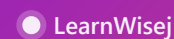


App in tempo reale con server push — Libro delle lezioni

L'insieme completo delle letture dei moduli di questo corso, in un unico documento.



Indice

01	Architettura web in tempo reale in Wisej.NET	Modulo 1 · Guida della lezione
02	Architettura web in tempo reale in Wisej.NET	Modulo 1 · Guida al lab / esame
03	Contesto applicativo, sessioni e ciclo di vita	Modulo 2 · Guida della lezione
04	Contesto applicativo, sessioni e ciclo di vita	Modulo 2 · Guida al lab / esame
05	Server push per una singola sessione con attività in background	Modulo 3 · Guida della lezione
06	Server push per una singola sessione con attività in background	Modulo 3 · Guida al lab / esame
07	Timer, polling di riserva e cadenza degli aggiornamenti	Modulo 4 · Guida della lezione
08	Timer, polling di riserva e cadenza degli aggiornamenti	Modulo 4 · Guida al lab / esame
09	Data binding dal vivo e griglie in tempo reale	Modulo 5 · Guida della lezione
10	Data binding dal vivo e griglie in tempo reale	Modulo 5 · Guida al lab / esame
11	Push multiutente con servizi ed event hub	Modulo 6 · Guida della lezione
12	Push multiutente con servizi ed event hub	Modulo 6 · Guida al lab / esame
13	Irrobustimento per la produzione, rilascio e capstone	Modulo 7 · Guida della lezione
14	Irrobustimento per la produzione, rilascio e capstone	Modulo 7 · Guida al lab / esame

Architettura web in tempo reale in Wisej.NET

L'architettura in tempo reale guidata dal server: push via WebSocket, aggiornamenti nella richiesta e polling di riserva nel modello lato server di Wisej.NET. Il concetto, il modello mentale e le insidie in produzione — leggilo prima del lab.



Focus di produzione

L'architettura in tempo reale guidata dal server: push via WebSocket, aggiornamenti nella richiesta e polling di riserva nel modello lato server di Wisej.NET.

Perché esiste questo modulo

Le applicazioni web in tempo reale si associano di solito a chat, giochi, quotazioni di borsa, schermate di monitoraggio e dashboard. Nei sistemi gestionali la stessa esigenza compare in forme meno spettacolari ma più preziose: un supervisore vede i nuovi ticket appena arrivano, un operatore di produzione vede una macchina fermarsi, un dispatcher vede che un intervento è stato riassegnato, oppure un addetto alla fatturazione vede che un'importazione è ancora in corso senza premere Aggiorna.

Wisej.NET è costruito attorno a un modello applicativo lato server. Lo sviluppatore lavora in C# o VB.NET, usa controlli, eventi, form, pagine, timer, data binding e il designer di Visual Studio, mentre Wisej.NET mantiene l'albero dei widget nel browser e lo sincronizza con lo stato lato server. Questo modello è particolarmente importante per il server push, perché lo stato dell'applicazione è già in mano al server. Un'attività in background può cambiare una label, aggiungere una riga a una griglia, aggiornare una progress bar o aprire una notifica, e Wisej.NET può consegnare al browser le modifiche della UI in sospeso quando esiste una connessione WebSocket attiva.

Un corso per principianti insegna di solito a creare pagine, finestre, controlli, menu e semplici gestori di evento. Questo corso intermedio dà per scontate queste competenze e si concentra sulle domande di progettazione per la produzione che emergono quando la UI si aggiorna da sola. La domanda centrale non è "Posso aggiornare una label da un thread?". La domanda centrale è "Come aggiorni la sessione dell'utente giusto, alla frequenza giusta, dal servizio giusto, senza perdere stato, senza memory leak, senza bloccare il server e senza sommergere il browser?"

Che cosa significa tempo reale in questo corso

In questo corso, un'app Wisej.NET in tempo reale è un'applicazione che riceve gli eventi dell'utente con un ritardo minimo e che può anche aggiornare il browser dell'utente dal server, senza aspettare un altro clic su un pulsante. La parte importante è il server push. L'applicazione deve poter cambiare la UI visibile perché è successo qualcosa sul server: un job in background è andato avanti, una coda è cambiata, è arrivato un messaggio o un altro utente ha compiuto un'azione.

Non ridurre il tempo reale a un "refresh veloce". Una griglia che si aggiorna ogni 30 secondi a volte è accettabile, ma non è la stessa cosa del push. Il push è guidato dagli eventi: l'applicazione si aggiorna perché il modello è cambiato. Il polling chiede ripetutamente se qualcosa è cambiato. Entrambi possono essere utili, ma hanno costi, latenza e comportamento in caso di guasto diversi.

Il modello mentale del tempo reale in Wisej.NET

Un modello mentale utile è questo:

1. Il browser ospita i widget JavaScript di Wisej.NET.
2. Il server ospita la sessione dell'applicazione Wisej.NET e i controlli lato server.
3. Le azioni dell'utente viaggiano dal browser al server come eventi.
4. Il codice lato server modifica i controlli e lo stato dell'applicazione.
5. Wisej.NET rimanda le modifiche al browser come aggiornamenti compatti.
6. Con WebSocket, un'attività in background può inviare con un push gli aggiornamenti in sospeso senza aspettare la successiva richiesta del browser.

Non si tratta di screen scraping e non è un desktop remoto. Il browser continua a disegnare widget web nativi. Il server non trasmette pixel. Il punto architetturale importante è che la logica dell'applicazione e lo stato autorevole della UI vivono sul server. Quando cambia una proprietà di un controllo Wisej.NET, il widget corrispondente nel browser può essere sincronizzato.

Push via WebSocket e aggiornamenti guidati dalle richieste

Una normale richiesta web inizia quando il browser contatta il server e termina quando il server risponde. Se non esiste un canale aperto dal server al browser, il server non può contattare il browser di sua iniziativa: può inviare le modifiche solo quando il browser fa un'altra richiesta. WebSocket cambia le cose. Con una connessione WebSocket, Wisej.NET può inviare con un push gli aggiornamenti della UI al client connesso direttamente dal codice lato server.

In Wisej.NET, un'attività in background può aggiornare i controlli della UI e chiamare `Application.Update()` per inviare le modifiche in sospeso quando WebSocket è disponibile. Se WebSocket non è disponibile, `Application.Update()` non può fare il push perché non c'è una connessione attiva. In quel caso l'applicazione ha bisogno di una soluzione di riserva, come il polling di Wisej.NET o richieste guidate da un timer. La soluzione di riserva non invalida l'app: cambia semplicemente il meccanismo di consegna.

Tre meccanismi di aggiornamento da distinguere

1. Aggiornamenti nella richiesta

È il caso più semplice. L'utente fa clic su un pulsante, parte un gestore di evento, il gestore modifica i controlli e Wisej.NET rimanda le modifiche insieme alla risposta. Qui di solito non chiami `Application.Update()`, perché l'aggiornamento torna al browser in modo naturale alla fine della richiesta.

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

2. Push out-of-bound via WebSocket

È l'argomento principale del corso. Un'attività in background gira al di fuori dell'evento originale del browser. Modifica la UI lato server e chiama `Application.Update()` per inviare le modifiche tramite la connessione WebSocket.

```
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 o refresh guidato da un timer

Quando un browser o un server non può usare WebSocket, il server non può inviare modifiche out-of-band. In quel caso l'applicazione può usare il polling, così che il browser contatti periodicamente il server e riceva le modifiche in sospeso. Si può usare anche un `Wisej.Web.Timer` per scatenare eventi lato server a un intervallo definito.

```
private void MainPage_Load(object sender, EventArgs e)
{
    // Ignorato quando WebSocket è disponibile. Utile come soluzione di riserva quando non lo è.
    Application.StartPolling(1000);
}
```

Il corso insegna a progettare prima per WebSocket, e poi a degradare in modo prevedibile.

Il progetto del corso: TicketOps Live

Tutti i moduli costruiscono parti della stessa applicazione, **TicketOps Live**. È una console operativa di assistenza in tempo reale con queste funzionalità:

- una barra di stato del sistema dal vivo;
- un monitor dell'avanzamento dei job sul server;
- una board dei ticket che si aggiorna quando i ticket vengono creati o modificati;

- un pannello di notifiche per gli eventi multiutente;
- una dashboard operativa con semplici indicatori di salute del server;
- una checklist di produzione per il rilascio dietro IIS o un load balancer.

Il progetto è volutamente semplice dal punto di vista del business. Lo scopo è imparare l'architettura in tempo reale, non implementare un helpdesk completo. Ogni lab dovrebbe lasciare allo studente codice riutilizzabile in applicazioni reali.

Regola di progettazione: prima i modelli, poi la UI

Viene la tentazione di lasciare che le attività in background manipolino direttamente ogni controllo che deve cambiare. Per piccole demo funziona, ma le applicazioni reali hanno bisogno di una separazione tra dati e presentazione. Di solito l'attività in background dovrebbe aggiornare un modello o sollevare un evento di dominio, e la UI specifica della sessione dovrebbe decidere come mostrarlo.

Per una finestra di avanzamento di un singolo utente, aggiornare direttamente i controlli è accettabile. Per le dashboard multiutente, l'accesso globale diretto ai controlli è sbagliato. I controlli appartengono alle sessioni. Un servizio globale non dovrebbe conservare per sempre istanze di form qualsiasi. Le sessioni invece si sottoscrivono a un servizio, ricevono gli eventi e aggiornano i propri controlli nel proprio contesto applicativo.

Un vocabolario sicuro per questo corso

Sessione indica un'istanza in esecuzione dell'applicazione Wisej.NET per un contesto di scheda/finestra del browser. Non coincide con un utente autenticato: uno stesso utente può avere più sessioni.

Contesto applicativo indica il contesto Wisej.NET di cui il codice ha bisogno per accedere alle proprietà statiche di `Application` specifiche della sessione e per aggiornare il browser corretto.

Codice in-bound indica il codice che gira come parte di una richiesta del browser o di un evento Wisej.NET.

Codice out-of-bound indica il codice che gira al di fuori di un evento del browser, come un thread in background, un evento di un hosted service, una callback di una coda o una callback di un timer.

Push indica che il server consegna al browser gli aggiornamenti della UI in sospeso tramite una connessione attiva.

Polling indica che il browser chiede periodicamente al server se ci sono aggiornamenti.



Errori comuni

L'errore più comune è aggiornare un controllo da un thread in background senza preservare il contesto applicativo corretto. Il secondo errore più comune è fare push troppo spesso. Una progress bar non ha bisogno di 500 aggiornamenti al secondo. Il push è potente, ma ogni aggiornamento ha un costo: serializzazione, traffico di rete, lavoro del browser e schedulazione sul server.

Un altro errore comune è usare campi statici per dati specifici della sessione. I campi statici sono condivisi da tutti gli utenti e da tutte le sessioni nell'application domain. Per gli oggetti specifici della sessione usa `Application.Session`, un servizio per sessione oppure `Application.GetInstance()`. Usa servizi statici solo per dati davvero condivisi, e in quel caso progetta con cura la concorrenza.



Punto di verifica

Alla fine di questo modulo lo studente dovrebbe saper disegnare il percorso completo di un evento ticket generato dal server: il servizio modifica i dati, il servizio solleva un evento, una sessione lo riceve, la sessione aggiorna i propri controlli e Wisej.NET invia con un push le modifiche della UI risultanti quando WebSocket è disponibile.

Architettura web in tempo reale in Wisej.NET

Che cosa costruirai nel lab pratico, le attività, l'implementazione suggerita e i criteri di accettazione.



Lab pratico

Crea la prima schermata di TicketOps Live: una pagina con una barra di stato dal vivo, una label con l'orologio, un indicatore di avanzamento e un heartbeat simulato del server che aggiorna il browser senza alcun clic dell'utente.

Punto di partenza

Crea una Wisej.NET Web Page Application chiamata `TicketOpsLive`. Aggiungi una pagina principale chiamata `MainPage`. Usa un qualsiasi tema integrato. Il lab presuppone C#, ma gli stessi concetti valgono per VB.NET.

Attività

1. Aggiungi un pannello in alto chiamato `statusPanel`.
2. Aggiungi le label `clockLabel`, `connectionLabel` e `activityLabel`.
3. Aggiungi una progress bar chiamata `serverLoadBar`.
4. Aggiungi i pulsanti `startButton` e `stopButton`.
5. Implementa un'attività in background che simula l'heartbeat quando si fa clic su `startButton`.
6. L'heartbeat deve aggiornare l'orologio, un valore di carico fittizio e un messaggio di attività una volta al secondo.
7. Ferma gli aggiornamenti quando si fa clic su `stopButton` o quando la pagina viene eliminata (disposed).

Implementazione suggerita

```
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";
}

```

Criteri di accettazione

- ✓ La UI si aggiorna una volta al secondo dopo che lo studente fa clic su Start.
- ✓ Il browser mostra le modifiche senza clic aggiuntivi.
- ✓ Stop disattiva il ciclo entro un secondo.
- ✓ Avviare due volte non crea due cicli in concorrenza.
- ✓ Il codice verifica se la pagina è stata eliminata (disposed).

Domande di riflessione

1. Perché questo lab chiama `Application.Update(this)` dentro il ciclo?
2. Che cosa cambia se WebSocket non è disponibile?
3. Perché `_running` non è un campo statico?
4. Quale sarebbe il rischio di aggiornare ogni 10 millisecondi?

Contesto applicativo, sessioni e ciclo di vita

Sessioni, contesto applicativo e ciclo di vita: gestisci la UI dell'utente giusto, evita la trappola dei campi statici e fai pulizia all'uscita e al timeout. Il concetto, il modello mentale e le insidie in produzione — leggilo prima del lab.



Focus di produzione

Sessioni, contesto applicativo e ciclo di vita: gestisci la UI dell'utente giusto, evita la trappola dei campi statici e fai pulizia all'uscita e al timeout.

Perché le sessioni contano di più nelle app in tempo reale

In una pagina tradizionale a richiesta/risposta, l'utente fa clic, il server risponde e la maggior parte del lavoro lato server finisce lì. In un'applicazione Wisej.NET in tempo reale, il lavoro può continuare dopo la richiesta originale. Un job in background può durare minuti. Una dashboard può sottoscrivere a un flusso globale di eventi. Un timer può continuare ad aggiornare un grafico. Poiché il lavoro sopravvive al clic che lo ha avviato, il ciclo di vita della sessione diventa parte dell'architettura dell'applicazione.

Una sessione Wisej.NET somiglia a un'istanza in esecuzione di un'applicazione desktop. Ha controlli, finestre, pagine, stato dei componenti e oggetti a livello di applicazione. Ma a differenza di un'applicazione desktop, browser e server possono disconnettersi, il browser può essere aggiornato e non esiste un thread della UI dedicato e permanente. Wisej.NET ripristina lo stato corretto della sessione a ogni richiesta e può mantenere lo stato lato server sia per le connessioni HTTP sia per quelle WebSocket.

Sessione non significa utente

Un utente autenticato e una sessione sono collegati ma non identici. Uno stesso utente può aprire due schede del browser e creare due sessioni. Un utente può autenticarsi dentro una sessione, disconnettersi e lasciare la sessione attiva. Un utente anonimo può avere una sessione prima del login. Un servizio in background può esistere senza appartenere ad alcuna sessione utente.

Questa distinzione è importante per il push. Quando un ticket cambia, l'applicazione può dover notificare tutte le sessioni di un tenant, una sola sessione di un operatore oppure ogni dashboard attiva. Non dare per scontato che "utente" e "browser" siano la stessa cosa.

Contesto applicativo

La classe `Application` di Wisej.NET è statica, ma le sue proprietà e operazioni sono isolate per sessione quando il codice gira nel contesto corretto. Quando il codice gira perché il browser ha sollevato un evento, il contesto è già disponibile. Quando il codice gira in un thread out-of-bound, potrebbe non sapere quale sessione deve aggiornare.

`Application.StartTask()` è il modo più semplice per avviare un'elaborazione in background da una sessione, perché avvia l'attività con il contesto della sessione chiamante. Quando il codice è già out-of-bound, puoi catturare `Application.Current` mentre sei nel contesto, e in seguito usare

`Application.Update(context, action)` oppure `Application.RunInContext(context, action)` per eseguire il codice sulla sessione corretta.

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

Il contesto non è una licenza globale per tenere in vita i controlli per sempre. Va usato con una strategia chiara di sottoscrizione e di pulizia.

Dove deve vivere lo stato

In un'applicazione Wisej.NET in tempo reale ci sono quattro posti comuni per lo stato.

Stato dei controlli: lo stato memorizzato direttamente nei controlli, come il testo di una label, la riga selezionata di una griglia, una finestra visibile, la scheda corrente o il valore di una progress bar. È semplice e spesso corretto per lo stato di presentazione.

Application.Session memorizza valori specifici della sessione che non devono essere visibili alle altre sessioni. Usala per metadati leggeri della sessione, il tenant selezionato, le preferenze dell'utente o oggetti per sessione quando non serve un servizio formale.

Servizi di sessione: oggetti creati per ogni sessione. Sono utili quando ti serve un design più testabile e strutturato rispetto a campi di sessione improvvisati. I pattern di Wisej.NET consapevoli della sessione aiutano a evitare condivisioni statiche accidentali.

Servizi globali: condivisi da tutte le sessioni. Sono adatti a code, event hub, cache, watcher in background o stato a livello di sistema. Devono essere thread-safe e non devono memorizzare controlli di sessione senza un ciclo di vita della sottoscrizione ben definito.

La trappola dei campi statici

I campi statici sono condivisi. Se memorizzi `CurrentTicketId` in un campo statico, ogni utente può sovrascrivere il ticket corrente di tutti gli altri. Se memorizzi un riferimento a `MainPage` in una lista statica e non lo rimuovi mai, il server può tenere in vita le sessioni anche dopo che gli utenti se ne sono andati. Se memorizzi in un campo statico un `CancellationTokenSource` per un job di un singolo utente, un utente può annullare il job di un altro.

Usa i campi statici solo per servizi davvero globali. Per i valori specifici della sessione usa i controlli, `Application.Session` o un oggetto per sessione.

Refresh, keep-alive e timeout

Un'applicazione Wisej.NET deve fare i conti con le realtà del web: gli utenti aggiornano il browser, i portatili vanno in sospensione, le reti spariscono e le schede restano aperte tutta la notte. Wisej.NET usa il timeout della sessione e il meccanismo di keep-alive per decidere se una sessione deve restare attiva o essere terminata. Quando una sessione va in timeout, viene rimossa dalla memoria. È più simile a chiudere un'applicazione desktop che a nascondere una pagina web.

Per le app in tempo reale, le conseguenze pratiche sono:

- L'elaborazione in background di lunga durata non deve dare per scontato che la UI esista ancora.
- Il codice che aggiorna la UI deve verificare se una pagina o un controllo è stato eliminato (disposed).
- Le sottoscrizioni a eventi globali devono essere rimosse quando la sessione termina o la vista viene eliminata.
- Se un job deve continuare dopo la fine di una sessione, deve appartenere a un servizio globale, non alla pagina.
- Se un job ha senso solo per la UI di un singolo utente, deve essere annullato quando la UI scompare.

Pattern di pulizia

Una vista in tempo reale che si sottoscrive a un servizio globale deve sempre annullare la sottoscrizione. L'evento esatto da usare dipende dalla struttura della vista, ma il pattern è lo stesso: sottoscriviti al caricamento o nel costruttore, annulla la sottoscrizione quando la vista viene eliminata o quando l'applicazione termina.

```

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);
    });
}

```

L'obiettivo non è solo evitare le eccezioni. L'obiettivo è rendere evidente chi possiede cosa. Il feed possiede gli eventi globali dei ticket. La pagina possiede i propri controlli. La sessione possiede il proprio contesto applicativo.

Gestire gli errori nelle attività in background

Un'attività in background non deve mai fallire in silenzio. In una demo didattica un'eccezione può limitarsi a fermare il ciclo. In produzione, i fallimenti silenziosi fanno mentire le dashboard. Usa un try/catch dentro l'attività, registra l'eccezione nel log e aggiorna la UI con un messaggio di errore sicuro se la UI esiste ancora.

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

In produzione non mostrare agli utenti finali i dettagli grezzi delle eccezioni. Usa log dettagliati per gli sviluppatori e messaggi sicuri per gli utenti.



Punto di verifica

Alla fine di questo modulo lo studente dovrebbe saper spiegare perché un'attività in background avviata da una pagina può aggiornare quella pagina, perché un servizio globale non deve possedere direttamente i controlli di una pagina e perché ogni sottoscrizione in un'app in tempo reale deve avere un piano per annullarla.

Contesto applicativo, sessioni e ciclo di vita

Che cosa costruirai nel lab pratico, le attività, l'implementazione suggerita e i criteri di accettazione.



Lab pratico

Aggiungi un pannello diagnostico che aiuti lo sviluppatore a vedere la sessione corrente, il contesto applicativo, i dettagli del browser e gli eventi del ciclo di vita rilevanti per gli aggiornamenti in tempo reale.

Attività

1. Aggiungi a `MainPage` un group box diagnostico.
2. Mostra l'ora corrente, l'id del client, il tipo di dispositivo del browser se disponibile e un contatore specifico della sessione.
3. Memorizza il contatore in `Application.Session` o in un oggetto per sessione.
4. Aggiungi i pulsanti per incrementare il contatore e per simulare un aggiornamento in background.
5. Aggiungi un log di pulizia quando l'applicazione termina.
6. Aggiorna il browser e osserva che cosa rimane.

Frammenti di codice suggeriti

```
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)
{
    // In produzione sostituiscilo con il tuo framework di logging.
    System.Diagnostics.Debug.WriteLine("TicketOps session exited: " + Application.ClientId);
}
```

Criteri di accettazione

- ✓ Il contatore è specifico della sessione, non statico.
- ✓ Un aggiornamento in background modifica la sessione del browser corretta.
- ✓ Il pannello diagnostico mostra quando arriva l'aggiornamento in background.
- ✓ Lo studente sa spiegare che cosa succede dopo un refresh e dopo il timeout della sessione.

Domande di riflessione

1. Perché `Application.Current` viene catturato prima di avviare l'azione out-of-bound?
2. Che cosa succederebbe se il contatore fosse statico?
3. Quale azione di pulizia aggiungerei per la sottoscrizione a un servizio globale?

Server push per una singola sessione con attività in background

Push per una singola sessione: esegui il lavoro lungo fuori dal gestore del clic, segnala l'avanzamento, supporta l'annullamento cooperativo e ripristina la UI nel finally. Il concetto, il modello mentale e le insidie in produzione — leggilo prima del lab.



Focus di produzione

Push per una singola sessione: esegui il lavoro lungo fuori dal gestore del clic, segnala l'avanzamento, supporta l'annullamento cooperativo e ripristina la UI nel finally.

Lo scenario di server push più comune

Il primo uso più comune del server push è un'operazione di lunga durata. Un utente avvia un'importazione, un report, un passo di migrazione, la generazione di un file, una sincronizzazione di dati o una routine di validazione. L'operazione richiede secondi o minuti. La UI non deve bloccarsi, l'utente deve vedere l'avanzamento e l'operazione, alla fine, deve segnalare il successo o l'errore.

Nelle applicazioni desktop, gli sviluppatori usano spesso un background worker o un task e riportano gli aggiornamenti sul thread della UI. In Wisej.NET il design equivalente è l'elaborazione in background nel contesto della sessione, più `Application.Update()` quando devi inviare con un push le modifiche della UI al browser. La UI lato server appartiene comunque alla sessione. L'attività in background deve rispettare il ciclo di vita, l'annullamento e la frequenza degli aggiornamenti.

Il pattern di base

Il pattern di fondo ha sei passi:

1. Disattiva o adatta la UI prima di iniziare il lavoro.
2. Crea lo stato di annullamento se l'utente può annullare.
3. Avvia l'attività in background nel contesto della sessione.
4. Esegui il lavoro a piccoli passi.
5. Aggiorna lo stato della UI a intervalli ragionevoli.
6. Riabilita la UI e mostra il completamento o l'errore.

```

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 ... ";
}

```

Il metodo dell'attività deve contenere un try/catch/finally, così che la UI non resti disattivata dopo un'eccezione.

```

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

            // Simula il lavoro.
            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);
}
}

```

Osserva la frequenza degli aggiornamenti. Il codice modifica la progress bar a ogni iterazione, ma fa il push solo ogni cinque iterazioni più l'aggiornamento finale. In questo modo si riducono i messaggi di rete e i ridisegni del browser, e l'interfaccia sembra comunque dal vivo.

Aggiornamento diretto della UI o callback nel contesto

Quando un'attività viene avviata con `Application.StartTask()` da un evento Wisej.NET, gira con il contesto applicativo corrente. In molti esempi vedrai modifiche dirette ai controlli seguite da `Application.Update(this)`. È accettabile per un job di una singola sessione posseduto dalla pagina.

Per il codice che parte al di fuori della sessione, usa l'overload con il contesto esplicito:

```

var context = Application.Current;

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

```

Così è chiaro che le modifiche della UI avvengono nel contesto della sessione corretta.

Evita i gestori di evento bloccanti

Non eseguire un'operazione lunga direttamente in un gestore del clic. La richiesta del browser resta aperta, l'utente non vede alcun aggiornamento intermedio via push e il thread del server resta occupato troppo a lungo. Avvia l'operazione e restituisci il controllo all'utente.

Sbagliato:

```

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

```

Meglio:

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

L'annullamento è una funzionalità, non un ripensamento

Una UI di avanzamento in tempo reale senza annullamento può essere peggio di una UI bloccante: dice all'utente che sta succedendo qualcosa ma non gli dà alcun controllo. L'annullamento va integrato nelle attività di lunga durata ovunque sia sicuro. Non terminare i thread. Usa l'annullamento cooperativo. Controlla il token tra un'unità di lavoro e l'altra e lascia il sistema in uno stato coerente.

L'annullamento è utile anche quando la pagina si chiude. Se l'elaborazione in background serve solo a quella pagina, annullalo al dispose o all'uscita dell'applicazione. Se il lavoro è un job del server che deve continuare, sposta la proprietà in un servizio e lascia che la pagina mostri lo stato del servizio.

Segnalazione degli errori

Un'attività in background può fallire dopo che il gestore di evento originale è già terminato. Senza una gestione esplicita degli errori, l'utente potrebbe vedere soltanto che l'avanzamento si è fermato. Intercetta sempre le eccezioni dentro l'elaborazione in background. Registra nel log i dettagli tecnici. Invia alla UI con un push uno stato di errore sicuro per l'utente. Conserva abbastanza dettagli per la diagnosi: id del job, id della sessione/client, id dell'utente se autenticato, nome dell'operazione e orario.

Raggruppamento e limitazione della frequenza

L'occhio umano non trae vantaggio da aggiornamenti illimitati. La maggior parte delle dashboard gestionali sembra dal vivo con 1-4 aggiornamenti al secondo. Le progress bar spesso appaiono fluide a 5-10 aggiornamenti al secondo, ma molte operazioni non ne hanno bisogno. La telemetria ad alta frequenza va aggregata prima di essere mostrata.

Regole pratiche:

- Invia subito con un push lo stato finale.
- Invia l'avanzamento visibile all'utente a un intervallo controllato.
- Raggruppa più modifiche a label e griglie in un'unica chiamata a `Application.Update()`.
- Evita il push dentro cicli stretti che usano molta CPU.
- Preferisci una coda o un timer per accorpare le raffiche provenienti da sistemi esterni.



Punto di verifica

Alla fine di questo modulo lo studente dovrebbe saper implementare un'operazione di lunga durata che parte subito, aggiorna l'avanzamento senza bloccare il browser, supporta l'annullamento, segnala gli errori e lascia la UI in uno stato coerente.

Server push per una singola sessione con attività in background

Che cosa costruirai nel lab pratico, le attività, l'implementazione suggerita e i criteri di accettazione.



Lab pratico

Aggiungi a TicketOps Live un pannello realistico per un'importazione di lunga durata. L'utente può avviare un'importazione, seguirne l'avanzamento, annullarla e controllarne il risultato.

Requisiti della UI

Aggiungi:

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

Attività

1. Implementa un ciclo di importazione con 200 record simulati.
2. Invia l'avanzamento con un push ogni 10 record, non a ogni record.
3. Aggiungi messaggi di log per l'avvio, ogni 50 record, l'annullamento, l'errore e il completamento.
4. Implementa l'annullamento con `CancellationTokenSource`.
5. Aggiungi un pulsante o un'opzione di errore deliberato che lancia un'eccezione al record 87.
6. Assicurati che la UI venga riabilitata in tutti gli esiti.

Ciclo sui record suggerito

```
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);
}
}

```

Criteri di accettazione

- ✓ Gli aggiornamenti di avanzamento compaiono mentre l'operazione è in corso.
- ✓ L'annullamento funziona e lascia la UI utilizzabile.
- ✓ L'errore viene segnalato senza esporre stack trace grezzi.
- ✓ L'aggiornamento finale arriva sempre al browser se la sessione è ancora attiva.
- ✓ Il codice non fa più push del necessario.

Sfida di estensione

Aggiungi un `JobId` a ogni importazione e includilo nei messaggi di log. Il capstone lo userà per collegare gli eventi della UI ai log del server.

Timer, polling di riserva e cadenza degli aggiornamenti

Cadenza degli aggiornamenti e soluzioni di riserva: scegli tra push, timer e polling, accorpa le raffiche di eventi e gestisci il ciclo di vita dei timer per ogni sessione. Il concetto, il modello mentale e le insidie in produzione — leggilo prima del lab.



Focus di produzione

Cadenza degli aggiornamenti e soluzioni di riserva: scegli tra push, timer e polling, accorpa le raffiche di eventi e gestisci il ciclo di vita dei timer per ogni sessione.

Il push è preferibile, ma la soluzione di riserva conta

Un'applicazione di produzione non può dare per scontato che ogni browser, proxy, server o percorso di rete si comporti in modo perfetto. Alcuni ambienti bloccano i WebSocket. Alcuni reverse proxy richiedono un supporto WebSocket esplicito. Alcune reti aziendali introducono dei timeout. Wisej.NET supporta il push in tempo reale con WebSocket e offre anche soluzioni di riserva, come il polling, per i sistemi che non possono usare WebSocket.

Lo sviluppatore di livello intermedio deve capire la differenza tra una soluzione di riserva e un errore di progettazione. Il polling non è sbagliato di per sé. È sbagliato quando sostituisce un modello a eventi chiaro o quando l'intervallo viene scelto senza considerare il costo per il server. Un intervallo di polling di un secondo per migliaia di sessioni può essere costoso. Un intervallo di dieci secondi per una board operativa critica può sembrare lento. La scelta giusta dipende dal valore per l'utente e dal costo.

Wisej.Web.Timer o attività in background

Un `Wisej.Web.Timer` è utile quando vuoi eventi periodici lato server associati a una sessione. Per esempio, una dashboard potrebbe ricalcolare un KPI ogni cinque secondi, anche se non è arrivato alcun evento da un servizio globale. Un'attività in background è più adatta quando un'operazione specifica parte, gira e si completa. Un servizio globale è più adatto quando un unico watcher condiviso alimenta molte sessioni.

Usa un timer per il refresh periodico della UI con ambito di sessione. Usa `Application.StartTask()` per il lavoro di lunga durata con ambito di sessione. Usa un servizio condiviso per lo stato globale e gli eventi multiutente.

Polling di riserva con StartPolling ed EndPolling

Quando WebSocket è disponibile, `Application.Update()` può inviare con un push gli aggiornamenti in sospeso al browser. Quando WebSocket non è disponibile, non ha una connessione attiva da usare.

Wisej.NET fornisce `Application.StartPolling(milliseconds)` e `Application.EndPolling()` come meccanismi di riserva. Il polling viene ignorato quando WebSocket è attivo, il che permette all'applicazione di richiedere il comportamento di riserva senza danneggiare le sessioni WebSocket.

Un pattern comune è:

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

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

            // Efficace quando WebSocket esiste. Il polling gestisce i casi senza WebSocket.
            Application.Update(this);
            Thread.Sleep(1000);
        }

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

Il codice va comunque progettato con attenzione. Se `_liveMode` non diventa mai false e la pagina viene eliminata, il ciclo diventa un leak. Prevedi sempre una condizione di arresto.

Scegliere la cadenza degli aggiornamenti

La cadenza è il ritmo degli aggiornamenti. In un'app Wisej.NET in tempo reale la cadenza ha quattro dimensioni:

Cadenza del modello: quanto spesso cambiano i dati sottostanti.

Cadenza della UI: quanto spesso l'utente ha bisogno di vedere modifiche visibili.

Cadenza del push: quanto spesso il server invia aggiornamenti della UI al browser.

Cadenza del polling: quanto spesso il browser chiede aggiornamenti al server quando il push non è disponibile.

Non coincidono sempre. Una macchina può produrre telemetria 100 volte al secondo, ma alla UI può bastare un riepilogo una volta al secondo. Una coda può cambiare 20 volte in una raffica, ma la dashboard può mostrare lo stato finale dopo 250 millisecondi.

Accorpare le raffiche

Una raffica si verifica quando arrivano molte modifiche tutte insieme. Se ogni modifica provoca un push separato, la UI può sprecare risorse e mostrare stati intermedi che gli utenti non riescono a capire.

Accorpare significa raccogliere più modifiche e applicarle insieme.

Una semplice strategia di accorpamento:

```

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();
}

```

L'evento segna il modello come modificato (dirty). Il timer applica gli aggiornamenti con una cadenza della UI stabile.

Chi possiede il timer

Un timer appartiene a qualcuno. Se è un componente Wisej.NET su una pagina, lo possiede la pagina. Se è un timer .NET in un servizio globale, lo possiede il servizio. La proprietà determina la pulizia. Un timer duplicato può raddoppiare il traffico di aggiornamenti e far sembrare casuali i bug.

Regole pratiche:

- Avvia i timer una sola volta.
- Ferma i timer quando la vista non è più attiva.
- Non creare un nuovo timer a ogni clic sul pulsante di refresh.
- Evita il lavoro di lunga durata dentro un tick del timer; avvia il lavoro o pianificalo.
- Usa un flag per impedire tick sovrapposti se un refresh dura più dell'intervallo.

Riconoscere una cadenza sbagliata

I sintomi di troppi aggiornamenti includono CPU alta, rendering a scatti nel browser, molto traffico WebSocket, rumore nei log del server e dashboard che sembrano sfarfallare. I sintomi di troppo pochi aggiornamenti includono utenti che premono refresh, lavoro duplicato, decisioni operative prese su dati vecchi e mancanza di fiducia nella dashboard.

Gli sviluppatori di livello intermedio dovrebbero strumentare le dashboard importanti. Conta gli eventi ricevuti, gli aggiornamenti della UI applicati, i push tentati, gli errori e il tempo medio impiegato per applicare un aggiornamento.



Punto di verifica

Alla fine di questo modulo lo studente dovrebbe saper scegliere un timer, un'attività in background, il polling di riserva o un evento di servizio in base al lavoro da svolgere, e spiegare il costo della cadenza scelta.

Timer, polling di riserva e cadenza degli aggiornamenti

Che cosa costruirai nel lab pratico, le attività, l'implementazione suggerita e i criteri di accettazione.



Lab pratico

Estendi TicketOps Live con un timer di aggiornamento della dashboard e con i controlli per sperimentare la cadenza degli aggiornamenti.

Requisiti della UI

Aggiungi:

- ✓ `liveModeCheckBox`
- ✓ `cadenceComboBox` con i valori `250 ms`, `1 sec`, `5 sec`
- ✓ `eventsReceivedLabel`
- ✓ `updatesAppliedLabel`
- ✓ `lastAppliedLabel`
- ✓ `refreshTimer`

Attività

1. Avvia il polling quando la modalità dal vivo è attiva.
2. Ferma il polling quando la modalità dal vivo è disattivata.
3. Usa un timer per applicare gli aggiornamenti della dashboard alla cadenza selezionata.
4. Simula modifiche al modello più rapide degli aggiornamenti della UI.
5. Conta separatamente gli eventi del modello e gli aggiornamenti della UI applicati.
6. Impedisci i tick del timer sovrapposti.

Idea di implementazione suggerita

```
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;
    }
}

```

Criteri di accettazione

- ✓ La cadenza del timer cambia in modo visibile.
- ✓ Il conteggio degli eventi ricevuti può crescere più in fretta di quello degli aggiornamenti della UI applicati.
- ✓ Il polling parte e si ferma insieme alla modalità dal vivo.
- ✓ Non si verificano tick del timer sovrapposti.
- ✓ Lo studente sa spiegare perché l'accorpamento degli aggiornamenti è utile.

Data binding dal vivo e griglie in tempo reale

Data binding dal vivo: aggiorna le collezioni in binding nel contesto della sessione senza rifare il binding della griglia, mantenendo selezione, ordinamento e scorrimento. Il concetto, il modello mentale e le insidie in produzione — leggilo prima del lab.



Focus di produzione

Data binding dal vivo: aggiorna le collezioni in binding nel contesto della sessione senza rifare il binding della griglia, mantenendo selezione, ordinamento e scorrimento.

Perché il data binding diventa più difficile quando i dati si muovono

Il data binding statico è semplice: carichi i record, colleghi la griglia, lasci modificare l'utente, salvi. Il data binding in tempo reale aggiunge movimento. Arrivano righe, cambiano campi, gli stati avanzano e i job in background aggiornano la stessa schermata che l'utente sta guardando. La sfida è tenere sincronizzata la UI senza distruggere e ricreare di continuo i binding.

Wisej.NET supporta il binding semplice per le singole proprietà dei controlli e il binding complesso per controlli come le griglie. Il componente `BindingSource` è particolarmente utile perché si colloca tra i controlli e le sorgenti dati, gestisce la posizione corrente e offre un punto coerente in cui reimpostare o aggiornare i binding.

Il modello Ticket

In questo corso usa un semplice modello di ticket:

```
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));
    }
}

```

Lo scopo di `INotifyPropertyChanged` è rendere visibili le modifiche alle proprietà all'infrastruttura di binding, dove è supportato. Anche quando chiami `BindingSource.ResetBindings(false)` per chiarezza, il modello esprime comunque di poter notificare le modifiche.

Collegare una lista dal vivo

Un punto di partenza pratico è `BindingList<Ticket>` insieme a `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;
}

```

Quando arriva un nuovo ticket, aggiungilo alla lista nel contesto della sessione corretta. Quando un campo cambia, aggiorna l'elemento e reimposta i binding se la UI non si aggiorna da sola.

```
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);
}
```

Non rifare il binding della griglia a ogni aggiornamento

Rifare il binding dell'intera griglia a volte è il modo più rapido per far funzionare una demo, ma in produzione spesso è sbagliato. Può azzerare la selezione, la posizione di scorrimento, l'ordinamento, la larghezza delle colonne e il contesto dell'utente. Può anche essere costoso. Preferisci modifiche incrementalmente alla lista in binding, e poi notifica la binding source.

Sbagliato:

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

Meglio:

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

La soluzione migliore per le schermate ad alto volume: raggruppa più modifiche e fai il refresh una volta per intervallo di cadenza.

Thread safety con le collezioni in binding

Una collezione in binding appartiene alla UI della sessione. Un servizio globale può ricevere aggiornamenti da un altro thread. Non modificare la collezione in binding da thread qualsiasi senza entrare nel contesto della sessione e senza valutare la sincronizzazione.

Un pattern multiutente sicuro è:

1. Il servizio globale riceve un evento ticket.
2. Il gestore di evento della sessione intercetta l'evento.
3. Il gestore usa `Application.Update(context, () => { ... })`.

4. Dentro la callback, aggiorna la lista in binding della sessione.
5. Reimposta i binding una sola volta.

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

Formattazione e parsing

Le griglie in tempo reale non riguardano solo il trasporto dei dati: devono restare leggibili. Usa la formattazione per date, valute, priorità e stati. Usa gli eventi di parse/format sui controlli in binding quando i valori richiedono una conversione. Evita di memorizzare stringhe formattate nel modello se il valore reale ha un tipo più forte.

Per esempio, `UpdatedAt` deve essere un `DateTime`, non una stringa già formattata. La griglia può mostrarlo come ora locale o come età relativa.

Stato delle righe e indicazioni visive

Una griglia dal vivo deve mostrare che cosa è cambiato senza diventare rumorosa. Pattern comuni:

- mostrare per qualche secondo un indicatore temporaneo "aggiornato";
- ordinare i nuovi ticket in cima;
- mantenere stabile, quando possibile, il ticket selezionato dall'utente;
- evitare di spostare una riga mentre l'utente la sta modificando;
- usare colonne di stato e piccoli indicatori invece di interruzioni modali.

La UI deve aiutare gli utenti a fidarsi dei dati. Troppe animazioni o salti di riga possono far pensare agli utenti che l'applicazione sia instabile.

Validazione e gestione dei conflitti

Dati in tempo reale significa che un altro utente o un servizio può modificare un record mentre l'utente corrente lo sta guardando o modificando. Decidi che cosa succede quando un ticket cambia durante la modifica. In questo corso usa una regola semplice: se un ticket è in modifica in un form di dettaglio, mostra un avviso non bloccante che il ticket è cambiato e lascia all'utente la scelta di ricaricare o continuare.

I sistemi di produzione possono aver bisogno della concorrenza ottimistica con numeri di versione, timestamp o row version del database. Wisej.NET non elimina la necessità della concorrenza nel livello dati: dà alla UI un modo per avvisare gli utenti al momento giusto.



Punto di verifica

Alla fine di questo modulo lo studente dovrebbe saper mantenere una griglia dal vivo che riceve nuove righe e aggiornamenti di righe da una sorgente in background, preservando il contesto dell'utente ed evitando di rifare il binding inutilmente.

Data binding dal vivo e griglie in tempo reale

Che cosa costruirai nel lab pratico, le attività, l'implementazione suggerita e i criteri di accettazione.



Lab pratico

Crea una `DataGridView` dal vivo che mostra i ticket di assistenza e si aggiorna quando arrivano eventi simulati sui ticket.

Attività

1. Aggiungi una `DataGridView` chiamata `ticketsGrid`.
2. Crea il modello `Ticket` e l'enum `TicketStatus`.
3. Collega una `BindingList<Ticket>` tramite una `BindingSource`.
4. Aggiungi un pulsante che crea un nuovo evento ticket ogni secondo per 20 secondi.
5. Aggiungi un pulsante che cambia a caso lo stato dei ticket.
6. Mantieni la riga selezionata quando possibile.
7. Mostra `UpdatedAt` in un formato leggibile.

Evento del modello suggerito

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

Metodo di aggiornamento suggerito

```
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);
}
```

Criteri di accettazione

- ✓ I nuovi ticket compaiono senza rifare da zero il binding della griglia.
- ✓ I ticket esistenti si aggiornano sul posto.
- ✓ La riga selezionata viene mantenuta se la riga esiste ancora.
- ✓ La frequenza degli aggiornamenti resta sotto controllo.
- ✓ Lo studente sa spiegare perché la lista in binding appartiene alla sessione.

Sfida di estensione

Aggiungi un filtro per i ticket **Escalated** e assicurati che gli aggiornamenti dal vivo funzionino ancora quando il filtro è attivo.

Push multiutente con servizi ed event hub

Push multiutente: un event hub globale solleva eventi di dominio mentre ogni sessione aggiorna la propria UI nel proprio contesto. Il concetto, il modello mentale e le insidie in produzione — leggilo prima del lab.



Focus di produzione

Push multiutente: un event hub globale solleva eventi di dominio mentre ogni sessione aggiorna la propria UI nel proprio contesto.

Da una sessione a molte sessioni

Il push per una singola sessione è utile, ma molte applicazioni gestionali hanno bisogno del push multiutente. Quando un dispatcher assegna un ticket, ogni dashboard dei supervisori deve aggiornarsi. Quando termina un'importazione notturna, tutti gli amministratori connessi devono vederne il risultato. Quando fallisce un health check di un servizio, le console operative attive devono mostrare l'allarme.

Il cambiamento architetturale chiave riguarda la proprietà. Una pagina possiede i propri controlli. Una sessione possiede il proprio contesto. Un servizio globale possiede i dati e gli eventi condivisi. Non lasciare che il servizio globale diventi un contenitore di riferimenti a form. Usa invece eventi di dominio e sottoscrizioni a livello di sessione.

Il pattern event hub

Un event hub è un servizio condiviso che riceve o crea eventi di dominio e li espone ai sottoscrittori. L'hub non sa come ogni UI mostra l'evento: annuncia semplicemente che è successo qualcosa.

```
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);
}
}

```

L'hub mantiene thread-safe lo stato globale e solleva un evento di dominio. Non chiama `Application.Update()`, perché non possiede alcuna sessione.

Il pattern del sottoscrittore di sessione

Ogni sessione si sottoscrive all'hub e aggiorna la propria 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;
}

```

Questo pattern dà a ogni sessione la propria UI. Due utenti possono filtrare o ordinare in modo diverso pur ricevendo gli stessi eventi di dominio.

Aggiornamento broadcast o mirato

Broadcast significa che ogni sottoscrittore riceve l'evento. Aggiornamento mirato significa che lo ricevono solo alcune sessioni. Esempi:

- Inviare in broadcast tutti gli allarmi di sistema gravi a tutte le console operative.
- Inviare gli eventi specifici di un tenant solo alle sessioni di quel tenant.
- Inviare le notifiche di assegnazione solo all'operatore assegnato.
- Inviare gli eventi amministrativi solo agli utenti con ruolo di amministratore.

L'hub può includere metadati sufficienti perché il sottoscrittore decida se mostrare l'evento. In alternativa, l'hub può gestire gruppi di sottoscrizione. Mantieni semplice la prima versione: includi metadati di tenant/utente/ruolo e lascia filtrare la sessione.

Thread safety e snapshot

I servizi globali mantengono spesso delle collezioni. Le collezioni vanno protette con lock o con collezioni concorrenti. Quando esponi i dati alle sessioni, preferisci gli snapshot ai riferimenti diretti alla collezione.

Uno snapshot evita che il codice della UI modifichi per errore lo stato globale ed evita di enumerare la collezione mentre il servizio la sta modificando.

Sbagliato:

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

Meglio:

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

Il clone non deve per forza essere costoso in ogni caso d'uso, ma il confine di proprietà deve essere chiaro.

Memory leak attraverso gli eventi

Gli eventi sono una fonte comune di memory leak. Se un servizio globale espone un evento e una pagina vi si sottoscrive, il servizio ora ha un riferimento al gestore della pagina. Se la pagina viene chiusa ma non annulla mai la sottoscrizione, il servizio può tenerla in vita. Nelle app in tempo reale questo leak è sia un problema di memoria sia un problema di correttezza, perché il servizio può tentare di aggiornare una sessione ormai morta.

Ogni sottoscrizione deve avere il suo annullamento. È una regola del corso.

Backpressure

Se l'hub emette modifiche più in fretta di quanto le sessioni riescano a mostrarle, il sistema ha bisogno di backpressure o di accorpamento. Strategie semplici:

- scartare gli eventi duplicati a bassa priorità;
- raggruppare le modifiche e inviare riepiloghi;
- segnare una sessione come da aggiornare (dirty) e fare il refresh con un timer;
- conservare solo lo stato più recente di una metrica della dashboard;
- accodare gli eventi per sessione con una dimensione massima.

Le app gestionali di solito non hanno bisogno di mostrare ogni stato intermedio. Hanno bisogno di mostrare lo stato su cui gli utenti possono agire.



Punto di verifica

Alla fine di questo modulo lo studente dovrebbe saper progettare un servizio condiviso che alimenta in sicurezza molte sessioni Wisej.NET senza memorizzare globalmente i controlli delle pagine, senza lasciare gestori di evento orfani e senza modificare le collezioni in binding della UI dal thread sbagliato.

Push multiutente con servizi ed event hub

Che cosa costruirai nel lab pratico, le attività, l'implementazione suggerita e i criteri di accettazione.



Lab pratico

Rifattorizza la simulazione dei ticket in un servizio globale `TicketHub` e fai in modo che più sessioni del browser ricevano gli eventi dei ticket dal vivo.

Attività

1. Crea un singleton `TicketHub` con una lista di ticket thread-safe.
2. Aggiungi gli eventi `TicketChanged`.
3. Carica uno snapshot all'avvio di `MainPage`.
4. Sottoscrivi la pagina a `TicketChanged`.
5. Annulla la sottoscrizione all'uscita dell'applicazione.
6. Apri due browser o due sessioni e verifica che entrambi ricevano gli aggiornamenti.
7. Aggiungi metadati di tenant o di ruolo e filtra almeno un tipo di evento.

Criteri di accettazione

- ✓ L'hub non memorizza riferimenti a pagine o controlli.
- ✓ Ogni sessione aggiorna la propria lista in binding.
- ✓ Chiudere o aggiornare una sessione non lascia indietro un sottoscrittore guasto.
- ✓ Più sessioni possono mostrare filtri diversi condividendo la stessa sorgente di eventi di dominio.
- ✓ Lo studente sa spiegare la differenza tra aggiornamento broadcast e mirato.

Sfida di estensione

Aggiungi un contatore delle notifiche per sessione. Il contatore deve essere specifico della sessione, anche se gli eventi arrivano da un servizio globale.

Irrobustimento per la produzione, rilascio e capstone

Irrobustimento per la produzione: verifica l'intero percorso WebSocket, le sticky session, gli health check, la sicurezza e l'osservabilità prima del rilascio. Il concetto, il modello mentale e le insidie in produzione — leggilo prima del lab.



Focus di produzione

Irrobustimento per la produzione: verifica l'intero percorso WebSocket, le sticky session, gli health check, la sicurezza e l'osservabilità prima del rilascio.

In produzione il tempo reale è architettura più operatività

Un'app in tempo reale può funzionare perfettamente sulla macchina dello sviluppatore e fallire in produzione, perché il percorso di rilascio è diverso. WebSocket può essere bloccato da un proxy. Un load balancer può instradare le richieste di un utente verso istanze del server diverse. Il timeout della sessione può interrompere il lavoro di lunga durata della UI. Un servizio globale può usare troppa memoria. Un'impostazione di sicurezza può essere sicura in sviluppo ma sbagliata dietro un reverse proxy.

Questo modulo trasforma i pattern del corso in una checklist di produzione.

Un rilascio che tiene conto di WebSocket

Il push in tempo reale di Wisej.NET funziona al meglio quando WebSocket è disponibile. In produzione, verifica il supporto WebSocket lungo tutto il percorso: browser, rete, reverse proxy, load balancer, web server e applicazione. Alcuni load balancer richiedono l'instradamento TCP o un supporto WebSocket esplicito. I reverse proxy come NGINX possono fare da proxy per WebSocket, ma devono essere configurati correttamente.

Fare i test solo dalla console del server non basta. Esegui i test dal percorso di rete dell'utente. Una VPN aziendale, un'appliance di sicurezza o un proxy datato possono cambiare il comportamento.

Bilanciamento del carico e affinità di sessione

Le applicazioni Wisej.NET mantengono lo stato della sessione lato server. In un rilascio su più server, le richieste di una data sessione devono raggiungere sempre la stessa istanza del server, a meno che l'architettura non supporti esplicitamente un'altra strategia. La regola pratica per il bilanciamento del carico è usare le sticky session o un altro meccanismo di instradamento che garantisca la coerenza tra utente e server.

Gli health check devono dire al load balancer quando un server deve smettere di ricevere traffico.

Wisej.NET supporta la configurazione degli health check tramite un file `HealthCheck.json`, con parametri come la memoria massima, il numero massimo di sessioni e la soglia di CPU. Le soglie esatte dipendono dal comportamento dell'applicazione e dalle dimensioni dell'hosting.

Aree di configurazione importanti

La configurazione riguarda sia `Web.config` sia i file di configurazione JSON di Wisej.NET. Le applicazioni Wisej.NET usano file applicativi JSON come `Default.json`, e un progetto può definire più applicazioni Wisej.NET, ciascuna con i propri file di configurazione.

Le impostazioni in tempo reale e gli aspetti di rilascio importanti includono:

- `sessionTimeout`: controlla quando le sessioni inattive vanno in timeout;
- `pollingInterval`: controlla il comportamento del polling automatico per i client senza WebSocket;
- `responseTimeout`: conta per le risposte HTTP iniziali lunghe, ma WebSocket non va in timeout allo stesso modo;
- `debug=false` in produzione, così che le librerie JavaScript vengano raggruppate e minificate come si deve;
- la chiave di licenza del server e le impostazioni del tema predefinito;
- le impostazioni di moduli e handler IIS per le app .NET Framework;
- il modello di hosting ASP.NET Core e il middleware per le app .NET Core/.NET;
- le impostazioni degli health check per i rilasci con bilanciamento del carico.

Esempio di frammento di `Default.json` per un'app didattica:

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

Non copiare alla cieca in produzione i valori usati per la formazione. Usa i test di carico e i requisiti degli utenti per scegliere le impostazioni.

Considerazioni di sicurezza per le app in tempo reale

Wisej.NET evita molti dei tradizionali problemi di concatenazione di stringhe HTML, perché sincronizza i controlli del server con i widget del browser invece di costruire pagine con stringhe HTML arbitrarie. Gli sviluppatori però possono comunque creare un rischio XSS se mostrano come HTML contenuti forniti dagli utenti. I controlli e i messaggi che accettano HTML vanno gestiti con attenzione. Mantieni codificati titoli dei ticket, commenti, nomi e messaggi inseriti dagli utenti, a meno che non esista un processo di sanitizzazione chiaro.

Dietro un reverse proxy o un load balancer, anche la gestione dell'IP del client deve essere deliberata. Gli header inoltrati vanno considerati affidabili solo quando l'app si trova dietro un proxy fidato che ripulisce gli header in ingresso. Altrimenti un attaccante può falsificare i valori degli header.

Le app in tempo reale devono anche proteggere le risorse del server. Imposta limiti per sessioni, code e frequenza degli eventi. Non permettere a un singolo utente di creare job in background o sottoscrizioni senza limiti.

Osservabilità

Un'app in tempo reale ha bisogno di log e metriche che spieghino che cosa è successo dopo la fine della richiesta originale dell'utente. Includi campi di correlazione:

- id del job;
- id dell'utente o marcatore di sessione anonima;
- id del client/della sessione dove opportuno;
- nome dell'operazione;
- tipo di evento;
- istanza del server;
- durata ed esito;
- numero di aggiornamenti della UI inviati o applicati.

In TicketOps Live ogni job di importazione deve avere un id del job. Ogni evento dell'hub deve avere un id dell'evento. La UI può mostrare uno stato semplificato, ma i log devono permettere al personale di supporto di ricostruire la sequenza.

Checklist delle prestazioni

Prima del rilascio, rivedi queste domande:

1. Ogni ciclo in background ha una condizione di arresto?
2. Ogni sottoscrizione a eventi globali viene annullata?
3. Gli aggiornamenti della UI sono raggruppati o limitati in frequenza?
4. Gli eventi del modello ad alta frequenza vengono accorpati?
5. Le collezioni in binding vengono aggiornate nel contesto della sessione?
6. I campi statici sono usati solo per lo stato globale?
7. Contatori, filtri e selezioni specifici della sessione sono davvero per sessione?
8. Le eccezioni vengono intercettate e registrate nel log dentro le attività in background?
9. L'app si comporta in modo accettabile quando WebSocket non è disponibile?
10. Il load balancer mantiene una sessione sulla stessa istanza del server?

Panoramica del capstone

Il capstone è una revisione per la produzione e il completamento di TicketOps Live. Gli studenti devono dimostrare:

- aggiornamenti di stato dal vivo;
- avanzamento delle importazioni in background con annullamento e gestione degli errori;
- aggiornamenti dal vivo della griglia dei ticket;
- notifiche multiutente sui ticket;
- strategia di polling di riserva;
- pulizia della sessione;

- checklist di rilascio e osservabilità.

Il capstone non viene valutato in base al numero di controlli sullo schermo, ma in base all'architettura: proprietà, contesto, ciclo di vita, cadenza degli aggiornamenti e prontezza per la produzione.



Punto di verifica

Alla fine di questo modulo lo studente dovrebbe saper spiegare come un'app Wisej.NET in tempo reale passa da demo ad applicazione rilasciabile: percorso WebSocket verificato, instradamento delle sessioni stabile, soluzione di riserva definita, sottoscrizioni ripulite, aggiornamenti limitati in frequenza, dati protetti e operatività osservabile.

Irrobustimento per la produzione, rilascio e capstone

Che cosa costruirai nel lab pratico, le attività, l'implementazione suggerita e i criteri di accettazione.



Lab pratico

Completa TicketOps Live e preparalo per una revisione dell'architettura di produzione.

Funzionalità richieste

1. Barra di stato dal vivo dal M01.
2. Diagnostica della sessione dal M02.
3. Monitor delle importazioni in background dal M03.
4. Controlli di cadenza e polling dal M04.
5. Board dei ticket dal vivo dal M05.
6. **TicketHub** multiutente dal M06.
7. Checklist di produzione e revisione della configurazione dal M07.

Deliverable di architettura

Crea una nota di architettura di una pagina che indichi:

- ✓ quale stato è specifico della sessione;
- ✓ quale stato è globale;
- ✓ dove parte l'elaborazione in background;
- ✓ come gli aggiornamenti della UI raggiungono il browser;
- ✓ come funziona l'annullamento;
- ✓ come vengono rimosse le sottoscrizioni;
- ✓ che cosa succede quando WebSocket non è disponibile;
- ✓ che cosa deve supportare l'ambiente di rilascio.

Copione della demo

1. Apri TicketOps Live in due sessioni del browser.
2. Avvia l'heartbeat.
3. Avvia un'importazione e mostra l'avanzamento.
4. Annulla un'importazione.
5. Avvia un'altra importazione e forza un errore simulato.
6. Genera nuovi eventi ticket e mostra che entrambe le sessioni si aggiornano.
7. Applica un filtro in una sessione e mostra che l'altra sessione è indipendente.
8. Disattiva la modalità dal vivo e spiega il comportamento di polling e cadenza.
9. Presenta la checklist di produzione.

Criteri di accettazione

- ✓ L'applicazione gira senza cicli duplicati né sottoscrizioni duplicate.
- ✓ La UI resta reattiva durante l'elaborazione in background.
- ✓ Ogni operazione di lunga durata ha gli stati di completamento, annullamento ed errore.
- ✓ L'hub multiutente non mantiene riferimenti diretti a pagine o controlli.
- ✓ Le griglie in binding si aggiornano in modo incrementale.
- ✓ Lo studente sa difendere tutte le ipotesi di rilascio.