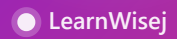


Postback, WebRequest, WebMethod, and Data Endpoints

Lesson Guide · Lab / Exam Guide



Lesson Guide

Feed JavaScript widgets that pull data using Ajax, while keeping requests in the context of the correct Wisej.NET component. What this means for a Wisej.NET integration developer — and how to read this module.



Production focus

Feed JavaScript widgets that pull data using Ajax, while keeping requests in the context of the correct Wisej.NET component.

1. The integration problem

Many advanced widgets, especially grids, pivot grids, reports, viewers, and uploaders, expect URLs or service endpoints rather than pushed data.

2. Key concepts

Concept	How to use it
Postback URL	Route browser requests back to the correct component instance.
WebMethod	Expose RPC-style calls for client callbacks or promises.

3. Production lesson

Some widgets are easiest to integrate when they pull data from a URL. This is common for data grids, pivot grids, spreadsheets, viewers, uploaders, and libraries with their own data-source abstraction. Wisej.NET provides a postback URL tied to the component instance, so the request arrives in the right session and the right control context. A postback handler behaves like a small web service. It receives an `HttpRequest` and writes an `HttpResponse`. The response can be JSON for a grid, binary content for a file, image data for a viewer, or any format required by the vendor library. The advantage is that the handler lives with the component instance, so it can use the component's state and application services. `[WebMethod]` is a different shape. It is ideal when the browser needs to call a public server method and receive a marshaled return value. It can be static or attached to a top-level container. It is usually cleaner for RPC-like calls, such as looking up a record, validating a value, or returning a small result set, while postback handlers remain the better fit for files, images, or large data responses that a vendor library expects to fetch from a URL.

Routing a URL to a component instance

Many advanced widgets fetch their own data. A grid asks for a page of rows, a viewer streams a document, an uploader posts a file. They all want a URL, and the challenge is that the URL must reach the right Wisej.NET session and the right component instance within it. `GetPostBackUrl()` on the server, or `getPostbackUrl()` on the client widget, returns exactly such a URL, encoded with the identifiers the framework needs to deliver the request to that component.

The wrapper passes that URL to the vendor as its data source address, usually in `init`, and can append its own query parameters to distinguish actions. Because the URL is bound to an instance, two grids on the same page get two different URLs and each request lands on the component that owns the data.

Treat the URL as session-scoped and short-lived. It is not a public API address, must not be stored, and must not be shared between users.

Handling the request in WebRequest

When a request arrives at the postback URL, the component's `WebRequest` event fires with the raw `HttpRequest` and `HttpResponse`. The handler reads the action and parameters, produces the data, sets the content type, and writes the response. JSON for a grid, a PDF stream for a viewer, an image for a thumbnail, or a CSV for an export are all just different bodies with different content types.

```
private void grid_WebRequest(object sender, WebRequestEventArgs e)
{
    var action = e.Request.QueryString["action"];
    if (action != "load") { e.Response.StatusCode = 400; return; }
    var rows = _service.LoadPage(CurrentTenant, ParsePage(e.Request));
    e.Response.ContentType = "application/json";
    e.Response.Write(JsonSerializer.Serialize(rows));
}
```

The handler runs in the component's context, so it can read the component's state and the current session. That is its whole advantage: the data query knows which tenant, which filters, and which permissions apply without any of that being passed from the browser.

WebMethod for RPC-style calls

When the caller is the integration's own JavaScript rather than a vendor data source, a URL is the wrong shape. A method marked with `[WebMethod]` is callable from the client as a function, with typed arguments marshaled from JavaScript and a return value marshaled back, and the client can await the result. It can be static, or an instance method on a top-level container such as the page or window.

The lab implements the same read-only dataset both ways to make the trade-off visible. The postback URL is right when the vendor library owns the fetching and expects an address it can call with its own paging arguments. The WebMethod is right when wrapper code needs a value, such as validating an entered code or fetching a lookup list, and would otherwise have to parse a raw HTTP response.

Both are entry points into the server, and both bypass the normal event pipeline, so both need the same validation and authorization a button handler would get.

Parameters, content types, and security

Every data endpoint accepts input from the browser, and the browser can be scripted by anyone. The action parameter is compared against a fixed list, not used to build a method name. Paging arguments are parsed as integers with an upper bound, so a request for a million rows is refused. Filter and sort values are mapped to known columns rather than passed into a query. Anything unexpected returns a client error and is logged with the session and correlation identifiers.

The content type is set explicitly on every response, because a JSON body served as text confuses some libraries and an HTML body served as JSON confuses all of them. The self-check's answer is `application/json` for JSON. File responses set the disposition header and the real MIME type so the browser handles them correctly.

- Is the current user allowed to see these rows in this tenant, checked in the handler.
- Are paging and filter parameters bounded and mapped to a whitelist.
- Does an error response reveal nothing about the server beyond a status code.



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 ``getPostbackUrl`` / ``GetPostbackURL`` to route browser requests to a component instance.
- ✓ Handle requests with ``WebRequest`` and return JSON, files, images, or any response the widget expects.
- ✓ Expose ``[WebMethod]`` methods for client calls that should behave like RPC rather than raw HTTP requests.
- ✓ Add action parameters, validation, content types, and security checks around widget data requests.

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

Implement both a postback endpoint and a WebMethod endpoint for the same read-only dataset. Connect one simple widget to each and compare the tradeoffs.

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

- ✓ Postback WebRequest handler
- ✓ WebMethod with arguments and return value
- ✓ Comparison note: URL data source vs client callback

Self-check before submitting

- ✓ When is postback better than WebMethod?
- ✓ Why should action parameters be validated?
- ✓ What content type should JSON responses use?