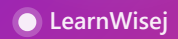


Signing, Distribution, Updates and the FieldOps Capstone

Lesson Guide · Lab / Exam Guide



Lesson Guide

Code signing, store submission, enterprise distribution, versioning, update strategies across shells, telemetry and the capstone delivery. What this means for a team shipping a Wisej.NET app beyond the browser tab — and how to read this module.



Production focus

Code signing, store submission, enterprise distribution, versioning, update strategies across shells, telemetry and the capstone delivery.

1. The production problem

FieldOps finishes this course as one server and four ways to reach it: a browser tab, an installed PWA, a Windows desktop shell and a mobile shell on Android and iOS. The three packaged ones are files an operating system refuses to install unless an identity it trusts signed them, and each travels a different channel with its own review queue and rollout speed. You can redeploy the server before lunch; the iOS build a technician is holding may be three weeks behind it, and it still has to work.

2. Key concepts

Concept	Why it matters
Authenticode signature	The signature a code-signing certificate applies to a Windows installer or MSIX, proving the file was not altered after signing and naming a vetted publisher.
countersignature timestamp	A trusted timestamp recorded at signing time, so the installer keeps validating after the signing certificate itself expires.
Android keystore and upload key	The key that signs an Android build; with Play App Signing you sign with an upload key and Google re-signs with the app signing key it holds for that package.
Apple distribution certificate and provisioning profile	The certificate identifies your team; the profile binds an App ID, that certificate, the entitlements and, outside the App Store, the allowed devices.
enterprise distribution	Delivering a private build to one organisation's devices — managed Google Play, Apple Business Manager custom apps, an MSIX pushed by Intune — instead of publishing publicly.
minimum shell version	The oldest shell build, per platform, the current server still supports; anything below it is stopped on connect before it reaches any data.
device-feature usage event	A record written whenever a shell capability runs — camera, signature, location, push, offline flush — carrying the outcome plus the shell platform and version.

3. Lesson narrative

What a signature proves, and the three FieldOps needs

A code signature makes exactly two claims: the bytes are the bytes that were signed, and a named identity signed them. It says nothing about whether the code is well behaved, and the whole distribution chain rests on those two claims and nothing else. Sign the Windows package with a code-signing certificate issued to your company and always add a trusted timestamp; without one the installer stops validating the day the certificate expires. If you ship the desktop shell as an MSIX, the certificate subject must match the publisher name in the package manifest, or Windows rejects it outright.

Android is a different model: the build is signed with a key held in a keystore, and the identity the device cares about is that key, not a certificate authority. Enrol FieldOps in Play App Signing, where you sign each release with an upload key and Google verifies it, strips it and re-signs with the app signing key it holds. That removes the worst failure in mobile distribution, a lost key that makes the app permanently un-updatable, because only the upload key is yours to lose and it can be reset.

Apple needs two artefacts. The distribution certificate identifies your team; the provisioning profile says that this App ID, signed by that certificate, may use these entitlements and — outside the App Store — may run on these devices. Push notifications, which FieldOps uses to announce a new work order, are an entitlement, so a profile that predates that capability produces a build that installs and then silently

receives nothing. The capstone asks for all three signed artefacts plus a note per platform recording who holds the key, where it lives and when it expires.

Store, enterprise or direct: one decision per platform

Three channels exist and each platform picks one. A public store listing suits an app anyone may install: Google Play and the App Store give you hosting, delta updates, staged rollouts and a review queue measured in hours to days. Enterprise distribution suits FieldOps better, because technicians are employees on managed devices: managed Google Play publishes privately to one organisation and assigns through their EMM, Apple Business Manager distributes a custom app to named organisations that push it with MDM, and on Windows the signed MSIX goes out through Intune. Direct download — your own server hosting the signed installer and a version feed — stays useful for the desktop shell, and is essentially unusable on iOS.

The browser and the PWA are the odd ones out, and that is the point of the whole course: no channel, no certificate, no review. You deploy the server and the next page load is the new version. Every hour spent keeping FieldOps fully usable in a plain browser is insurance against a store queue.

Preparing a submission is real work, so the lab asks for a store listing draft even where you distribute privately: the name and description written for a technician rather than a shopper, screenshots at each required device size, a privacy and data-safety declaration that honestly lists the camera, photo and location data FieldOps collects, and a support contact. The enterprise setup is the mirror image: assignment groups, managed configuration such as the server address, and who in the customer's IT team approves a new version.

The server sprints, the shells crawl

Because the shells cannot keep pace with the server, the compatibility rule must be explicit. Give the server its own version and a per-platform minimum shell version — the oldest build it is still willing to talk to. The shell sends its platform and version on connect, as parameters on the address it loads or through the bridge handshake, and the server records both in `Application.Session` so every later log line and telemetry event can name them. Below the minimum, the technician gets a blocking update screen pointing at the right destination for that channel and no work-order data is loaded at all. At or above the minimum but behind the latest, show a dismissible notice and let them keep working.

```
public ShellGate AdmitShell(string platform, Version shellVersion)
{
    var minimum = _config.MinimumShellVersion(platform); // e.g. 2.4.0
    if (shellVersion < minimum)
        return ShellGate.MustUpdate(minimum, _config.UpdateTarget(platform));

    Application.Session["ShellPlatform"] = platform;
    Application.Session["ShellVersion"] = shellVersion;

    return shellVersion < _config.LatestShellVersion(platform)
        ? ShellGate.UpdateAvailable(_config.ServerVersion)
        : ShellGate.Current(_config.ServerVersion);
}
```

Raising the minimum is a deliberate act, not a side effect of a release. Raise it only when the server starts depending on something older shells cannot do: a new bridge method, a changed payload shape, a permission the old build never requested. Everything else, including new screens, must degrade on the shells already in the field. The discipline is worth more than the check itself, because it forces the question before every deploy — what will the two-month-old Android build do with this?

Telemetry that names the shell, and the release checklist

With four deliveries in production, a bug report that says "it crashed" is nearly useless. Put a crash reporting SDK inside each shell so native crashes and unhandled exceptions in the shell process arrive with a stack trace, the device model and the build number, even though they happened outside your C# UI code. On the server, log unhandled exceptions with the session's shell platform and version attached, so one incident can be read from both ends. Then add a usage event per device feature: photo captured, signature saved, location resolved, push received, queued upload flushed, each with its outcome and the shell it came from. Two weeks of those answer what arguments cannot, such as whether the camera fallback fires because a permission was denied or because the shell is too old to have the capability at all.

Telemetry is data about people's work, so declare it. No photo contents, no signature images, no customer names, no raw coordinates where a work-order identifier will do — and whatever you do collect belongs in the privacy declaration you filed. The capstone then ends with a release checklist, because four deliveries have four different failure modes and the checklist is what makes a release repeatable by someone who was not there when it was designed.

- Browser: server deployed, version endpoint updated, cold-session smoke test.
- PWA: service worker cache version bumped, update flow verified from the previous build.
- Desktop: installer signed and timestamped, signature verified on a clean machine, Intune package replaced.
- Mobile: right keys and profiles, minimum shell version reviewed, staged rollout set, crash dashboard watched on day one.



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:

- ✓ Sign desktop and mobile packages and explain what each certificate proves.
- ✓ Choose between store, enterprise and direct distribution per platform and prepare each submission.
- ✓ Design a versioning and update strategy where the server can move faster than the shells.
- ✓ Add crash and usage telemetry across shells and deliver the FieldOps capstone.

Lab / Exam Guide

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



Hands-on lab

Complete the FieldOps capstone: sign the Windows installer and the Android and iOS packages, prepare an enterprise distribution for the mobile shells and a store listing draft, define a compatibility rule between server versions and shell versions with a minimum-version check on startup, add crash reporting and a usage event for each device feature, and deliver a release checklist covering all four delivery models.

Starting point

You open the FieldOps solution as Module 5 left it: the Wisej.NET server with the work-order list, detail and status flow, the PWA manifest and service worker, the WebView2 desktop shell, the .NET MAUI shell for Android and iOS, and the bridge that already exposes camera, signature, location and push behind capability checks. Everything currently runs from unsigned debug builds pointed at a development server.

Suggested approach

- ✓ Produce the three signing identities first — a code-signing certificate for Windows, a keystore with an upload key enrolled in Play App Signing, and an Apple distribution certificate with a provisioning profile that includes the push entitlement — and record for each who holds the key and when it expires.
- ✓ Build and sign a release package per platform, then verify each signature the way the target device will: check the Authenticode signature and its timestamp on a clean Windows machine, and confirm the mobile builds install from their intended channel rather than from a debug cable.
- ✓ Add the version handshake before anything else loads: expose the server version and the per-platform minimum shell version from configuration, have each shell report its platform and version on connect, and store both in the session.
- ✓ Build the update screen the gate shows an outdated shell, with the destination that matches its channel, and a dismissible notice for a shell that is merely behind the latest build.
- ✓ Wire crash reporting into each shell and a usage event into every bridge call, then write the release checklist by walking one release end to end across browser, PWA, desktop and mobile.

Required deliverables

- ✓ Signed Windows, Android and iOS packages
- ✓ Enterprise distribution setup and store listing draft
- ✓ Server and shell version compatibility check
- ✓ Crash and usage telemetry
- ✓ Release checklist for all delivery models

Student review questions

- ✓ A technician's Android shell is six weeks old and the server has shipped four times since. Which of those four server changes should have raised the minimum shell version, and how would you have known at the time?
- ✓ The Windows installer was signed but not timestamped, and the certificate expired last month. What do existing installations do, what do new downloads do, and what is the shortest path back to a trusted installer?
- ✓ The camera usage event shows a sharp rise in the browser fallback path on iOS only. Name three explanations that fit the data, and say which extra field in the event would tell them apart.



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.