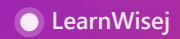


Architecture: EF Core inside a Wisej.NET Application

Lesson Guide · Lab / Exam Guide



Lesson Guide

Where EF Core belongs in a stateful Wisej.NET app: session, UI, request and unit-of-work lifetimes, a DbContextFactory with one context per operation, the Microsoft DI bridge, and the session-safety rules. What this means for a Wisej.NET developer working with real databases — and how to read this module.



Production focus

Where EF Core belongs in a stateful Wisej.NET app: session, UI, request and unit-of-work lifetimes, a DbContextFactory with one context per operation, the Microsoft DI bridge, and the session-safety rules.

1. The production problem

A Wisej.NET application feels like WinForms because forms, pages, controls, `BindingSource` components and event handlers all live on the server, and that familiarity invites a dangerous shortcut: treating a `DbContext` as one more long-lived server object. EF Core is server-side too, but it is not a UI state container; it is a short unit of work whose change tracker is neither thread-safe nor meant to be shared. Many browser sessions will open the Support Desk Data Console at once, so the first skill of this course is keeping the UI lifetimes and the database lifetime apart before any grid is bound.

2. Key concepts

Concept	Why it matters
session state	The per-browser-session server state Wisej.NET keeps alive: open pages, the selected row, filters, validation messages and pending edits, reachable through <code>Application.Session</code> and the live control tree.
UI object lifetime	How long a <code>Page</code> , <code>Form</code> or <code>BindingSource</code> exists, from construction until the user closes it or the session ends; often hours, far longer than any database call.
request/thread lifetime	One browser event or one <code>await</code> continuation on the server; the thread that started a click handler is not necessarily the one that finishes it.
unit of work	The lifetime of one <code>DbContext</code> : create it, query or change entities, call <code>SaveChangesAsync</code> , dispose it; seconds, not sessions.
DbContextFactory	The <code>IDbContextFactory<T></code> service registered by <code>AddDbContextFactory</code> that hands out a fresh, correctly configured context whenever an operation needs one.
Microsoft DI bridge	Registering ASP.NET Core's <code>IServiceProvider</code> with <code>Application.Services</code> so Wisej.NET pages can resolve the same services that startup configured.
cross-session leak	A static or shared mutable object (a context, a change tracker, a bound list, a selected entity) that one user's session can read after another user's session wrote it.

3. Lesson narrative

Four lifetimes on one server

Everything in a Wisej.NET process belongs to one of four lifetimes, and most data-binding bugs come from putting an object in the wrong one. Session state lives as long as the browser session: the user's identity, the open ticket browser, the current search filters. UI object lifetime belongs to a particular `Page` or `Form` and the components it owns, such as the `BindingSource` behind the ticket grid and the selected row; a modal editor exists only while the dialog is open. Request lifetime is one round trip: the browser reports a click, the server runs the handler, and any `await` inside it may resume on a different thread-pool thread, so a handler cannot assume thread affinity. The unit-of-work lifetime is EF Core's: a `DbContext` is created, queries or changes a handful of entities, saves, and is disposed.

They nest badly. A session outlives thousands of requests, a form outlives many database calls, and a context should outlive none of them. A context kept in a session or a form tracks every entity it ever loaded, grows until the session ends, and becomes a stale second copy of the database that the user believes is current. Two async operations on that same context make EF Core throw, because a context is not thread-safe. Those are the failure modes behind the module's first question, why a static or shared `DbContext` is dangerous.

Registering EF Core and bridging the two containers

Wisej.NET has its own services system and can also work with Microsoft's `IServiceProvider`. For EF Core, Microsoft DI is the natural home: `AddDbContextFactory<SupportDeskContext>` (or `AddPooledDbContextFactory` when connection churn matters) registers a factory that knows the provider, the connection string from configuration and the development-only diagnostics. Services that use the factory, such as the `TicketQueryService` the lab resolves from a page, go in the same container. After `builder.Build()`, one line hands Microsoft's provider to Wisej.NET so top-level UI containers can resolve those services, with `Application.Services.GetService<T>()` or through property injection with the `[Inject]` attribute on a `Page` or `Form`.

```
var builder = WebApplication.CreateBuilder(args);
var cs = builder.Configuration.GetConnectionString("SupportDesk")
    ?? throw new InvalidOperationException("Missing connection string 'SupportDesk'.");

builder.Services.AddDbContextFactory<SupportDeskContext>(options =>
{
    options.UseSqlServer(cs, sql => sql.EnableRetryOnFailure());
    if (builder.Environment.IsDevelopment())
        options.EnableDetailedErrors(); // sensitive-data logging: local development only
});
builder.Services.AddTransient<TicketQueryService>();

var app = builder.Build();
Application.Services.AddService<IServiceProvider>(app.Services); // the Wisej.NET bridge
```

Once the bridge is in place there is one rule: register every application service through Microsoft DI and do not mix containers, or you get two objects where you expected one. The connection string comes from configuration, never from a constant on a screen, and `EnableSensitiveDataLogging` is a local-development switch that must never reach production logs. The design-time factory in the data project exists only so the `dotnet ef` tools can build a context without starting the web application.

One context per operation, per editor form, or behind a service

The safest default for Wisej.NET data screens is the factory plus one context per operation. A grid search, a lookup load, a save command and a delete command each create and dispose their own context. That matches EF Core's own guidance for stateful server-side UI frameworks, where components are long-lived and a request-scoped context does not line up with the UI lifetime. In the Support Desk Data Console the shape is a service method: `await using var db = await _factory.CreateDbContextAsync();` followed by `await db.Tickets.CountAsync()`, which is exactly the first async query the lab runs behind a loading flag and an error message, without binding anything yet.

A longer-lived context is acceptable in one place: inside a modal editor, when the user opens a ticket, edits tracked entities, navigates a few small related collections such as its comments, and saves or cancels as one unit. Even then it is private to that editor instance, guarded so no two operations run on it concurrently, and disposed in the form's disposal path, never in a shared service, a static field or an application-wide cache. The third option is the repository or service boundary: pages talk to `TicketQueryService` and `TicketCommandService`, which own the factory, so the UI never sees a context

at all. That boundary grows through the rest of the labs, and it is why the lab resolves a query service, not a context, from the page.

Session safety: what may be shared and what never may

Each browser session has its own UI objects, selected row, validation messages, filters and pending edits; those are session or form concerns. EF Core entities, `DbContext` instances and change trackers must not cross sessions. A static cache may hold immutable reference data, such as the ticket categories once loaded, or compiled query delegates, because nobody mutates them. It must never hold a mutable `DbContext`, a `BindingSource` or a selected entity, because a static is shared by every session on the server and the last writer silently wins.

The same rule applies to background work: a task started with `Application.StartTask` creates its own context inside the operation and updates the UI through Wisej.NET's session-aware mechanisms, rather than borrowing a context a page created. Inside a handler, every EF Core call is awaited before the same context is reused, buttons are disabled while a load or save runs, and a friendly status message replaces the raw exception; the lab's loading flag is the smallest version of this discipline.



Common pitfalls

Registering the `DbContext` as a global shared Wisej.NET service, so every session tracks entities in the same object. Sharing entities or a `BindingSource` across sessions through a static field or cache. Starting parallel EF Core operations on the same context, such as two queries launched without awaiting the first. Keeping a context alive in a page because it seems faster, until its change tracker holds every ticket ever opened. Mixing services between Wisej.NET's registry and Microsoft DI after the bridge is installed.



Data-binding habit

Keep the two lifetimes apart. Wisej.NET forms, `BindingSources` and the selected row live for the session; a `DbContext` lives for one operation. Create it from the factory, query or save, dispose it, and never let it become session state.



Learning objectives

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

- ✓ Explain the difference between Wisej.NET session state, UI object lifetime, request/thread lifetime, and EF Core unit-of-work lifetime.
- ✓ Register EF Core with Microsoft dependency injection and expose the service provider to Wisej.NET.
- ✓ Choose between one `DbContext` per operation, one `DbContext` per editor form, and a repository/service boundary.
- ✓ Avoid shared static `DbContext` instances and accidental cross-session data leaks.

Lab / Exam Guide

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



Hands-on lab

Create the Support Desk Data Console solution with Web, Data, Services and Tests projects, add the EF Core provider packages to the data project, create `SupportDeskContext` with a design-time factory, register `AddDbContextFactory` in ASP.NET Core startup and expose Microsoft's `IServiceProvider` to Wisej.NET, then resolve a query service from a Wisej.NET Page and run a first async count query behind a loading flag and an error message, without binding anything yet.

Starting point

Open Visual Studio and create a new solution named `SupportDesk` with a Wisej.NET web application (`SupportDesk.Web`) plus empty class libraries `SupportDesk.Data` and `SupportDesk.Services` and a test project `SupportDesk.Tests` ; a local SQL Server LocalDB or SQLite database is the only other prerequisite. The web template gives you a `Program.cs` and an empty main `Page` ; no entities, no context and no services exist yet, and the full model arrives in Module 2.

Suggested approach

- ✓ Add the EF Core provider package (SQL Server or SQLite) and the design package to `SupportDesk.Data` , then create `SupportDeskContext` with a minimal `Ticket` entity and `DbSet<Ticket> Tickets` so the count query has something to count.
- ✓ Add `SupportDeskDesignTimeFactory` implementing `IDesignTimeDbContextFactory<SupportDeskContext>` so the `dotnet ef` tools can build the context without starting the web application.
- ✓ In `Program.cs` , read the connection string from configuration, call `AddDbContextFactory<SupportDeskContext>` with development-only diagnostics, register `TicketQueryService` , and after `Build()` call `Application.Services.AddService<IServiceProvider>(app.Services)` .
- ✓ Write `TicketQueryService` in `SupportDesk.Services` with an `IDbContextFactory<SupportDeskContext>` constructor dependency and a `CountTicketsAsync` method that creates, uses and disposes one context.
- ✓ In `TicketBrowserPage` , resolve the service with an `[Inject]` property, then wire `countButton_Click` as an async handler with a `_loading` flag, a disabled button, a status label, a try / catch that shows a friendly message, and a finally block that restores the button.

Required deliverables

- ✓ Solution with Web, Data, Services and Tests projects and the EF Core provider packages
- ✓ `SupportDeskContext` registered through `AddDbContextFactory` with a design-time factory
- ✓ Microsoft `IServiceProvider` exposed to Wisej.NET and a query service resolved from a Page
- ✓ First async count query with a loading guard and an error message

- ✓ No static DbContext anywhere in the solution

Student review questions

- ✓ Which lifetime did you choose for the DbContext behind `CountTicketsAsync`, and what would go wrong if the page kept that context alive between clicks?
- ✓ Two users open the Support Desk console at the same time and both click Count. Which objects are separate per session, which are shared, and why is that split safe?
- ✓ If the database is unreachable, what does the user see, what does the button do, and how does the loading flag end up false again?



Data-binding habit

Keep the two lifetimes apart. Wisej.NET forms, BindingSources and the selected row live for the session; a DbContext lives for one operation. Create it from the factory, query or save, dispose it, and never let it become session state.