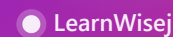


# JavaScript Object Model, Browser APIs, and Secure Interop Contracts

Lesson Guide · Lab / Exam Guide



## Lesson Guide

Advanced JS interop, client events, browser capabilities, contract validation. What this means for an advanced Wisej.NET team — and how to read this module.



### Production focus

Advanced JS interop, client events, browser capabilities, contract validation.

## 1. The production problem

Wisej.NET lets developers add JavaScript where it is valuable, but advanced interop can become fragile unless every boundary is treated as a small API.

## 2. Key concepts

| Concept                   | Why it matters                                                                                                                                                      |
|---------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| App object model          | The client-side representation of Wisej.NET components that JavaScript can reach. Use it deliberately, through documented entry points, not by poking at internals. |
| client event              | A browser-side event that is forwarded to the server. Only meaningful events should cross the wire, with compact payloads.                                          |
| remote method             | A server method the browser can call directly, such as a WebMethod, which returns a result to JavaScript. It needs the same validation as any endpoint.             |
| browser feature detection | Checking what the browser supports before using an API. Use it for browser-side value only, while business rules stay on the server.                                |
| interop contract          | A named, documented description of what JavaScript sends to the server and what it receives back, so both sides can be tested and versioned.                        |
| lifecycle timing          | Knowing when a client widget is created, rendered, updated, and destroyed, so scripts run at the right moment and clean up after themselves.                        |
| input validation          | Treating every value that arrives from the browser as untrusted and validating it on the server, because client code can be modified by anyone.                     |

### 3. Lesson narrative

#### JavaScript is a tool, not the architecture

Advanced Wisej.NET applications can use browser APIs, keyboard handling, clipboard, camera, scanners, and specialized widgets. The server remains the authority for workflow, permission, and persistence.

#### Name the contract

Every interop method should have a name, purpose, input schema, output schema, validation rule, and failure behavior. Treat it like a tiny public API between the browser and the server.

#### Browser capabilities without browser business rules

JavaScript earns its place in an enterprise Wisej.NET application when the browser can do something the server cannot: react to a keystroke before a round trip, read the clipboard, talk to a scanner, or report screen size and network status. The command palette in the lab is a good example. Capturing Ctrl+K, filtering a list as the user types, and highlighting a match are pure browser concerns, so they belong in a small client script.

Deciding what a command does is a different matter. When the user presses Enter on `approve`, the script sends a named request to the server, and the server decides whether the current user, in the current tenant, may approve this work order. The permission check, the state transition, the audit entry, and the persistence all run in a service on the server. If the script were allowed to change a status field directly, every rule in that service could be bypassed from the developer tools.

A useful test in code review is to ask what would break if the browser lied. If the answer is only a cosmetic glitch, the code is in the right place. If the answer is a wrong approval or a leaked record, that logic must move to the server.

#### Small contracts with named inputs and outputs

An interop contract is a written agreement about one boundary crossing. The lab captures it in a document and in code as a request and a result type, such as a `ClientCommandRequest` carrying a command name, a target ID, and an optional argument, and a server method such as `RunClientCommand` that returns a typed result. Every field has a type, an allowed range, and a default. Nothing else is accepted.

Keeping the contract small is what makes it safe. A payload with three named fields can be validated in a few lines, logged with its correlation ID, and versioned when a field is added. A payload that forwards an arbitrary JavaScript object cannot be validated at all, because the server does not know what it is looking at. Validation runs before any service is called, and a failure returns a named error such as

`PERMISSION_DENIED` or `INVALID_TARGET` rather than an exception message.

- Name: what the call is for, in one sentence.
- Input schema: every field, its type, and its validation rule.
- Output schema: the success shape and every named failure.
- Owner: which server service enforces the rules.

## Client events and server callbacks

Traffic between browser and server flows in two directions. A client event carries something that happened in the browser, such as a palette selection or a detected capability change, to the server. A server callback is the server calling into a client widget to update it, focus it, or read a value back. Both directions cross the same trust boundary, so both need the same discipline.

For client events, forward only the events that carry a business meaning, with a compact payload. A palette that raised an event for every keystroke would flood the session with round trips and give the server nothing useful to act on. For server callbacks, expect that the widget may be missing, disposed, or not yet initialized, and make the callback safe to run twice. A callback that appends a row to the DOM should check whether the row already exists.

Every callback and every event handler should be attributable. Log the user, tenant, correlation ID, and command name on the server side of each crossing, so a support engineer can reconstruct what the browser asked for and what the server answered.

## Timing, lifecycle, and capability differences

The most common interop failure is a script that runs before its target exists. The lab wires the palette script so it attaches only after the host widget has been created and rendered, not on page load. The same rule applies to any code that reads a DOM element, subscribes to a widget event, or calls a vendor API. If the element is created later, the script must wait for the creation event or be invoked from the widget itself.

Lifecycle has an end as well as a start. When a form closes or a widget is disposed, handlers that were attached to the window or the document keep running unless they are removed. Over a long session those leftovers accumulate, hold references to dead widgets, and produce errors that appear unrelated to the screen that caused them. Every attach needs a matching detach.

Browsers also differ. The capability panel in the lab exists precisely because clipboard access, camera access, and notification permission vary by browser, device, and security context. Detect the feature, report the result to the server as data, and let the server choose a fallback such as a file upload instead of a camera capture. Never assume a capability from the user agent string.



### 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:

- ✓ Use JavaScript for browser capabilities without moving business rules to the client.
- ✓ Define small interop contracts with named inputs, outputs, and validation.
- ✓ Use client events and server callbacks safely.
- ✓ Handle timing, widget lifecycle, and browser capability differences.

## Lab / Exam Guide

---

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



### Hands-on lab

Add an advanced keyboard command palette and browser capability panel. Use JavaScript to collect browser features and trigger server-side commands, then enforce permissions server-side before running the command.

## 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

- ✓ Interop contract document
- ✓ Command palette script
- ✓ Server callback methods
- ✓ Browser capability panel
- ✓ Security review notes

## Student review questions

- ✓ What happens before the target widget exists?
- ✓ Which values from JavaScript are trusted?
- ✓ Which business rules remain server-side?



### 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.