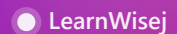


Desktop Packaging with a WebView Shell

Lesson Guide · Lab / Exam Guide



Lesson Guide

Hosting the app in a native desktop window, WebView2 on Windows, window chrome, file system and OS integration, and installers. What this means for a team shipping a Wisej.NET app beyond the browser tab — and how to read this module.



Production focus

Hosting the app in a native desktop window, WebView2 on Windows, window chrome, file system and OS integration, and installers.

1. The production problem

FieldOps technicians spend an entire shift inside the app: a laptop in the van, a docking station at the depot. The browser build from the first two modules does the work, but it has no icon in the Start menu, no presence in the notification area when a job is assigned, no way to drop an exported report into a folder the technician picked, and nothing IT can push through their usual software channel. This module wraps that same application, unchanged, in a Windows shell that supplies exactly those four things.

2. Key concepts

Concept	Why it matters
WebView2 control	The Chromium engine Microsoft ships as the redistributable Evergreen Runtime, embedded in a WinForms or WPF window so your page renders exactly as in Edge.
desktop shell	A small native executable that owns the window, the tray icon and the OS calls, and points a WebView2 control at your Wisej.NET application.
window chrome	The title, icon, minimum size, remembered position and single-instance behaviour that make the app feel installed rather than embedded.
message bridge	The two-way channel between page and shell: <code>PostWebMessageAsJson</code> down, <code>WebMessageReceived</code> up, or a host object published with <code>AddHostObjectToScript</code> .
JavaScript interop	How server-side C# reaches that channel: <code>Application.Eval</code> and <code>Application.Call</code> run script in the session's browser context and the answer returns as a client event.
server topology	Whether the shell navigates to a remote company server or starts a self-hosted Wisej.NET process on loopback — decided at launch and logged, never guessed.
update channel	A version endpoint the shell asks at start-up, plus the installer it fetches when the answer is newer than the version it is running.

3. Lesson narrative

A native window around the same page

The shell is smaller than developers expect: a WinForms or WPF executable containing one `WebView2` control docked to fill the client area, initialised asynchronously, then navigated to your application address. You can equally start from the Wisej.NET Hybrid desktop target, which packages the same arrangement for you. The page inside is the ordinary FieldOps session — the same server-side `Wisej.Web` controls, theme and layout logic. Nothing in the app knows it is being framed.

What the shell adds is everything outside the page. It sets the window title to the technician's name once the session reports who signed in, gives the window an icon and a minimum size, restores its last position, and refuses to open twice — a second launch activates the running window. A `NotifyIcon` puts FieldOps in the notification area with a short menu: Open, Sync now, Exit. Closing hides the window to the tray rather than ending the process, so the toast still arrives and the session stays where the technician left it.

Two things must not be lost. WebView2 requires the Evergreen Runtime, so the installer detects and bootstraps it rather than failing on first launch. And every capability the shell adds is an addition, never a replacement: the same build must still run in a plain browser tab, which is what the lab's capability check is for.

One bridge, two directions

WebView2 gives you the channel. From the shell to the page, `CoreWebView2.PostWebMessageAsJson` delivers a JSON object that arrives as a message event. From the page back, script calls

`window.chrome.webview.postMessage` and the shell handles `WebMessageReceived`, reading `WebMessageAsJson`. A third option, `AddHostObjectToScript`, publishes a COM-visible object script calls like a local API — convenient, and a much larger surface, because anything loaded into that page can call it.

Your C# lives on the server, so it reaches that channel through JavaScript interop: `Application.Eval` runs a fragment in the session's browser context and `Application.Call` invokes a function you registered on the client. The shell's answer returns as a client event that raises a server-side handler, which updates the controls and lets Wisej.NET repaint them. Because the round trip is asynchronous, every request carries an identifier — otherwise two exports started in quick succession will swap their results.

FieldOps uses that bridge twice. Export sends a `pickFolder` request; the shell opens a native folder dialog, returns the chosen path, and the server writes the report there. Assignment works in reverse: the server pushes the new order into the session, the page posts a `notify` message, and the shell raises a toast from the tray icon that activates the window on that order. In a plain browser neither request has a listener, so the capability check hides the native path and the export falls back to `Application.Download`.

```
// Shell (WinForms): answer one request from the page.
webView.CoreWebView2.WebMessageReceived += (s, e) =>
{
    var req = JsonSerializer.Deserialize<ShellRequest>(e.WebMessageAsJson);
    if (req.Action != "pickFolder") return;

    using var dlg = new FolderBrowserDialog();
    var path = dlg.ShowDialog() == DialogResult.OK ? dlg.SelectedPath : null;

    webView.CoreWebView2.PostWebMessageAsJson(
        JsonSerializer.Serialize(new { id = req.Id, path }));
};
```

Where the server actually runs

The shell is not the application, so you still have to say where the application is. The remote topology points the WebView2 control at the company server: one deployment, one database, an update that reaches every technician the moment they reconnect. The local topology bundles a self-hosted Wisej.NET process the shell starts on a loopback port; it works with no signal at all, but every van then carries a copy of the app and a store somebody must reconcile.

FieldOps configures both. A small `shell.config.json` beside the executable names the primary server, the probe timeout and whether the local fallback is permitted. On launch the shell calls a health endpoint; if it does not answer in a few seconds it starts the bundled server and shows a persistent strip reading **Working offline — local server**. The mode is visible in the window and written to the shell log, so a support call never begins by guessing which server the technician was talking to.

Each mode has consequences. Cookies from the remote server mean nothing to the local one, so local mode signs in against its own store and marks what it captures for later sync. The loopback port is picked at runtime — a fixed port is the one another product already took — and the shell stops the child process on exit, including after a crash.

Installing it, and keeping it current

Two packaging routes are reasonable. MSIX gives a per-user install, a clean uninstall, version handling by Windows itself and deployment through Intune, at the price of signing and a constrained install-time story. A classic WiX Toolset or Inno Setup installer gives full control: shortcuts, the WebView2 runtime bootstrapper, registry values, a service. Choose by how the software is distributed — FieldOps ships an Inno Setup package because the vans are re-imaged and the runtime bootstrap must be part of the install.

The update channel is small work with a large payoff. On launch, before navigating, the shell reads its own assembly version and asks a version endpoint for the current one. If the answer is newer it shows a non-blocking prompt with what changed, downloads the installer, checks it, runs it and exits. It never interrupts a technician mid-job, and an unanswered check is a logged warning rather than a failed start-up.

Two version numbers now exist. The server should move faster than the installed shells, so the endpoint also publishes the minimum shell version it still supports, letting the server warn — or refuse — when a van has been offline for two months. Signing and distribution belong to the last module; here the channel only has to exist and be honest.

- The Evergreen Runtime is bootstrapped by the installer, not assumed.
- Every bridge request carries an id, and every reply is matched to it.
- The server mode is visible in the window and written to the log.



Packaging habit

The app is still a Wisej.NET web app served by your server; the shell is a delivery vehicle. Keep every feature working in a plain browser, then add device capabilities as progressive enhancements behind a capability check.



Learning objectives

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

- ✓ Host the Wisej.NET app in a desktop shell built on WebView2 with a native window, title bar and system tray.
- ✓ Bridge the web app and the shell so the app can call native features such as file dialogs, printing and notifications.
- ✓ Decide whether the server runs locally alongside the shell or remotely, and configure each.
- ✓ Build an installer and an update channel for the desktop package.

Lab / Exam Guide

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



Hands-on lab

Package FieldOps for Windows: a WebView2 shell with a native window, custom title and tray icon, a bridge that lets the app open a native folder picker to export work-order reports and show a native notification when a new order arrives, a configuration that points the shell at the remote server with a local fallback, and an installer with an update check on launch.

Starting point

You open the FieldOps solution as it stands after Module 2: the work-order list, the work-order detail page and the status update flow all run in a browser, and the app already installs as a PWA with its manifest and service worker. Nothing in that web project changes in this lab — you add a second project, a Windows shell executable, alongside it.

Suggested approach

- ✓ Add a WinForms or WPF shell project with a single WebView2 control filling the window, initialise it asynchronously, and navigate it to the FieldOps address before you write any bridge code.
- ✓ Give the window its chrome: the FieldOps icon, a title that shows the signed-in technician, a remembered size and position, single-instance activation, and a NotifyIcon with Open, Sync now and Exit that keeps the process alive when the window is closed.
- ✓ Build the bridge in both directions — handle `WebMessageReceived` in the shell for a `pickFolder` and a notify request, reply with `PostWebMessageAsJson` carrying the request id, and on the server reach the page with `Application.Eval` and receive the answer as a client event.
- ✓ Write `shell.config.json` beside the executable with the remote server address, a probe timeout and the local fallback switch, then start the bundled self-hosted server and show the offline strip when the health probe does not answer.
- ✓ Produce the installer with the WebView2 runtime bootstrapper included, and add the launch-time version check that compares the shell's assembly version with the version endpoint and offers the new installer without blocking start-up.

Required deliverables

- ✓ WebView2 desktop shell with window and tray
- ✓ Native bridge for folder picker and notifications
- ✓ Remote and local server configuration
- ✓ Installer package
- ✓ Update check on launch

Student review questions

- ✓ A technician starts two exports within a second of each other and the shell answers both folder dialogs. What in your bridge design stops the second path from being written into the first report, and what would go wrong without it?
- ✓ Your Export button now opens a native folder dialog. Describe precisely what that same button does when the page is opened in Edge instead of the shell, and where in your code that difference is decided.
- ✓ The version endpoint is unreachable because the van has no signal. Argue for what the shell should do on that launch, and explain what a technician loses under each of the alternatives you rejected.



Packaging habit

The app is still a Wisej.NET web app served by your server; the shell is a delivery vehicle. Keep every feature working in a plain browser, then add device capabilities as progressive enhancements behind a capability check.