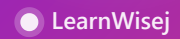


Custom Controls, Extensions, Widget Wrappers, Reusable Components

Lesson Guide · Lab / Exam Guide



Lesson Guide

Reusable controls, third-party widget wrappers, custom properties/events, resource packaging. What this means for an advanced Wisej.NET team — and how to read this module.



Production focus

Reusable controls, third-party widget wrappers, custom properties/events, resource packaging.

1. The production problem

Advanced teams often need reusable enterprise components or third-party JavaScript widgets. The challenge is to expose them as maintainable Wisej.NET controls without losing designer productivity or server-side safety.

2. Key concepts

Concept	Why it matters
custom UserControl	A composite control built from existing Wisej.NET controls that packages layout and behavior for reuse across screens.
inherited control	A subclass of a built-in control that adds defaults, validation, or behavior while keeping full designer and theme support.
Widget wrapper	A control that hosts a third-party JavaScript library inside Wisej.NET. It should expose only browser-side value while business rules stay on the server.
embedded resources	Scripts, styles, and images compiled into the control's assembly so the component ships as one package with no loose files to deploy.
client/server event contract	The agreed set of events and payload shapes exchanged between the browser widget and its server control, documented so both sides can evolve safely.
designer defaults	Sensible default property values and attributes so a control dropped on a form works immediately and looks consistent with the rest of the application.
component API	The public properties, methods, and events a reusable control exposes. A small, intentional API is easier to support than one that leaks implementation details.

3. Lesson narrative

Reuse needs an API

A reusable control should be easier to use correctly than incorrectly. Give it clear properties, events, defaults, and documentation. Do not require every screen developer to know the internal layout or JavaScript details.

Wrap behavior, not just visuals

A third-party widget wrapper should translate business-friendly C# calls into client behavior and translate client events back into named server events. Keep the API small, stable, and explicit.

UserControls and inherited controls

Wisej.NET offers two ways to build a reusable component from existing controls. A `UserControl` composes several controls into one designable surface with its own layout, which fits a component such as the lab's `StatusTimeline`: a header, a list of status entries, and a footer that together represent a work order's history. An inherited control extends a single built-in control, such as a grid that always shows the tenant's standard columns, and keeps every feature of its base while adding defaults and behavior.

Both are placed from the Toolbox and both appear in the Designer, which is the point. A screen developer drags a `StatusTimeline` onto a form, sets its properties, and handles its events without knowing how it draws itself. The lab's first review question, whether another developer can use the component without reading its internals, decides whether the component is finished.

Choose composition when the component is a new thing made of several parts, and inheritance when it is a specialized version of one existing control. Mixing the two produces components that are hard to theme and hard to explain.

A stable C# API around a client widget

The chart in the lab is a third-party JavaScript widget, and the application never speaks to it directly. A `WorkOrderChartWidget` class wraps it, exposing a method such as `SetItems` that accepts a list of typed points and a server event such as `SegmentClicked` that carries a segment identifier. Inside the wrapper, the method turns the list into the vendor's data shape and the client event is translated from the vendor's callback into the named server event.

Stability comes from keeping the API small and expressed in application terms. A method named after what the application wants, not after the vendor's function, survives a vendor upgrade or a vendor replacement. The wrapper is also where failure is handled: when the vendor script is blocked or fails to load, the wrapper renders a plain fallback and raises no errors into the screen, which is the lab's polite fallback.

The second review question, what data crosses from server to client, is answered by the wrapper's DTOs and nothing else. A wrapper that sends the work order entity to the chart has leaked the domain into the browser.

Packaging resources consistently

A reusable component depends on scripts, stylesheets, images, and sometimes fonts, and every one of them must arrive with the component wherever it is used. The reliable way is to embed them as resources

in the component's assembly and reference them through the framework's resource mechanism, so that adding the assembly to a project adds everything the component needs. Loose files copied into each application drift out of sync within a release.

The embedded resource package in the lab follows one convention: a folder per component, versioned file names for vendor libraries, and one place that lists what is loaded and in what order. Images used by the component are theme-aware where possible, so a dark theme does not show a light icon. The component's own stylesheet loads after the vendor's so its overrides win.

- Can the component be added to a new project by referencing one assembly.
- Is the vendor version visible in the resource names and the release notes.
- Does upgrading the vendor change one folder and one list.

Properties, events, and design-time defaults

The public surface of a component is what other developers see in the Properties window and in IntelliSense. Each property has a type that matches its meaning, a default value attribute so the Designer knows what not to serialize, a category, and a description. Each event has a typed argument class carrying exactly what a handler needs, such as the clicked segment, and a name that says what happened rather than which vendor callback fired.

Design time is a first-class scenario. The Designer creates the component with no services and no data, so the component must detect design mode and render sample data safely. The lab's sample mode shows a plausible timeline and chart in the Designer, which lets a screen developer lay out the form without running the application, and never calls a service or throws. The third review question, whether the component renders safely in the Designer, is checked by dropping it on an empty form.

Document the API on the component itself, in summary comments and descriptions, because that is the only documentation most users of the component will ever read.



Advanced production habit

Do not stop when the screen looks correct. Explain state ownership, security implications, failure paths, and production behavior before considering the module complete.



Learning objectives

By the end of this module you'll be able to:

- ✓ Design custom UserControls and inherited controls for reuse.
- ✓ Wrap a client widget behind a stable C# API.
- ✓ Package scripts, styles, images, and resources consistently.
- ✓ Expose properties, events, and design-time defaults that other developers can use safely.

Lab / Exam Guide

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



Hands-on lab

Build a reusable StatusTimeline control and a wrapped chart/visualization Widget for work-order history. Expose C# properties, server events, embedded resources, and a design-time sample mode.

Suggested approach

- ✓ Read the production problem and identify the architecture risk before touching the UI.
- ✓ Create or update the relevant service, command, model, or component boundary first.
- ✓ Build the Wisej.NET screen or component using the established project standards.
- ✓ Add a failure-path demonstration, not only the happy path.
- ✓ Record what changed in the learning log and update the relevant checklist.

Required deliverables

- ✓ StatusTimeline UserControl
- ✓ Widget wrapper or custom control
- ✓ Embedded resource package
- ✓ Event contract document
- ✓ Usage example screen

Student review questions

- ✓ Can another developer use the component without reading its internals?
- ✓ What data crosses from server to client?
- ✓ Does the component render safely in the designer?



Advanced production habit

Do not stop when the screen looks correct. Explain state ownership, security implications, failure paths, and production behavior before considering the module complete.