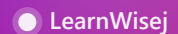


Device APIs through the Hybrid Bridge

Lesson Guide · Lab / Exam Guide



Lesson Guide

Camera, files, geolocation, push notifications and biometrics from the app through the shell, with capability checks and permission prompts. What this means for a team shipping a Wisej.NET app beyond the browser tab — and how to read this module.



Production focus

Camera, files, geolocation, push notifications and biometrics from the app through the shell, with capability checks and permission prompts.

1. The production problem

A FieldOps technician photographs a corroded valve, collects a signature on the glass and drives to the next address — while the dispatcher who assigned the job watches the same work order in a desktop browser. Every control in FieldOps is a C# object on your server, which cannot touch the camera, the GPS chip or the notification tray. This module is the round trip that closes that gap, and what the screen does where the round trip does not exist.

2. Key concepts

Concept	Why it matters
device bridge	The message channel a hybrid shell exposes to the page it hosts: your server asks the shell to perform a device operation, and the shell answers with a message the app turns back into a server-side event.
host object	The object a WebView injects into the hosted page so JavaScript there can call native code. Server-side C# reaches it only through the app's JavaScript interop.
capability descriptor	The features a shell reports when a session starts — camera, location, notifications, biometrics — cached and consulted before any device action is shown.
progressive enhancement	Designing each screen so the plain-browser path is a complete feature and the native path is a better version of it, never the only one that works.
just-in-time permission	Asking the platform for camera, location or notification access at the moment the user presses the button that needs it, right after a sentence explaining why.
bridge result event	The asynchronous reply — photo bytes, coordinates or a refusal — arriving on a later request than the click that asked for it, so it needs its own push.
storage outside the web root	Writing captured photos and signatures where no URL maps to them, then serving them through a handler that checks whether this session may see that work order.

3. Lesson narrative

Asking the shell, and waiting for the answer

The Take Photo button does not open a camera. Its handler composes a request — a still, longest edge 1600 pixels, JPEG quality 70 — and hands it down two hops. The first is the one Wisej.NET already gives you: server-side code runs JavaScript in the hosted page and gets a value back, the mechanism behind `Application.Call` and the widget interop layer. The second is the shell's: that JavaScript calls the host object the WebView injected, and the shell invokes the platform camera. The reply returns the same way, raised as an ordinary .NET event carrying a byte array.

What makes this different from a normal click is time. The technician may spend forty seconds framing the shot, so the handler cannot block; it returns immediately and the request stays pending in `Application.Session` under a token you generated. The answer arrives on a different request, so the code that stores the bytes and fills the picture box runs outside the original event, and its control changes will not reach the browser on their own — finish with `Application.Update`, exactly as for `Application.StartTask` work.

Cancellation and failure are results too, not exceptions: the user backed out, the request timed out, the shell was suspended and resumed. All arrive through the same event, and the token is what lets you drop an answer nobody is waiting for.

```
private void btnPhoto_Click(object sender, EventArgs e)
{
    if (!Shell.Current.Has("camera")) { uploadPhoto.Visible = true; return; }
    var token = Shell.Current.RequestPhoto(maxEdge: 1600, quality: 70);
    this.pendingPhotos[token] = this.workOrder.Id;
}

private void OnPhotoCaptured(object sender, ShellResultEventArgs e)
{
    if (!this.pendingPhotos.TryGetValue(e.Token, out var id)) return;
    this.photoStore.Save(id, e.Bytes);
    this.picPhoto.Image = Image.FromStream(new MemoryStream(e.Bytes));
    Application.Update(this);
}
```

One screen, four shells

FieldOps ships to a browser, a PWA, a WebView2 desktop window and a .NET MAUI mobile shell, and the work-order page is the same page in all four. What it shows comes from a capability check performed once per session: the app asks the page whether a host object is present and what the shell reports it can do, and that answer goes into the session for every screen to read. What you must not do is parse the user agent — it describes the browser, not whether this build of your shell exposes a camera.

The matrix has a row per capability and a column per shell. Camera is native in the mobile shell and, on a phone browser, the **Upload** control, which the browser satisfies straight from the camera app; the desktop shell offers a file dialog instead. Location is native on mobile and the browser Geolocation API reached by JavaScript interop elsewhere. Notifications come from the OS in the two shells and from web push in the browser and PWA. Biometrics exist only where the platform has an authenticator, so FieldOps uses them to re-unlock a session, never as the only way in.

Take Photo and **Upload** are therefore two renderings of one action, and the rest of the screen never knows which ran. A capability the shell does not report simply does not appear: a disabled button with no explanation is worse than an absent one, and one that throws when pressed is worse than both.

Permission at the moment of use

Two gates stand in front of every device feature: the platform permission, where the OS or browser asks the user to allow the camera, the location or notifications, and your own explanation, which comes first. FieldOps shows one sentence — the location shows the distance to the next site and stamps the arrival time — and only then triggers the platform prompt.

Timing decides the answer. A location prompt on the login screen, before the technician has seen a work order, is answered "no", and on several platforms that no is permanent; the same prompt raised by Navigate on a job whose address is on screen is answered "yes" almost every time. So FieldOps asks for notifications after the session's first assignment, and for the camera when Take Photo is pressed.

- Granted: continue, and remember the grant for this session.
- Denied once: keep the action and the reason; the next job may get a different answer.

- Denied permanently: swap in the manual alternative — attach an existing photo, type the address — and show where the setting lives.

The one behaviour that guarantees a permanent refusal is re-prompting in a loop. Record the outcome of every request per capability per session, log it, and treat a refusal as an ordinary state of the screen rather than an error.

Getting the bytes home

A photo from a current phone is four to eight megabytes, and a technician on a rural connection attaches three per job. Resize before the bytes cross the bridge: the shell request carries a bounded longest edge and a JPEG quality, and the browser fallback draws the picked file into a canvas at the same bound and re-encodes it before **Upload** posts it. Six megabytes becomes roughly three hundred kilobytes — the difference between a usable and an unusable app on a weak signal.

On the server, stream rather than buffer: copy the incoming stream into storage and keep only the identifier in session state, because a byte array parked on a screen object is multiplied by every concurrent technician. Write the file outside the web root under a random name, and serve it through a handler that checks this session is entitled to that work order.

The signature is the same pipeline with a different source: a drawing surface collects the strokes, the flattened image is exported as PNG and posted, and the server stores it with the work-order identifier, the signer's typed name and a server-generated timestamp — never a time or identity supplied by the client. Location is the smallest and most sensitive payload: store latitude, longitude, accuracy radius and server time together, since a two-kilometre fix is not proof of arrival, and capture it only on an explicit action.



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:

- ✓ Call device features from server-side code through the shell bridge and receive results as events.
- ✓ Check capabilities at runtime so the same screen works in a browser, a PWA and a native shell.
- ✓ Request permissions at the right moment and handle denial gracefully.
- ✓ Move photos, signatures and location data to the server safely and efficiently.

Lab / Exam Guide

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



Hands-on lab

Add device features to FieldOps: a Take Photo action that uses the device camera in the native shell and falls back to an Upload control in the browser, a signature capture uploaded as an image, a Navigate action that reads the device location and opens the maps app, a push notification when a work order is assigned, and a capability check that shows or hides each action per shell.

Starting point

You open FieldOps as Modules 1 to 4 left it: the work-order list, the detail page and the status update flow all run in a plain browser, the PWA installs and updates itself, the WebView2 desktop shell hosts the app in a native window, and the .NET MAUI shell runs it on Android and iOS. The detail page already has a Photos panel, a Signature panel and a Navigate button, all of them inert placeholders, and the shells expose a bridge that nothing in the web app has called yet.

Suggested approach

- ✓ Write a capability service that runs once when the session starts: ask the hosted page whether the shell's host object is present, record which features it reports — camera, location, notifications, biometrics — in the session, and expose them to every screen through one method.
- ✓ Implement Take Photo as one server-side action with two paths behind it: a bridge request bounded to a 1600-pixel longest edge and JPEG quality 70 in the shells, and an **Upload** control resized in the browser before it posts; store the returned bytes outside the web root and render the thumbnail through an authorized handler.
- ✓ Add the signature pad on a drawing surface in the page, capture the strokes, export the flattened PNG and upload it with the work-order identifier, the signer's typed name and a server-generated timestamp.
- ✓ Wire Navigate to a location request that shows its reason first, store latitude, longitude, accuracy radius and server time on the visit record, then hand off to the platform maps app through the shell, or open a maps link in a new browser tab when there is no bridge.
- ✓ Raise a notification when a work order is assigned — through the shell where the OS provides one, web push otherwise — request notification permission after the first assignment of the session rather than at login, and log every permission outcome per capability.

Required deliverables

- ✓ Camera capture with browser upload fallback
- ✓ Signature capture and upload
- ✓ Geolocation and maps hand-off
- ✓ Push notification on assignment

- ✓ Capability check matrix across shells

Student review questions

- ✓ The camera result arrives forty seconds after the click that asked for it. Explain which request it arrives on, what session state it must find in order to know what to do with the bytes, and why the picture box does not appear in the browser without an explicit push.
- ✓ A technician refused the location permission permanently three weeks ago on their own phone. Describe exactly what pressing Navigate should do today, and name the one thing the app must never do in response.
- ✓ Take Photo works in the mobile shell and the Upload fallback works in desktop Chrome. Name one thing that is genuinely different for the user between those two paths, and argue whether that difference is acceptable for a field technician.



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.