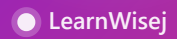


Client-Server Calls, Serialization, and Object Access

Lesson Guide · Lab / Exam Guide



Lesson Guide

Move data and commands safely between .NET components and browser widgets. What this means for a Wisej.NET integration developer — and how to read this module.



Production focus

Move data and commands safely between .NET components and browser widgets.

1. The integration problem

A useful integration needs more than initial rendering. The server must call client methods, the client may need to call server methods, and both sides need predictable data contracts.

2. Key concepts

Concept	How to use it
Call / CallAsync	Invoke client widget methods from server code.
DTO	Use small data transfer objects for stable serialized contracts.

3. Production lesson

Wisej.NET provides a communication layer that makes server-to-client calls feel natural from C#. The server can call a client method and continue, or it can await a value from the browser when the workflow requires it. The client can also call server methods exposed with `[WebMethod]`, using either callback style or promise/await style where supported. Serialization is a strength but also a responsibility. Wisej.NET can serialize anonymous objects, DTOs, and primitive values, but that does not mean every server object should be sent to the browser. A good integration uses small objects that match the JavaScript widget's needs. It avoids leaking internal domain models, database entities, or security-sensitive state into client options. Named components are reflected in the client object model. This is convenient for demos and controlled application code, but integration wrappers should avoid becoming dependent on fragile global paths.

Call, CallAsync, and EvalAsync

The three server-to-client calls differ in one thing: whether the server waits. `Call` names a client method, passes serialized arguments, and returns immediately; the invocation is queued and sent with the next

response. It is the right choice for commands the server does not need an answer to, such as setting the gauge value or resetting an animation. `CallAsync` names a client method too, but returns a Task that completes with the method's return value once the browser replies. `EvalAsync` is the same mechanism for an arbitrary JavaScript expression evaluated in the page.

Awaiting a client result suspends the server-side handler until the browser round trip completes. That is acceptable when the workflow genuinely depends on the value, such as reading the rendered size before choosing a layout, and wasteful when the value is only logged. The lab's rule is to use `Call` by default and reach for an awaiting variant only when the next server statement cannot run without the answer.

- Is the client result used in the same handler; if not, use `Call`.
- What happens if the browser never replies, for example because the tab was closed.
- Is the awaited value validated before it is trusted.

Serialization and camel-casing

Every argument passed to the client and every value returned from it goes through Wisej.NET's JSON serializer. Primitive values, arrays, dictionaries, anonymous objects, and ordinary classes all serialize, and property names are converted to camel case on the way out, so a C# `SelectedIndex` arrives in JavaScript as `selectedIndex`. Client code that reads the Pascal-case name gets undefined and no error.

The reverse direction has its own rules. A JavaScript object returned from `CallAsync` arrives on the server as a dynamic value, and the C# code must read it by the JavaScript names or convert it into a typed DTO. Dates, numbers that exceed JavaScript's safe integer range, and enums deserve a look in every contract, because they are the values most often mangled in transit.

Serialization is also where data leaks happen. An entity with a navigation property to a customer record serializes the customer too. The safe habit is to never pass a domain object across the wire, only an object built for the crossing.

Reaching into the client object model

Wisej.NET exposes its component tree to page scripts. `App.MainPage` is the current main page, controls that have a name are reachable as properties of their container, and `widget(id)` returns any component by its client identifier. In a demo, this makes it easy for a script to grab a button, read a text box, or find the gauge by walking the tree.

An integration wrapper should not depend on that tree. A path such as the main page's third panel's gauge breaks the moment a developer renames a control or moves it into a group box. The wrapper already has a reference to itself as `this` and to its own container, and everything it needs from the server arrives through options and calls. The lab's gauge commands are all invoked on the widget directly, never looked up by path.

Reserve the client object model for page-level scripts under the application's control, such as a keyboard shortcut handler, and treat any use of it inside a wrapper as a code review finding.

DTOs that stay small and stable

A data transfer object for the wire is a class with a handful of primitive properties, named for what the widget needs, and nothing else. The lab returns the gauge's selected client state as one such object: the

current value, whether it is above threshold, and the rendered width and height. Four numbers and a boolean, serialized in a few dozen bytes, and readable in a log entry.

Small DTOs are stable because they change rarely. Adding a property is safe, since the client ignores what it does not know. Renaming or removing one is a breaking change that must be versioned, and a DTO with four properties is versioned in minutes. A DTO that mirrors a domain entity changes every time the entity does, and every change is a chance to break the client silently.

Write the DTOs first, before the calls that carry them. Once the shapes crossing the wire are fixed, both the server code and the InitScript have something concrete to be tested against.



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:

- ✓ Use ``Call``, ``CallAsync``, and ``EvalAsync`` from server code to interact with the client widget.
- ✓ Understand Wisej.NET JSON serialization, including camel-casing of property names.
- ✓ Use the client object model such as ``App.MainPage``, named widgets, and ``widget(id)`` appropriately.
- ✓ Define small, stable DTOs instead of passing large or accidental server objects.

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

Add commands to the reusable gauge: set value, read rendered size, reset animation, and return selected client-side state to the server.

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

- ✓ Server-to-client method calls
- ✓ One ``CallAsync`` or ``EvalAsync`` example
- ✓ DTO used for client state return

Self-check before submitting

- ✓ When should server code await a client result?
- ✓ Why does camel-casing matter when passing objects to JavaScript?
- ✓ Why should DTOs be small?