone integrates the course in one application, `OperationsConsole` : the navigable shell (Modules 1 and 3), the validated customer editor (Module 2), the TreeView and ListView explorer (Module 4), the Orders `DataGridView` (Module 5), the dashboard and content screen (Module 6), and this module's custom widget or extension screen. Each screen is judged on the questions the labs asked: which control does the most important work, where state lives, what happens with two sessions or ten times the data, and which code belongs in a reusable `UserControl` or service.

The rubric weights control selection and data controls at 20% each, and layout and reuse, validation and feedback, the custom widget contract, and polish and documentation at 15% each. Prepare a five-minute demo that follows the capstone script, and write `CapstoneNotes.md` from the workbook template: controls used and why, data-size assumptions, validation and layout choices, reusable code extracted, and one thing you would improve in production. The code should be understandable by another developer and explainable by an AI assistant grounded on your notes.



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 difference between native controls, widgets, components, extender providers, and icon packs.
- ✓ Use the Widget control to load JavaScript/CSS packages, initialize a client widget, and exchange events/data with server code.

- ✓ Apply theming, responsive properties, accessibility notes, and AI grounding notes to a custom control package.
- ✓ Complete and present the Operations Console capstone using native controls plus at least one custom widget or extension.

Lab / Exam Guide

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



Hands-on lab

Wrap a small JavaScript rating or indicator widget with the Widget control, or configure a Wisej.NET extension package: load its JavaScript and CSS packages, raise a server-side event from the client with the widget event pattern, call a client-side function from the server with CallAsync or Eval, theme it to match the console, then polish the complete Operations Console capstone and prepare a 5-minute demo.

Starting point

Open the `OperationsConsole` solution you have carried through Modules 1 to 6: the shell, the customer editor, the document explorer, the Orders grid and the dashboard all exist, and the Widgets tab is still the placeholder page from Module 1. Keep this module's work in its own folder or namespace so the capstone can reuse it.

Suggested approach

- ✓ Decide, and write down, why no native control fits the rating or indicator you are adding; if a native control does fit, use it and pick a different widget target.
- ✓ Drop a `Widget` control on the Widgets page, add the JavaScript and CSS files to its `Packages` collection in load order, and write an `InitScript` whose `init` creates the client object inside `this.container` and is safe to run again after a refresh.
- ✓ In the client callback capture `var me = this` and call `me.fireWidgetEvent("ratingChanged", { value })`; on the server handle `WidgetEvent`, check `e.Type`, validate and normalise `e.Data`, then save through a service.
- ✓ Confirm back to the browser with `CallAsync("setSaved", value)` or `Eval`, give the user a Toast or AlertBox, and theme the widget through theme appearances, packaged CSS or the StyleSheet extender rather than colours hard-coded in JavaScript.
- ✓ Write the contract next to the code (packages, properties, events, payload format, debugging notes), then walk every capstone screen against the rubric and rehearse the five-minute demo script.

Required deliverables

- ✓ Widget or extension-based screen with its JavaScript and CSS packages loaded
- ✓ Client-to-server event raised with the widget event pattern or a native extension event
- ✓ Server-to-client call through CallAsync, Eval or the extension API
- ✓ Documented client/server contract: packages, properties, events and payload format
- ✓ Polished Operations Console capstone with a 5-minute demo script

Student review questions

- ✓ What would happen if a user edited the `ratingChanged` payload in the browser's developer tools before it reached your handler, and which line of your code stops it?
- ✓ Your widget renders blank after the Widgets tab is hidden and shown again: which three things do you check, in which order, and why is the startup HTML not the fix?
- ✓ Which rule in your widget could you move to JavaScript as an optimisation, and which one must never leave the server?

Capstone hand-in

This module's lab is also the capstone submission. The Operations Console must include six screens: the application shell with navigation, command and status areas; the validated customer or contact editor; the TreeView/ListView explorer with stable IDs and a lazy or virtual strategy; the Orders DataGridView with explicit columns, formatting and a large-data strategy; the dashboard and content screen with ChartJS and a document preview; and the custom Widget or extension screen from this lab. Submit the source project, screenshots of each screen, `CapstoneNotes.md` written from the workbook template, and a five-minute demo (live or recorded) that follows this script:

- ✓ Explain the control-library architecture of the application, then show the editor and its validation behaviour.
- ✓ Show tree and list navigation with detail loading, then the grid and how it handles large data.
- ✓ Show the dashboard and content controls, then the custom widget or extension including its event communication in both directions.
- ✓ Explain what your AI assistant should answer if asked why each control was chosen.

Rubric area	Weight	Excellent looks like
Control selection	20%	Native controls, extenders, UserControls, widgets and extensions used appropriately with clear rationale.
Layout and reuse	15%	Containers, docking, anchoring, splitters, tabs and UserControls used cleanly.
Data controls	20%	TreeView, ListView and DataGridView patterns with correct identity, binding and large-data strategy.
Validation and feedback	15%	Editor properties, ErrorProvider and notifications used clearly and consistently.
Custom widget or extension	15%	A documented client/server contract, handled events, no misplaced business logic.
Polish and documentation	15%	Theme, responsive behaviour, comments and AI grounding notes are production-minded.



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.