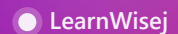


Hosting Models Beyond the Browser

Lesson Guide · Lab / Exam Guide



Lesson Guide

Browser, progressive web app, desktop shell and mobile shell compared: architecture, capabilities, update model and cost. What this means for a team shipping a Wisej.NET app beyond the browser tab — and how to read this module.



Production focus

Browser, progressive web app, desktop shell and mobile shell compared: architecture, capabilities, update model and cost.

1. The production problem

FieldOps is a field-service application: technicians receive work orders, capture photos and signatures, navigate to sites and work with patchy connectivity. Built as an ordinary Wisej.NET application it already runs in any browser, which is enough for the dispatchers in the office and not enough for a technician standing in a plant room with one bar of signal and a phone in a glove. The question this module answers is not "can we build it" but "how does it reach the people who use it" — and that answer is an architecture decision, not a packaging afterthought.

2. Key concepts

Concept	Why it matters
delivery model	The route the application takes to a user: a browser tab, an installed PWA, a desktop window or a store-distributed mobile app. It changes distribution, updates and device access — never the C# that builds the screen.
host browser engine	Every model renders the same server-driven UI in a browser engine: the user's own browser, or an embedded one — WebView2 on Windows, WebKit on macOS and iOS, the system WebView on Android.
Wisej.NET Hybrid	The framework's .NET MAUI-based shell that hosts your existing Wisej.NET application on Android, iOS and desktop and adds a bridge to device features such as camera, geolocation, files and notifications.
device bridge	The two-way channel between your server-side C# and the native shell: the server asks for a device action, the shell performs it and the result comes back to your code as an event, not as a return value.
capability check	A runtime test of what the current shell can actually do, used to show, hide or substitute a feature so one screen behaves correctly in every delivery model.
application shell cache	The static HTML, CSS, scripts and icons a PWA service worker can keep locally. It is what still loads without a network — the interactive, server-rendered part of the app is not in it.
update asymmetry	The server can be redeployed at any moment and every session gets the new build on reload, while a desktop installer or a store package needs a release and an install first.

3. Lesson narrative

One application, four front doors

A Wisej.NET application is a server process. Your `Window` and `Page` objects, their controls and their event handlers live in server memory, and the browser holds a rendered mirror that sends user actions back over an HTTP and WebSocket connection. Nothing in that arrangement cares which browser engine is on the other end. That single fact is what makes four delivery models possible from one deployment: the same URL, the same session, the same `Application.Session` state.

In a browser tab, the user types or clicks a link and gets the app. As a progressive web app, the same URL is installed from the browser: a manifest gives it a name, an icon and a standalone window with no address bar, and a service worker caches the static shell so the launch feels immediate. In a desktop shell, a small native program opens a window that contains an embedded browser engine — WebView2 on Windows, WebKit on macOS — and points it at your server; the user sees a taskbar entry and a title bar, and the application inside is the same one the dispatchers use. In a mobile shell, Wisej.NET Hybrid hosts the application inside a .NET MAUI app for Android and iOS, which is installed from a store or an enterprise channel and can reach the device.

For FieldOps that means the work-order list is written once and never ported or forked. The shells are containers: the difference between them is what surrounds the app, not what runs inside it.

The comparison that actually decides it

Five columns decide the model, and the first is the honest one. **Offline behaviour**: a server-side UI needs its server. When the connection drops, no shell can keep running your event handlers, because the handlers are not on the device. A PWA or a native shell can cache the shell and show a designed offline screen with the last synced work-order summary, and it can queue an action the technician takes so it is sent when the signal returns — but it cannot open a work order the device has never seen. Promising more than that is how field apps lose their users' trust in week one.

Device access is where the shells separate. A browser and a PWA get what the web platform grants after a permission prompt: file selection, an image through an upload control, coarse geolocation, web notifications where supported. A native shell adds the bridge — a real camera session, background location, the file system, push notifications, biometrics. **Distribution** runs from a URL you email, through an installer, to a store review that can take days. **Updates** follow the same order: the server is instant, a desktop package needs an update channel, a store build needs a submission. **Cost** is the sum of certificates, developer accounts, test devices and the review cycles you now own forever.

- Dispatchers on a desk: browser, and nothing more.
- Technicians who need camera, location and notifications: mobile shell.
- Warehouse kiosks that must survive a flaky Wi-Fi drop: PWA with an honest offline page.

One server, one codebase, many shells

The architecture that keeps this cheap is deliberately boring: one Wisej.NET server deployment, one solution, and shells that contain no business logic at all. The work-order list, the detail page and the status update flow are ordinary Wisej.NET screens calling a `WorkOrderService`. The shell projects added in later modules do exactly two things — host the browser engine pointed at your server, and expose device features to it.

The joint between them is the capability check. Rather than testing "am I on Android", test what you need: can this shell take a photo, read a location, raise a notification. Resolve those capabilities once when the session starts, keep them in session state, and let each screen read them. A screen then adapts instead of branching per platform, and a new shell added next year needs no edit to the screens.

```
public sealed class ShellCapabilities
{
    public bool HasCamera { get; set; }
    public bool HasLocation { get; set; }
    public bool CanNotify { get; set; }

    public static ShellCapabilities Current =>
        Application.Session["Shell"] as ShellCapabilities
        ?? new ShellCapabilities(); // plain browser: everything off
}

// In the work-order detail page:
btnTakePhoto.Visible = ShellCapabilities.Current.HasCamera;
uploadPhoto.Visible = !ShellCapabilities.Current.HasCamera;
```

Note the fallback: not a disabled button, but an **Upload** control that still gets a photo onto the work order. Every capability needs that answer before the feature is designed, which is why the check is a Module 1 deliverable and the device features are not.

Writing the delivery decision record

The last deliverable is a short document, and the one your team will actually re-read. It names the targets, the order you build them in, and the exit criteria that say a target is done. FieldOps' record might read: browser first, because dispatchers need it and it proves the app; then the Android shell, because that is what the technicians carry; then the PWA for warehouse kiosks; iOS only if the supervisors' tablets are approved.

Exit criteria stop a target drifting. "The Android shell is done when a technician completes a work order end to end on a supplied device, the session survives the screen locking, and a signed build installs through the enterprise channel" is checkable; "mobile support" is not. Write the record before any shell project exists, because after the first installer is built every argument about scope becomes an argument about sunk effort.



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:

- ✓ Explain how a Wisej.NET app runs unchanged inside a browser, a PWA, a desktop WebView shell and a mobile shell.
- ✓ Compare the four delivery models on offline behaviour, device access, distribution, updates and cost.
- ✓ Design the FieldOps architecture so one server and one codebase serve every shell.

- ✓ Write a delivery decision record that names the targets, the order to build them and the exit criteria.

Lab / Exam Guide

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



Hands-on lab

Create the FieldOps solution with a work-order list, a work-order detail page and a status update flow that work in a plain browser. Write the delivery decision record: compare browser, PWA, desktop and mobile shells for the technicians' needs, choose the targets and their order, and define the capability checks the app will use to enable device features.

Starting point

You start from a new, empty Wisej.NET Web Application named FieldOps, with nothing in it but the generated start-up window and the default configuration files. Everything in this lab is yours to create — the work-order model and service, the three screens, the capability abstraction and the written decision record — and every shell added in Modules 2 to 6 wraps the server you finish here without changing it.

Suggested approach

- ✓ Model the domain first: a WorkOrder with number, customer, site address, priority, status and assigned technician, behind a WorkOrderService that returns the technician's open orders and applies a status change.
- ✓ Build the work-order list as a grid bound to the service, with a filter for the signed-in technician, and make it usable at a narrow window width so the same screen survives a phone-sized viewport later.
- ✓ Build the detail page for one work order and a status update flow — Accepted, On site, Completed — that validates the transition in the service, writes it once, and refreshes the list.
- ✓ Add a ShellCapabilities object resolved when the session starts and stored in Application.Session, with flags for camera, location and notifications, all false in a plain browser, and have the detail page read it to decide what it shows.
- ✓ Write the comparison table and the delivery decision record, then draw the architecture diagram showing one server, one codebase and the shells that will point at it.

Required deliverables

- ✓ FieldOps solution with list, detail and status update
- ✓ Delivery model comparison table
- ✓ Delivery decision record with target order
- ✓ Capability check design
- ✓ Architecture diagram with one server and multiple shells

Student review questions

- ✓ A technician loses signal halfway through updating a work order. Describe exactly which parts of FieldOps can still be shown from a cached shell, which cannot, and what your screen should do

at the moment the connection drops.

- ✓ Your decision record puts the Android shell before the PWA. What would have to be true about the technicians' devices and network for that order to be wrong?
- ✓ The detail page needs a photo. Explain why a capability check plus an Upload fallback is a better design than a build-time switch that produces a mobile-only version of the screen.



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.