

# Startup, Configuration, Session State & Application Lifetime

Go deeper on session state: a session-scoped context wired without static fields, the startup and configuration pipeline, and the static-field trap.

● LearnWisej



## Applied goal

Wire the TicketOps Console startup pipeline so that global configuration is read once, a per-session `SessionContext` is registered with session lifetime and injected, and a diagnostics screen can show — side by side — what is shared by the whole app versus what belongs to just this Work Order operator.

## 1. The startup pipeline: where each thing gets read

TicketOps has three layers of intake at boot, and mixing them up is where production bugs start. `Program` composes the app and registers services. **Configuration sources** — `appsettings.json` for defaults, environment variables for per-deployment overrides (RDS host, integration keys), and the Wisej `Web.config` / `Default.json` for session timeout and upload limits — are read *once per process*. What each session then receives is the *result* of that composition, not a re-read of the files.

| Value                     | Where it lives                                               | Read how often                 |
|---------------------------|--------------------------------------------------------------|--------------------------------|
| RDS connection string     | Environment variable (never in source)                       | Once, at process start         |
| Dispatch API base URL     | <code>appsettings.json</code> , env override per environment | Once, at process start         |
| Idle session timeout      | Wisej config ( <code>Default.json</code> )                   | Once; applied to every session |
| Current operator & tenant | <code>SessionContext</code> (session scope)                  | Per session, on connect        |
| Selected Work Order id    | <code>SessionContext</code> or the Form                      | Per interaction                |

Rule of thumb: if the value is the **same for everyone on this deployment**, it comes from configuration and is read once. If it differs **per connected operator**, it comes from `SessionContext`.

## 2. Registering SessionContext with session lifetime

The lesson guide introduced the `SessionContext` shape. The production move is *how* you hand it out. Register a factory with **session lifetime** so Wisej builds exactly one instance per connected session and disposes it when that session ends. Every Form and service then *asks* for it — it is never fetched from a `static` field.

```
// Program / startup - compose once, resolve per session.
services.AddSingleton<IDispatchApi>(_ => new DispatchApi(config["Dispatch:BaseUrl"]));

// One SessionContext per connected TicketOps operator.
services.AddScoped<SessionContext>(_ => new SessionContext
{
    SessionId      = Application.SessionId,
    CurrentUser    = Application.User?.Identity?.Name,
    Tenant         = ResolveTenant(Application.User),
    Theme          = Application.Theme,
    OpenWorkOrderId = null    // filled as the operator navigates
});
```

Inside a Work Order Form you take a dependency on it rather than reaching outward:

```
public partial class WorkOrderForm : Wisej.Web.Form
{
    private readonly SessionContext _ctx;

    public WorkOrderForm(SessionContext ctx)    // injected, session-scoped
    {
        _ctx = ctx;
        InitializeComponent();
        this.Text = $"Work Order - {_ctx.Tenant} / {_ctx.CurrentUser}";
    }

    private void OnOpen(int workOrderId)
    {
        _ctx.OpenWorkOrderId = workOrderId;    // stays in THIS session only
    }
}
```



#### Production habit

Give shared services (API clients, caches) singleton lifetime and per-operator state session lifetime. When you write `AddScoped<SessionContext>`, you are declaring "one per Work Order operator" — let the container enforce it instead of managing instances by hand.

### 3. Application vs Session vs Request — deciding the scope

The most common design question in TicketOps is simply "where does this variable go?" Answer it with lifetime, not convenience.

| Scope                   | TicketOps example                                              | Why there                                                          |
|-------------------------|----------------------------------------------------------------|--------------------------------------------------------------------|
| <b>Application</b>      | Priority-code lookup table, dispatch API client, feature flags | Identical for everyone and safe to share read-only                 |
| <b>Session</b>          | Current operator, tenant, active Work Order, unsaved filter    | Belongs to one connected instance; must not bleed across operators |
| <b>Request / thread</b> | A single "assign technician" call, a validation result         | Lives only for the round-trip; discarded after                     |

Note what is *not* here: durable business data. The operator's saved Work Order changes belong in the database, not in session state. Session state is the **working set** of one live console — lose it on disconnect and nothing is corrupted.

#### 4. Lifetime, timeout and disposal

A TicketOps session begins on connect and ends when the operator disconnects or the **idle timeout** (configured once, in step 1) elapses. When it ends, the session-scoped `SessionContext` and anything it holds are **disposed**. This is why session state is the right home for transient work: you get automatic cleanup and you never have to reconcile one operator's leftovers with the next connection.

If a `SessionContext` holds something disposable — an open reader, a temp-file handle for an attachment upload — implement `IDisposable` so it is released with the session. Do not let a long-lived singleton hold a reference back into a session object, or you pin it past its timeout and leak memory across the process.



##### Common pitfall

Caching the "current operator" or the open Work Order id in a `static` field to avoid passing `SessionContext` around. It compiles, it demos fine with one operator — and the moment a second dispatcher connects, one console starts showing the other's Work Order. Static means process-wide, which means shared by every session.

#### 5. A diagnostics page that separates global from per-session

Give operations a read-only diagnostics screen with two clearly labelled panels. The **Application** panel shows values that are identical no matter who is logged in; the **This Session** panel shows values resolved from the injected `SessionContext`. Seeing them side by side is the fastest way to catch a scoping mistake — if a "per-session" value looks suspiciously global, something is being read from the wrong place.

| Application panel (global)       | This Session panel (per operator)     |
|----------------------------------|---------------------------------------|
| Build/version, environment name  | <code>SessionContext.SessionId</code> |
| Dispatch API base URL            | Current operator & tenant             |
| Configured idle timeout          | Active theme, open Work Order id      |
| Priority-code table size (cache) | Session start time / idle countdown   |

Populate the right-hand panel by injecting `SessionContext` into the diagnostics Form exactly as the Work Order Form does — never by reading `Application.Session` ad hoc from a helper, which invites the very static-style shortcuts this page exists to expose.

### Decision cheat-sheet

- **Same for the whole deployment?** → Configuration, read once in `Program`, held as a singleton.
- **Different per connected operator?** → `SessionContext`, registered with session lifetime, injected.
- **Only needed for one round-trip?** → A local / request-scoped value; do not persist it.
- **Must survive disconnect?** → It is durable data — put it in the database, not in session state.
- **Tempted to use `static`?** → Only for truly immutable, shared-read data; never for operator, tenant, or the open Work Order.