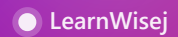


Real-Time Apps with Server Push — Lesson Book

The complete set of module readings for this course, in one document.



Contents

01	Real-Time Web Architecture in Wisej.NET	Module 1 · Lesson Guide
02	Real-Time Web Architecture in Wisej.NET	Module 1 · Lab / Exam Guide
03	Application Context, Sessions, and Lifecycle	Module 2 · Lesson Guide
04	Application Context, Sessions, and Lifecycle	Module 2 · Lab / Exam Guide
05	Single-Session Server Push with Background Tasks	Module 3 · Lesson Guide
06	Single-Session Server Push with Background Tasks	Module 3 · Lab / Exam Guide
07	Timers, Polling Fallbacks, and Update Cadence	Module 4 · Lesson Guide
08	Timers, Polling Fallbacks, and Update Cadence	Module 4 · Lab / Exam Guide
09	Live Data Binding and Real-Time Grids	Module 5 · Lesson Guide
10	Live Data Binding and Real-Time Grids	Module 5 · Lab / Exam Guide
11	Multi-User Push with Services and Event Hubs	Module 6 · Lesson Guide
12	Multi-User Push with Services and Event Hubs	Module 6 · Lab / Exam Guide
13	Production Hardening, Deployment, and Capstone	Module 7 · Lesson Guide
14	Production Hardening, Deployment, and Capstone	Module 7 · Lab / Exam Guide

Real-Time Web Architecture in Wisej.NET

Server-driven real-time architecture: WebSocket push, in-request updates, and polling fallback in the server-side Wisej.NET model. The concept, the mental model, and the production pitfalls — read this before the lab.



Production focus

Server-driven real-time architecture: WebSocket push, in-request updates, and polling fallback in the server-side Wisej.NET model.

Why this module exists

Real-time web applications are usually associated with chat, games, stock tickers, monitoring screens, and dashboards. In line-of-business systems the same requirement appears in less spectacular but more valuable forms: a supervisor sees new tickets as they arrive, a production operator sees a machine stop, a dispatcher sees that a job was reassigned, or a billing clerk sees that an import is still running without pressing Refresh.

Wisej.NET is built around a server-side application model. The developer works in C# or VB.NET, uses controls, events, forms, pages, timers, data binding, and the Visual Studio designer, while Wisej.NET maintains the browser-side widget tree and synchronizes it with the server-side state. This model is especially important for server push because the server already owns the application state. A background task can change a label, add a grid row, update a progress bar, or open a notification, and Wisej.NET can deliver the pending UI changes to the browser when a live WebSocket connection exists.

A beginner course usually teaches how to create pages, windows, controls, menus, and basic event handlers. This intermediate course assumes those skills and focuses on the production design questions that appear when the UI updates itself. The central question is not "Can I update a label from a thread?" The central question is "How do I update the right user's session, at the right rate, from the right service, without losing state, leaking memory, blocking the server, or overwhelming the browser?"

What real-time means in this course

For this course, a Wisej.NET real-time app is an application that can receive events from the user with low delay and can also update the user's browser from the server without waiting for another button click. Server push is the important part. The application should be able to change the visible UI because something happened on the server: a background job advanced, a queue changed, a message arrived, or another user performed an action.

Do not reduce real time to "fast refresh." A grid that refreshes every 30 seconds is sometimes acceptable, but it is not the same as push. Push is event-driven. The application updates because the model changed. Polling asks repeatedly whether something changed. Both can be useful, but they have different cost, latency, and failure behavior.

Wisej.NET's real-time mental model

A useful mental model is this:

1. The browser hosts Wisej.NET JavaScript widgets.
2. The server hosts the Wisej.NET application session and server-side controls.
3. User actions travel from browser to server as events.
4. Server-side code changes controls and application state.
5. Wisej.NET sends the changes back to the browser as compact updates.
6. With WebSocket, a background task can push pending updates without waiting for the next browser request.

This is not screen scraping and it is not a remote desktop. The browser still renders native web widgets. The server does not stream pixels. The important architectural point is that the application logic and the authoritative UI state live on the server. When a Wisej.NET control's property changes, the browser-side widget can be synchronized.

WebSocket push vs. request-driven updates

A normal web request starts when the browser contacts the server and ends when the server responds. If there is no open server-to-browser channel, the server cannot spontaneously contact the browser. It can only send changes when the browser makes another request. WebSocket changes that. With a WebSocket connection, Wisej.NET can push UI updates to the connected client from server-side code.

In Wisej.NET, a background task can update UI controls and call `Application.Update()` to push the pending changes when WebSocket is available. If WebSocket is not available, `Application.Update()` cannot push because there is no live connection. The application then needs a fallback approach such as Wisej.NET polling or timer-driven requests. The fallback does not invalidate the app. It simply changes the delivery mechanism.

Three update mechanisms you must distinguish

1. In-request updates

This is the simplest case. The user clicks a button, an event handler runs, the handler changes controls, and Wisej.NET sends changes back with the response. You usually do not call `Application.Update()` here because the update is naturally returned to the browser at the end of the request.

```
private void buttonRefresh_Click(object sender, EventArgs e)
{
    statusLabel.Text = "Refreshed at " + DateTime.Now.ToLongTimeString();
}
```

2. Out-of-bound WebSocket push

This is the main subject of the course. A background task runs outside the original browser event. It changes the server-side UI and calls `Application.Update()` to push changes through the WebSocket connection.

```
private void buttonStart_Click(object sender, EventArgs e)
{
    buttonStart.Enabled = false;
    progressBar.Value = 0;

    Application.StartTask(() =>
    {
        for (int i = 1; i ≤ 100; i++)
        {
            progressBar.Value = i;
            statusLabel.Text = $"Processing {i}%";
            Application.Update(this);
            Thread.Sleep(100);
        }

        buttonStart.Enabled = true;
        statusLabel.Text = "Done";
        Application.Update(this);
    });
}
```

3. Polling or timer-driven refresh

When a browser or server cannot use WebSocket, the server cannot push out-of-band changes. In that case, the application can use polling so that the browser periodically contacts the server and receives pending changes. A `Wisej.Web.Timer` can also be used to fire server-side events at a defined interval.

```
private void MainPage_Load(object sender, EventArgs e)
{
    // Ignored when WebSocket is available. Useful as fallback when it is not.
    Application.StartPolling(1000);
}
```

The course teaches how to design for WebSocket first, then degrade predictably.

The course project: TicketOps Live

All modules build pieces of the same application, **TicketOps Live**. It is a real-time support operations console with these features:

- a live system status strip;
- a server job progress monitor;
- a ticket board that updates when tickets are created or changed;
- a notification panel for multi-user events;
- an operations dashboard with simple server health indicators;
- a production checklist for deployment behind IIS or a load balancer.

The project is deliberately simple in business terms. The purpose is to learn real-time architecture, not to implement a full helpdesk system. Every lab should leave the student with code that can be reused in real

applications.

Design rule: update models first, UI second

It is tempting to let background tasks directly manipulate every control that needs to change. That works for small demos, but real applications need a separation between data and presentation. The background task should usually update a model or raise a domain event, and the session-specific UI should decide how to display it.

For a single-user progress window, direct control updates are acceptable. For multi-user dashboards, direct global access to controls is wrong. Controls belong to sessions. A global service should not store random form instances forever. Instead, sessions subscribe to a service, receive events, and update their own controls in their own application context.

A safe vocabulary for this course

Session means one running Wisej.NET application instance for one browser tab/window context. It is not the same as a logged-in user. One user may have multiple sessions.

Application context means the Wisej.NET context needed by code to access session-specific static `Application` properties and update the correct browser.

In-bound code means code running as part of a browser request or Wisej.NET event.

Out-of-bound code means code running outside a browser event, such as a background thread, a hosted service event, a queue callback, or a timer callback.

Push means the server delivers pending UI updates to the browser over a live connection.

Polling means the browser periodically asks the server for updates.



Common mistakes

The most common mistake is updating a control from a background thread without preserving the right application context. The second most common mistake is calling push too frequently. A progress bar does not need 500 updates per second. Push is powerful, but every update has cost: serialization, network traffic, browser work, and server scheduling.

Another common mistake is to use static fields for session-specific data. Static fields are shared by all users and all sessions in the application domain. Use `Application.Session`, a per-session service, or `Application.GetInstance()` for session-specific objects. Use static services only for truly shared data, and then design concurrency carefully.



Checkpoint

At the end of this module, the student should be able to draw the end-to-end path of a server-generated ticket event: service changes data, service raises an event, a session receives it, the session updates its controls, and Wisej.NET pushes the resulting UI changes when WebSocket is available.

Real-Time Web Architecture in Wisej.NET

What you'll build in the hands-on lab, the tasks, the suggested implementation, and the acceptance criteria.



Hands-on lab

Create the first screen of TicketOps Live: a page with a live status strip, a clock label, a progress indicator, and a simulated server heartbeat that updates the browser without a user click.

Starting point

Create a Wisej.NET Web Page Application named `TicketOpsLive`. Add a main page named `MainPage`. Use any built-in theme. The lab assumes C#, but the same concepts apply to VB.NET.

Tasks

1. Add a top panel named `statusPanel`.
2. Add labels named `clockLabel`, `connectionLabel`, and `activityLabel`.
3. Add a progress bar named `serverLoadBar`.
4. Add buttons named `startButton` and `stopButton`.
5. Implement a simulated heartbeat background task when `startButton` is clicked.
6. The heartbeat should update the clock, a fake load value, and an activity message once per second.
7. Stop updates when `stopButton` is clicked or when the page is disposed.

Suggested implementation

```
private volatile bool _running;

private void startButton_Click(object sender, EventArgs e)
{
    if (_running)
        return;

    _running = true;
    startButton.Enabled = false;
    stopButton.Enabled = true;
    connectionLabel.Text = "Server heartbeat running";

    Application.StartTask(() =>
    {
        var random = new Random();
        var counter = 0;

        while (_running && !this.IsDisposed)
        {
            counter++;
        }
    });
}
```

```

        clockLabel.Text = DateTime.Now.ToString("HH:mm:ss");
        serverLoadBar.Value = random.Next(5, 95);
        activityLabel.Text = $"Heartbeat #{counter}";

        Application.Update(this);
        Thread.Sleep(1000);
    }
});
}

private void stopButton_Click(object sender, EventArgs e)
{
    _running = false;
    startButton.Enabled = true;
    stopButton.Enabled = false;
    connectionLabel.Text = "Server heartbeat stopped";
}

```

Acceptance criteria

- ✓ The UI updates once per second after the student clicks Start.
- ✓ The browser displays changes without additional clicks.
- ✓ Stop disables the loop within one second.
- ✓ Starting twice does not create two competing loops.
- ✓ The code checks whether the page has been disposed.

Reflection questions

1. Why does this lab call `Application.Update(this)` inside the loop?
2. What changes if WebSocket is not available?
3. Why is `_running` not a static field?
4. What would be the risk of updating every 10 milliseconds?

Application Context, Sessions, and Lifecycle

Sessions, application context and lifecycle: own the right user's UI, avoid the static-field trap, and clean up on exit and timeout. The concept, the mental model, and the production pitfalls — read this before the lab.



Production focus

Sessions, application context and lifecycle: own the right user's UI, avoid the static-field trap, and clean up on exit and timeout.

Why sessions matter more in real-time apps

In a traditional request/response page, a user clicks, the server responds, and most server-side work ends. In a real-time Wisej.NET application, work may continue after the original request. A background job may run for minutes. A dashboard may subscribe to a global event stream. A timer may keep updating a chart. Because work outlives the click that started it, the session lifecycle becomes part of the application architecture.

A Wisej.NET session is similar to a running desktop application instance. It has controls, windows, pages, component state, and application-level objects. But unlike a desktop application, the browser and server can disconnect, the browser may refresh, and there is no permanent dedicated UI thread. Wisej.NET restores the correct session state for each request and can maintain server-side state for both HTTP and WebSocket connections.

Session is not user

A logged-in user and a session are related but not identical. A single user can open two browser tabs and create two sessions. A user can authenticate inside a session, sign out, and leave the session alive. An anonymous user can have a session before login. A background service can exist without belonging to any user session.

This distinction matters for push. When a ticket changes, the application may need to notify all sessions for a tenant, one session for one operator, or every active dashboard. Do not assume "user" and "browser" are the same thing.

Application context

Wisej.NET's `Application` class is static, but its properties and operations are isolated by session when code is running in the correct context. When code runs because the browser raised an event, the context is already available. When code runs in an out-of-bound thread, it may not know which session it should update.

`Application.StartTask()` is the simplest way to start background work from a session because it starts the task with the calling session context. When code is already out-of-bound, you can capture `Application.Current` while you are in context, then later use `Application.Update(context, action)` or `Application.RunInContext(context, action)` to execute code against the correct session.

```
private IWISEJComponent _context;

private void MainPage_Load(object sender, EventArgs e)
{
    _context = Application.Current;
}

private void OnExternalEvent(object sender, TicketChangedEventArgs e)
{
    Application.Update(_context, () =>
    {
        notificationsList.Items.Add(e.Message);
    });
}
```

The context is not a global license to keep controls alive forever. It should be used with a clear subscription and cleanup strategy.

Where state should live

There are four common places for state in a Wisej.NET real-time application.

Control state is state stored directly in controls: a label text, selected grid row, visible window, current tab, or progress bar value. It is easy and often correct for presentation state.

Application.Session stores session-specific values that should not be visible to other sessions. Use it for lightweight session metadata, selected tenant, user preferences, or per-session objects when a formal service is not necessary.

Session services are objects created per session. They are useful when you need a more testable and structured design than ad-hoc session fields. Wisej.NET's session-aware patterns can help avoid accidental static sharing.

Global services are shared by all sessions. They are appropriate for queues, event hubs, cache, background watchers, or system-wide status. They must be thread-safe and must not store session controls without a defined subscription lifecycle.

The static field trap

Static fields are shared. If you store `CurrentTicketId` in a static field, every user can overwrite every other user's current ticket. If you store a `MainPage` reference in a static list and never remove it, the server may keep sessions alive after users leave. If you store a `CancellationTokenSource` in a static field for a per-user job, one user may cancel another user's job.

Use static fields for truly global services only. For session-specific values, use controls, `Application.Session`, or a per-session object.

Refresh, keep-alive, and timeout

A Wisej.NET application must deal with web realities: users refresh the browser, laptops sleep, networks disappear, and tabs remain open overnight. Wisej.NET uses session timeout and keep-alive behavior to

decide whether a session should remain alive or be terminated. When a session times out, it is removed from memory. That is closer to terminating a desktop application than hiding a web page.

For real-time apps, the practical implications are:

- Long-running background work should not assume the UI still exists.
- UI update code should check whether a page/control has been disposed.
- Global event subscriptions should be removed when the session exits or the view is disposed.
- If a job must continue after a session ends, it should be owned by a global service, not by the page.
- If a job is only meaningful for a single user's UI, it should be cancelled when the UI disappears.

Cleanup patterns

A real-time view that subscribes to a global service should always unsubscribe. The exact event used depends on your view structure, but the pattern is the same: subscribe on load or construction, unsubscribe when disposed or when the application exits.

```
private readonly TicketFeed _feed = TicketFeed.Instance;
private IWisejComponent _context;

private void MainPage_Load(object sender, EventArgs e)
{
    _context = Application.Current;
    _feed.TicketChanged += Feed_TicketChanged;
    Application.ApplicationExit += Application_ApplicationExit;
}

private void Application_ApplicationExit(object sender, EventArgs e)
{
    _feed.TicketChanged -= Feed_TicketChanged;
}

private void Feed_TicketChanged(object sender, TicketChangedEventArgs e)
{
    if (this.IsDisposed)
        return;

    Application.Update(_context, () =>
    {
        ticketLog.Items.Insert(0, e.Message);
    });
}
```

The goal is not just to avoid exceptions. The goal is to make ownership obvious. The feed owns global ticket events. The page owns its controls. The session owns its application context.

Handling errors in background tasks

A background task should never fail silently. In a training demo, an exception may only stop the loop. In production, silent failures make dashboards lie. Use a try/catch inside the task, log the exception, and update the UI with a safe error message when the UI still exists.

```
Application.StartTask(() =>
{
    try
    {
        RunLongOperation();
    }
    catch (Exception ex)
    {
        Log.Error(ex);
        if (!this.IsDisposed)
        {
            labelStatus.Text = "The operation failed. See the server log.";
            Application.Update(this);
        }
    }
});
```

Do not display raw exception details to end users in production. Use detailed logs for developers and safe messages for users.



Checkpoint

At the end of this module, the student should be able to explain why a background task started from a page can update that page, why a global service should not directly own page controls, and why every subscription in a real-time app must have an unsubscribe plan.

Application Context, Sessions, and Lifecycle

What you'll build in the hands-on lab, the tasks, the suggested implementation, and the acceptance criteria.



Hands-on lab

Add a diagnostics panel that helps the developer see the current session, application context, browser details, and lifecycle events relevant to real-time updates.

Tasks

1. Add a diagnostics group box to `MainPage`.
2. Display the current time, client id, browser device type if available, and a session-specific counter.
3. Store the counter in `Application.Session` or a per-session object.
4. Add buttons to increment the counter and to simulate a background update.
5. Add cleanup logging when the application exits.
6. Refresh the browser and observe what persists.

Suggested snippets

```
private void MainPage_Load(object sender, EventArgs e)
{
    if (Application.Session.Counter == null)
        Application.Session.Counter = 0;

    clientIdLabel.Text = Application.ClientId;
    titleLabel.Text = "TicketOps Live - " + DateTime.Now.ToShortTimeString();
    Application.Title = "TicketOps Live";

    Application.ApplicationExit += Application_ApplicationExit;
}

private void incrementButton_Click(object sender, EventArgs e)
{
    Application.Session.Counter = (int)Application.Session.Counter + 1;
    counterLabel.Text = Application.Session.Counter.ToString();
}

private void backgroundButton_Click(object sender, EventArgs e)
{
    var context = Application.Current;
    Application.StartTask(() =>
    {
        Thread.Sleep(1500);
        Application.Update(context, () =>
        {
            lifecycleList.Items.Add("Background update at " + DateTime.Now);
        });
    });
}
```

```
    });  
}  
  
private void Application_ApplicationExit(object sender, EventArgs e)  
{  
    // Replace with your logging framework in production.  
    System.Diagnostics.Debug.WriteLine("TicketOps session exited: " + Application.ClientId);  
}
```

Acceptance criteria

- ✓ The counter is session-specific, not static.
- ✓ A background update modifies the correct browser session.
- ✓ The diagnostics panel shows when the background update arrives.
- ✓ The student can explain what happens after refresh and after session timeout.

Reflection questions

1. Why is `Application.Current` captured before starting the out-of-bound action?
2. What would happen if the counter were static?
3. Which cleanup action would you add for a global service subscription?

Single-Session Server Push with Background Tasks

Single-session push: run long work off the click handler, report progress, support cooperative cancellation, and restore UI in finally. The concept, the mental model, and the production pitfalls — read this before the lab.



Production focus

Single-session push: run long work off the click handler, report progress, support cooperative cancellation, and restore UI in finally.

The most common server-push scenario

The most common first use of server push is a long-running operation. A user starts an import, report, migration step, file generation, data synchronization, or validation routine. The operation takes seconds or minutes. The UI should not freeze, the user should see progress, and the operation should report success or failure when finished.

In desktop applications, developers often use a background worker or task and marshal updates back to the UI thread. In Wisej.NET, the equivalent design is session-context background work plus

`Application.Update()` when you need to push UI changes to the browser. The server-side UI still belongs to the session. The background task must respect lifecycle, cancellation, and update frequency.

Basic pattern

The core pattern has six steps:

1. Disable or adjust the UI before starting work.
2. Create cancellation state if the user can cancel.
3. Start the background task in the session context.
4. Run the work in small steps.
5. Update UI state at reasonable intervals.
6. Re-enable the UI and show completion or failure.

```

private CancellationTokenSource _cts;

private void startImportButton_Click(object sender, EventArgs e)
{
    _cts = new CancellationTokenSource();
    startImportButton.Enabled = false;
    cancelImportButton.Enabled = true;
    importProgress.Value = 0;
    importStatus.Text = "Starting import ... ";

    Application.StartTask(() => RunImport(_cts.Token));
}

private void cancelImportButton_Click(object sender, EventArgs e)
{
    _cts?.Cancel();
    cancelImportButton.Enabled = false;
    importStatus.Text = "Cancelling ... ";
}

```

The task method should contain a try/catch/finally so that the UI is not left disabled after an exception.

```

private void RunImport(Cancellation token)
{
    try
    {
        for (int i = 1; i ≤ 100; i++)
        {
            token.ThrowIfCancellationRequested();

            // Simulate work.
            Thread.Sleep(75);

            importProgress.Value = i;
            importStatus.Text = $"Imported {i}%";

            if (i % 5 == 0)
                Application.Update(this);
        }

        importStatus.Text = "Import complete.";
    }
    catch (OperationCanceledException)
    {
        importStatus.Text = "Import cancelled.";
    }
    catch (Exception ex)
    {
        LogError(ex);
        importStatus.Text = "Import failed. See the server log.";
    }
    finally

```

```

{
    startImportButton.Enabled = true;
    cancelImportButton.Enabled = false;
    Application.Update(this);
}

```

Notice the update frequency. The code changes the progress bar on every iteration but pushes only every five iterations plus the final update. This reduces the number of network messages and browser repaints while still feeling live.

Direct UI update vs. context callback

When a task is started with `Application.StartTask()` from a Wisej.NET event, it runs with the current application context. In many examples you will see direct changes to controls followed by `Application.Update(this)`. This is acceptable for a single-session job that is owned by the page.

For code that starts outside the session, use the explicit context overload:

```

var context = Application.Current;

ExternalWorker.JobAdvanced += (s, e) =>
{
    Application.Update(context, () =>
    {
        progressBar.Value = e.Percent;
        labelStatus.Text = e.Message;
    });
};

```

This makes it clear that the UI changes happen in the correct session context.

Avoid blocking event handlers

Do not run a long operation directly in a click handler. The browser request remains open, the user sees no intermediate push updates, and the server thread is occupied for too long. Start the operation and return control to the user.

Bad:

```

private void button_Click(object sender, EventArgs e)
{
    RunTenMinuteImport();
    labelStatus.Text = "Done";
}

```

Better:

```
private void button_Click(object sender, EventArgs e)
{
    Application.StartTask(() => RunTenMinuteImportWithProgress());
    labelStatus.Text = "Import started";
}
```

Cancellation is a feature, not an afterthought

A real-time progress UI without cancellation can be worse than a blocking UI. It tells the user something is happening but gives them no control. Cancellation should be built into long-running tasks wherever safe. Do not terminate threads. Use cooperative cancellation. Check the token between units of work and leave the system in a consistent state.

Cancellation is also useful when the page closes. If the background work is only for that page, cancel it on disposal or application exit. If the work is a server job that should continue, move ownership to a service and let the page display the service state.

Error reporting

A background task can fail after the original event handler has returned. Without explicit error handling, the user may only see that progress stopped. Always catch exceptions inside background work. Log the technical details. Push a user-safe error state to the UI. Keep enough detail to support troubleshooting: job id, session/client id, user id if authenticated, operation name, and time.

Batching and throttling

The human eye does not benefit from unlimited updates. Most business dashboards feel live with 1-4 updates per second. Progress bars often feel smooth at 5-10 updates per second, but many operations do not need that. High-frequency telemetry should be aggregated before display.

Practical rules:

- Push final state immediately.
- Push user-visible progress at a controlled interval.
- Batch multiple label/grid changes into one `Application.Update()` call.
- Avoid pushing inside tight CPU loops.
- Prefer a queue or timer to coalesce bursts from external systems.



Checkpoint

At the end of this module, the student should be able to implement a long-running operation that starts immediately, updates progress without blocking the browser, supports cancellation, reports errors, and leaves the UI in a consistent state.

Single-Session Server Push with Background Tasks

What you'll build in the hands-on lab, the tasks, the suggested implementation, and the acceptance criteria.



Hands-on lab

Add a realistic long-running import panel to TicketOps Live. The user can start an import, watch progress, cancel it, and inspect the result.

UI requirements

Add:

- ✓ `startImportButton`
- ✓ `cancelImportButton`
- ✓ `importProgressBar`
- ✓ `importStatusLabel`
- ✓ `importLogListBox`
- ✓ `recordsImportedLabel`

Tasks

1. Implement an import loop with 200 simulated records.
2. Push progress every 10 records, not every record.
3. Add log messages for start, every 50 records, cancellation, failure, and completion.
4. Implement cancellation with `CancellationTokenSource`.
5. Add a deliberate failure button or option that throws at record 87.
6. Ensure the UI is re-enabled in all outcomes.

Suggested record loop

```
private void RunImport(CancellationToken token, bool failAt87)
{
    try
    {
        for (int i = 1; i ≤ 200; i++)
        {
            token.ThrowIfCancellationRequested();
            Thread.Sleep(25);

            if (failAt87 && i == 87)
                throw new InvalidOperationException("Simulated malformed record.");

            recordsImportedLabel.Text = i.ToString();
            importProgressBar.Value = i / 2;
        }
    }
}
```

```

        if (i % 50 == 0)
            importLogListBox.Items.Insert(0, $"Imported {i} records");

        if (i % 10 == 0)
            Application.Update(this);
    }

    importStatusLabel.Text = "Import completed successfully.";
}
catch (OperationCanceledException)
{
    importStatusLabel.Text = "Import cancelled by user.";
    importLogListBox.Items.Insert(0, "Cancelled at " + DateTime.Now);
}
catch (Exception ex)
{
    LogError(ex);
    importStatusLabel.Text = "Import failed. Review the server log.";
    importLogListBox.Items.Insert(0, "Failed at " + DateTime.Now);
}
finally
{
    startImportButton.Enabled = true;
    cancelImportButton.Enabled = false;
    Application.Update(this);
}
}

```

Acceptance criteria

- ✓ Progress updates appear while the operation is running.
- ✓ Cancellation works and leaves the UI usable.
- ✓ Failure is reported without exposing raw stack traces.
- ✓ The final update always reaches the browser when the session is still alive.
- ✓ The code does not push more than necessary.

Extension challenge

Add a `JobId` to every import and include it in log messages. The capstone will use this to connect UI events to server logs.

Timers, Polling Fallbacks, and Update Cadence

Update cadence and fallback: choose push vs. timers vs. polling, coalesce bursts, and own timer lifecycle per session. The concept, the mental model, and the production pitfalls — read this before the lab.



Production focus

Update cadence and fallback: choose push vs. timers vs. polling, coalesce bursts, and own timer lifecycle per session.

Push is preferred, but fallback matters

A production application cannot assume that every browser, proxy, server, or network path will behave perfectly. Some environments block WebSockets. Some reverse proxies require explicit WebSocket support. Some corporate networks introduce timeouts. Wisej.NET supports real-time push with WebSocket, and it also provides fallback approaches such as polling for systems that cannot use WebSocket.

The intermediate developer must understand the difference between a fallback and a design mistake. Polling is not automatically wrong. It is wrong when it is used as a substitute for a clear event model or when the interval is chosen without regard to server cost. A one-second polling interval for thousands of sessions can be expensive. A ten-second interval for a critical operations board may feel slow. The right choice depends on user value and cost.

Wisej.Web.Timer vs. background task

A `Wisej.Web.Timer` is useful when you want periodic server-side events associated with a session. For example, a dashboard might refresh a calculated KPI every five seconds, even if no global service event has arrived. A background task is better when a specific operation starts, runs, and completes. A global service is better when one shared watcher feeds many sessions.

Use a timer for session-scoped periodic UI refresh. Use `Application.StartTask()` for session-scoped long-running work. Use a shared service for global state and multi-user events.

Polling fallback with StartPolling and EndPolling

When WebSocket is available, `Application.Update()` can push pending updates to the browser. When WebSocket is not available, it has no live connection to use. Wisej.NET provides `Application.StartPolling(milliseconds)` and `Application.EndPolling()` as fallback mechanisms. Polling is ignored when WebSocket is active, which allows the application to request fallback behavior without harming WebSocket sessions.

A common pattern is:

```
private void BeginLiveMode()
{
    Application.StartPolling(1000);

    Application.StartTask(() =>
    {
        while (_liveMode && !this.IsDisposed)
        {
            UpdateDashboardModel();
            ApplyDashboardToControls();

            // Effective when WebSocket exists. Polling handles non-WebSocket cases.
            Application.Update(this);
            Thread.Sleep(1000);
        }

        Application.EndPolling();
    });
}
```

The code must still be designed carefully. If `_liveMode` never becomes false and the page is disposed, the loop becomes a leak. Always have a stop condition.

Choosing update cadence

Cadence is the rhythm of updates. In a Wisej.NET real-time app, cadence has four dimensions:

Model cadence: how often the underlying data changes.

UI cadence: how often the user needs to see visible changes.

Push cadence: how often the server sends UI updates to the browser.

Polling cadence: how often the browser asks the server for updates when push is unavailable.

These are not always the same. A machine may produce telemetry 100 times per second, but the UI may only need a summary once per second. A queue may change 20 times in a burst, but the dashboard can display the final state after 250 milliseconds.

Coalescing bursts

A burst occurs when many changes arrive at once. If each change causes a separate push, the UI may waste resources and show intermediate states that users cannot understand. Coalescing means collecting multiple changes and applying them together.

Simple coalescing strategy:

```

private readonly object _sync = new object();
private bool _dirty;

private void OnModelChanged()
{
    lock (_sync)
    {
        _dirty = true;
    }
}

private void refreshTimer_Tick(object sender, EventArgs e)
{
    bool shouldRefresh;

    lock (_sync)
    {
        shouldRefresh = _dirty;
        _dirty = false;
    }

    if (!shouldRefresh)
        return;

    ApplyModelToControls();
}

```

The event marks the model as dirty. The timer applies updates at a stable UI cadence.

Timer ownership

A timer belongs somewhere. If it is a Wisej.NET control component on a page, the page owns it. If it is a .NET timer in a global service, the service owns it. Ownership determines cleanup. A duplicated timer can double update traffic and make bugs appear random.

Practical rules:

- Start timers once.
- Stop timers when the view is no longer active.
- Do not create a new timer on every refresh button click.
- Avoid long-running work inside a timer tick; start work or schedule it instead.
- Use a flag to prevent overlapping ticks if a refresh takes longer than the interval.

Detecting bad cadence

Symptoms of too many updates include high CPU, choppy browser rendering, large WebSocket traffic, server log noise, and dashboards that appear to flicker. Symptoms of too few updates include users pressing refresh, duplicate work, stale operational decisions, and lack of trust in the dashboard.

Intermediate developers should instrument important dashboards. Count events received, UI updates applied, pushes attempted, errors, and average time spent applying an update.



Checkpoint

At the end of this module, the student should be able to choose a timer, background task, polling fallback, or service event based on the job to be done, and explain the cost of the chosen cadence.

Timers, Polling Fallbacks, and Update Cadence

What you'll build in the hands-on lab, the tasks, the suggested implementation, and the acceptance criteria.



Hands-on lab

Extend TicketOps Live with a dashboard refresh timer and controls to experiment with update cadence.

UI requirements

Add:

- ✓ `liveModeCheckBox`
- ✓ `cadenceComboBox` with values `250 ms`, `1 sec`, `5 sec`
- ✓ `eventsReceivedLabel`
- ✓ `updatesAppliedLabel`
- ✓ `lastAppliedLabel`
- ✓ `refreshTimer`

Tasks

1. Start polling when live mode is enabled.
2. Stop polling when live mode is disabled.
3. Use a timer to apply dashboard updates at the selected cadence.
4. Simulate faster model changes than UI updates.
5. Count model events and applied UI updates separately.
6. Prevent overlapping timer ticks.

Suggested implementation idea

```
private int _eventsReceived;
private int _updatesApplied;
private bool _refreshInProgress;
private volatile bool _dashboardDirty;

private void liveModeCheckBox_CheckedChanged(object sender, EventArgs e)
{
    if (liveModeCheckBox.Checked)
    {
        Application.StartPolling(1000);
        refreshTimer.Interval = GetSelectedInterval();
        refreshTimer.Start();
    }
    else
    {
        refreshTimer.Stop();
        Application.EndPolling();
    }
}
```

```

    }
}

private void SimulatedModelChanged()
{
    _eventsReceived++;
    _dashboardDirty = true;
}

private void refreshTimer_Tick(object sender, EventArgs e)
{
    if (_refreshInProgress || !_dashboardDirty)
        return;

    try
    {
        _refreshInProgress = true;
        _dashboardDirty = false;
        _updatesApplied++;

        eventsReceivedLabel.Text = _eventsReceived.ToString();
        updatesAppliedLabel.Text = _updatesApplied.ToString();
        lastAppliedLabel.Text = DateTime.Now.ToString("HH:mm:ss.fff");
    }
    finally
    {
        _refreshInProgress = false;
    }
}

```

Acceptance criteria

- ✓ The timer cadence changes visibly.
- ✓ The events received count can grow faster than the UI updates applied count.
- ✓ Polling starts and stops with live mode.
- ✓ No overlapping timer ticks occur.
- ✓ The student can explain why update coalescing is useful.

Live Data Binding and Real-Time Grids

Live data binding: update bound collections in session context without rebinding the grid, preserving selection, sort and scroll. The concept, the mental model, and the production pitfalls — read this before the lab.



Production focus

Live data binding: update bound collections in session context without rebinding the grid, preserving selection, sort and scroll.

Why data binding becomes harder when data moves

Static data binding is easy: load records, bind the grid, let the user edit, save. Real-time data binding adds movement. Rows arrive, fields change, statuses advance, and background jobs update the same screen the user is viewing. The challenge is to keep the UI synchronized without constantly destroying and recreating bindings.

Wisej.NET supports simple binding for single control properties and complex binding for controls such as grids. The `BindingSource` component is especially useful because it sits between controls and data sources, manages currency, and provides a consistent place to reset or refresh bindings.

The Ticket model

Use a simple ticket model in this course:

```
public enum TicketStatus
{
    New,
    Assigned,
    Waiting,
    Resolved,
    Escalated
}

public class Ticket : INotifyPropertyChanged
{
    private TicketStatus _status;
    private string _owner;
    private DateTime _updatedAt;

    public int Id { get; set; }
    public string Title { get; set; }
    public string Customer { get; set; }

    public string Owner
    {
        get => _owner;
        set { _owner = value; OnPropertyChanged(nameof(Owner)); }
    }
}
```

```

public TicketStatus Status
{
    get => _status;
    set { _status = value; OnPropertyChanged(nameof(Status)); }
}

public DateTime UpdatedAt
{
    get => _updatedAt;
    set { _updatedAt = value; OnPropertyChanged(nameof(UpdatedAt)); }
}

public event PropertyChangedEventHandler PropertyChanged;

protected void OnPropertyChanged(string propertyName)
{
    PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
}
}

```

The point of `INotifyPropertyChanged` is to make property changes visible to binding infrastructure where supported. Even when you call `BindingSource.ResetBindings(false)` for clarity, the model still expresses that it can notify changes.

Binding a live list

A practical starting point is `BindingList<Ticket>` plus `BindingSource`.

```

private BindingList<Ticket> _tickets;
private BindingSource _ticketSource;

private void MainPage_Load(object sender, EventArgs e)
{
    _tickets = new BindingList<Ticket>();
    _ticketSource = new BindingSource();
    _ticketSource.DataSource = _tickets;

    ticketsGrid.AutoGenerateColumns = true;
    ticketsGrid.DataSource = _ticketSource;
}

```

When a new ticket arrives, add it to the list in the correct session context. When a field changes, update the item and reset bindings if the UI does not refresh automatically.

```
private void AddTicket(Ticket ticket)
{
    _tickets.Insert(0, ticket);
    _ticketSource.ResetBindings(false);
}

private void UpdateTicket(int id, TicketStatus status, string owner)
{
    var ticket = _tickets.FirstOrDefault(t => t.Id == id);
    if (ticket == null)
        return;

    ticket.Status = status;
    ticket.Owner = owner;
    ticket.UpdatedAt = DateTime.Now;
    _ticketSource.ResetBindings(false);
}
```

Do not rebind the grid for every update

Rebinding the whole grid is sometimes the fastest way to make a demo work, but it is often wrong in production. It can reset selection, scroll position, sorting, column widths, and user context. It may also be expensive. Prefer incremental changes to the bound list, then notify the binding source.

Bad:

```
ticketsGrid.DataSource = null;
ticketsGrid.DataSource = LoadAllTicketsAgain();
```

Better:

```
ApplyTicketChange(change);
_ticketSource.ResetBindings(false);
```

Best for high-volume screens: batch several changes and refresh once per cadence interval.

Thread safety with bound collections

A bound collection belongs to the session UI. A global service may receive updates from another thread. Do not mutate the bound collection from arbitrary threads without entering the session context and considering synchronization.

A safe multi-user pattern is:

1. Global service receives a ticket event.
2. Session event handler captures the event.
3. Handler uses `Application.Update(context, () => { ... })`.
4. Inside the callback, update the session's bound list.
5. Reset bindings once.

```
private void Feed_TicketChanged(object sender, TicketChangedEventArgs e)
{
    Application.Update(_context, () =>
    {
        ApplyTicketChange(e.Change);
        _ticketSource.ResetBindings(false);
    });
}
```

Formatting and parsing

Real-time grids are not only about data transport. They must remain readable. Use formatting for dates, currency, priorities, and statuses. Use parse/format events for bound controls when values require conversion. Avoid storing formatted strings as your model if the real value has a stronger type.

For example, `UpdatedAt` should be a `DateTime`, not a preformatted string. The grid can display it as local time or relative age.

Row status and visual cues

A live grid should show what changed without becoming noisy. Common patterns:

- show a temporary "updated" marker for a few seconds;
- sort new tickets to the top;
- keep the user's selected ticket stable when possible;
- avoid moving a row while the user is editing it;
- use status columns and small indicators rather than modal interruptions.

The UI should help users trust the data. Too many animations or row jumps can make users think the application is unstable.

Validation and conflict handling

Real-time data means another user or service can change a record while the current user is viewing or editing it. Decide what happens when a ticket changes during editing. For this course, use a simple rule: if a ticket is being edited in a details form, show a non-blocking warning that the ticket changed and allow the user to reload or continue.

Production systems may need optimistic concurrency with version numbers, timestamps, or database row versions. Wisej.NET does not remove the need for data-layer concurrency. It gives the UI a way to notify users at the right moment.



Checkpoint

At the end of this module, the student should be able to maintain a live grid that receives new rows and row updates from a background source, while preserving the user's context and avoiding wasteful rebinding.

Live Data Binding and Real-Time Grids

What you'll build in the hands-on lab, the tasks, the suggested implementation, and the acceptance criteria.



Hands-on lab

Create a live `DataGridView` that displays support tickets and updates when simulated ticket events arrive.

Tasks

1. Add a `DataGridView` named `ticketsGrid`.
2. Create the `Ticket` model and `TicketStatus` enum.
3. Bind a `BindingList<Ticket>` through a `BindingSource`.
4. Add a button that creates a new ticket event every second for 20 seconds.
5. Add a button that randomly changes ticket statuses.
6. Preserve the selected row when possible.
7. Show `UpdatedAt` in a readable format.

Suggested model event

```
public class TicketChangedEventArgs : EventArgs
{
    public Ticket Ticket { get; set; }
    public string ChangeType { get; set; }
    public string Message { get; set; }
}
```

Suggested update method

```
private void ApplyTicketEvent(TicketChangedEventArgs e)
{
    var selectedId = ticketsGrid.CurrentRow?.DataBoundItem is Ticket selected
        ? selected.Id
        : (int?)null;

    var existing = _tickets.FirstOrDefault(t => t.Id == e.Ticket.Id);

    if (existing == null)
        _tickets.Insert(0, e.Ticket);
    else
    {
        existing.Status = e.Ticket.Status;
        existing.Owner = e.Ticket.Owner;
        existing.UpdatedAt = DateTime.Now;
    }

    _ticketSource.ResetBindings(false);

    if (selectedId.HasValue)
        RestoreSelection(selectedId.Value);
}
```

Acceptance criteria

- ✓ New tickets appear without rebinding the grid from scratch.
- ✓ Existing tickets update in place.
- ✓ The selected row is preserved when the row still exists.
- ✓ The update rate remains controlled.
- ✓ The student can explain why the bound list is session-owned.

Extension challenge

Add a filter for `Escalated` tickets and make sure live updates still work when the filter is active.

Multi-User Push with Services and Event Hubs

Multi-user push: a global event hub raises domain events while each session updates its own UI in its own context. The concept, the mental model, and the production pitfalls — read this before the lab.



Production focus

Multi-user push: a global event hub raises domain events while each session updates its own UI in its own context.

From one session to many sessions

Single-session push is useful, but many business applications need multi-user push. When one dispatcher assigns a ticket, every supervisor dashboard should update. When a nightly import finishes, all connected administrators should see the result. When a service health check fails, active operations consoles should show the alert.

The key architecture change is ownership. A page owns its controls. A session owns its context. A global service owns shared data and events. Do not let the global service become a bag of form references. Instead, use domain events and session-level subscriptions.

The event hub pattern

An event hub is a shared service that receives or creates domain events and exposes them to subscribers. The hub does not know how every UI displays the event. It simply announces that something happened.

```
public sealed class TicketHub
{
    private static readonly Lazy<TicketHub> _instance =
        new Lazy<TicketHub>(() => new TicketHub());

    public static TicketHub Instance => _instance.Value;

    public event EventHandler<TicketChangedEventArgs> TicketChanged;

    private readonly object _sync = new object();
    private readonly List<Ticket> _tickets = new List<Ticket>();

    private TicketHub() { }

    public IReadOnlyList<Ticket> GetSnapshot()
    {
        lock (_sync)
            return _tickets.Select(CloneTicket).ToList();
    }

    public void AddOrUpdate(Ticket ticket)
    {
        TicketChangedEventArgs args;
```

```

lock (_sync)
{
    var existing = _tickets.FirstOrDefault(t => t.Id == ticket.Id);
    if (existing == null)
        _tickets.Insert(0, ticket);
    else
        CopyTicket(ticket, existing);

    args = new TicketChangedEventArgs
    {
        Ticket = CloneTicket(ticket),
        ChangeType = existing == null ? "Added" : "Updated",
        Message = $"Ticket {ticket.Id} {ticket.Status}"
    };

    TicketChanged?.Invoke(this, args);
}
}

```

The hub keeps global state thread-safe and raises a domain event. It does not call `Application.Update()` because it does not own a session.

Session subscriber pattern

Each session subscribes to the hub and updates its own UI.

```

private readonly TicketHub _hub = TicketHub.Instance;
private IWisejComponent _context;

private void MainPage_Load(object sender, EventArgs e)
{
    _context = Application.Current;
    LoadSnapshot(_hub.GetSnapshot());
    _hub.TicketChanged += Hub_TicketChanged;
    Application.ApplicationExit += Application_ApplicationExit;
}

private void Hub_TicketChanged(object sender, TicketChangedEventArgs e)
{
    if (this.IsDisposed)
        return;

    Application.Update(_context, () =>
    {
        ApplyTicketEvent(e);
        notificationsList.Items.Insert(0, e.Message);
    });
}

private void Application_ApplicationExit(object sender, EventArgs e)
{
    _hub.TicketChanged -= Hub_TicketChanged;
}

```

This pattern gives every session its own UI. Two users can filter or sort differently while receiving the same domain events.

Broadcast vs. targeted update

Broadcast means every subscriber receives the event. Targeted update means only some sessions receive it. Examples:

- Broadcast all severe system alerts to all operations consoles.
- Send tenant-specific events only to sessions in that tenant.
- Send assignment notifications only to the assigned operator.
- Send admin events only to users with admin roles.

The hub can include enough metadata for the subscriber to decide whether to display the event. Alternatively, the hub can manage subscription groups. Keep the first version simple: include tenant/user/role metadata and let the session filter.

Thread safety and snapshots

Global services often maintain collections. Collections should be protected by locks or concurrent collections. When exposing data to sessions, prefer snapshots over direct collection references. A snapshot

avoids letting UI code modify global state accidentally and avoids enumeration while the service changes the collection.

Bad:

```
public List<Ticket> Tickets => _tickets;
```

Better:

```
public IReadOnlyList<Ticket> GetSnapshot()
{
    lock (_sync)
        return _tickets.Select(CloneTicket).ToList();
}
```

The clone does not have to be expensive for every use case, but the ownership boundary must be clear.

Memory leaks through events

Events are a common source of memory leaks. If a global service has an event and a page subscribes, the service now has a reference to the page's handler. If the page is closed but never unsubscribes, the service may keep it alive. In real-time apps, this leak is both a memory problem and a correctness problem because the service may try to update a dead session.

Every subscription must have an unsubscribe. This is a course rule.

Backpressure

If the hub emits changes faster than sessions can render them, the system needs backpressure or coalescing. Simple strategies include:

- drop low-priority duplicate events;
- batch changes and send summaries;
- mark a session as dirty and refresh on a timer;
- keep only the latest state for a dashboard metric;
- queue events per session with a maximum size.

Business apps usually do not need to render every intermediate state. They need to render the state users can act upon.



Checkpoint

At the end of this module, the student should be able to design a shared service that safely feeds many Wisej.NET sessions without storing page controls globally, leaking event handlers, or mutating bound UI collections from the wrong thread.

Multi-User Push with Services and Event Hubs

What you'll build in the hands-on lab, the tasks, the suggested implementation, and the acceptance criteria.



Hands-on lab

Refactor the ticket simulation into a global `TicketHub` service and make multiple browser sessions receive live ticket events.

Tasks

1. Create a singleton `TicketHub` with a thread-safe ticket list.
2. Add `TicketChanged` events.
3. Load a snapshot when `MainPage` starts.
4. Subscribe the page to `TicketChanged`.
5. Unsubscribe on application exit.
6. Open two browsers or two sessions and verify that both receive updates.
7. Add tenant or role metadata and filter at least one event type.

Acceptance criteria

- ✓ The hub does not store page or control references.
- ✓ Each session updates its own bound list.
- ✓ Closing or refreshing a session does not leave a broken subscriber behind.
- ✓ Multiple sessions can show different filters while sharing the same domain event source.
- ✓ The student can explain broadcast vs. targeted update.

Extension challenge

Add a per-session notification count. The count should be session-specific even though the events come from a global service.

Production Hardening, Deployment, and Capstone

Production hardening: verify the full WebSocket path, sticky sessions, health checks, security and observability before deploy. The concept, the mental model, and the production pitfalls — read this before the lab.



Production focus

Production hardening: verify the full WebSocket path, sticky sessions, health checks, security and observability before deploy.

Real-time production is architecture plus operations

A real-time app can work perfectly on a developer machine and fail in production because the deployment path is different. WebSocket may be blocked by a proxy. A load balancer may route a user's requests to different server instances. A session timeout may terminate long-running UI work. A global service may use too much memory. A security setting may be safe in development but wrong behind a reverse proxy.

This module converts the course patterns into a production checklist.

WebSocket-aware deployment

Wisej.NET real-time push works best when WebSocket is available. In production, verify WebSocket support across the full path: browser, network, reverse proxy, load balancer, web server, and application. Some load balancers need TCP routing or explicit WebSocket support. Reverse proxies such as NGINX can proxy WebSocket, but they must be configured correctly.

Testing only from the server console is not enough. Test from the user's network path. A corporate VPN, security appliance, or older proxy can change behavior.

Load balancing and session affinity

Wisej.NET applications maintain server-side session state. In a multi-server deployment, requests for a given session must consistently reach the same server instance unless the architecture specifically supports another strategy. The practical load-balancing rule is to use sticky sessions or another routing mechanism that guarantees user-server consistency.

Health checks should tell the load balancer when a server should stop receiving traffic. Wisej.NET supports health check configuration using a `HealthCheck.json` file, with parameters such as maximum memory, maximum sessions, and CPU threshold. The exact thresholds depend on application behavior and hosting size.

Important configuration areas

Configuration belongs to both `Web.config` and Wisej.NET JSON configuration files. Wisej.NET applications use JSON application files such as `Default.json`, and a project can define multiple Wisej.NET applications with their own configuration files.

Important real-time settings and deployment concerns include:

- `sessionTimeout` : controls when inactive sessions time out;
- `pollingInterval` : controls automatic polling behavior for non-WebSocket clients;
- `responseTimeout` : matters for long initial HTTP responses, but WebSocket does not time out in the same way;
- `debug=false` in production so JavaScript libraries are bundled/minified appropriately;
- server license key and default theme settings;
- IIS module/handler settings for .NET Framework apps;
- ASP.NET Core hosting model and middleware for .NET Core/.NET apps;
- health check settings for load-balanced deployments.

Example `Default.json` fragment for a training app:

```
{
  "mainWindow": "TicketOpsLive.MainPage, TicketOpsLive",
  "theme": "Bootstrap-4",
  "sessionTimeout": 180,
  "pollingInterval": 1000
}
```

Do not copy training values blindly into production. Use load testing and user requirements to choose settings.

Security considerations for real-time apps

Wisej.NET avoids many traditional HTML string-concatenation problems because it synchronizes server controls and browser widgets rather than building pages with arbitrary HTML strings. However, developers can still create XSS risk if they display user-supplied content as HTML. Controls and messages that allow HTML must be handled carefully. Keep user-entered ticket titles, comments, names, and messages encoded unless there is a clear sanitization process.

Behind a reverse proxy or load balancer, client IP handling must also be deliberate. Forwarded headers should only be trusted when the app is behind a trusted proxy that sanitizes incoming headers. Otherwise, an attacker may forge header values.

Real-time apps should also protect server resources. Use limits for sessions, queues, and event rates. Do not let one user create unlimited background jobs or subscriptions.

Observability

A real-time app needs logs and metrics that explain what happened after the original user request ended. Include correlation fields:

- job id;
- user id or anonymous session marker;
- client id/session id where appropriate;

- operation name;
- event type;
- server instance;
- duration and outcome;
- number of UI updates pushed or applied.

For TicketOps Live, every import job should have a job id. Every hub event should have an event id. The UI can show simplified status, but logs should allow support staff to reconstruct the sequence.

Performance checklist

Before deployment, review these questions:

1. Does every background loop have a stop condition?
2. Does every global event subscription unsubscribe?
3. Are UI updates batched or throttled?
4. Are high-frequency model events coalesced?
5. Are bound collections updated in the session context?
6. Are static fields used only for global state?
7. Are session-specific counters, filters, and selections per session?
8. Are exceptions caught and logged inside background tasks?
9. Does the app behave acceptably when WebSocket is unavailable?
10. Does the load balancer keep a session on the same server instance?

Capstone overview

The capstone is a production review and completion of TicketOps Live. Students must demonstrate:

- live status updates;
- background import progress with cancellation and error handling;
- live ticket grid updates;
- multi-user ticket notifications;
- polling fallback strategy;
- session cleanup;
- deployment and observability checklist.

The capstone is not graded by the number of controls on the screen. It is graded by architecture: ownership, context, lifecycle, update cadence, and production readiness.



Checkpoint

At the end of this module, the student should be able to explain how a Wisej.NET real-time app moves from a demo to a deployable application: WebSocket path verified, session routing stable, fallback defined, subscriptions cleaned up, updates throttled, data protected, and operations observable.

Production Hardening, Deployment, and Capstone

What you'll build in the hands-on lab, the tasks, the suggested implementation, and the acceptance criteria.



Hands-on lab

Complete TicketOps Live and prepare it for a production architecture review.

Required features

1. Live status strip from M01.
2. Session diagnostics from M02.
3. Background import monitor from M03.
4. Cadence and polling controls from M04.
5. Live ticket board from M05.
6. Multi-user **TicketHub** from M06.
7. Production checklist and configuration review from M07.

Architecture deliverable

Create a one-page architecture note with:

- ✓ which state is session-specific;
- ✓ which state is global;
- ✓ where background work starts;
- ✓ how UI updates reach the browser;
- ✓ how cancellation works;
- ✓ how subscriptions are removed;
- ✓ what happens when WebSocket is unavailable;
- ✓ what the deployment environment must support.

Demo script

1. Open TicketOps Live in two browser sessions.
2. Start the heartbeat.
3. Start an import and show progress.
4. Cancel one import.
5. Start another import and force a simulated failure.
6. Generate new ticket events and show both sessions update.
7. Apply a filter in one session and show that the other session is independent.
8. Disable live mode and explain polling/cadence behavior.
9. Present the production checklist.

Acceptance criteria

- ✓ The application runs without duplicate loops or duplicate subscriptions.
- ✓ The UI remains responsive during background work.
- ✓ Every long-running operation has completion, cancellation, and failure states.
- ✓ The multi-user hub does not keep direct references to pages or controls.
- ✓ Bound grids update incrementally.
- ✓ The student can defend all deployment assumptions.