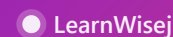


Services, Dependency Injection & Testable UI Patterns

Use Wisej.NET service registration, session-scoped services, property injection, constructor injection for services, and simple presenter / application-service patterns.



Lesson goal

Use Wisej.NET service registration, session-scoped services, property injection, constructor injection for services, and simple presenter / application-service patterns to make UI testable — so a fake service can stand in for the real one with no browser involved.

In Module 1 we settled a habit: **controls display and collect state; services decide what the application means**. In Module 2 we learned that a Wisej.NET app keeps a live **session** of objects per user between clicks. This module joins those two ideas. If a screen is going to hand its real work to a service, two questions follow immediately: *where does that service come from*, and *how long does it live*? Dependency injection (DI) answers the first; service lifetimes answer the second. Getting both right is what makes UI logic something you can actually test.

1. Why DI makes UI testable

Suppose a ticket editor needs to save a ticket. The lazy version creates the service itself:

The Form welds itself to one concrete implementation

```
private void btnSave_Click(object sender, EventArgs e)
{
    var service = new TicketService(new SqlTicketRepository(_cs)); // hard-wired
    service.Save(ReadTicketFromForm());
}
```

This screen can now *only* run with a real database behind it. To test the save logic you would need a browser, a live session, a SQL connection and seed data — for what is really a one-line decision. The dependency is invisible and un-swappable because it is created inside the handler.

The fix is to depend on an **interface** and let something else supply the implementation. The Form no longer knows or cares which `ITicketService` it holds:

Depend on the interface; the concrete type is supplied from outside

```
// The contract – everything the UI needs, nothing about SQL
public interface ITicketService
{
    SaveResult Save(Ticket ticket);
}

// The Form holds the interface, not a concrete class
public partial class TicketEditor : Wisej.Web.Page
{
    private ITicketService _tickets;    // supplied by DI, see §3
}
```

Now a test can hand the presenter (§5) a **fake** `ITicketService` that records what it was asked to save and returns a canned result — no database, no browser, no session. That substitution is the whole point of DI: *the thing under test names what it needs by interface, and the test decides what to plug in*. When your dependencies are visible and replaceable, DI is doing its job. When you add registration ceremony but still **new** up the real class inside the handler, you have paid the cost and kept none of the benefit.

2. Registering services and choosing a lifetime

Wisej.NET exposes an application-level service container (reached through `Application.Services` / an `IServiceProvider`). You register a contract, its implementation and a **lifetime** once at startup, and from then on the framework resolves it for you. The lifetime is the decision that most often goes wrong, so treat it deliberately.

Registering three services, each with the lifetime its state demands

```
// Typically in Program/Startup where the container is configured.
services.AddSingleton<IExchangeRateCache, ExchangeRateCache>(); // app-wide, read-mostly
services.AddScoped<ICurrentUser, SessionCurrentUser>();         // per session (per user)
services.AddTransient<ITicketService, TicketService>();         // fresh each resolve
```

The exact registration API name can vary by Wisej.NET version — describe the intent (contract → implementation → lifetime) and match your project's convention.

Lifetime	One instance per...	Use it for	Never put here
Application / Singleton (Shared)	The whole server process — shared by <i>all</i> users.	Stateless helpers, caches of shared reference data, config.	Anything specific to one user. Must be thread-safe.
Session-scoped	One user's session (Module 2).	The current user, their permissions, a selected-ticket context.	State that must outlive a single user's visit.
Transient	Every request/resolve — a brand-new object each time.	Short-lived, lightweight command/workflow services.	Anything expensive to build on every call.

The single most dangerous mistake is a **longer-lived service capturing per-session state**. A singleton lives once for the whole server; if it stores "the currently selected ticket" or "the logged-in user", every user on the server shares that one field — user B suddenly sees user A's selection. That is the same trap as a **static** field from Module 1, just wearing a DI costume. Per-user context belongs in a **session-scoped** service, whose instance is tied to the one session that owns it.



The lifetime question

For every service ask: "If two users hit this at once, is sharing one instance correct?" If yes and it holds no per-user state → singleton. If it holds per-user state → session-scoped. If it is a cheap one-shot worker → transient.

3. Two ways to get a service in: constructor vs. property injection

How a class receives its dependencies depends on *who creates the class*.

Constructor injection is the default for ordinary services — classes you (or the container) create. The dependency is a required constructor parameter, so an instance simply cannot exist without it. This is the most honest form: the constructor signature is a complete list of what the class needs.

A service asks for its collaborators in its constructor

```
public class TicketService : ITicketService
{
    private readonly ITicketRepository _repo;
    private readonly ILog _log;

    // Everything this class needs is visible right here.
    public TicketService(ITicketRepository repo, ILog log)
    {
        _repo = repo;
        _log = log;
    }

    public SaveResult Save(Ticket ticket)
    {
        if (string.IsNullOrEmpty(ticket.Title))
            return SaveResult.Invalid("Title is required.");

        _repo.Insert(ticket);
        _log.Info($"Saved ticket {ticket.Id}");
        return SaveResult.Ok(ticket.Id);
    }
}
```

Property injection is used when *the framework* constructs the object and you do not own the constructor — which is exactly the case for Wisej.NET **Form**, **Page** and **UserControl** instances. Wisej.NET creates these as part of building the UI, so you cannot pass a service through their constructor. Instead you mark a property (or field) so the framework fills it in after construction — commonly with an inject attribute.

A framework-created Page receives its service by property injection

```

public partial class TicketEditor : Wisej.Web.Page
{
    // The framework creates this Page, then resolves and assigns the service.
    [Inject]
    public ITicketService Tickets { get; set; }

    private TicketPresenter _presenter;

    protected override void OnLoad(EventArgs e)
    {
        base.OnLoad(e);
        // Hand the injected service to a plain, testable presenter.
        _presenter = new TicketPresenter(Tickets);
    }

    private void btnSave_Click(object sender, EventArgs e)
    {
        var outcome = _presenter.Save(ReadTicketFromForm()); // UI → presenter → service
        ShowResult(outcome);                                // result → UI
    }
}

```

The attribute name shown is illustrative; confirm the exact injection attribute your Wisej.NET version uses. An `IServiceProvider` resolve is a valid alternative when an attribute doesn't fit.

If neither injection style fits a spot, you can always resolve a service explicitly through the provider (`Application.Services.GetService<ITicketService>()` or the equivalent). Prefer constructor injection where you own the constructor, property injection for framework-created UI, and explicit resolution only as a fallback.

4. Keep the container out of your logic

A subtle rule: services should receive their dependencies, not go hunting for them. A `TicketService` that calls `Application.Services.GetService<...>()` in the middle of a method has hidden a dependency the constructor never advertised — and made itself hard to test again, because the test can no longer choose what it gets. Ask for collaborators up front (constructor); let the container do the wiring in one place at startup.

5. A tiny presenter you can unit-test

The Form is created by Wisej.NET and lives in a session, which makes it awkward to instantiate in a test. So push the decision-shaped code one step away, into a **presenter** (an application-service pattern): a plain C# class that takes the service by constructor and exposes methods the Form calls. The Form becomes a thin translator — read controls, call the presenter, show the result — and the presenter becomes ordinary, browser-free, unit-testable code.

The presenter: plain class, constructor-injected, no Wisej.NET types

```

public class TicketPresenter
{
    private readonly ITicketService _tickets;

    public TicketPresenter(ITicketService tickets) => _tickets = tickets;

    // Pure decision logic – no controls, no session, no browser.
    public SaveOutcome Save(Ticket ticket)
    {
        var result = _tickets.Save(ticket);
        return result.IsValid
            ? new SaveOutcome(true, $"Saved #{result.Id}.")
            : new SaveOutcome(false, result.Error);
    }
}

```

A unit test replaces the real service with a fake — no browser, no database

```

// A hand-written fake standing in for the real ITicketService.
class FakeTicketService : ITicketService
{
    public Ticket LastSaved;
    public SaveResult Save(Ticket t) { LastSaved = t; return SaveResult.Ok(42); }
}

[Test]
public void Save_reports_success_and_forwards_the_ticket()
{
    var fake = new FakeTicketService();
    var presenter = new TicketPresenter(fake); // inject the fake by constructor

    var outcome = presenter.Save(new Ticket { Title = "Printer down" });

    Assert.IsTrue(outcome.Ok);
    Assert.AreEqual("Printer down", fake.LastSaved.Title); // verify the interaction
}

```

Nothing in that test starts a server, opens a browser or touches SQL. Because `TicketEditor` depended on the `ITicketService` interface and delegated to a plain presenter, the exact same logic the user exercises can run in milliseconds inside a test runner. That is DI paying off: the seam you designed in §1 is the seam the test plugs into here.



Learning objectives

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

- ✓ Register shared/singleton, session-scoped, and transient services with the lifetime their state demands.

- ✓ Inject services by constructor into plain classes and by property into framework-created Forms and Pages.
- ✓ Move UI decisions into a presenter/application-service that runs without a browser.
- ✓ Replace a real service with a fake behind the same interface to unit-test UI logic.
- ✓ Explain why a longer-lived service must never capture per-session state.

6. Common mistakes

Mistake	Why it hurts	Do instead
<code>new</code> -ing a service inside a handler instead of resolving it.	Welds the screen to one implementation; it can only run with the real dependency, so it can't be tested.	Depend on the interface and let DI supply it (constructor or property injection).
A singleton/shared service holding per-user data (selected ticket, current user).	One instance is shared by all users, so they clobber each other's state — a <code>static</code> field in disguise.	Make it session-scoped so each session gets its own instance (Module 2).
UI logic welded to controls — rules read <code>txtTitle.Text</code> directly.	The logic can't run without a live control, so it's untestable and stuck in one screen.	Read the controls into data, then pass that data to a presenter/service.
A service that calls the container mid-method to fetch what it needs.	Hides a dependency the constructor never listed; the test can't substitute it.	Ask for every collaborator in the constructor; wire once at startup.
Registering everything as transient (or everything as singleton).	Loses the meaning of lifetime — either rebuilds costly state constantly, or shares state that must not be shared.	Choose per service by asking whether sharing one instance is correct.

Common pitfall to avoid



Common pitfall

Using DI ceremony without making dependencies easier to replace or test — registering interfaces at startup but still `new`-ing the concrete class inside the handler. The container is not the goal; the replaceable seam is.

Key vocabulary

- **Dependency injection (DI)** — a class names what it needs by interface and is handed an implementation from outside, rather than creating one itself.
- **Service container** / `IServiceProvider` — the registry that maps a contract to its implementation and lifetime, and resolves instances on demand.
- **Service lifetime** — how long a resolved instance lives: application/singleton (shared), session-scoped (per user), or transient (per resolve).

- **Session-scoped service** — one instance per user session; the correct home for per-user context (Module 2).
- **Constructor injection** — dependencies supplied as constructor parameters; used for classes you or the container create.
- **Property injection** — a dependency assigned to a marked property after construction; used for framework-created Forms/Pages you don't build yourself.
- **Presenter / application service** — a plain, browser-free class that holds the decision logic a Form delegates to, making that logic unit-testable.
- **Fake / test double** — a stand-in implementation of an interface used in a test in place of the real service.

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 — it lives in a presenter or service.
- ✓ Services are resolved through DI, never `new` -ed inside a handler.
- ✓ No per-user state is held in a shared/singleton service or a `static` field.
- ✓ 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

Which services must be session-scoped, and why? What breaks if a shared/singleton service stores a selected ticket ID? For each screen, could you unit-test its decision by handing a presenter a fake service — with no browser? If yes to all three, you are ready.

Companion code

`DependencyInjectionPatterns.cs` collects the patterns from this module in one compilable file.

Download it and read it alongside the lab.

↓ [Download DependencyInjectionPatterns.cs](#)