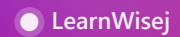


Echtzeit-Apps mit Server-Push — Lektionsbuch

Alle Modul-Lektüren dieses Kurses in einem Dokument.



Inhalt

01	Echtzeit-Webarchitektur in Wisej.NET	Modul 1 · Lektionsleitfaden
02	Echtzeit-Webarchitektur in Wisej.NET	Modul 1 · Lab- und Prüfungsleitfaden
03	Anwendungskontext, Sessions und Lebenszyklus	Modul 2 · Lektionsleitfaden
04	Anwendungskontext, Sessions und Lebenszyklus	Modul 2 · Lab- und Prüfungsleitfaden
05	Server-Push in einer Session mit Hintergrundaufgaben	Modul 3 · Lektionsleitfaden
06	Server-Push in einer Session mit Hintergrundaufgaben	Modul 3 · Lab- und Prüfungsleitfaden
07	Timer, Polling-Fallbacks und Aktualisierungstakt	Modul 4 · Lektionsleitfaden
08	Timer, Polling-Fallbacks und Aktualisierungstakt	Modul 4 · Lab- und Prüfungsleitfaden
09	Live-Data-Binding und Echtzeit-Grids	Modul 5 · Lektionsleitfaden
10	Live-Data-Binding und Echtzeit-Grids	Modul 5 · Lab- und Prüfungsleitfaden
11	Push an mehrere Benutzer mit Services und Event-Hubs	Modul 6 · Lektionsleitfaden
12	Push an mehrere Benutzer mit Services und Event-Hubs	Modul 6 · Lab- und Prüfungsleitfaden
13	Härtung für den Produktivbetrieb, Deployment und Abschlussprojekt	Modul 7 · Lektionsleitfaden
14	Härtung für den Produktivbetrieb, Deployment und Abschlussprojekt	Modul 7 · Lab- und Prüfungsleitfaden

Echtzeit-Webarchitektur in Wisej.NET

Servergesteuerte Echtzeit-Architektur: WebSocket-Push, Aktualisierungen innerhalb eines Requests und Polling als Fallback im serverseitigen Modell von Wisej.NET. Das Konzept, das mentale Modell und die Fallstricke im Produktivbetrieb – lesen Sie das vor dem Lab.



Fokus: Produktivbetrieb

Servergesteuerte Echtzeit-Architektur: WebSocket-Push, Aktualisierungen innerhalb eines Requests und Polling als Fallback im serverseitigen Modell von Wisej.NET.

Warum es dieses Modul gibt

Echtzeit-Webanwendungen verbindet man meist mit Chats, Spielen, Börsentickern, Monitoring-Ansichten und Dashboards. In Geschäftsanwendungen taucht dieselbe Anforderung in weniger spektakulärer, aber wertvollerer Form auf: Eine Teamleiterin sieht neue Tickets, sobald sie eingehen, ein Anlagenbediener sieht, dass eine Maschine stillsteht, ein Disponent sieht, dass ein Auftrag neu zugewiesen wurde, oder eine Sachbearbeiterin in der Fakturierung sieht, dass ein Import noch läuft – ohne auf „Aktualisieren“ zu klicken.

Wisej.NET basiert auf einem serverseitigen Anwendungsmodell. Sie arbeiten in C# oder VB.NET und verwenden Controls, Events, Formulare, Pages, Timer, Data-Binding und den Visual-Studio-Designer, während Wisej.NET den Widget-Baum im Browser verwaltet und mit dem Zustand auf dem Server synchronisiert. Für Server-Push ist dieses Modell besonders wichtig, weil der Anwendungszustand ohnehin dem Server gehört. Eine Hintergrundaufgabe kann ein Label ändern, eine Grid-Zeile hinzufügen, eine ProgressBar aktualisieren oder eine Benachrichtigung öffnen, und Wisej.NET kann die ausstehenden UI-Änderungen an den Browser ausliefern, sobald eine aktive WebSocket-Verbindung besteht.

Ein Einsteigerkurs vermittelt in der Regel, wie man Pages, Fenster, Controls, Menüs und einfache Event-Handler erstellt. Dieser Kurs für Fortgeschrittene setzt diese Kenntnisse voraus und konzentriert sich auf die Designfragen für den Produktivbetrieb, die entstehen, sobald sich die UI selbst aktualisiert. Die zentrale Frage lautet nicht: „Kann ich ein Label aus einem Thread aktualisieren?“ Die zentrale Frage lautet: „Wie aktualisiere ich die Session des richtigen Benutzers, im richtigen Takt, aus dem richtigen Service – ohne Zustand zu verlieren, Speicherlecks zu erzeugen, den Server zu blockieren oder den Browser zu überlasten?“

Was „Echtzeit“ in diesem Kurs bedeutet

In diesem Kurs ist eine Wisej.NET-Echtzeit-App eine Anwendung, die Events des Benutzers mit geringer Verzögerung empfängt und außerdem den Browser des Benutzers vom Server aus aktualisieren kann, ohne auf den nächsten Button-Klick zu warten. Entscheidend ist dabei der Server-Push. Die Anwendung soll die sichtbare UI ändern können, weil auf dem Server etwas passiert ist: Ein Hintergrundjob ist vorangekommen, eine Warteschlange hat sich geändert, eine Nachricht ist eingetroffen oder ein anderer Benutzer hat eine Aktion ausgeführt.

Reduzieren Sie Echtzeit nicht auf „schnelles Aktualisieren“. Ein Grid, das sich alle 30 Sekunden aktualisiert, ist manchmal akzeptabel, aber es ist nicht dasselbe wie Push. Push ist ereignisgesteuert: Die Anwendung

aktualisiert sich, weil sich das Modell geändert hat. Polling fragt wiederholt nach, ob sich etwas geändert hat. Beides kann nützlich sein, doch Kosten, Latenz und Verhalten im Fehlerfall unterscheiden sich.

Das mentale Echtzeit-Modell von Wisej.NET

Ein hilfreiches mentales Modell sieht so aus:

1. Der Browser beherbergt die JavaScript-Widgets von Wisej.NET.
2. Der Server beherbergt die Session der Wisej.NET-Anwendung und die serverseitigen Controls.
3. Benutzeraktionen gelangen als Events vom Browser zum Server.
4. Serverseitiger Code ändert Controls und Anwendungszustand.
5. Wisej.NET sendet die Änderungen als kompakte Updates zurück an den Browser.
6. Mit WebSocket kann eine Hintergrundaufgabe ausstehende Updates pushen, ohne auf den nächsten Browser-Request zu warten.

Das ist weder Screen-Scraping noch ein Remote-Desktop. Der Browser rendert weiterhin native Web-Widgets, und der Server streamt keine Pixel. Architektonisch wichtig ist, dass die Anwendungslogik und der maßgebliche UI-Zustand auf dem Server liegen. Ändert sich eine Eigenschaft eines Wisej.NET-Controls, kann das zugehörige Widget im Browser synchronisiert werden.

WebSocket-Push oder request-gesteuerte Updates

Ein normaler Web-Request beginnt, wenn der Browser den Server kontaktiert, und endet, wenn der Server antwortet. Gibt es keinen offenen Kanal vom Server zum Browser, kann der Server den Browser nicht von sich aus kontaktieren. Er kann Änderungen nur senden, wenn der Browser einen weiteren Request stellt. WebSocket ändert das: Über eine WebSocket-Verbindung kann Wisej.NET UI-Updates aus serverseitigem Code an den verbundenen Client pushen.

In Wisej.NET kann eine Hintergrundaufgabe UI-Controls aktualisieren und `Application.Update()` aufrufen, um die ausstehenden Änderungen zu pushen, sofern WebSocket verfügbar ist. Ist WebSocket nicht verfügbar, kann `Application.Update()` nicht pushen, weil keine aktive Verbindung besteht. Die Anwendung braucht dann einen Fallback, etwa das Polling von Wisej.NET oder timergesteuerte Requests. Der Fallback macht die App nicht unbrauchbar – er ändert lediglich den Zustellmechanismus.

Drei Update-Mechanismen, die Sie unterscheiden müssen

1. Updates innerhalb eines Requests

Das ist der einfachste Fall. Der Benutzer klickt auf einen Button, ein Event-Handler läuft, der Handler ändert Controls, und Wisej.NET sendet die Änderungen mit der Antwort zurück. Hier rufen Sie `Application.Update()` normalerweise nicht auf, weil das Update am Ende des Requests ohnehin an den Browser zurückgeht.

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

2. WebSocket-Push außerhalb eines Requests

Das ist das Hauptthema dieses Kurses. Eine Hintergrundaufgabe läuft außerhalb des ursprünglichen Browser-Events. Sie ändert die serverseitige UI und ruft `Application.Update()` auf, um die Änderungen über die WebSocket-Verbindung zu pushen.

```
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 oder timergesteuerte Aktualisierung

Wenn Browser oder Server kein WebSocket nutzen können, kann der Server keine Änderungen außerhalb eines Requests pushen. In diesem Fall kann die Anwendung Polling verwenden: Der Browser kontaktiert den Server in regelmäßigen Abständen und erhält dabei die ausstehenden Änderungen. Alternativ kann ein `Wisej.Web.Timer` in einem festgelegten Intervall serverseitige Events auslösen.

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

Der Kurs zeigt Ihnen, wie Sie zuerst für WebSocket entwerfen und dann vorhersehbar auf einfachere Mechanismen zurückfallen.

Das Kursprojekt: TicketOps Live

Alle Module bauen Teile derselben Anwendung, **TicketOps Live**. Es handelt sich um eine Echtzeit-Konsole für den Support-Betrieb mit folgenden Funktionen:

- eine Live-Statusleiste des Systems;
- ein Fortschrittsmonitor für Serverjobs;

- ein Ticketboard, das sich aktualisiert, wenn Tickets erstellt oder geändert werden;
- ein Benachrichtigungsbereich für Events mehrerer Benutzer;
- ein Betriebs-Dashboard mit einfachen Indikatoren für den Serverzustand;
- eine Checkliste für den Produktivbetrieb für das Deployment hinter IIS oder einem Load Balancer.

Fachlich ist das Projekt bewusst einfach gehalten. Ziel ist es, Echtzeit-Architektur zu lernen, nicht ein vollständiges Helpdesk-System zu implementieren. Jedes Lab soll Ihnen Code hinterlassen, den Sie in echten Anwendungen wiederverwenden können.

Designregel: erst das Modell, dann die UI aktualisieren

Es ist verlockend, Hintergrundaufgaben direkt jedes Control ändern zu lassen, das sich ändern muss. Für kleine Demos funktioniert das, echte Anwendungen brauchen aber eine Trennung zwischen Daten und Darstellung. Die Hintergrundaufgabe sollte in der Regel ein Modell aktualisieren oder ein Domain-Event auslösen, und die sessionspezifische UI entscheidet, wie sie das anzeigt.

Bei einem Fortschrittsfenster für einen einzelnen Benutzer sind direkte Updates der Controls akzeptabel. Bei Dashboards für mehrere Benutzer ist globaler Direktzugriff auf Controls falsch. Controls gehören zu Sessions. Ein globaler Service sollte nicht beliebige Formularinstanzen dauerhaft speichern. Stattdessen abonnieren Sessions einen Service, empfangen Events und aktualisieren ihre eigenen Controls in ihrem eigenen Anwendungskontext.

Ein sicheres Vokabular für diesen Kurs

Session bezeichnet eine laufende Instanz einer Wisej.NET-Anwendung für den Kontext eines Browser-Tabs bzw. -Fensters. Sie ist nicht dasselbe wie ein angemeldeter Benutzer: Ein Benutzer kann mehrere Sessions haben.

Anwendungskontext bezeichnet den Wisej.NET-Kontext, den Code benötigt, um auf die sessionspezifischen statischen **Application**-Eigenschaften zuzugreifen und den richtigen Browser zu aktualisieren.

In-bound-Code bezeichnet Code, der als Teil eines Browser-Requests oder eines Wisej.NET-Events ausgeführt wird.

Out-of-bound-Code bezeichnet Code, der außerhalb eines Browser-Events läuft, etwa in einem Hintergrund-Thread, einem Event eines Hosted Service, einem Callback einer Warteschlange oder einem Timer-Callback.

Push bedeutet, dass der Server ausstehende UI-Updates über eine aktive Verbindung an den Browser ausliefert.

Polling bedeutet, dass der Browser den Server in regelmäßigen Abständen nach Updates fragt.



Typische Fehler

Der häufigste Fehler besteht darin, ein Control aus einem Hintergrund-Thread zu aktualisieren, ohne den richtigen Anwendungskontext beizubehalten. Der zweithäufigste Fehler ist, zu oft zu pushen. Eine ProgressBar braucht keine 500 Updates pro Sekunde. Push ist mächtig, aber jedes Update kostet etwas: Serialisierung, Netzwerkverkehr, Arbeit im Browser und Scheduling auf dem Server.

Ein weiterer typischer Fehler ist die Verwendung statischer Felder für sessionspezifische Daten. Statische Felder werden von allen Benutzern und allen Sessions in der Anwendungsdomäne geteilt. Verwenden Sie für sessionspezifische Objekte `Application.Session`, einen Service pro Session oder `Application.GetInstance()`. Setzen Sie statische Services nur für wirklich gemeinsam genutzte Daten ein, und planen Sie die Nebenläufigkeit dann sorgfältig.



Kontrollpunkt

Am Ende dieses Moduls sollten Sie den vollständigen Weg eines serverseitig erzeugten Ticket-Events skizzieren können: Der Service ändert Daten, der Service löst ein Event aus, eine Session empfängt es, die Session aktualisiert ihre Controls, und Wisej.NET pusht die daraus resultierenden UI-Änderungen, sofern WebSocket verfügbar ist.

Echtzeit-Webarchitektur in Wisej.NET

Was Sie im Praxis-Lab bauen, die Aufgaben, die vorgeschlagene Implementierung und die Abnahmekriterien.



Praxis-Lab

Erstellen Sie die erste Maske von TicketOps Live: eine Page mit einer Live-Statusleiste, einem Uhr-Label, einer Fortschrittsanzeige und einem simulierten Server-Heartbeat, der den Browser ohne Klick des Benutzers aktualisiert.

Ausgangspunkt

Erstellen Sie eine Wisej.NET-Web-Page-Anwendung mit dem Namen `TicketOpsLive`. Fügen Sie eine Hauptseite namens `MainPage` hinzu. Verwenden Sie ein beliebiges integriertes Theme. Das Lab geht von C# aus, die Konzepte gelten aber genauso für VB.NET.

Aufgaben

1. Fügen Sie oben ein Panel namens `statusPanel` hinzu.
2. Fügen Sie die Labels `clockLabel`, `connectionLabel` und `activityLabel` hinzu.
3. Fügen Sie eine ProgressBar namens `serverLoadBar` hinzu.
4. Fügen Sie die Buttons `startButton` und `stopButton` hinzu.
5. Implementieren Sie eine Hintergrundaufgabe, die einen Heartbeat simuliert, wenn auf `startButton` geklickt wird.
6. Der Heartbeat soll einmal pro Sekunde die Uhr, einen fiktiven Lastwert und eine Aktivitätsmeldung aktualisieren.
7. Beenden Sie die Updates, wenn auf `stopButton` geklickt oder die Page verworfen (disposed) wird.

Vorgeschlagene Implementierung

```
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";
}

```

Abnahmekriterien

- ✓ Die UI aktualisiert sich einmal pro Sekunde, nachdem auf „Start“ geklickt wurde.
- ✓ Der Browser zeigt Änderungen ohne weitere Klicks an.
- ✓ „Stop“ beendet die Schleife innerhalb einer Sekunde.
- ✓ Zweimaliges Starten erzeugt keine zwei konkurrierenden Schleifen.
- ✓ Der Code prüft, ob die Page verworfen (disposed) wurde.

Reflexionsfragen

1. Warum ruft dieses Lab `Application.Update(this)` innerhalb der Schleife auf?
2. Was ändert sich, wenn WebSocket nicht verfügbar ist?
3. Warum ist `_running` kein statisches Feld?
4. Welches Risiko entstünde, wenn alle 10 Millisekunden aktualisiert würde?

Anwendungskontext, Sessions und Lebenszyklus

Sessions, Anwendungskontext und Lebenszyklus: die UI des richtigen Benutzers ansprechen, die Falle statischer Felder vermeiden und beim Beenden und beim Timeout aufräumen. Das Konzept, das mentale Modell und die Fallstricke im Produktivbetrieb – lesen Sie das vor dem Lab.



Fokus: Produktivbetrieb

Sessions, Anwendungskontext und Lebenszyklus: die UI des richtigen Benutzers ansprechen, die Falle statischer Felder vermeiden und beim Beenden und beim Timeout aufräumen.

Warum Sessions in Echtzeit-Apps wichtiger sind

Bei einer klassischen Request/Response-Seite klickt der Benutzer, der Server antwortet, und der Großteil der serverseitigen Arbeit ist erledigt. In einer Wisej.NET-Echtzeitanwendung kann die Arbeit nach dem ursprünglichen Request weiterlaufen. Ein Hintergrundjob kann minutenlang laufen, ein Dashboard kann einen globalen Event-Stream abonnieren, ein Timer kann ein Diagramm fortlaufend aktualisieren. Weil die Arbeit den Klick überdauert, der sie gestartet hat, wird der Session-Lebenszyklus Teil der Anwendungsarchitektur.

Eine Wisej.NET-Session ähnelt einer laufenden Instanz einer Desktop-Anwendung. Sie besitzt Controls, Fenster, Pages, Komponentenzustand und Objekte auf Anwendungsebene. Anders als bei einer Desktop-Anwendung können sich Browser und Server jedoch voneinander trennen, der Browser kann neu laden, und es gibt keinen dauerhaft reservierten UI-Thread. Wisej.NET stellt für jeden Request den richtigen Session-Zustand wieder her und kann den serverseitigen Zustand sowohl für HTTP- als auch für WebSocket-Verbindungen verwalten.

Eine Session ist kein Benutzer

Ein angemeldeter Benutzer und eine Session hängen zusammen, sind aber nicht identisch. Ein einzelner Benutzer kann zwei Browser-Tabs öffnen und damit zwei Sessions erzeugen. Ein Benutzer kann sich innerhalb einer Session anmelden, wieder abmelden und die Session weiterlaufen lassen. Ein anonymes Benutzer kann schon vor der Anmeldung eine Session haben. Und ein Hintergrund-Service kann existieren, ohne zu irgendeiner Benutzer-Session zu gehören.

Für Push ist diese Unterscheidung wichtig. Ändert sich ein Ticket, muss die Anwendung womöglich alle Sessions eines Mandanten, eine einzelne Session eines Bearbeiters oder jedes aktive Dashboard benachrichtigen. Gehen Sie nicht davon aus, dass „Benutzer“ und „Browser“ dasselbe sind.

Anwendungskontext

Die Klasse `Application` von Wisej.NET ist statisch, ihre Eigenschaften und Operationen sind aber pro Session isoliert, sofern der Code im richtigen Kontext läuft. Wird Code ausgeführt, weil der Browser ein Event ausgelöst hat, ist der Kontext bereits vorhanden. Läuft Code dagegen in einem Out-of-bound-Thread, weiß er unter Umständen nicht, welche Session er aktualisieren soll.

`Application.StartTask()` ist der einfachste Weg, Hintergrundverarbeitung aus einer Session heraus zu starten, weil die Aufgabe mit dem Kontext der aufrufenden Session gestartet wird. Ist der Code bereits out-of-bound, können Sie `Application.Current` erfassen, solange Sie sich noch im Kontext befinden, und später mit `Application.Update(context, action)` oder `Application.RunInContext(context, action)` Code gegen die richtige Session ausführen.

```
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);
    });
}
```

Der Kontext ist kein Freibrief, Controls für immer am Leben zu halten. Er sollte nur zusammen mit einer klaren Strategie für Abonnement und Aufräumen verwendet werden.

Wo der Zustand liegen sollte

In einer Wisej.NET-Echtzeitanwendung gibt es vier übliche Orte für Zustand.

Control-Zustand ist Zustand, der direkt in Controls gespeichert ist: der Text eines Labels, die ausgewählte Grid-Zeile, ein sichtbares Fenster, der aktuelle Tab oder der Wert einer ProgressBar. Er ist einfach zu handhaben und für Darstellungszustand oft genau richtig.

Application.Session speichert sessionspezifische Werte, die für andere Sessions nicht sichtbar sein sollen. Nutzen Sie es für schlanke Session-Metadaten, den ausgewählten Mandanten, Benutzereinstellungen oder Objekte pro Session, wenn kein eigener Service nötig ist.

Session-Services sind Objekte, die pro Session erzeugt werden. Sie sind nützlich, wenn Sie ein besser testbares und strukturierteres Design brauchen als spontan angelegte Session-Felder. Die sessionbewussten Muster von Wisej.NET helfen, versehentliches statisches Teilen zu vermeiden.

Globale Services werden von allen Sessions gemeinsam genutzt. Sie eignen sich für Warteschlangen, Event-Hubs, Caches, Hintergrund-Watcher oder den systemweiten Status. Sie müssen threadsicher sein und dürfen Controls einer Session nicht ohne definierten Abonnement-Lebenszyklus speichern.

Die Falle statischer Felder

Statische Felder werden geteilt. Speichern Sie `CurrentTicketId` in einem statischen Feld, kann jeder Benutzer das aktuelle Ticket jedes anderen Benutzers überschreiben. Speichern Sie eine `MainPage` - Referenz in einer statischen Liste und entfernen sie nie, hält der Server Sessions womöglich am Leben,

nachdem die Benutzer längst gegangen sind. Speichern Sie eine `CancellationTokenSource` für einen benutzerbezogenen Job in einem statischen Feld, kann ein Benutzer den Job eines anderen abbrechen.

Verwenden Sie statische Felder ausschließlich für wirklich globale Services. Für sessionspezifische Werte verwenden Sie Controls, `Application.Session` oder ein Objekt pro Session.

Neuladen, Keep-Alive und Timeout

Eine Wisej.NET-Anwendung muss mit den Realitäten des Webs umgehen: Benutzer laden den Browser neu, Laptops gehen in den Ruhezustand, Netzwerke verschwinden, und Tabs bleiben über Nacht offen.

Wisej.NET entscheidet anhand von Session-Timeout und Keep-Alive, ob eine Session bestehen bleibt oder beendet wird. Läuft eine Session ab, wird sie aus dem Speicher entfernt. Das entspricht eher dem Beenden einer Desktop-Anwendung als dem Ausblenden einer Webseite.

Für Echtzeit-Apps hat das folgende praktische Konsequenzen:

- Lang laufende Hintergrundverarbeitung darf nicht davon ausgehen, dass die UI noch existiert.
- Code, der die UI aktualisiert, sollte prüfen, ob eine Page bzw. ein Control verworfen (disposed) wurde.
- Abonnements globaler Events sollten entfernt werden, wenn die Session endet oder die Ansicht verworfen wird.
- Muss ein Job nach dem Ende einer Session weiterlaufen, sollte er einem globalen Service gehören, nicht der Page.
- Ist ein Job nur für die UI eines einzelnen Benutzers sinnvoll, sollte er abgebrochen werden, sobald diese UI verschwindet.

Muster zum Aufräumen

Eine Echtzeit-Ansicht, die einen globalen Service abonniert, sollte das Abonnement immer wieder kündigen. Welches Event Sie dafür nutzen, hängt vom Aufbau Ihrer Ansicht ab, das Muster ist aber immer gleich: beim Laden oder im Konstruktor abonnieren, beim Verwerfen oder beim Beenden der Anwendung wieder abmelden.

```

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);
    });
}

```

Es geht nicht nur darum, Exceptions zu vermeiden. Es geht darum, Zuständigkeiten offensichtlich zu machen: Der Feed besitzt die globalen Ticket-Events, die Page besitzt ihre Controls, und die Session besitzt ihren Anwendungskontext.

Fehlerbehandlung in Hintergrundaufgaben

Eine Hintergrundaufgabe darf niemals stillschweigend scheitern. In einer Schulungsdemo stoppt eine Exception vielleicht nur die Schleife. Im Produktivbetrieb führen stille Fehler dazu, dass Dashboards falsche Informationen anzeigen. Verwenden Sie innerhalb der Aufgabe ein try/catch, protokollieren Sie die Exception und zeigen Sie in der UI eine unbedenkliche Fehlermeldung an, sofern die UI noch existiert.

```
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);
        }
    }
});
```

Zeigen Sie Endbenutzern im Produktivbetrieb keine rohen Exception-Details an. Nutzen Sie ausführliche Logs für Entwickler und unbedenkliche Meldungen für Benutzer.



Kontrollpunkt

Am Ende dieses Moduls sollten Sie erklären können, warum eine aus einer Page gestartete Hintergrundaufgabe diese Page aktualisieren kann, warum ein globaler Service Controls einer Page nicht direkt besitzen sollte und warum jedes Abonnement in einer Echtzeit-App einen Plan zur Abmeldung braucht.

Anwendungskontext, Sessions und Lebenszyklus

Was Sie im Praxis-Lab bauen, die Aufgaben, die vorgeschlagene Implementierung und die Abnahmekriterien.



Praxis-Lab

Fügen Sie ein Diagnose-Panel hinzu, in dem Sie als Entwickler die aktuelle Session, den Anwendungskontext, Browserdetails und die für Echtzeit-Updates relevanten Lebenszyklus-Events sehen.

Aufgaben

1. Fügen Sie `MainPage` eine GroupBox für die Diagnose hinzu.
2. Zeigen Sie die aktuelle Uhrzeit, die Client-ID, den Gerätetyp des Browsers (falls verfügbar) und einen sessionspezifischen Zähler an.
3. Speichern Sie den Zähler in `Application.Session` oder in einem Objekt pro Session.
4. Fügen Sie Buttons hinzu, um den Zähler zu erhöhen und ein Hintergrund-Update zu simulieren.
5. Protokollieren Sie das Aufräumen, wenn die Anwendung beendet wird.
6. Laden Sie den Browser neu und beobachten Sie, was erhalten bleibt.

Vorgeschlagene Code-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);
}
```

Abnahmekriterien

- ✓ Der Zähler ist sessionspezifisch, nicht statisch.
- ✓ Ein Hintergrund-Update ändert die richtige Browser-Session.
- ✓ Das Diagnose-Panel zeigt an, wann das Hintergrund-Update eintrifft.
- ✓ Sie können erklären, was nach einem Neuladen und nach einem Session-Timeout passiert.

Reflexionsfragen

1. Warum wird `Application.Current` erfasst, bevor die Out-of-bound-Aktion gestartet wird?
2. Was würde passieren, wenn der Zähler statisch wäre?
3. Welche Aufräumaktion würden Sie für das Abonnement eines globalen Service ergänzen?

Server-Push in einer Session mit Hintergrundaufgaben

Push innerhalb einer Session: lange Arbeit außerhalb des Klick-Handlers ausführen, Fortschritt melden, kooperativen Abbruch unterstützen und die UI im finally-Block wiederherstellen. Das Konzept, das mentale Modell und die Fallstricke im Produktivbetrieb – lesen Sie das vor dem Lab.



Fokus: Produktivbetrieb

Push innerhalb einer Session: lange Arbeit außerhalb des Klick-Handlers ausführen, Fortschritt melden, kooperativen Abbruch unterstützen und die UI im finally-Block wiederherstellen.

Das häufigste Server-Push-Szenario

Der häufigste erste Einsatz von Server-Push ist eine lang laufende Operation. Ein Benutzer startet einen Import, einen Bericht, einen Migrationsschritt, eine Dateierzeugung, eine Datensynchronisation oder eine Validierungsroutine. Die Operation dauert Sekunden oder Minuten. Die UI darf dabei nicht einfrieren, der Benutzer soll den Fortschritt sehen, und die Operation soll am Ende Erfolg oder Fehlschlag melden.

In Desktop-Anwendungen nutzen Entwickler häufig einen BackgroundWorker oder einen Task und leiten Updates zurück an den UI-Thread. In Wisej.NET besteht das entsprechende Design aus Hintergrundverarbeitung im Session-Kontext plus `Application.Update()`, wenn UI-Änderungen an den Browser gepusht werden sollen. Die serverseitige UI gehört weiterhin der Session. Die Hintergrundaufgabe muss Lebenszyklus, Abbruch und Update-Frequenz respektieren.

Das Grundmuster

Das Kernmuster besteht aus sechs Schritten:

1. Die UI vor dem Start der Arbeit deaktivieren oder anpassen.
2. Einen Abbruchzustand anlegen, falls der Benutzer abbrechen kann.
3. Die Hintergrundaufgabe im Session-Kontext starten.
4. Die Arbeit in kleinen Schritten ausführen.
5. Den UI-Zustand in sinnvollen Intervallen aktualisieren.
6. Die UI wieder aktivieren und Abschluss oder Fehler anzeigen.

```

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 ... ";
}

```

Die Methode der Aufgabe sollte ein try/catch/finally enthalten, damit die UI nach einer Exception nicht deaktiviert bleibt.

```

private void RunImport(CancellationToken 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);
}
}

```

Achten Sie auf die Update-Frequenz. Der Code ändert die ProgressBar bei jedem Durchlauf, pusht aber nur alle fünf Durchläufe plus ein letztes Update. Das reduziert die Zahl der Netzwerknachrichten und der Neuzeichnungen im Browser und wirkt trotzdem live.

Direktes UI-Update oder Kontext-Callback

Wird eine Aufgabe mit `Application.StartTask()` aus einem Wisej.NET-Event gestartet, läuft sie mit dem aktuellen Anwendungskontext. In vielen Beispielen sehen Sie direkte Änderungen an Controls, gefolgt von `Application.Update(this)`. Für einen Job innerhalb einer Session, der der Page gehört, ist das in Ordnung.

Für Code, der außerhalb der Session startet, verwenden Sie die Überladung mit explizitem Kontext:

```

var context = Application.Current;

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

```

So ist klar erkennbar, dass die UI-Änderungen im richtigen Session-Kontext stattfinden.

Event-Handler nicht blockieren

Führen Sie eine lange Operation nicht direkt in einem Klick-Handler aus. Der Browser-Request bleibt offen, der Benutzer sieht keine Push-Updates zwischendurch, und der Server-Thread ist viel zu lange belegt. Starten Sie die Operation und geben Sie die Kontrolle an den Benutzer zurück.

Schlecht:

```

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

```

Besser:

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

Abbruch ist ein Feature, kein Nachgedanke

Eine Echtzeit-Fortschrittsanzeige ohne Abbruchmöglichkeit kann schlimmer sein als eine blockierende UI. Sie signalisiert dem Benutzer, dass etwas passiert, gibt ihm aber keine Kontrolle. Bauen Sie den Abbruch überall dort in lang laufende Aufgaben ein, wo er gefahrlos möglich ist. Beenden Sie keine Threads gewaltsam, sondern verwenden Sie kooperativen Abbruch. Prüfen Sie das Token zwischen einzelnen Arbeitseinheiten und hinterlassen Sie das System in einem konsistenten Zustand.

Der Abbruch ist auch nützlich, wenn die Page geschlossen wird. Dient die Hintergrundverarbeitung nur dieser Page, brechen Sie sie beim Verwerfen (Dispose) oder beim Beenden der Anwendung ab. Handelt es sich um einen Serverjob, der weiterlaufen soll, übertragen Sie die Zuständigkeit an einen Service und lassen Sie die Page den Zustand des Service anzeigen.

Fehlermeldung

Eine Hintergrundaufgabe kann fehlschlagen, nachdem der ursprüngliche Event-Handler bereits zurückgekehrt ist. Ohne explizite Fehlerbehandlung sieht der Benutzer womöglich nur, dass der Fortschritt stehen geblieben ist. Fangen Sie Exceptions in der Hintergrundverarbeitung immer ab. Protokollieren Sie die technischen Details. Pushen Sie einen für den Benutzer unbedenklichen Fehlerzustand in die UI. Halten Sie genug Details für die Fehlersuche fest: Job-ID, Session- bzw. Client-ID, Benutzer-ID (falls angemeldet), Name der Operation und Zeitpunkt.

Bündeln und Drosseln

Das menschliche Auge profitiert nicht von beliebig vielen Updates. Die meisten Business-Dashboards wirken mit 1–4 Updates pro Sekunde live. ProgressBars wirken oft bei 5–10 Updates pro Sekunde flüssig, viele Operationen brauchen das aber gar nicht. Hochfrequente Telemetrie sollte vor der Anzeige aggregiert werden.

Praktische Regeln:

- Den Endzustand sofort pushen.
- Sichtbaren Fortschritt in einem kontrollierten Intervall pushen.
- Mehrere Änderungen an Labels oder Grids in einem einzigen Aufruf von `Application.Update()` bündeln.
- Nicht innerhalb enger CPU-Schleifen pushen.
- Lastspitzen aus externen Systemen bevorzugt über eine Warteschlange oder einen Timer zusammenfassen.



Kontrollpunkt

Am Ende dieses Moduls sollten Sie eine lang laufende Operation implementieren können, die sofort startet, den Fortschritt aktualisiert, ohne den Browser zu blockieren, einen Abbruch unterstützt, Fehler meldet und die UI in einem konsistenten Zustand hinterlässt.

Server-Push in einer Session mit Hintergrundaufgaben

Was Sie im Praxis-Lab bauen, die Aufgaben, die vorgeschlagene Implementierung und die Abnahmekriterien.



Praxis-Lab

Fügen Sie TicketOps Live ein realistisches Panel für einen lang laufenden Import hinzu. Der Benutzer kann einen Import starten, den Fortschritt verfolgen, ihn abbrechen und das Ergebnis prüfen.

Anforderungen an die UI

Fügen Sie hinzu:

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

Aufgaben

1. Implementieren Sie eine Importschleife mit 200 simulierten Datensätzen.
2. Pushen Sie den Fortschritt alle 10 Datensätze, nicht bei jedem Datensatz.
3. Fügen Sie Log-Meldungen für den Start, alle 50 Datensätze, den Abbruch, den Fehlerfall und den Abschluss hinzu.
4. Implementieren Sie den Abbruch mit `CancellationTokenSource`.
5. Fügen Sie einen Button oder eine Option für einen absichtlichen Fehler hinzu, der bei Datensatz 87 eine Exception auslöst.
6. Stellen Sie sicher, dass die UI in allen Fällen wieder aktiviert wird.

Vorgeschlagene Datensatzschleife

```
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);
}
}

```

Abnahmekriterien

- ✓ Fortschritts-Updates erscheinen, während die Operation läuft.
- ✓ Der Abbruch funktioniert und hinterlässt eine benutzbare UI.
- ✓ Fehler werden gemeldet, ohne rohe Stacktraces preiszugeben.
- ✓ Das letzte Update erreicht den Browser immer, solange die Session noch existiert.
- ✓ Der Code pusht nicht häufiger als nötig.

Zusatzaufgabe

Vergeben Sie für jeden Import eine `JobId` und nehmen Sie sie in die Log-Meldungen auf. Das Abschlussprojekt nutzt sie, um UI-Events mit den Server-Logs zu verknüpfen.

Timer, Polling-Fallbacks und Aktualisierungstakt

Aktualisierungstakt und Fallback: zwischen Push, Timern und Polling wählen, Lastspitzen zusammenfassen und den Lebenszyklus von Timern pro Session verantworten. Das Konzept, das mentale Modell und die Fallstricke im Produktivbetrieb – lesen Sie das vor dem Lab.



Fokus: Produktivbetrieb

Aktualisierungstakt und Fallback: zwischen Push, Timern und Polling wählen, Lastspitzen zusammenfassen und den Lebenszyklus von Timern pro Session verantworten.

Push ist die erste Wahl, doch der Fallback zählt

Eine Produktivanwendung kann nicht davon ausgehen, dass sich jeder Browser, jeder Proxy, jeder Server und jeder Netzwerkpfad perfekt verhält. Manche Umgebungen blockieren WebSockets. Manche Reverse-Proxys benötigen eine explizite WebSocket-Unterstützung. Manche Firmennetzwerke erzwingen Timeouts. Wisej.NET unterstützt Echtzeit-Push über WebSocket und bietet außerdem Fallbacks wie Polling für Systeme, die kein WebSocket nutzen können.

Als fortgeschrittener Entwickler müssen Sie den Unterschied zwischen einem Fallback und einem Designfehler verstehen. Polling ist nicht automatisch falsch. Falsch ist es, wenn es ein klares Event-Modell ersetzt oder wenn das Intervall ohne Rücksicht auf die Serverkosten gewählt wird. Ein Polling-Intervall von einer Sekunde kann bei Tausenden von Sessions teuer werden. Ein Intervall von zehn Sekunden kann sich bei einem kritischen Betriebs-Board zu langsam anfühlen. Die richtige Wahl hängt vom Nutzen für den Benutzer und von den Kosten ab.

Wisej.Web.Timer oder Hintergrundaufgabe

Ein `Wisej.Web.Timer` ist nützlich, wenn Sie periodische serverseitige Events brauchen, die an eine Session gebunden sind. Ein Dashboard kann zum Beispiel alle fünf Sekunden eine berechnete Kennzahl aktualisieren, auch wenn kein Event eines globalen Service eingetroffen ist. Eine Hintergrundaufgabe ist besser geeignet, wenn eine bestimmte Operation startet, läuft und endet. Ein globaler Service ist besser geeignet, wenn ein gemeinsamer Watcher viele Sessions versorgt.

Verwenden Sie einen Timer für periodische UI-Aktualisierungen im Rahmen einer Session. Verwenden Sie `Application.StartTask()` für lang laufende Arbeit im Rahmen einer Session. Verwenden Sie einen gemeinsamen Service für globalen Zustand und Events für mehrere Benutzer.

Polling-Fallback mit StartPolling und EndPolling

Ist WebSocket verfügbar, kann `Application.Update()` ausstehende Updates an den Browser pushen. Ist WebSocket nicht verfügbar, steht keine aktive Verbindung zur Verfügung. Wisej.NET bietet dafür `Application.StartPolling(milliseconds)` und `Application.EndPolling()` als Fallback-Mechanismen. Bei aktivem WebSocket wird Polling ignoriert. So kann die Anwendung Fallback-Verhalten anfordern, ohne WebSocket-Sessions zu beeinträchtigen.

Ein gängiges Muster sieht so aus:

```

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();
    });
}

```

Auch dieser Code muss sorgfältig entworfen werden. Wird `_liveMode` nie false und die Page verworfen, entsteht mit der Schleife ein Leck. Sehen Sie immer eine Abbruchbedingung vor.

Den Aktualisierungstakt wählen

Der Takt ist der Rhythmus der Updates. In einer Wisej.NET-Echtzeit-App hat er vier Dimensionen:

Modelltakt: wie oft sich die zugrunde liegenden Daten ändern.

UI-Takt: wie oft der Benutzer sichtbare Änderungen sehen muss.

Push-Takt: wie oft der Server UI-Updates an den Browser sendet.

Polling-Takt: wie oft der Browser den Server nach Updates fragt, wenn kein Push verfügbar ist.

Diese Takte sind nicht immer gleich. Eine Maschine liefert vielleicht 100-mal pro Sekunde Telemetrie, die UI braucht aber nur einmal pro Sekunde eine Zusammenfassung. Eine Warteschlange ändert sich vielleicht 20-mal in einem Schub, das Dashboard kann aber nach 250 Millisekunden den Endzustand anzeigen.

Lastspitzen zusammenfassen

Eine Lastspitze entsteht, wenn viele Änderungen auf einmal eintreffen. Löst jede Änderung einen eigenen Push aus, verschwendet die UI womöglich Ressourcen und zeigt Zwischenzustände, mit denen Benutzer nichts anfangen können. Zusammenfassen (Coalescing) bedeutet, mehrere Änderungen zu sammeln und gemeinsam anzuwenden.

Eine einfache Strategie zum Zusammenfassen:

```

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();
}

```

Das Event markiert das Modell als geändert („dirty“). Der Timer wendet die Updates in einem stabilen UI-Takt an.

Zuständigkeit für Timer

Ein Timer gehört immer irgendwohin. Ist er eine Wisej.NET-Komponente auf einer Page, gehört er der Page. Ist er ein .NET-Timer in einem globalen Service, gehört er dem Service. Die Zuständigkeit bestimmt das Aufräumen. Ein doppelt angelegter Timer kann den Update-Verkehr verdoppeln und Bugs scheinbar zufällig auftreten lassen.

Praktische Regeln:

- Timer nur einmal starten.
- Timer stoppen, wenn die Ansicht nicht mehr aktiv ist.
- Nicht bei jedem Klick auf einen Aktualisieren-Button einen neuen Timer erzeugen.
- Keine lang laufende Arbeit in einem Timer-Tick ausführen, sondern die Arbeit starten oder einplanen.
- Mit einem Flag überlappende Ticks verhindern, falls eine Aktualisierung länger dauert als das Intervall.

Einen ungeeigneten Takt erkennen

Anzeichen für zu viele Updates sind hohe CPU-Last, ruckelndes Rendering im Browser, viel WebSocket-Verkehr, Rauschen in den Server-Logs und Dashboards, die zu flackern scheinen. Anzeichen für zu wenige Updates sind Benutzer, die ständig auf „Aktualisieren“ klicken, doppelte Arbeit, betriebliche Entscheidungen auf Basis veralteter Daten und mangelndes Vertrauen in das Dashboard.

Fortgeschrittene Entwickler sollten wichtige Dashboards instrumentieren: Zählen Sie empfangene Events, angewendete UI-Updates, Push-Versuche und Fehler, und messen Sie die durchschnittliche Zeit für das Anwenden eines Updates.



Kontrollpunkt

Am Ende dieses Moduls sollten Sie je nach Aufgabe zwischen Timer, Hintergrundaufgabe, Polling-Fallback und Service-Event wählen und die Kosten des gewählten Takts erklären können.

Timer, Polling-Fallbacks und Aktualisierungstakt

Was Sie im Praxis-Lab bauen, die Aufgaben, die vorgeschlagene Implementierung und die Abnahmekriterien.



Praxis-Lab

Erweitern Sie TicketOps Live um einen Timer zur Aktualisierung des Dashboards und um Controls, mit denen Sie mit dem Aktualisierungstakt experimentieren können.

Anforderungen an die UI

Fügen Sie hinzu:

- ✓ `liveModeCheckBox`
- ✓ `cadenceComboBox` mit den Werten `250 ms`, `1 sec`, `5 sec`
- ✓ `eventsReceivedLabel`
- ✓ `updatesAppliedLabel`
- ✓ `lastAppliedLabel`
- ✓ `refreshTimer`

Aufgaben

1. Starten Sie das Polling, wenn der Live-Modus aktiviert wird.
2. Beenden Sie das Polling, wenn der Live-Modus deaktiviert wird.
3. Verwenden Sie einen Timer, um Dashboard-Updates im ausgewählten Takt anzuwenden.
4. Simulieren Sie Modelländerungen, die schneller eintreffen als die UI-Updates.
5. Zählen Sie Modell-Events und angewendete UI-Updates getrennt.
6. Verhindern Sie überlappende Timer-Ticks.

Vorschlag für die Implementierung

```
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;
    }
}

```

Abnahmekriterien

- ✓ Der Takt des Timers ändert sich sichtbar.
- ✓ Die Zahl der empfangenen Events kann schneller wachsen als die Zahl der angewendeten UI-Updates.
- ✓ Das Polling startet und stoppt mit dem Live-Modus.
- ✓ Es treten keine überlappenden Timer-Ticks auf.
- ✓ Sie können erklären, warum das Zusammenfassen von Updates nützlich ist.

Live-Data-Binding und Echtzeit-Grids

Live-Data-Binding: gebundene Collections im Session-Kontext aktualisieren, ohne das Grid neu zu binden, und dabei Auswahl, Sortierung und Scrollposition erhalten. Das Konzept, das mentale Modell und die Fallstricke im Produktivbetrieb – lesen Sie das vor dem Lab.



Fokus: Produktivbetrieb

Live-Data-Binding: gebundene Collections im Session-Kontext aktualisieren, ohne das Grid neu zu binden, und dabei Auswahl, Sortierung und Scrollposition erhalten.

Warum Data-Binding schwieriger wird, wenn Daten in Bewegung sind

Statisches Data-Binding ist einfach: Datensätze laden, das Grid binden, den Benutzer bearbeiten lassen, speichern. Echtzeit-Data-Binding bringt Bewegung ins Spiel. Zeilen kommen hinzu, Felder ändern sich, Status schreiten voran, und Hintergrundjobs aktualisieren genau die Maske, die der Benutzer gerade ansieht. Die Herausforderung besteht darin, die UI synchron zu halten, ohne Bindings ständig zu zerstören und neu zu erzeugen.

Wisej.NET unterstützt einfaches Binding für einzelne Control-Eigenschaften und komplexes Binding für Controls wie Grids. Die Komponente `BindingSource` ist dabei besonders nützlich: Sie sitzt zwischen Controls und Datenquellen, verwaltet die aktuelle Position und bietet eine zentrale Stelle, um Bindings zurückzusetzen oder zu aktualisieren.

Das Ticket-Modell

Verwenden Sie in diesem Kurs ein einfaches Ticket-Modell:

```
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));
    }
}

```

Der Zweck von `INotifyPropertyChanged` ist, Eigenschaftsänderungen für die Binding-Infrastruktur sichtbar zu machen, wo dies unterstützt wird. Auch wenn Sie der Klarheit halber `BindingSource.ResetBindings(false)` aufrufen, drückt das Modell weiterhin aus, dass es Änderungen melden kann.

Eine Live-Liste binden

Ein praxistauglicher Ausgangspunkt ist `BindingList<Ticket>` zusammen mit `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;
}

```

Trifft ein neues Ticket ein, fügen Sie es im richtigen Session-Kontext der Liste hinzu. Ändert sich ein Feld, aktualisieren Sie das Element und setzen die Bindings zurück, falls sich die UI nicht automatisch aktualisiert.

```
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);
}
```

Das Grid nicht bei jedem Update neu binden

Das gesamte Grid neu zu binden ist manchmal der schnellste Weg, eine Demo zum Laufen zu bringen, im Produktivbetrieb aber oft falsch. Dabei können Auswahl, Scrollposition, Sortierung, Spaltenbreiten und der Arbeitskontext des Benutzers verloren gehen, und es kann teuer sein. Bevorzugen Sie inkrementelle Änderungen an der gebundenen Liste und benachrichtigen Sie anschließend die BindingSource.

Schlecht:

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

Besser:

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

Am besten für Masken mit hohem Datenaufkommen: mehrere Änderungen bündeln und nur einmal pro Taktintervall aktualisieren.

Threadsicherheit bei gebundenen Collections

Eine gebundene Collection gehört zur UI der Session. Ein globaler Service empfängt Updates womöglich aus einem anderen Thread. Ändern Sie die gebundene Collection nicht aus beliebigen Threads heraus, ohne in den Session-Kontext zu wechseln und die Synchronisation zu bedenken.

Ein sicheres Muster für mehrere Benutzer sieht so aus:

1. Der globale Service empfängt ein Ticket-Event.
2. Der Event-Handler der Session nimmt das Event entgegen.
3. Der Handler verwendet `Application.Update(context, () => { ... })`.

4. Innerhalb des Callbacks wird die gebundene Liste der Session aktualisiert.
5. Die Bindings werden einmal zurückgesetzt.

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

Formatieren und Parsen

Bei Echtzeit-Grids geht es nicht nur um den Datentransport – sie müssen lesbar bleiben. Formatieren Sie Datumswerte, Währungen, Prioritäten und Status. Nutzen Sie bei gebundenen Controls die Parse- und Format-Events, wenn Werte konvertiert werden müssen. Speichern Sie keine formatierten Strings in Ihrem Modell, wenn der eigentliche Wert einen aussagekräftigeren Typ hat.

So sollte `UpdatedAt` beispielsweise ein `DateTime` sein, kein vorformatierter String. Das Grid kann den Wert dann als Ortszeit oder als relatives Alter anzeigen.

Zeilenstatus und visuelle Hinweise

Ein Live-Grid sollte zeigen, was sich geändert hat, ohne unruhig zu wirken. Gängige Muster:

- für einige Sekunden eine temporäre Markierung „aktualisiert“ anzeigen;
- neue Tickets oben einsortieren;
- das vom Benutzer ausgewählte Ticket nach Möglichkeit stabil halten;
- eine Zeile nicht verschieben, während der Benutzer sie bearbeitet;
- Statusspalten und kleine Indikatoren statt modaler Unterbrechungen verwenden.

Die UI sollte Benutzern helfen, den Daten zu vertrauen. Zu viele Animationen oder springende Zeilen können den Eindruck erwecken, die Anwendung sei instabil.

Validierung und Konfliktbehandlung

Echtzeitdaten bedeuten, dass ein anderer Benutzer oder ein Service einen Datensatz ändern kann, während der aktuelle Benutzer ihn ansieht oder bearbeitet. Legen Sie fest, was passiert, wenn sich ein Ticket während der Bearbeitung ändert. In diesem Kurs gilt eine einfache Regel: Wird ein Ticket in einem Detailformular bearbeitet, zeigen Sie einen nicht blockierenden Hinweis an, dass sich das Ticket geändert hat, und lassen den Benutzer neu laden oder weiterarbeiten.

Produktivsysteme benötigen unter Umständen optimistische Nebenläufigkeit mit Versionsnummern, Zeitstempeln oder Zeilenversionen in der Datenbank. Wisej.NET macht Nebenläufigkeitskontrolle in der Datenschicht nicht überflüssig. Es gibt der UI aber die Möglichkeit, Benutzer im richtigen Moment zu informieren.



Kontrollpunkt

Am Ende dieses Moduls sollten Sie ein Live-Grid pflegen können, das neue Zeilen und Zeilen-Updates aus einer Hintergrundquelle erhält, dabei den Kontext des Benutzers bewahrt und unnötiges Neu-Binden vermeidet.

Live-Data-Binding und Echtzeit-Grids

Was Sie im Praxis-Lab bauen, die Aufgaben, die vorgeschlagene Implementierung und die Abnahmekriterien.



Praxis-Lab

Erstellen Sie eine Live- `DataGridView` , die Support-Tickets anzeigt und sich aktualisiert, wenn simulierte Ticket-Events eintreffen.

Aufgaben

1. Fügen Sie eine `DataGridView` namens `ticketsGrid` hinzu.
2. Erstellen Sie das Modell `Ticket` und die Enumeration `TicketStatus` .
3. Binden Sie eine `BindingList<Ticket>` über eine `BindingSource` .
4. Fügen Sie einen Button hinzu, der 20 Sekunden lang jede Sekunde ein neues Ticket-Event erzeugt.
5. Fügen Sie einen Button hinzu, der die Status von Tickets zufällig ändert.
6. Erhalten Sie nach Möglichkeit die ausgewählte Zeile.
7. Zeigen Sie `UpdatedAt` in einem gut lesbaren Format an.

Vorgeschlagenes Modell-Event

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

Vorgeschlagene Update-Methode

```
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);
}
```

Abnahmekriterien

- ✓ Neue Tickets erscheinen, ohne dass das Grid von Grund auf neu gebunden wird.
- ✓ Bestehende Tickets werden an Ort und Stelle aktualisiert.
- ✓ Die ausgewählte Zeile bleibt erhalten, solange die Zeile noch existiert.
- ✓ Die Update-Rate bleibt kontrolliert.
- ✓ Sie können erklären, warum die gebundene Liste der Session gehört.

Zusatzaufgabe

Fügen Sie einen Filter für Tickets mit dem Status `Escalated` hinzu und stellen Sie sicher, dass Live-Updates auch bei aktivem Filter funktionieren.

Push an mehrere Benutzer mit Services und Event-Hubs

Push an mehrere Benutzer: Ein globaler Event-Hub löst Domain-Events aus, während jede Session ihre eigene UI in ihrem eigenen Kontext aktualisiert. Das Konzept, das mentale Modell und die Fallstricke im Produktivbetrieb – lesen Sie das vor dem Lab.



Fokus: Produktivbetrieb

Push an mehrere Benutzer: Ein globaler Event-Hub löst Domain-Events aus, während jede Session ihre eigene UI in ihrem eigenen Kontext aktualisiert.

Von einer Session zu vielen Sessions

Push innerhalb einer Session ist nützlich, doch viele Geschäftsanwendungen brauchen Push an mehrere Benutzer. Weist ein Disponent ein Ticket zu, sollte sich das Dashboard jeder Teamleitung aktualisieren. Ist ein nächtlicher Import abgeschlossen, sollten alle verbundenen Administratoren das Ergebnis sehen. Schlägt ein Health-Check eines Service fehl, sollten die aktiven Betriebskonsolen den Alarm anzeigen.

Die entscheidende Änderung an der Architektur betrifft die Zuständigkeit. Eine Page besitzt ihre Controls. Eine Session besitzt ihren Kontext. Ein globaler Service besitzt gemeinsame Daten und Events. Lassen Sie den globalen Service nicht zu einer Sammelstelle für Formularreferenzen werden. Verwenden Sie stattdessen Domain-Events und Abonnements auf Session-Ebene.

Das Event-Hub-Muster

Ein Event-Hub ist ein gemeinsam genutzter Service, der Domain-Events empfängt oder erzeugt und sie Abonnenten zur Verfügung stellt. Der Hub weiß nicht, wie jede einzelne UI das Event darstellt. Er meldet lediglich, dass etwas passiert ist.

```
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);
}
}

```

Der Hub hält den globalen Zustand threadsicher und löst ein Domain-Event aus. Er ruft `Application.Update()` nicht auf, weil ihm keine Session gehört.

Das Muster für Session-Abonnenten

Jede Session abonniert den Hub und aktualisiert ihre eigene 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;
}

```

Mit diesem Muster hat jede Session ihre eigene UI. Zwei Benutzer können unterschiedlich filtern oder sortieren und trotzdem dieselben Domain-Events empfangen.

Broadcast oder gezieltes Update

Broadcast bedeutet, dass jeder Abonnent das Event erhält. Ein gezieltes Update bedeutet, dass nur bestimmte Sessions es erhalten. Beispiele:

- Alle schwerwiegenden Systemalarme per Broadcast an alle Betriebskonsolen senden.
- Mandantenspezifische Events nur an Sessions dieses Mandanten senden.
- Benachrichtigungen über Zuweisungen nur an den zugewiesenen Bearbeiter senden.
- Admin-Events nur an Benutzer mit Administratorrolle senden.

Der Hub kann genügend Metadaten mitliefern, damit der Abonnent entscheiden kann, ob er das Event anzeigt. Alternativ kann der Hub Abonnementgruppen verwalten. Halten Sie die erste Version einfach: Liefern Sie Metadaten zu Mandant, Benutzer und Rolle mit und lassen Sie die Session filtern.

Threadsicherheit und Snapshots

Globale Services verwalten oft Collections. Diese sollten durch Locks oder nebenläufige Collections geschützt sein. Wenn Sie Daten an Sessions herausgeben, bevorzugen Sie Snapshots gegenüber direkten

Referenzen auf die Collection. Ein Snapshot verhindert, dass UI-Code versehentlich den globalen Zustand ändert, und vermeidet eine Enumeration, während der Service die Collection verändert.

Schlecht:

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

Besser:

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

Das Klonen muss nicht in jedem Anwendungsfall teuer sein, aber die Grenze der Zuständigkeit muss klar sein.

Speicherlecks durch Events

Events sind eine häufige Ursache für Speicherlecks. Hat ein globaler Service ein Event und abonniert eine Page dieses Event, hält der Service nun eine Referenz auf den Handler der Page. Wird die Page geschlossen, meldet sich aber nie ab, hält der Service sie womöglich am Leben. In Echtzeit-Apps ist dieses Leck sowohl ein Speicher- als auch ein Korrektheitsproblem, weil der Service versuchen kann, eine nicht mehr existierende Session zu aktualisieren.

Zu jedem Abonnement gehört eine Abmeldung. Das ist eine feste Regel dieses Kurses.

Backpressure

Sendet der Hub Änderungen schneller, als die Sessions sie darstellen können, braucht das System Backpressure oder Zusammenfassung. Einfache Strategien sind:

- doppelte Events mit niedriger Priorität verwerfen;
- Änderungen bündeln und Zusammenfassungen senden;
- eine Session als „dirty“ markieren und per Timer aktualisieren;
- für eine Dashboard-Kennzahl nur den neuesten Zustand vorhalten;
- Events pro Session in einer Warteschlange mit Maximalgröße puffern.

Geschäftsanwendungen müssen in der Regel nicht jeden Zwischenzustand darstellen. Sie müssen den Zustand darstellen, auf dessen Grundlage Benutzer handeln können.



Kontrollpunkt

Am Ende dieses Moduls sollten Sie einen gemeinsam genutzten Service entwerfen können, der viele WISEJ.NET-Sessions sicher versorgt, ohne Controls von Pages global zu speichern, Event-Handler als Lecks zurückzulassen oder gebundene UI-Collections aus dem falschen Thread zu verändern.

Push an mehrere Benutzer mit Services und Event-Hubs

Was Sie im Praxis-Lab bauen, die Aufgaben, die vorgeschlagene Implementierung und die Abnahmekriterien.



Praxis-Lab

Überführen Sie die Ticket-Simulation per Refactoring in einen globalen `TicketHub` -Service und sorgen Sie dafür, dass mehrere Browser-Sessions Live-Ticket-Events empfangen.

Aufgaben

1. Erstellen Sie einen Singleton `TicketHub` mit einer threadsicheren Ticketliste.
2. Fügen Sie `TicketChanged` -Events hinzu.
3. Laden Sie beim Start von `MainPage` einen Snapshot.
4. Lassen Sie die Page `TicketChanged` abonnieren.
5. Melden Sie das Abonnement beim Beenden der Anwendung ab.
6. Öffnen Sie zwei Browser oder zwei Sessions und prüfen Sie, dass beide Updates erhalten.
7. Ergänzen Sie Metadaten zu Mandant oder Rolle und filtern Sie mindestens einen Event-Typ.

Abnahmekriterien

- ✓ Der Hub speichert keine Referenzen auf Pages oder Controls.
- ✓ Jede Session aktualisiert ihre eigene gebundene Liste.
- ✓ Das Schließen oder Neuladen einer Session hinterlässt keinen defekten Abonnenten.
- ✓ Mehrere Sessions können unterschiedliche Filter anzeigen und dabei dieselbe Quelle für Domain-Events nutzen.
- ✓ Sie können den Unterschied zwischen Broadcast und gezieltem Update erklären.

Zusatzaufgabe

Fügen Sie einen Benachrichtigungszähler pro Session hinzu. Der Zähler soll sessionspezifisch sein, obwohl die Events von einem globalen Service stammen.

Härtung für den Produktivbetrieb, Deployment und Abschlussprojekt

Härtung für den Produktivbetrieb: vor dem Deployment den gesamten WebSocket-Pfad, Sticky Sessions, Health-Checks, Sicherheit und Beobachtbarkeit prüfen. Das Konzept, das mentale Modell und die Fallstricke im Produktivbetrieb – lesen Sie das vor dem Lab.



Fokus: Produktivbetrieb

Härtung für den Produktivbetrieb: vor dem Deployment den gesamten WebSocket-Pfad, Sticky Sessions, Health-Checks, Sicherheit und Beobachtbarkeit prüfen.

Echtzeit im Produktivbetrieb ist Architektur plus Betrieb

Eine Echtzeit-App kann auf dem Entwicklerrechner einwandfrei funktionieren und im Produktivbetrieb scheitern, weil der Deployment-Pfad ein anderer ist. WebSocket wird womöglich von einem Proxy blockiert. Ein Load Balancer leitet die Requests eines Benutzers womöglich an verschiedene Serverinstanzen weiter. Ein Session-Timeout kann lang laufende UI-Arbeit beenden. Ein globaler Service kann zu viel Speicher belegen. Eine Sicherheitseinstellung kann in der Entwicklung unbedenklich, hinter einem Reverse-Proxy aber falsch sein.

Dieses Modul überführt die Muster des Kurses in eine Checkliste für den Produktivbetrieb.

WebSocket-fähiges Deployment

Echtzeit-Push mit Wisej.NET funktioniert am besten, wenn WebSocket verfügbar ist. Prüfen Sie im Produktivbetrieb die WebSocket-Unterstützung über den gesamten Pfad: Browser, Netzwerk, Reverse-Proxy, Load Balancer, Webserver und Anwendung. Manche Load Balancer benötigen TCP-Routing oder eine explizite WebSocket-Unterstützung. Reverse-Proxys wie NGINX können WebSocket weiterleiten, müssen dafür aber korrekt konfiguriert sein.

Tests nur von der Serverkonsole aus reichen nicht. Testen Sie über den Netzwerkpfad der Benutzer. Ein Firmen-VPN, eine Security-Appliance oder ein älterer Proxy kann das Verhalten verändern.

Lastverteilung und Session-Affinität

Wisej.NET-Anwendungen halten den Session-Zustand auf dem Server. In einem Deployment mit mehreren Servern müssen die Requests einer Session daher stets dieselbe Serverinstanz erreichen, sofern die Architektur nicht ausdrücklich eine andere Strategie unterstützt. Die praktische Regel für die Lastverteilung lautet: Verwenden Sie Sticky Sessions oder einen anderen Routing-Mechanismus, der die Zuordnung von Benutzer zu Server konsistent hält.

Health-Checks sollten dem Load Balancer mitteilen, wann ein Server keinen Verkehr mehr erhalten soll. Wisej.NET unterstützt die Konfiguration von Health-Checks über eine Datei `HealthCheck.json` mit Parametern wie maximalem Speicher, maximaler Anzahl von Sessions und CPU-Schwellenwert. Die genauen Schwellenwerte hängen vom Verhalten der Anwendung und von der Größe des Hostings ab.

Wichtige Konfigurationsbereiche

Die Konfiguration verteilt sich auf `Web.config` und die JSON-Konfigurationsdateien von Wisej.NET.

Wisej.NET-Anwendungen verwenden JSON-Anwendungsdateien wie `Default.json`, und ein Projekt kann mehrere Wisej.NET-Anwendungen mit jeweils eigenen Konfigurationsdateien definieren.

Zu den wichtigen Echtzeit-Einstellungen und Deployment-Themen gehören:

- `sessionTimeout` : steuert, wann inaktive Sessions ablaufen;
- `pollingInterval` : steuert das automatische Polling für Clients ohne WebSocket;
- `responseTimeout` : relevant für lange initiale HTTP-Antworten, während WebSocket nicht auf die gleiche Weise in ein Timeout läuft;
- `debug=false` im Produktivbetrieb, damit JavaScript-Bibliotheken passend gebündelt bzw. minifiziert werden;
- Lizenzschlüssel für den Server und Einstellungen für das Standard-Theme;
- IIS-Module und -Handler für Apps auf .NET Framework;
- ASP.NET-Core-Hostingmodell und Middleware für Apps auf .NET Core bzw. .NET;
- Health-Check-Einstellungen für Deployments mit Lastverteilung.

Beispiel für ein `Default.json`-Fragment einer Schulungs-App:

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

Übernehmen Sie Werte aus der Schulung nicht blind in den Produktivbetrieb. Wählen Sie die Einstellungen auf Basis von Lasttests und Benutzeranforderungen.

Sicherheitsaspekte für Echtzeit-Apps

Wisej.NET vermeidet viele klassische Probleme durch das Zusammensetzen von HTML-Strings, weil es Server-Controls und Browser-Widgets synchronisiert, statt Seiten aus beliebigen HTML-Strings aufzubauen. Dennoch können Entwickler ein XSS-Risiko schaffen, wenn sie vom Benutzer stammende Inhalte als HTML anzeigen. Controls und Meldungen, die HTML zulassen, müssen sorgfältig behandelt werden. Halten Sie von Benutzern eingegebene Tickettitel, Kommentare, Namen und Nachrichten kodiert, sofern es keinen klaren Bereinigungsprozess gibt.

Hinter einem Reverse-Proxy oder Load Balancer muss auch der Umgang mit der Client-IP bewusst gestaltet sein. Forwarded-Headern sollte nur vertraut werden, wenn die App hinter einem vertrauenswürdigen Proxy läuft, der eingehende Header bereinigt. Andernfalls kann ein Angreifer Header-Werte fälschen.

Echtzeit-Apps sollten außerdem die Serverressourcen schützen. Setzen Sie Grenzen für Sessions, Warteschlangen und Event-Raten. Lassen Sie nicht zu, dass ein einzelner Benutzer unbegrenzt viele Hintergrundjobs oder Abonnements erzeugt.

Beobachtbarkeit

Eine Echtzeit-App braucht Logs und Metriken, die erklären, was nach dem Ende des ursprünglichen Benutzer-Requests passiert ist. Nehmen Sie Korrelationsfelder auf:

- Job-ID;
- Benutzer-ID oder Kennung einer anonymen Session;
- Client-ID bzw. Session-ID, wo sinnvoll;
- Name der Operation;
- Event-Typ;
- Serverinstanz;
- Dauer und Ergebnis;
- Anzahl der gepushten oder angewendeten UI-Updates.

In TicketOps Live sollte jeder Importjob eine Job-ID haben und jedes Hub-Event eine Event-ID. Die UI kann einen vereinfachten Status anzeigen, die Logs sollten es dem Support aber ermöglichen, den Ablauf zu rekonstruieren.

Performance-Checkliste

Gehen Sie vor dem Deployment diese Fragen durch:

1. Hat jede Hintergrundschleife eine Abbruchbedingung?
2. Wird jedes Abonnement eines globalen Events wieder abgemeldet?
3. Werden UI-Updates gebündelt oder gedrosselt?
4. Werden hochfrequente Modell-Events zusammengefasst?
5. Werden gebundene Collections im Session-Kontext aktualisiert?
6. Werden statische Felder nur für globalen Zustand verwendet?
7. Sind sessionspezifische Zähler, Filter und Auswahlen pro Session angelegt?
8. Werden Exceptions in Hintergrundaufgaben abfangen und protokolliert?
9. Verhält sich die App akzeptabel, wenn WebSocket nicht verfügbar ist?
10. Hält der Load Balancer eine Session auf derselben Serverinstanz?

Überblick über das Abschlussprojekt

Das Abschlussprojekt besteht aus einem Produktiv-Review und der Fertigstellung von TicketOps Live. Sie müssen Folgendes demonstrieren:

- Live-Status-Updates;
- Fortschritt eines Hintergrundimports mit Abbruch und Fehlerbehandlung;
- Live-Updates im Ticket-Grid;
- Ticket-Benachrichtigungen für mehrere Benutzer;
- Polling-Fallback-Strategie;
- Aufräumen von Sessions;

- Checkliste für Deployment und Beobachtbarkeit.

Das Abschlussprojekt wird nicht nach der Anzahl der Controls auf der Maske bewertet, sondern nach der Architektur: Zuständigkeit, Kontext, Lebenszyklus, Aktualisierungstakt und Produktivreife.



Kontrollpunkt

Am Ende dieses Moduls sollten Sie erklären können, wie eine Wisej.NET-Echtzeit-App von der Demo zur auslieferbaren Anwendung wird: WebSocket-Pfad geprüft, Session-Routing stabil, Fallback definiert, Abonnements aufgeräumt, Updates gedrosselt, Daten geschützt und der Betrieb beobachtbar.

Härtung für den Produktivbetrieb, Deployment und Abschlussprojekt

Was Sie im Praxis-Lab bauen, die Aufgaben, die vorgeschlagene Implementierung und die Abnahmekriterien.



Praxis-Lab

Stellen Sie TicketOps Live fertig und bereiten Sie die Anwendung auf ein Architektur-Review für den Produktivbetrieb vor.

Geforderte Funktionen

1. Live-Statusleiste aus M01.
2. Session-Diagnose aus M02.
3. Monitor für Hintergrundimporte aus M03.
4. Steuerung von Takt und Polling aus M04.
5. Live-Ticketboard aus M05.
6. **TicketHub** für mehrere Benutzer aus M06.
7. Checkliste für den Produktivbetrieb und Konfigurations-Review aus M07.

Ergebnis: Architekturnotiz

Erstellen Sie eine einseitige Architekturnotiz mit folgenden Punkten:

- ✓ welcher Zustand sessionspezifisch ist;
- ✓ welcher Zustand global ist;
- ✓ wo Hintergrundverarbeitung startet;
- ✓ wie UI-Updates den Browser erreichen;
- ✓ wie der Abbruch funktioniert;
- ✓ wie Abonnements entfernt werden;
- ✓ was passiert, wenn WebSocket nicht verfügbar ist;
- ✓ was die Deployment-Umgebung unterstützen muss.

Demo-Ablauf

1. Öffnen Sie TicketOps Live in zwei Browser-Sessions.
2. Starten Sie den Heartbeat.
3. Starten Sie einen Import und zeigen Sie den Fortschritt.
4. Brechen Sie einen Import ab.
5. Starten Sie einen weiteren Import und erzwingen Sie einen simulierten Fehler.
6. Erzeugen Sie neue Ticket-Events und zeigen Sie, dass sich beide Sessions aktualisieren.

7. Wenden Sie in einer Session einen Filter an und zeigen Sie, dass die andere Session davon unabhängig ist.
8. Deaktivieren Sie den Live-Modus und erklären Sie das Verhalten von Polling und Takt.
9. Präsentieren Sie die Checkliste für den Produktivbetrieb.

Abnahmekriterien

- ✓ Die Anwendung läuft ohne doppelte Schleifen oder doppelte Abonnements.
- ✓ Die UI bleibt während der Hintergrundverarbeitung reaktionsfähig.
- ✓ Jede lang laufende Operation hat die Zustände Abschluss, Abbruch und Fehler.
- ✓ Der Hub für mehrere Benutzer hält keine direkten Referenzen auf Pages oder Controls.
- ✓ Gebundene Grids werden inkrementell aktualisiert.
- ✓ Sie können alle Annahmen zum Deployment begründet vertreten.