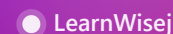


Il modello delle prestazioni di Wisej.NET

Guida della lezione · Guida al lab / esame



Guida della lezione

Il ciclo di richieste ed eventi, il push via WebSocket, lo stato di sessione come unità di capacità, i cinque contenitori di costo e la disciplina scenario-più-baseline con cui si confronta ogni traccia successiva. Che cosa significa per uno sviluppatore Wisej.NET che diagnostica i colli di bottiglia in un'app in produzione — e come affrontare questo modulo.



Focus di produzione

Il ciclo di richieste ed eventi, il push via WebSocket, lo stato di sessione come unità di capacità, i cinque contenitori di costo e la disciplina scenario-più-baseline con cui si confronta ogni traccia successiva.

1. Il problema in produzione

Un team di supporto segnala che WisejPerfLab "è diventato lento". Quella frase non contiene nessuna misura, nessuno scenario e nessun livello. La dashboard potrebbe essere in attesa di SQL, il gestore dell'aggiornamento potrebbe formattare cinquantamila celle, il browser potrebbe ridisegnare una griglia che non avrebbe mai dovuto essere resa per intero, oppure il server potrebbe semplicemente trattenere i form di duecento sessioni chiusi un'ora fa. Ognuno di questi casi ha una correzione diversa e uno strumento di Visual Studio diverso, e lo sviluppatore che apre il profiler prima di decidere di quale si tratta raccoglierà una traccia che non può rispondere alla domanda. Questo modulo costruisce prima il modello mentale: che cosa succede davvero tra un clic nel browser e una schermata ridisegnata, quanto costa ciascuno di quei passi e come mettere per iscritto uno scenario e un budget, così che i sei moduli successivi abbiano qualcosa con cui confrontarsi.

2. Concetti chiave

Concetto	Perché è importante
UI lato server	I controlli, le loro proprietà e il loro stato vivono nel processo del server e il browser rende ciò che il server descrive, quindi ogni oggetto di UI che crei è memoria del server e ogni modifica alla UI è CPU del server più un payload.
round trip di un evento	Il percorso completo di un'azione dell'utente: evento nel browser, trasporto al server, il tuo gestore, accesso ai dati, modifiche ai controlli, l'aggiornamento serializzato di ritorno, il ridisegno nel browser — l'unità che l'utente percepisce davvero.
la sessione come unità di capacità	Una sessione possiede pagine, form, controlli, timer, selezioni, binding e dati in cache; alle domande sulla capacità si risponde per sessione, non per richiesta, perché quella memoria resta residente tra un clic e l'altro.
push via WebSocket	Il canale attraverso cui il server invia aggiornamenti che l'utente non ha chiesto; rende economico consegnare dati in tempo reale e rende costoso un task in background senza limitazione, perché ogni stato intermedio diventa un frame.
i cinque contenitori di costo	CPU nel tuo codice, blocchi e attese, allocazione e ritenzione, accesso ai dati e I/O, e payload degli aggiornamenti — la classificazione che decide quale strumento del profiler può anche solo vedere il tuo problema.
latenza e throughput	La latenza è quanto aspetta un singolo utente; il throughput è quante sessioni il server riesce a sostenere. Una correzione che elimina un'attesa aiuta la latenza; una correzione che elimina lavoro o memoria aiuta entrambi, e alcune correzioni della latenza costano throughput.
scenario e baseline	Una sequenza di azioni dell'utente con un nome e ripetibile, più l'ambiente, la configurazione di build, la dimensione del dataset e i numeri misurati che ha prodotto — senza di essa una traccia è rumore e un confronto successivo non ha significato.

3. Il filo della lezione

Quanto costa davvero un clic

In un'applicazione web stateless l'unità di misura naturale è la richiesta: arriva, viene servita, il processo se ne dimentica. Wisej.NET ti offre invece il modello di programmazione di WinForms — una **Page** o un **Form** con controlli di cui imposti le proprietà — e lo paga mantenendo vivo quel grafo di oggetti sul server per tutto il tempo in cui l'utente è connesso. Il browser contiene una resa della UI; il server contiene la UI.

Segui un clic dall'inizio alla fine. Il browser genera un evento e lo invia sulla connessione aperta. Il server individua la sessione, esegue il dispatch nel tuo gestore e da lì in poi è tutto normale C#: interroghi un database, costruisci stringhe, imposti **Text** su una label, assegni un **DataSource**, aggiungi righe. Wisej.NET registra quelle modifiche alle proprietà e, quando il gestore termina, serializza la differenza risultante e la invia al browser, che ridisegna. Su quel percorso stanno cinque costi distinti — il rendering del browser, rete e payload, la CPU del tuo gestore, l'attesa di database e I/O e l'eventuale elaborazione in

background avviata dal clic — e un sintomo che all'utente sembra identico ("la schermata è lenta") può venire da uno qualsiasi di questi.

Due conseguenze contano subito. Primo, gli eventi client di una sessione vengono elaborati uno alla volta, quindi un gestore che impiega quattro secondi blocca quell'utente per quattro secondi, qualunque altra cosa stia facendo il server. Secondo, un gestore che consuma CPU non rallenta solo la propria sessione; compete per gli stessi core con ogni altra sessione dell'istanza. Per questo il corso misura il costo per scenario e poi ragiona sulla capacità per sessione, invece di dichiarare "abbastanza veloce" un clic veloce con un solo utente.

Cinque contenitori, cinque strumenti diversi

La classificazione non è accademica — è il modo in cui scegli uno strumento. Il lavoro di CPU nel tuo codice è ciò che mostrano CPU Usage e il flame graph. Il tempo passato in blocchi e attese è in gran parte invisibile a CPU Usage ed è a questo che servono Instrumentation e .NET Async. La pressione sulle allocazioni e la memoria trattenuta richiedono rispettivamente .NET Object Allocation e gli snapshot di Memory Usage, e sono problemi diversi: uno scenario può allocare molto e non trattenere nulla, oppure allocare quasi nulla e far perdere un intero form. La latenza di accesso ai dati e di I/O spetta agli strumenti Database e File I/O. Il costo del payload degli aggiornamenti è l'unico contenitore che Visual Studio non vede affatto, e le prove devono fornirle gli strumenti per sviluppatori del browser.

L'abitudine che ne deriva è una domanda da farti prima di avviare una raccolta: *in quale contenitore credo che stia il problema, e che cosa mi dimostrerebbe che sbaglio?* Una schermata lenta con poca CPU sul server non è un problema di CPU, per quante tracce di CPU tu raccolga. Il Modulo 2 trasforma tutto questo in una matrice di scelta degli strumenti e in un flusso di raccolta ripetibile; questo modulo è quello in cui impari a formulare l'ipotesi in modo esplicito, così che la traccia possa smentirla.

Rendere misurabile WisejPerfLab

Il lab crea WisejPerfLab, la dashboard per le operazioni di supporto che l'intero corso ottimizza, e poi fa il lavoro poco affascinante di renderla misurabile. Lo strumento è un piccolo servizio `ScenarioProbe` registrato nell'host builder e iniettato nelle pagine. `Measure()` restituisce uno scope `IDisposable` che avvia uno `Stopwatch`, scrive una riga di log strutturata "PERF start" e, al rilascio, scrive "PERF end" con i millisecondi trascorsi e il numero di righe. Il logging strutturato qui è importante: il nome dello scenario, l'azione dell'utente e il numero di righe sono campi separati, così gli stessi tre scenari possono essere filtrati e confrontati in ogni esecuzione successiva.

```
using (_probe.Measure("Dashboard", "Refresh", rows.Count))
{
    // tutto ciò che sta dentro questo scope è un unico scenario misurato:
    // query, proiezione, modifica dei controlli e l'aggiornamento che segue
    RefreshKpis(rows);
    RefreshGrid(rows);
}
// PERF end Dashboard Refresh elapsedMs=1840 rows=5000
```

Tre scenari ricevono una sonda: aggiornamento della dashboard, ricerca dei ticket ed espansione dell'albero dei clienti. Sono stati scelti perché cadono in contenitori diversi — l'aggiornamento si rivelerà

CPU nel codice di formattazione, la ricerca si rivelerà tempo di database, l'espansione dell'albero si rivelerà payload e numero di controlli trattenuti — e al Modulo 7 avrai i numeri prima e dopo per tutti e tre.

Baseline e budget, messi per iscritto

Una baseline è lo scenario più tutto ciò che potrebbe cambiarne il risultato: la macchina, la configurazione di build, la dimensione del dataset, se il database è locale, quale browser, quante sessioni erano connesse e se l'app è stata prima riscaldata. Salta uno qualsiasi di questi elementi e il confronto che farai al Modulo 3 sarà un'opinione, non una prova. Release conta più di quanto gli sviluppatori si aspettino — Debug cambia l'ottimizzazione del JIT, aggiunge overhead e può produrre stack di chiamate che ti mandano a inseguire la funzione sbagliata.

Il budget è l'altra metà: una soglia di accettazione per scenario, scritta prima che inizi l'ottimizzazione. "Aggiornamento della dashboard sotto gli 800 ms di tempo del gestore sul server dopo il riscaldamento, con meno di 40 MB trattenuti per sessione inattiva" è un budget. Dà a ogni modulo successivo un superato o non superato invece di una sensazione, ti dice quando *smettere* di ottimizzare e al Modulo 7 diventa la fonte delle soglie degli health check e degli alert di produzione. Scrivi il budget nello stesso documento della baseline e annota accanto a ogni soglia quale strumento la dimostrerebbe.



Errori tipici

Fare il profiling di una build Debug e trattare il risultato come prova di produzione. Ottimizzare un metodo prima di aver definito lo scenario end-to-end. Ignorare la memoria di sessione trattenuta perché la CPU sembra accettabile. Scambiare un ritardo di rete o di WebSocket per CPU del server. Misurare una sola sessione e dare per scontato che il numero valga anche con duecento.



La regola delle prove

Definisci lo scenario prima di aprire un profiler, raccogli da una build Release senza debugger, cambia una sola cosa, poi riesegui lo stesso scenario e confronta. Classifica il ritardo in uno dei cinque contenitori — CPU nel tuo codice, blocchi e attese, allocazione e ritenzione, accesso ai dati e I/O, oppure payload degli aggiornamenti — perché ogni contenitore ha il suo strumento di Visual Studio, e un'ottimizzazione che non puoi difendere con una traccia prima e dopo è solo un'ipotesi.



Obiettivi di apprendimento

Alla fine di questo modulo sarai in grado di:

- ✓ Spiegare il ciclo di richieste ed eventi di Wisej.NET, il push via WebSocket, lo stato di sessione e il modello di memoria del server.
- ✓ Distinguere il tempo di rendering del browser, il tempo di rete e payload, il tempo dei gestori di eventi sul server, il tempo di database e il tempo dei task in background.
- ✓ Creare uno scenario di prestazioni ripetibile con metriche di baseline e soglie di accettazione.
- ✓ Individuare quali domande sulle prestazioni richiedono gli strumenti di Visual Studio per CPU, memoria, database, async, contatori o instrumentation.

Guida al lab / esame

Che cosa costruirai nel lab pratico, l'approccio suggerito e i deliverable richiesti.



Lab pratico

Crea la soluzione WisejPerfLab con una dashboard di supporto, una griglia dei ticket e un albero dei clienti alimentati da dati di seed, poi rendila misurabile. Aggiungi un servizio ScenarioProbe che racchiude uno scenario in uno scope con Stopwatch e scrive log PERF strutturati di inizio/fine con il nome dello scenario, l'azione dell'utente, i millisecondi trascorsi e il numero di righe, registralo nell'host builder e racchiudici tre scenari: aggiornamento della dashboard, ricerca dei ticket ed espansione dell'albero dei clienti. Esegui ogni scenario una volta di riscaldamento, poi registra la baseline e scrivi un budget di prestazioni che indica ambiente, configurazione di build, dimensione del dataset e una soglia di accettazione per ogni scenario.

Punto di partenza

Parti da un'applicazione Wisej.NET vuota e da Visual Studio 2022 con gli strumenti di profiling installati. Non esiste ancora nulla: questo lab costruisce le tre schermate che il resto del corso ottimizza e le popola con dati sufficienti a renderle lente di proposito — circa 50.000 ticket e una gerarchia di clienti profonda diverse migliaia di nodi. Non correggere nulla di ciò che noti. Lo scopo del Modulo 1 è misurare l'app così com'è costruita.

Approccio suggerito

- ✓ Crea la soluzione WisejPerfLab e tre schermate: una `Page` di dashboard con label per i KPI, un pulsante di aggiornamento e un grafico; una griglia dei ticket collegata ai dati di seed dei ticket; e un `TreeView` dei clienti. Popola i dati da uno script o da un `IHostedService` così che ogni esecuzione parta dallo stesso dataset.
- ✓ Scrivi `ScenarioProbe` come piccolo servizio: `Measure(scenario, userAction, rowCount)` restituisce uno scope `IDisposable` che avvia uno `Stopwatch` e scrive nel log "PERF start", e in `Dispose` scrive "PERF end" con `elapsedMs` e `rows` come campi strutturati, non con concatenazione di stringhe.
- ✓ Registralo come singleton nell'host builder accanto al provider di logging su console, e rendi il container raggiungibile dalle tue pagine Wisej.NET così che ogni schermata possa ottenere la sonda di cui ha bisogno.
- ✓ Racchiudi i tre scenari: aggiornamento della dashboard, ricerca dei ticket ed espansione dell'albero dei clienti. Metti lo scope della sonda attorno all'intera azione dell'utente — query, proiezione e modifica dei controlli — non solo attorno alla query, perché il lavoro di UI fa parte di ciò che l'utente aspetta.
- ✓ Passa a Release, esegui ogni scenario una volta per assorbire il rumore di avvio e del JIT, poi eseguillo tre volte e registra la mediana. Scrivi insieme la nota di baseline e il budget, e accanto a ogni soglia indica lo strumento di Visual Studio che la dimostrerebbe o la smentirebbe.

Deliverable richiesti

- ✓ Soluzione WisejPerfLab con dashboard, griglia dei ticket e albero dei clienti su dati di seed
- ✓ Servizio ScenarioProbe che scrive log PERF strutturati di inizio/fine con millisecondi trascorsi e numero di righe
- ✓ Tre scenari strumentati: aggiornamento della dashboard, ricerca dei ticket ed espansione dell'albero dei clienti
- ✓ Nota di baseline che registra ambiente, configurazione di build, dimensione del dataset e procedura di riscaldamento
- ✓ Budget di prestazioni con una soglia di accettazione per scenario e lo strumento che dimostrerebbe ciascuna

Domande di revisione per lo studente

- ✓ L'aggiornamento della dashboard registra 1.900 ms ma la CPU del server si muove appena durante lo scenario. Quale dei cinque contenitori esclude questo dato, verso quale punta, e qual è la misura successiva più economica?
- ✓ La sonda misura il gestore. Indica due costi che l'utente subisce sullo stesso clic e che il tuo numero `elapsedMs` non include, e spiega come misureresti ciascuno.
- ✓ Perché la nota di baseline deve registrare la dimensione del dataset e la configurazione di build, quando la parte interessante sembrano essere il nome dello scenario e il tempo trascorso?



La regola delle prove

Definisci lo scenario prima di aprire un profiler, raccogli da una build Release senza debugger, cambia una sola cosa, poi riesegui lo stesso scenario e confronta. Classifica il ritardo in uno dei cinque contenitori — CPU nel tuo codice, blocchi e attese, allocazione e ritenzione, accesso ai dati e I/O, oppure payload degli aggiornamenti — perché ogni contenitore ha il suo strumento di Visual Studio, e un'ottimizzazione che non puoi difendere con una traccia prima e dopo è solo un'ipotesi.