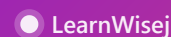


Deployment, Diagnostics, Load Balancing & Capstone Delivery

Prepare a Wisej.NET application for deployment and operations with hosting choices, health checks, logging, diagnostics, load balancing, release notes, and capstone presentation.



Lesson goal

Prepare a Wisej.NET application for deployment and operations with hosting choices, health checks, logging, diagnostics, load balancing, release notes, and capstone presentation.

This is the module where a working application becomes a *running service that other people depend on*. The code you shipped in the earlier modules already does the right thing; the job now is to place it somewhere it can be hosted, watched, scaled and recovered when something goes wrong. Deployment is not a publish button — it is a small set of operational decisions you make once and live with every day the app is in production.

Everything in this module hangs off one fact you already learned in Module 2: **Wisej.NET sessions are stateful and live on the server**. A user's controls, forms and session objects are real .NET objects sitting in the memory of a specific machine. That single property shapes how you load-balance, how you scale, and what "restart the server" costs a logged-in user. We will keep coming back to it.

1. Why stateful sessions drive every deployment decision

Recall the round-trip model: a click travels to the server, Wisej.NET finds *your* live control instance, runs your handler, and sends back only what changed. That live instance is the whole point — it is why state persists between clicks without you serializing anything. But it means the user is **bound to the node that holds their session**. If a later request lands on a different node that has never seen that session, the user's objects are not there.

Two consequences follow, and they run the rest of this lesson:

- **Load balancing needs affinity.** When you put two or more app instances behind a load balancer, every request from a given user must return to the node that owns their session. This is called a *sticky session* (session affinity). Plain round-robin — spraying requests across nodes evenly — will drop Wisej.NET sessions and log users out mid-action.
- **A restart is a session eviction.** Recycling the IIS app pool, redeploying, or scaling a node down destroys the in-memory sessions on that node. Users on it are dropped. That is why release timing, drain windows and rollback plans matter — they are not paperwork, they are the difference between a clean deploy and hundreds of interrupted users.



Plain-English definition

A **sticky session** (session affinity) is a load-balancer rule that always sends a given user back to the same server node. Because a Wisej.NET session lives in that node's memory, affinity is usually mandatory, not optional.

2. Key concepts

The production focus of this module, concept by concept:

#	Concept	What it gives you
1	IIS	Windows-native hosting with app-pool management and mature tooling.
2	Kestrel	The cross-platform .NET web server for self-hosted or containerized deploys.
3	NGINX/Apache reverse proxy	A front door that terminates TLS and, critically, applies affinity to Kestrel.
4	Azure/AWS	Cloud hosting for App Service, VMs, or container platforms with managed load balancers.
5	Linux support	Running the app on Linux via .NET + Kestrel, usually in a container.
6	Health check	An endpoint the load balancer or orchestrator polls to know a node is alive.
7	Sticky sessions	Affinity that keeps each user pinned to their session's node.
8	Diagnostics page	A role-protected in-app view of runtime state, safe of secrets.
9	Release checklist	Notes, rollback plan and demo evidence that make a deploy repeatable.

3. Lesson narrative

Deployment is architecture, because hosting affects logging, configuration, scaling, identity, WebSocket routing, health checks, and operational support. Wisej.NET uses a persistent connection (WebSocket, with fallback) between the browser widget and the server, so whatever sits in front of your app — a proxy, a load balancer, a cloud gateway — must be configured to **route and keep alive that connection**, not just plain HTTP requests.

Wisej.NET session state lives on the server, so load-balanced deployments usually need session affinity (sticky sessions). A diagnostics page must be role-protected and must avoid exposing secrets.

Choosing a host

There is no single right host — there is the right host *for your constraints*. The three common shapes:

Host	Best when	Trade-offs
IIS (Windows)	Windows-centric shops, existing IIS estate, Windows auth, familiar ops.	Windows-only; app-pool recycling can silently evict sessions; scaling means more Windows nodes.
Kestrel / self-host	Cross-platform, Linux, or when you want a lean process with a reverse proxy in front.	You own TLS, process supervision and affinity config at the proxy; more moving parts to get right.
Container (Docker)	Reproducible builds, cloud orchestration (Azure/AWS), horizontal scale.	Sessions are in-container memory — scaling out <i>requires</i> affinity at the ingress; restarts drop sessions.

Notice the common thread: whichever host you pick, the moment you have more than one instance you must answer the affinity question. Single-instance deployments dodge it — which is exactly why many teams run one large node rather than several small ones until traffic forces the issue.

A health-check endpoint

A **health check** is a cheap endpoint that answers one question: "is this node able to serve requests right now?" Load balancers and container orchestrators poll it and stop sending traffic to a node that fails. Keep it fast and honest — check the things whose failure should take the node out of rotation (database reachable, disk writable), and nothing that is merely slow.

A minimal health-check handler and what it should and shouldn't do

```
GET /health
// The balancer polls this every few seconds.
// 200 = "send me traffic"; anything else = "take me out of rotation".

if (CanReachDatabase() && CanWriteTempFiles())
    return 200 OK    // { "status": "healthy" }
else
    return 503 Service Unavailable // degraded – pull me out

// DO check: dependencies whose failure means this node can't work.
// DON'T: run heavy queries, or leak versions/secrets in the body.
```

The response body should be tiny and non-secret. A health check is public-facing infrastructure, not a diagnostics dump.

Structured logging you can actually query

Structured logging means writing log events as data — a level, a message, and named fields — instead of pasting values into a free-text string. The difference shows up the day production breaks and you need to filter thousands of lines down to "errors for user 4821 in the last hour."

Free-text vs. structured — same event, very different searchability

```
// Hard to search: everything is baked into one string.
log.Info("Saved ticket 501 for user 4821 in 320ms");

// Structured: level + message template + named fields.
log.Information("Saved ticket {TicketId} for {UserId} in {ElapsedMs}ms",
    ticketId, userId, elapsedMs);

// Now your log store can filter by UserId, sum ElapsedMs,
// and alert when error-level events spike — no regex archaeology.
```

Give every request or session a **correlation id** and attach it to each log event. When a user reports a problem, that one id lets you follow their exact path across the log — invaluable in a load-balanced world where their requests all landed on one node but your logs may be aggregated from many.

Load-balancer affinity, sketched

Behind a reverse proxy (NGINX/Apache) or a cloud load balancer, affinity is usually a cookie the balancer issues and reads. Here is the shape of the rule, described rather than tied to one product's exact syntax:

The affinity rule every multi-node Wisej.NET deploy needs

```
// Reverse proxy in front of two Kestrel nodes:
upstream wisej_app {
    // Pin each client to one node by a balancer-issued cookie.
    sticky cookie affinity;    // name varies by product
    server app-node-1:5000;
    server app-node-2:5000;
}

// Also required: forward the WebSocket upgrade so the
// persistent Wisej.NET connection survives the proxy.
proxy_set_header Upgrade $http_upgrade;
proxy_set_header Connection "upgrade";
```

Two failures hide here. Forget the sticky cookie and users bounce between nodes and lose their session. Forget the WebSocket upgrade headers and the Wisej.NET connection can't establish through the proxy at all. Both must be right.

A safe diagnostics page

A **diagnostics page** is an in-app view — reachable only by an operator role — that surfaces runtime facts: app version, environment name, uptime, session count, last errors, dependency status. It turns "is the server okay?" into something you can read at a glance. Two rules keep it from becoming a liability: it must be **role-protected** (never anonymous), and it must **never render secrets** — no connection strings, API keys or tokens, even partially.



Production habit

Prepare health checks, safe diagnostics, release notes, rollback notes, and demo evidence.



Learning objectives

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

- ✓ Explain common Wisej.NET deployment targets and when to choose each one.
- ✓ Prepare a deployment checklist for IIS, Kestrel/reverse proxy, Linux, Azure, or AWS.
- ✓ Configure health checks and understand sticky-session / load-balancer requirements.
- ✓ Package, demonstrate, and defend the capstone application.

4. Release notes and rollback

A deploy without a written plan is a gamble. **Release notes** record what changed, why, and any migration or configuration steps; a **rollback plan** records exactly how to get back to the previous known-good version if the new one misbehaves. Because a Wisej.NET restart evicts sessions, your plan should also state *when* you deploy (a low-traffic window) and whether you drain a node before recycling it.

The minimum a release note should carry

```
Release 4.2.0 - 2026-07-03
// What changed, in user terms
- Ticket export now includes closed tickets
- Fixed session drop under load (affinity cookie corrected)

// Operational steps
- DB migration 0031 must run before deploy
- Config: set Reporting__Timeout=60

// Rollback: redeploy tag v4.1.3; migration 0031 is additive (safe to leave)
```

Common mistakes

Mistake	Why it hurts	Do instead
Round-robin load balancing with no affinity.	Requests land on nodes that don't hold the user's session; users are logged out mid-action.	Enable sticky sessions (affinity cookie) at the balancer or proxy.
No health check, or one that always returns 200.	The balancer keeps sending traffic to a dead or degraded node.	Expose a real <code>/health</code> that checks live dependencies.
Logging nothing useful — or free-text only.	An incident becomes regex archaeology; you can't filter by user or correlate a request.	Structured logging with levels and a correlation id per session.
Diagnostics page open to anyone / showing secrets.	Leaks connection strings, versions and attack surface.	Role-protect it and render no secrets, ever.
No release notes or rollback plan.	A bad deploy has no defined way back; recovery is improvised under pressure.	Write notes + a tested rollback before you deploy.
Recycling / redeploying at peak with no drain.	Every live session on the node is destroyed at once.	Deploy in a low-traffic window; drain the node first where possible.

Key vocabulary

- **Health check** — a lightweight endpoint reporting whether a node can serve traffic; polled by balancers and orchestrators.
- **Sticky session / affinity** — a balancer rule pinning each user to the node that holds their in-memory Wisej.NET session.
- **Structured logging** — log events written as data (level + message template + named fields) so they can be filtered, aggregated and alerted on.
- **Diagnostics** — a role-protected in-app view of runtime state (version, uptime, sessions, errors) that never exposes secrets.
- **Release notes** — a record of what changed, the operational steps required, and the rollback path to the prior version.
- **Reverse proxy** — a front-end (NGINX/Apache/cloud gateway) that terminates TLS, forwards the WebSocket upgrade, and applies affinity.

Common pitfall to avoid



Common pitfall

Treating deployment as only a publish button instead of an operational plan.

Capstone delivery checklist

Your capstone is judged as a deliverable someone could operate, not just run. Walk this list before you present:

- ✓ The chosen host is stated, with a reason (IIS vs Kestrel/self-host vs container).
- ✓ If more than one instance runs, sticky sessions / affinity is configured and demonstrated.
- ✓ A working `/health` endpoint returns a real status, checked by the balancer or manually.
- ✓ Structured logging is in place, with a correlation id you can follow through an incident.
- ✓ A role-protected diagnostics view shows runtime state and leaks no secrets.
- ✓ Environment-specific values live in configuration, not in code.
- ✓ Release notes and a rollback plan are written and included in the package.
- ✓ Demo evidence — screenshots or a short recording — shows the app running and recovering.

Designer & code review checklist

- ✓ The screen remains usable in the Visual Studio / Wisej.NET Designer.
- ✓ Controls are named clearly enough for a teammate to follow the event code.
- ✓ Business logic is not trapped in visual event handlers.
- ✓ Failure paths are visible, logged, and explained without leaking internals.
- ✓ The deliverable can be reviewed without running the whole course.



Self-check before the lab

Does the load balancer need sticky sessions? Where are environment-specific values stored? What is the rollback plan if deployment fails? If you can answer all three without hedging — and point to the health check and diagnostics page — you are ready.

Companion code

`DeploymentAndDiagnosticsPatterns.cs` collects the patterns from this module in one compilable file. Download it and read it alongside the lab.

↓ [Download DeploymentAndDiagnosticsPatterns.cs](#)