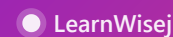


Startup, Configuration, Session State & Application Lifetime

Application vs session vs user vs browser-tab vs request state — and why a session is closer to a desktop app instance than a user account.



Lesson goal

Tell apart the five places state can live in a Wisej.NET app — application, session, user, browser tab and request/thread — and know which one each piece of data belongs in.

In Module 1 you learned that a Wisej.NET control is a live .NET object on the server, mirrored into the browser and kept in sync for you. This module answers the question that model immediately raises: *if my objects stay alive between clicks, exactly where do they live, for how long, and who else can see them?* Getting this right is the difference between an app that behaves correctly for one tester and one that quietly corrupts data the moment a second person logs in.

1. A session is not a user

The single biggest mental shift in this module: **a session is closer to a desktop application instance than to a user account.** One person can have two sessions open (two browser tabs, two devices); a session ends when the connection does. The *user* is who they are; the *session* is one running instance of the app for them.

Here is why the analogy works. When you launch a desktop app twice, you get two independent windows — each with its own open document, its own undo stack, its own in-memory variables. Nothing one instance holds leaks into the other, even though both run the same executable and log in as the same Windows account. A Wisej.NET session is that second window. When a browser connects, the server spins up a fresh set of objects — your `Application.MainPage`, every Form, every field you set — dedicated to that connection. Two tabs from the same person are two instances, just like two launches of the desktop app. The identity of the person is a *separate* fact that you attach to the session; it is not the session itself.



Plain-English definition

A *session* is one running instance of your app for one browser connection — its own private set of server-side objects. A *user* is the identity signed in to that instance. One user, many possible sessions.

2. The five scopes of state

Every piece of data in your app has a natural lifetime, and choosing the wrong scope is the root of most "it worked on my machine" bugs. Walk down this table and, for each thing you are about to store, ask *how long should this survive and who should be able to see it?*

Scope	Lives as long as...	Good for
Application	The whole app process.	Read-only config, caches shared safely across everyone.
Session	One connected instance of the app.	Current user, tenant, theme, selected record, filters.
User	The person's identity (across sessions).	Who they are and what they're allowed to do.
Browser tab	One tab/window.	Per-tab view state when you support multiple tabs.
Request / thread	A single round-trip.	Short-lived work; never store durable state here.

Two of these deserve a second look because they are easy to confuse. **Session** and **user** feel like the same thing until you support multiple tabs or a "remember me" login — then a single user identity clearly spans several independent sessions, and per-session data (the record I am editing right now) must not bleed between them. **Request/thread** state is the most dangerous to misuse: a request is over in milliseconds, so anything you park there vanishes almost immediately. It is for scratch values inside one round-trip — a computed total, a validation result — never for anything the user expects to still be there on the next click.

Wisej.NET gives you concrete handles for the durable scopes. Per-session data belongs in `Application.Session` — a per-session object bag that is isolated so one user's keys never collide with another's — or, better, in a session-scoped object you design yourself, which we build in the second half of this lesson. Application-wide configuration comes from `Application.Configuration`, and genuinely shared, immutable data can live in your own read-only statics. Across a session's life Wisej.NET raises session-start, session-timeout and session-end events, and gives every session a stable `Application.SessionId`, so you always know which instance you are serving.



Don't confuse these two

`Application.Session` is server-side memory for the current session — that's what you want for per-user state. `Application.SessionStorage` is a different thing: it maps to the browser's own `sessionStorage`. When you mean "per-session server state," reach for `Application.Session`.

3. The static-state trap

Static fields are shared by every session. Put a selected ticket ID, the current user, a temporary filter or a pending upload in a `static` field and you've just shared it with every other user on the server.

Why does this happen? A `static` field belongs to the *class*, and there is exactly one copy of the class per server process — not one per session. In WinForms that distinction was invisible: one process *was* one user, so a static "current customer" was effectively per-user by accident. Move that same habit to Wisej.NET, where one process serves hundreds of concurrent sessions, and the static field becomes a shared blackboard everyone scribbles on. The first session writes ticket 42; a second session writes ticket

99; now the first session saves changes to the wrong ticket. Worse, these bugs rarely show up in testing, because a single tester never triggers the collision.

Before — per-user state smuggled into a static field

```
public static class Workspace
{
    // ONE copy for the whole server – shared by every session!
    public static int SelectedTicketId;
    public static string CurrentUser;
}

// Session A sets 42, Session B sets 99, and now A saves to 99.
Workspace.SelectedTicketId = ticket.Id;
```

After — the same values held per session

```
public class SessionContext          // one instance PER session
{
    public int SelectedTicketId { get; set; }
    public string CurrentUser { get; set; }
}

// Read the instance that belongs to THIS session (see §6 below).
var ctx = SessionContext.Current;
ctx.SelectedTicketId = ticket.Id;    // isolated – B can't see A's value
```

The fix is not "never use static." Static is perfectly safe for data that is truly the same for everyone and never mutated per user — a lookup table of country codes, a compiled regex, an immutable config object loaded at startup. The rule is precise: **static is for shared, read-only (or thread-safe) data; per-user or per-workflow state must be session-scoped.**



Common pitfall

Using `static` fields for user-specific or workflow-specific state. It looks fine with one user and corrupts data the moment a second session connects.



Self-check

Which fields would be unsafe if static? What should happen when a browser refreshes? Where does authentication end and session state begin?

Application vs session — a side-by-side

Question	Application scope	Session scope
How many copies exist?	One per server process.	One per connected browser instance.
Who can see it?	Every session.	Only that one session.
When is it disposed?	When the process stops or recycles.	When the session ends or times out.
Safe to mutate freely?	Only if read-only or thread-safe.	Yes — it's private to one session.
Typical contents	Config, lookup tables, shared caches.	Current user, selected record, filters, wizard step.

Key vocabulary

- **Session** — one running instance of the app for a connection.
- **Static field** — process-wide storage shared by all sessions.
- **User context** — identity and permissions for the person.
- **Request/thread state** — short-lived state for a single round-trip.
- **Application storage** — process-wide state shared across all sessions.
- **Session storage** — per-session state private to one connection.
- **Session lifetime** — a DI registration that creates one instance per session.



Lesson goal

Use startup and configuration files intentionally, and design a `SessionContext` object that per-session services and Forms can depend on.

1. Configuration is a deployable contract

Treat configuration — `Default.json` and friends — as a **deployable contract**, not a scratchpad.

Document each setting so a deployer knows what it does.

The word *contract* is deliberate. The person who deploys your app to a customer's server is often not you, months later, and does not read your source. Everything they can safely change lives in configuration; everything they must not change lives in code. A well-kept config file tells them exactly which knobs exist and what turning each one does. A sloppy one — undocumented keys, values duplicated in three places, secrets checked into source control — turns every deployment into a guessing game.

Setting	Controls
Theme	Default look of the app.
Timeout	How long an idle session survives.
Upload limits	Max size / type of uploads.
Health checks	Endpoints that report the app is alive.
Logging	Levels and destinations for diagnostics.
Integration endpoints	URLs/keys for external systems (kept out of source).

A practical division of labour: put **app-wide** settings — theme, timeout, upload limits, logging levels, service endpoints — in configuration, read once at startup, and expose them through a read-only application-scoped object. Put **per-session** choices — the theme *this* user switched to, the tenant they logged into, the filters they set — in session state, never in the config file. The config file describes the app's defaults for everyone; session state records one instance's departures from those defaults.

2. Startup runs once; each session starts fresh

Application startup and application events are where you wire things up — but remember the session lifecycle: each new connection starts its own instance. Use startup to register services with the right **lifetime**, then let each session get its own per-session objects.

It helps to hold three distinct moments in your head. `Program` /startup runs **once** when the process boots — this is where you read configuration, register services and build anything shared.

`Application.Startup` (the session-start event) runs **once per session**, when a new browser connects — this is where you create that session's main page and its per-session objects. And the session-end path runs when the connection closes or the idle timeout expires — this is where you release per-session resources. Confusing "once per process" with "once per session" is exactly how a shared object ends up where a private one belonged, and vice versa.

Moment	Runs	Put here
<code>Program</code> / startup	Once per process.	Read config, register services, build shared/read-only state.
<code>Application.Startup</code> (session start)	Once per session.	Create the main page; initialise per-session objects.
Session end / timeout	Once per session.	Dispose per-session resources; flush anything pending.

Startup: shared registration once, per-session context per connection

```

public static class Program
{
    // Runs ONCE per process – read config, register shared services.
    static void Main()
    {
        var config = ConfigLoader.Load();           // app-wide, read-only
        Services.AddSingleton(config);              // one copy for everyone
        Services.AddSessionScoped<SessionContext>(); // one PER session
    }
}

public class Application : Wisej.Web.ApplicationBase
{
    // Runs ONCE PER SESSION – a fresh instance for each connection.
    protected override void OnSessionStart(EventArgs e)
    {
        var ctx = SessionContext.Current;          // this session's own object
        ctx.SessionId = Application.SessionId;
        this.MainPage = new MainPage();            // this session's UI
    }
}

```

Note the symmetry with Module 1's round-trip model: the objects created in `OnSessionStart` are the very same live objects that later handle clicks and hold state between them. Startup is simply the moment they are born.

3. Design a SessionContext

A `SessionContext` is a small object that holds the per-session values your screens and services keep reaching for — and it can be **injected** instead of read from a static field.

Think of it as the one legitimate home for "the state of this instance." Instead of scattering the current user, tenant and theme across static fields (unsafe) or passing them as parameters through every method (tedious), you collect them into a single object that Wisej.NET creates once per session and hands to whatever asks for it. Screens and services depend on the *type*; the framework supplies the *instance* that belongs to the current session.

```

public class SessionContext
{
    public string SessionId { get; set; }
    public string CurrentUser { get; set; }
    public string Tenant { get; set; }
    public string Theme { get; set; }
    public string ClientProfile { get; set; }
}

// Registered with session lifetime, then injected into
// Forms and services – never stored in a static field.

```

Consuming the context — inject the type, get this session's instance

```

public class TicketService
{
    private readonly SessionContext _ctx;

    // The framework hands us THIS session's context - no statics.
    public TicketService(SessionContext ctx) => _ctx = ctx;

    public Ticket CreateForCurrentUser(string title) => new Ticket
    {
        Title = title,
        Tenant = _ctx.Tenant,           // scoped to this session
        Author = _ctx.CurrentUser,      // safe under concurrent sessions
    };
}

```

Because the service depends on `SessionContext` rather than reaching into a static or reading a control, it can be unit-tested by passing in a hand-made context — no browser, no session, no server required. That is the same "services decide, without a control" test you met in Module 1, now applied to per-session state.



Production habit

Keep per-user state in session-scoped services and keep shared state read-only or thread-safe. If two sessions could fight over a field, it doesn't belong in shared state.

Common mistakes with startup, config and state

Mistake	Why it hurts	Do instead
Holding the current user or selected record in a <code>static</code> field.	Shared across every session — corrupts data under concurrent users.	Store it on a session-scoped <code>SessionContext</code> .
Doing per-session setup in <code>Program.Main</code> .	<code>Main</code> runs once per process, not per connection, so every session shares it.	Do per-session work in the session-start event.
Hardcoding endpoints, secrets or timeouts in code.	Deployers can't change them, and secrets leak into source control.	Read them from configuration; keep secrets out of source.
Parking durable data in request/thread state.	It vanishes when the round-trip ends.	Promote it to session state if it must survive the click.
Building a shared cache you mutate per user.	Application scope means every session sees the writes.	Keep shared caches read-only; put mutable data in session scope.

Companion code

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

↓ [Download SessionContextPattern.cs](#)



Self-check before the lab

Can you name, for each field in your app, whether it is application, session or request scope — and why?

Can you point to the one place per-session state lives (a `SessionContext`), rather than several statics?

Do you know which of your startup code runs once per process versus once per session? If yes to all three, you are ready.