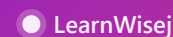


Services, Dependency Injection & Testable UI Patterns

Go deeper on services and DI: clean interfaces with fake vs production profiles, choosing service lifetimes, and a presenter that makes a screen testable.



Applied goal

Design clean service interfaces for the TicketOps Console, pair each with a fake and a production implementation, choose the right lifetime, inject them into Forms and UserControls the Wisej.NET way, and move one Work Order screen's decisions into a testable presenter.

1. Interfaces first: the TicketOps service surface

An interface is a contract the UI depends on without knowing who fulfils it. In TicketOps the Work Order screens never touch SQL or HTTP directly; they talk to a small set of role-focused interfaces, and each interface owns one slice of the domain.

Interface	Responsibility	Typical members
<code>ITicketService</code>	Load, create, and transition work orders	<code>GetOpenTickets()</code> , <code>Assign(id, userId)</code> , <code>Close(id, reason)</code>
<code>IUserService</code>	Current operator and roster lookups	<code>Current</code> , <code>FindTechnicians()</code>
<code>IPermissionService</code>	Answers "may this user do X?"	<code>CanClose(ticket)</code> , <code>CanReassign(ticket)</code>
<code>INotificationService</code>	Tells operators something happened	<code>Notify(userId, message)</code>
<code>IAuditLog</code>	Records who did what, when	<code>Record(action, ticketId, userId)</code>

Keep interfaces narrow. A screen that only closes tickets should depend on `ITicketService` , `IPermissionService` , and `IAuditLog` — not on a single god-service that exposes everything.

2. Fake vs. production implementations

Each interface gets two concrete classes. The *fake* holds in-memory data and returns predictable results, so you can build and demo screens before any backend exists. The *production* class talks to the real database or API. Because both satisfy the same contract, the Work Order form cannot tell them apart.

```
// Fake: predictable, no I/O, ideal for building and testing the UI
public class FakeTicketService : ITicketService
{
    private readonly List<Ticket> _tickets = SeedData.WorkOrders();

    public IReadOnlyList<Ticket> GetOpenTickets() =>
        _tickets.Where(t => t.Status == TicketStatus.Open).ToList();

    public void Close(int id, string reason)
    {
        var t = _tickets.First(x => x.Id == id);
        t.Status = TicketStatus.Closed;
        t.CloseReason = reason;
    }
}

// Production: same contract, real persistence
public class SqlTicketService : ITicketService
{
    private readonly TicketOpsDb _db;
    public SqlTicketService(TicketOpsDb db) => _db = db;

    public IReadOnlyList<Ticket> GetOpenTickets() =>
        _db.Tickets.Where(t => t.Status == TicketStatus.Open).ToList();

    public void Close(int id, string reason)
    {
        var t = _db.Tickets.Find(id);
        t.Status = TicketStatus.Closed;
        t.CloseReason = reason;
        _db.SaveChanges();
    }
}
```



Production habit

Write the fake and the production class in the same sitting, from the same interface. If a member is awkward to fake, it is usually a sign the contract is doing too much — split it before the UI depends on it.

3. Swapping registration profiles

The only place that knows *which* implementation is live is the registration method. Group registrations into a fake profile and a production profile, then pick one at startup. The screens are identical in both modes.

```
// One method per profile; called from Application startup
static void RegisterFakeServices()
{
    Application.Services.AddShared<IPermissionService, RolePermissionService>();
    Application.Services.AddShared<IAuditLog, ConsoleAuditLog>();
    Application.Services.AddSession<IUserService, FakeUserService>();
    Application.Services.AddSession<ITicketService, FakeTicketService>();
    Application.Services.AddSession<INotificationService, InMemoryNotifier>();
}

static void RegisterProductionServices()
{
    Application.Services.AddShared<IPermissionService, RolePermissionService>();
    Application.Services.AddShared<IAuditLog, SqlAuditLog>();
    Application.Services.AddSession<IUserService, SessionUserService>();
    Application.Services.AddSession<ITicketService, SqlTicketService>();
    Application.Services.AddSession<INotificationService, EmailNotifier>();
}
```

Because the profiles register the *same interfaces*, no screen, presenter, or workflow changes when you flip from fake to production. That single switch is what makes the fake-first habit from the Lesson Guide pay off.

4. Choosing a lifetime

Lifetime decides how many instances exist and how long they live. Getting it wrong is the most common DI defect: shared state leaking between users, or per-user state trapped in a global object.

Lifetime	Instances	Use when...	TicketOps example
Shared (application)	One, app-wide	Stateless or global, and thread-safe	<code>IPermissionService</code> , <code>IAuditLog</code>
Session	One per user session	Holds user-specific context	<code>IUserService</code> , <code>ITicketService</code>
Transient	New every resolve	Cheap, short-lived, no shared state	A per-request report builder



Common pitfall

Never store session data — the current operator, a selected work order, an open ticket filter — in a **shared** service. One user's selection would bleed into every other session, and the state is not thread-safe. Anything user-specific belongs in a *session* service.

5. Injecting services into Forms and UserControls

Top-level containers — a `Page` , a `Form` , or a docked `UserControl` — receive services through the `Inject` attribute on a property. Wisej.NET resolves them from `Application.Services` when the

container is created. You do not `new` the services yourself.

```
public partial class WorkOrderForm : Wisej.Web.Page
{
    [Inject] public ITicketService Tickets { get; set; }
    [Inject] public IPermissionService Permissions { get; set; }
    [Inject] public IAuditLog Audit { get; set; }

    // Event handler stays thin: validate UI state, then delegate
    private void closeButton_Click(object sender, EventArgs e)
    {
        var ticket = (Ticket)grid.CurrentRow.Tag;
        if (!Permissions.CanClose(ticket)) { ShowDenied(); return; }

        Tickets.Close(ticket.Id, reasonBox.Text);
        Audit.Record("close", ticket.Id, /* userId */ 0);
        RefreshGrid();
    }
}
```

Plain service classes that are themselves resolved from the container prefer *constructor injection* instead — it makes the dependency mandatory and visible. Reserve property `[Inject]` for the visual containers Wisej.NET instantiates for you.

6. A presenter that makes the screen testable

The event handler above still mixes a UI decision (which row is selected) with a domain decision (may this ticket be closed, and what gets logged). Move the domain decisions into a presenter that talks to services and exposes a plain result — no controls involved.

```

public class WorkOrderPresenter
{
    private readonly ITicketService _tickets;
    private readonly IPermissionService _permissions;
    private readonly IAuditLog _audit;

    public WorkOrderPresenter(ITicketService tickets,
        IPermissionService permissions, IAuditLog audit)
    {
        _tickets = tickets; _permissions = permissions; _audit = audit;
    }

    // Pure workflow: no Form, no grid, fully unit-testable
    public CloseResult CloseTicket(int ticketId, string reason, int userId)
    {
        var ticket = _tickets.GetOpenTickets().FirstOrDefault(t => t.Id == ticketId);
        if (ticket == null) return CloseResult.NotFound;
        if (!_permissions.CanClose(ticket)) return CloseResult.Denied;

        _tickets.Close(ticketId, reason);
        _audit.Record("close", ticketId, userId);
        return CloseResult.Ok;
    }
}

```

Now a test can construct `WorkOrderPresenter` with the three *fake* services, call `CloseTicket`, and assert on the returned `CloseResult` — without ever instantiating a `Page`, a grid, or the Wisej.NET runtime. The Form shrinks to reading the selected row and rendering whatever result the presenter hands back.



Where to draw the line

If a rule can change the outcome — permission checks, status transitions, what gets audited — it belongs in the presenter or a service. If it only affects appearance — enabling a button, showing a message — it can stay in the Form.

Decision cheat-sheet

- ✓ **Interface** — a narrow contract the UI depends on; one domain slice each.
- ✓ **Fake implementation** — in-memory, predictable; build and test screens without a backend.
- ✓ **Production implementation** — same contract, real persistence; swapped in via the registration profile.
- ✓ **Shared lifetime** — one thread-safe, stateless instance app-wide (permissions, audit).
- ✓ **Session lifetime** — one instance per user, holds user context (current operator, tickets).
- ✓ **Transient lifetime** — a fresh instance each resolve; no shared state.
- ✓ **[Inject] property** — how Forms and UserControls receive services from `Application.Services`.
- ✓ **Presenter** — a control-free class holding the workflow decisions, testable with fakes alone.