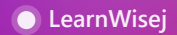


Progressive Web App Setup

Lesson Guide · Lab / Exam Guide



Lesson Guide

Web app manifest, icons and splash screens, the service worker, installability, offline shell and the update flow for a PWA. What this means for a team shipping a Wisej.NET app beyond the browser tab — and how to read this module.



Production focus

Web app manifest, icons and splash screens, the service worker, installability, offline shell and the update flow for a PWA.

1. The production problem

FieldOps technicians do not open a browser and type a URL. They tap an icon on a phone that has been in a van all morning, in a plant room where the signal drops to nothing halfway through a job. A progressive web app gives them that icon, a window without browser chrome and a page that still says something useful when the network disappears — but FieldOps is a Wisej.NET app, so its user interface lives on the server, and pretending otherwise is how a PWA turns into a support ticket.

2. Key concepts

Concept	Why it matters
web app manifest	A small JSON file, linked from the start-up page, that tells the browser the app's name, icons, colors, display mode and start URL — without it the browser has nothing to install.
display mode	The <code>display</code> member of the manifest. <code>standalone</code> launches FieldOps in its own window with no address bar, which is what makes an installed app stop feeling like a bookmark.
maskable icons	Icons declared with <code>purpose: "maskable"</code> and safe padding, so Android can crop them into its own shape without clipping the artwork.
service worker	A script the browser runs beside the page, served from the app's own origin over HTTPS, that can answer the page's requests from a cache when the connection is slow or gone.
application shell	The static files a session needs before the server can say anything: the start-up page, the client scripts, the theme and the icons. It is what the worker is allowed to cache.
cache-first vs network-only	The routing rule inside the worker's fetch handler: static shell files come from the cache first, while everything the Wisej.NET session exchanges with the server must go straight to the network, never a cached copy.
update lifecycle	A new worker installs while the old one still controls the page and then waits. <code>skipWaiting</code> , <code>clients.claim</code> and a controlled reload are what turn that waiting worker into the version the technician actually runs.

3. Lesson narrative

The manifest is the install contract

Installability is not a switch in Wisej.NET; it is a set of conditions the browser checks. FieldOps must be served over HTTPS, its start-up page must link a valid web app manifest, and a service worker must be registered for the page's scope. The manifest lives with the application's other static files — the same folder as the start-up page, the theme and the icons — and is referenced from `Default.html`, which is Wisej.NET's start-up document and the one place in the project where you own the browser's HTML.

The members that matter are few. `name` and `short_name` decide what appears under the icon and in the task switcher. `icons` needs a 192-pixel and a 512-pixel PNG plus maskable variants with padded artwork, and iOS additionally wants an `apple-touch-icon` link on the page itself. `theme_color` tints the system bars around the standalone window while `background_color` paints the splash screen the browser generates at launch, so both should match the FieldOps app bar. `display` is `standalone`, and `start_url` with `scope` point at the application root so a launch lands on the work-order list and every URL under the app stays inside the installed window.

When the browser is satisfied it fires `beforeinstallprompt` on the page instead of showing its own banner. The handler in `Default.html` stops that default, keeps the event and exposes a small function that calls its `prompt()` method. Two rules follow. The prompt only opens from a real click inside the browser, so the button that triggers it belongs in the page, not on the server. And the server decides *when*

the offer is worth making — the lab asks for it after the first successful session — revealing the affordance through `Application.Eval` or the equivalent JavaScript interop:

```
private void OnFirstSessionCompleted()
{
    // Default.html stashed the beforeinstallprompt event; the server only
    // decides when the offer becomes visible. prompt() still runs from the
    // page's own click handler, inside a real user gesture.
    Application.Session["InstallOffered"] = true;
    Application.Eval("window.fieldOps.showInstallButton()");
}
```

A worker that caches the shell, never the session

Register the worker from `Default.html` with `navigator.serviceWorker.register`, pointing at an `sw.js` served from the application root so its scope covers the whole app. On `install` the worker opens a named cache and puts the shell into it: the start-up page, the client scripts and theme files the page loads before the session exists, the manifest, the icons and the offline page. On `activate` it deletes every cache whose name is not the current one.

The `fetch` handler is where a PWA over a server-driven UI succeeds or fails. FieldOps is a Wisej.NET application: the controls are C# objects on the server, and what crosses the wire during a session is a stream of requests and responses reflecting that server state, plus a WebSocket for pushed updates. None of it is cacheable in any meaningful sense — a replayed response describes a session that no longer exists. So the handler matches on the request: a `GET` for a known shell file is answered cache-first, non-`GET` requests are never touched, and everything else goes to the network untouched. When a navigation fails because the server is unreachable, the handler answers with the cached offline page instead of the browser's error screen.

Honest offline for a server-driven UI

Say plainly what offline means here: the shell loads, the app does not run. Without the server there is no session, no work-order data and no way to change a status, and no amount of caching moves a server-side UI into the browser. The design work is making that state useful rather than broken. FieldOps writes a small summary after each successful load — open work-order count, next site, and the time of the sync — into browser storage, and the offline page renders it behind a clear "not connected" banner and a retry.

- Nothing captured on the offline page is presented as saved; queued edits are a later, deliberate feature, not a side effect of a cache.
- Every offline surface carries the timestamp of the last sync, so old data is never mistaken for current data.
- When connectivity returns, the app reloads into a fresh session rather than resuming a stale one.

Reconnection deserves the same honesty. A Wisej.NET session can time out or be lost while the device sleeps in a van; the reliable recovery is to reload the start-up page and let the server build a new session, not to leave a dead UI on screen looking live.

Shipping a new version past a cache that is doing its job

The cache that makes FieldOps fast is also what leaves a technician on last month's build. Version the cache name — `fieldops-shell-v7` — and change it in the same commit as any shell file. The new worker then installs into a new cache, and its `activate` handler deletes the old ones. Call `skipWaiting` so the new worker does not sit behind the old one, and `clients.claim` so it controls the open page immediately.

The page still has to act on that. Listening for `controllerchange`, or for `updatefound` and `statechange` on the registration, tells it a new version has taken over; FieldOps shows a small "A new version is ready — Reload" bar and reloads on the technician's tap, so nobody loses a half-filled form to a surprise refresh. Two deployment details complete the flow: `sw.js` and the manifest are served with no-cache headers so the browser can notice they changed at all, and the app compares its build stamp with the server's at session start, catching a shell that survived a deployment instead of letting it talk to a server it no longer matches.



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:

- ✓ Add a web app manifest with icons, colors and display mode so the app can be installed from the browser.
- ✓ Register a service worker that caches the application shell and serves it while the network is slow or absent.
- ✓ Explain what can and cannot work offline in a server-side UI and design the offline experience honestly.
- ✓ Handle PWA updates so users get the new version without a stale cache.

Lab / Exam Guide

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



Hands-on lab

Turn FieldOps into a PWA: add the manifest with icons, theme color and standalone display, register a service worker that caches the shell and static assets, show an install prompt after the first successful session, display an offline page with the last synced work-order summary when the server is unreachable, and implement the update flow that reloads when a new version is available.

Starting point

You open the FieldOps solution you built in Module 1: a Wisej.NET web application with a work-order list, a work-order detail page and a status update flow that already run correctly in a plain browser tab. Nothing about the app is installable yet — there is no manifest, no service worker and no offline page — and the static files you will add live beside the existing `Default.html`, theme and icon files.

Suggested approach

- ✓ Serve FieldOps over HTTPS locally, then write `manifest.webmanifest` with the app name, 192 and 512 pixel icons including maskable variants, the FieldOps theme and background colors, `display: standalone` and a start URL at the application root, and link it plus an apple-touch-icon from `Default.html`.
- ✓ Add `sw.js` at the application root with a versioned cache name, an install handler that precaches the shell — start-up page, client scripts, theme, icons and the offline page — and an activate handler that deletes every cache from an older version.
- ✓ Write the fetch handler so shell requests are answered cache-first while every request the Wisej.NET session makes to the server goes to the network untouched, and a failed navigation falls back to the cached offline page.
- ✓ Capture `beforeinstallprompt` in the page, keep the event, and have the server reveal the in-page Install button through JavaScript interop only after the technician has completed one successful session with the work-order list loaded.
- ✓ Write a compact work-order summary to browser storage after every successful load so the offline page can render it with a last-synced timestamp, then bump the cache version, deploy, and confirm the update bar appears and reloads into the new build.

Required deliverables

- ✓ Web app manifest with icons and colors
- ✓ Service worker caching the shell
- ✓ Install prompt after first session
- ✓ Offline page with last synced summary
- ✓ Update flow and version check

Student review questions

- ✓ A reviewer asks why the service worker does not simply cache every response so FieldOps keeps working underground. Explain what a replayed Wisej.NET session response would actually do to the technician's screen, and where the honest boundary lies.
- ✓ Your install offer works on your desk but never appears on the technician's phone. Name three separate conditions that could be responsible, and say how you would tell them apart from the device.
- ✓ A technician has the app installed and open when you deploy a new build. Describe every step between your deployment and that phone running the new shell, and identify which step would silently fail if the cache name never changed.



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.