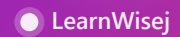


Production Readiness, Troubleshooting, and Capstone

Lesson Guide · Lab / Exam Guide



Lesson Guide

Prepare integrations for real applications: performance, diagnostics, background updates, packaging, versioning, and maintainability. What this means for a Wisej.NET integration developer — and how to read this module.



Production focus

Prepare integrations for real applications: performance, diagnostics, background updates, packaging, versioning, and maintainability.

1. The integration problem

A widget can work in a demo and still fail in production because of race conditions, broken resource paths, poor event filtering, missing cleanup, design-time crashes, or untested data volumes.

2. Key concepts

Concept	How to use it
Diagnostics	Make failures visible through logs, tools, and repeatable checks.
Disposal	Clean up client resources and event handlers when widgets are destroyed.

3. Production lesson

Production readiness is mostly about discipline. The integration must load resources deterministically, survive repeated creation and disposal, update without memory leaks, handle missing or malformed data, avoid excessive server round trips, and fail in ways developers can diagnose. It must also work in the Visual Studio designer when that is a stated requirement. Troubleshooting begins by isolating the problem. If the widget does not work in plain JavaScript, it is not a Wisej.NET problem yet. If it works in plain JavaScript but fails inside Wisej.NET, inspect the container, script order, timing, context, and server-call flow. Use ``debugger``, ``sourceURL``, the browser console, and the client object model. On the server, use normal Visual Studio debugging and remember that designer-loaded assemblies can be cached. The capstone asks students to integrate a new JavaScript widget family that is not Kendo or DevExtreme. Kendo and DevExtreme were the worked examples throughout the course. The capstone proves that the same lifecycle, event, and data-loading patterns transfer to an unfamiliar library, which is the real test of whether the integration approach has been understood rather than memorized.

Troubleshooting the client side

Client-side problems are diagnosed by narrowing the location. The first step is the vendor's own demo: if the widget fails there with the same options, the problem is the vendor and no Wisej.NET code is involved. If the demo works, the injected InitScript is next, and a `sourceURL` comment gives it a name in the developer tools so breakpoints can be set on real lines. A breakpoint in `init` shows whether the container exists, whether the packages have loaded, and what the options contain.

The console tells the rest. An undefined-symbol error at load time means a package order problem. A method-not-found error inside a callback means a lost `this`. A network entry with a 404 means a resource path that worked in development and not on the server. A guard clause that throws a clear message when a required option is missing turns a silent blank widget into a one-line diagnosis.

Record each diagnosis in the troubleshooting notes with its symptom, its cause, and its fix. The capstone's notes are the start of a team playbook.

Troubleshooting the server side

Server problems use the normal Visual Studio tools, with two Wisej.NET-specific habits. First, set the exception settings to break when thrown for the exceptions the integration raises, because an exception inside a `WebRequest` handler or a `WidgetEvent` handler may otherwise be caught by the framework and surface only as a missing result in the browser. Second, remember that the Designer runs the control's server code in design mode, so a crash there is a server crash and can be debugged by attaching to the Designer process.

The Designer also caches loaded assemblies. After changing a control's code, the Designer can keep rendering the old version until Visual Studio is restarted, which looks like a change that had no effect. Restarting is part of the discipline rather than a workaround. Design-mode code should detect design mode and render sample data instead of calling services, so that the Designer never needs a database.

When the server handler is correct and the browser still shows nothing, check the response body in the network panel before touching the server again. Most remaining failures are a content type or a serialization shape.

Background updates done safely

Widgets that show live data need updates that originate on the server without a user request. The pattern has three parts. `Application.StartTask` runs the work on a background thread while keeping the session context, so the code can still reach the page and the widget. Inside that task the server changes options or issues a `Call` on the widget, exactly as a button handler would. Then `Application.Update` pushes the pending changes to the browser, because without a request in flight nothing else would send them.

Care is needed in three places. Update frequency must be bounded, since every call is a round trip and a widget updated a hundred times a second is a slow page. The session may end while the task is running, so the task must handle a disposed widget without throwing. And the vendor call must run on the client after the option change it depends on, which the batching guarantees only when both are issued in the same update.

Batch the changes, push once per meaningful change, and stop the task when the page closes.

The production checklist

The capstone ships with a checklist that a reviewer can apply to any integration library, and each section is a place where demos fail in production. Resources: every package path is an application path or embedded resource, versions are pinned, and no CDN is required. Versioning: the vendor version, the wrapper version, and the breaking-change notes are recorded, and an upgrade has a test page. Security: every endpoint validates and authorizes, no domain object crosses the wire, and raw HTML from the vendor is sanitized.

Performance: the first render loads one page of data, events are filtered on the client, background updates are bounded, and disposal destroys the vendor instance and detaches every handler so a long session does not leak. Design time: the control renders sample data in the Designer without services and without throwing.

- Can the wrapper be created and disposed a hundred times without memory growth.
- Can the failure of the vendor script be told apart from a failure of the Wisej.NET infrastructure.
- Does another developer have everything they need to upgrade the vendor without the original author.



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:

- ✓ Troubleshoot client code with Chrome DevTools, dynamic script names, breakpoints, console inspection, and vendor demos.
- ✓ Troubleshoot server code with Visual Studio, exception settings, design-mode debugging, and designer restart discipline.
- ✓ Use background updates carefully with ``Application.StartTask``, client calls, and ``Application.Update``.
- ✓ Prepare resource, versioning, security, and performance checklists for an integration library.

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

Capstone: integrate an unfamiliar third-party widget as a reusable Wisej.NET wrapper, including server calls, at least one event, and either postback or WebMethod data loading.

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

- ✓ Capstone implementation
- ✓ Integration ADR
- ✓ Resource and security checklist
- ✓ Demo script and troubleshooting notes

Self-check before submitting

- ✓ How do you prove a failure is vendor usage rather than Wisej.NET infrastructure?
- ✓ What should be cleaned up when a widget is disposed?
- ✓ What belongs in a production integration checklist?