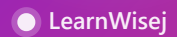


Reusable Widget Classes and Designer-Friendly Properties

Lesson Guide · Lab / Exam Guide



Lesson Guide

Move from one-off Widget instances to reusable server-side classes with typed properties and controlled setup. What this means for a Wisej.NET integration developer — and how to read this module.



Production focus

Move from one-off Widget instances to reusable server-side classes with typed properties and controlled setup.

1. The integration problem

The prototype works, but the same widget will be used on many screens. Copying Packages, Options, and InitScript into every page creates maintenance risk.

2. Key concepts

Concept	How to use it
Typed properties	Expose stable .NET properties instead of raw vendor options.
Resource setup	Keep package and script setup inside the wrapper class.

3. Production lesson

A reusable widget class is the natural next step after a successful prototype. The application developer should not have to remember the vendor script order, the exact initialization code, or the raw option names for every screen. The wrapper class centralizes those decisions. The server class still derives from `Wisej.Web.Widget`, but the public surface becomes a .NET API. For example, a gauge can expose `Value`, `Minimum`, `Maximum`, `Label`, and `AnimationEnabled` instead of requiring every developer to edit a dynamic Options object. The class can still keep an escape hatch for advanced options, but the common path should be typed, discoverable, and designer-friendly. This is also where naming, default values, categories, descriptions, and type converters matter. The wrapper is no longer a quick script host. It is becoming part of the application framework used by other developers.

A class that owns its setup

The lab refactors the prototype into `IntegrationLab.Controls.SimpleGauge`, a class that derives from `Wisej.Web.Widget`. Its constructor, or its initialization override, assigns the Packages, the InitScript, and

the default Options that were previously typed on every page. A screen that needs a gauge now declares a `SimpleGauge` field, drops it from the toolbox, and never sees a line of JavaScript.

The `InitScript` itself moves into an embedded resource or a constant in the class, so it is versioned with the C# code that depends on it. Vendor package paths become constants in one place. When the vendor releases a new version, one class changes, and every screen picks it up on the next build.

A wrapper class is also the first place where a decision record becomes code. The class name, its namespace, and its summary comment state what the widget is for and which vendor library it wraps, which is what a developer two years from now needs to know first.

Hiding the escape hatches

Once the class owns the setup, the raw members become dangerous. A developer who edits `Packages` on one instance breaks that instance and nobody else's, which is the hardest kind of bug to find. The wrapper therefore hides or protects `Packages`, `InitScript`, and the raw `Options` object so that they do not appear in the Properties window and cannot be changed casually from application code.

In practice this means overriding the members as protected, or marking them so the Designer does not browse or serialize them. The public surface of the class is then only the typed properties and events it deliberately exposes. Advanced scenarios can still be served by a protected method that lets a derived class extend the options, which keeps the escape hatch available to people who understand the consequences.

The self-check in the lab asks which members should be hidden from normal users of the wrapper. The answer is every member whose misuse could break the widget silently.

Typed properties that drive Options

A typed property is a small piece of C# that reads from and writes to the `Options` object while presenting a normal .NET type. `Value` is a double with a range check, `Minimum` and `Maximum` are doubles that validate against each other, and `Caption` is a string. Each setter assigns the camel-cased option, and Wisej.NET renders the change and calls `update` on the client exactly as it did for the prototype.

Attributes make the properties designer-friendly. A default value attribute tells the Designer what not to serialize, a category groups the properties in the Properties window, and a description explains each one in the help pane. A property that is expensive to render, such as one that forces the vendor to rebuild, should say so in its description and should only assign the option when the value actually changed.

- Which property changes must force a client update, and which are read-only after creation.
- Which values are validated in the setter, and what exception a bad value raises.
- Which defaults match the vendor's defaults, so a fresh instance renders sensibly.

Wrapper concerns versus application concerns

The wrapper is responsible for the vendor: loading its files, creating and disposing its instance, translating options, and forwarding events. The application is responsible for meaning: which reading the gauge shows, what threshold matters, and what happens when it is crossed. A wrapper that contains a database call or a business rule has crossed that line, and so has a page that contains vendor initialization code.

The demo page in the lab is the proof. It places a `SimpleGauge` , binds `Value` to a reading from a service, and handles a .NET event. There is no `InitScript` on the page and no vendor name in the page's code. If the vendor library were replaced tomorrow, the page would not change.

A useful boundary test is to imagine shipping the wrapper as a NuGet package to another team. Anything they would need to edit inside the package belongs in the application instead.



Production habit

Keep Wisej.NET state ownership and client widget behavior clear. Document the client/server contract before relying on the integration in production.



Common mistake to avoid

Copying demo JavaScript into screens without documenting lifecycle, resources, event contracts, cleanup, or security boundaries.



Learning goals

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

- ✓ Create a class that derives from `Wisej.Web.Widget`.
- ✓ Hide or protect Packages, `InitScript`, and raw Options where appropriate.
- ✓ Expose typed .NET properties that update the Options object safely.
- ✓ Separate vendor resource setup from application usage.

Lab / Exam Guide

What you'll build in the hands-on lab, the recommended workflow, the required deliverables, and the self-check before you submit.



Hands-on lab

Refactor the gauge prototype into `IntegrationLab.Controls.SimpleGauge``, with typed properties and default resource registration.

Recommended workflow

- ✓ Open the existing Wisej.NET integration course project and run it locally.
- ✓ Identify the server component, client widget, resource file, or endpoint involved in this module.
- ✓ Make one small change at a time and test it in the browser before moving on.
- ✓ Keep vendor-specific JavaScript inside the adapter or wrapper instead of scattering it across pages.
- ✓ Update the UI so the learner can clearly see success, failure, progress, or communication changes.
- ✓ Take screenshots or notes that prove the integration works and explain the design decision.

Required deliverables

- ✓ Reusable Widget-derived class
- ✓ Typed properties with defaults
- ✓ Demo page using the class without custom InitScript on the page

Self-check before submitting

- ✓ Which members should be hidden from normal users of the wrapper?
- ✓ What property changes should force client updates?
- ✓ What belongs in the wrapper and what belongs in the application?