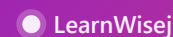


Deployment, Diagnostics, Load Balancing & Capstone Delivery

Go deeper on deployment: hosting trade-offs, health checks and diagnostics, sticky sessions for load balancing, and a clean release and rollback plan.



Applied goal

Understand the operational shape of shipping the TicketOps Console: how a Wisej.NET app is hosted, how the platform probes its health, how support diagnoses it, why load balancing needs session affinity, and how a controlled release with a rollback plan actually reads.

1. Hosting the TicketOps Console: IIS vs container/Azure

A Wisej.NET application is an ASP.NET Core app: it publishes to a folder and runs behind Kestrel. Where that process lives is the first architectural decision, and it changes how you configure logging, secrets, TLS, and scaling. For the capstone the realistic choice is between classic **IIS on a Windows VM** and a **Linux container on Azure** (App Service or Container Apps).

Concern	IIS on Windows VM	Container / Azure App Service
Runtime host	Kestrel behind the IIS ASP.NET Core Module	Kestrel directly, behind the platform proxy
Deploy unit	Published folder + <code>web.config</code>	Docker image or zip-deploy artifact
TLS & certificates	Bound in IIS site settings	Managed by the platform
Configuration	<code>appsettings.json</code> + machine env vars	App settings / container env + Key Vault
Scaling out	Manual (add servers behind a load balancer)	Instance count / autoscale rules
Best when	Existing Windows infrastructure, AD, familiar ops	Elastic scale, repeatable builds, Linux hosting

Neither is "correct" — the choice follows the team. If TicketOps already runs on a managed Windows fleet with Active Directory sign-in, IIS keeps operations boring and known. If the goal is repeatable image builds and elastic scale, a container on Azure earns its keep. The lesson guide listed these targets; the applied point is that *the host you pick dictates the rest of this reading*.

2. A HealthCheck endpoint the platform can probe

Every host above wants a URL it can poll to decide whether an instance is alive and whether it should receive traffic. Expose a lightweight, unauthenticated `/health` endpoint that returns a small JSON

document and an HTTP status the load balancer reads: 200 for healthy, 503 for degraded. Keep it cheap — a health check that runs a heavy query becomes the outage it was meant to detect.

```
// GET /health → 200 healthy, 503 degraded. No secrets, no PII.
{
  "status": "healthy",
  "service": "TicketOps.Console",
  "version": "2.4.1",
  "utc": "2026-07-03T14:22:05Z",
  "checks": {
    "database": "ok",
    "sessionStore": "ok",
    "diskFreePct": 61
  }
}
```

Distinguish **liveness** (the process is up and answering) from **readiness** (its dependencies — database, session store — are reachable). A load balancer that only checks liveness will happily route users to an instance that can no longer reach its database. Return 503 when a required dependency fails so the balancer stops sending new sessions to that node.



Habit worth keeping

Version the health payload and read it in your own release checks. When TicketOps reports "version": "2.4.1" after a deploy, you have proof the new build is actually the one serving traffic — not the previous one still cached on an instance.

3. A diagnostics page for support

The health endpoint is for machines; a diagnostics page is for a human on the support rota at 2 a.m. It answers "what is this instance doing right now?" — build version, environment name, current server time and time zone, active session count, last error timestamp, and the reachability of each dependency. Because it reveals internals, it must be **role-protected** (a support/admin role only) and must **never render secrets** — no connection strings, no keys, no tokens.

Show on the diagnostics page	Never show
Build version and commit/tag	Connection strings
Environment (Prod / Staging)	API keys, secrets, tokens
Server UTC time and time zone	Passwords or password hashes
Dependency reachability (db, store)	Full stack traces to end users
Active session count, uptime	Another user's session data

A well-built diagnostics page turns a vague "TicketOps is slow" ticket into "instance 2 shows the session store unreachable since 02:14" — which is the difference between a guess and a fix.

4. Load balancing and Wisej.NET server-side session state

This is the concept most likely to cause a production incident, so it deserves care. Wisej.NET keeps UI session state **on the server** — the open windows, control state, and the WebSocket connection all belong to the instance that first served the user. If you place several TicketOps instances behind a load balancer with plain round-robin routing, a user's second request can land on an instance that has never seen their session. The result is a broken or reset UI, dropped WebSocket updates, and confused support tickets.

There are two accepted answers:

- **Sticky sessions (session affinity):** the load balancer pins each user to the instance that started their session, usually via an affinity cookie. Simple and the default recommendation for Wisej.NET.
- **Shared session backing store:** externalize session state so any instance can serve any request. More work, but it survives a single instance dying mid-session and enables smoother scaling.



Pitfall: load balancing without affinity

Scaling TicketOps to two instances with round-robin and no sticky sessions and no shared store *will* break server-side session state. The app "works" for one instance and fails intermittently once traffic spreads — the hardest kind of bug to reproduce. Enable affinity (or a shared store) *before* you scale out, not after the tickets arrive.

5. Deployment checklist and rollback plan

A deploy is an operation, not a button. Walk the same list every time so releases stay boring, and decide the rollback path *before* you start — the middle of a failed deploy is the worst place to invent one.

Step	Before deploy	After deploy
Version	Tag the build; confirm the version string	<code>/health</code> reports the new version
Config	Env-specific settings set for the target	No dev/test values leaked into prod
Data	Migrations reviewed and backward-safe	Migration applied; app connects
Affinity	Sticky sessions / shared store confirmed	Multi-instance session test passes
Smoke test	Key TicketOps workflow scripted	Create-and-resolve-a-ticket passes
Rollback	Previous artifact retained; plan written	Rollback trigger and owner named

The **rollback plan** is the part teams skip and regret. Keep the previous published artifact (or image tag) available, write down the exact trigger that means "roll back" — for example, `/health` stays `503` for five minutes, or the smoke test fails — and name who makes the call. A rollback you rehearse is a shrug; a rollback you improvise is an outage.

6. Structured release notes

Release notes are how the next engineer — often future-you — understands what changed and why. Keep them structured and factual rather than a wall of commit messages. A useful shape for a TicketOps release:

```
// TicketOps Console - Release 2.4.1 - 2026-07-03
Added:    Diagnostics page (support/admin role only)
Changed:  /health now returns readiness for db + session store
Fixed:    WebSocket updates dropped after failover on scaled hosts
Ops:      Sticky sessions required - affinity cookie enabled on LB
Migration: 2026_07_ticket_index (backward-safe, additive)
Rollback: redeploy image tag 2.4.0; drops the index only if forced
```

Notice the *Ops* and *Rollback* lines — they carry the operational facts that a changelog of features alone would hide. Anyone reading this a month later knows the deploy required session affinity and how to reverse it, without spelunking through the code.

Decision cheat-sheet

- **Hosting:** known Windows/AD fleet → IIS. Elastic scale, repeatable images, Linux → container/Azure.
- **Health check:** cheap, unauthenticated, versioned; 200 healthy / 503 degraded; readiness covers dependencies.
- **Diagnostics page:** role-protected, secret-free, dependency-aware — built for the on-call human.
- **Scaling out:** Wisej.NET state is server-side → enable sticky sessions or a shared store *before* adding instances.
- **Deploy:** same checklist every time; retain the prior artifact and write the rollback trigger up front.
- **Release notes:** structured — Added / Changed / Fixed / Ops / Migration / Rollback.