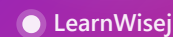


Rutas críticas de CPU, controladores de eventos y trabajo de interfaz en el servidor

Guía de la lección · Guía del laboratorio / examen



Guía de la lección

Código de interfaz costoso en el servidor dentro de controladores de eventos, temporizadores, refrescos de layout y lógica de formato, y los patrones de caché, agrupación y view model que vuelven a hacer barato un clic. Qué significa esto para un desarrollador de Wisej.NET que diagnostica cuellos de botella en una aplicación en producción, y cómo leer este módulo.



Enfoque en producción

Código de interfaz costoso en el servidor dentro de controladores de eventos, temporizadores, refrescos de layout y lógica de formato, y los patrones de caché, agrupación y view model que vuelven a hacer barato un clic.

1. El problema en producción

El Módulo 2 terminó con un nombre: `TicketFormatter.FormatRow`, un 68,9 % de CPU propia, llamado cincuenta mil veces en un solo clic. Esa es la forma de casi todos los problemas de CPU en una interfaz del lado del servidor. Nadie escribió un método lento; alguien escribió un método razonable y luego hizo que se ejecutara una vez por celda, en cada actualización, para siempre. El segundo culpable del dashboard de WisejPerfLab es su imagen especular: `RebuildKpiPanel` se llama una sola vez, pero libera y vuelve a crear un panel lleno de controles que no necesitaban cambiar. Uno es un problema de número de llamadas y el otro un problema de trabajo por llamada, y este módulo trata de reconocer cuál tienes y corregirlo sin conjeturas.

2. Conceptos clave

Concepto	Por qué importa
coste del controlador de eventos	Un clic, un cambio de selección o un tick de temporizador se ejecuta en el servidor y bloquea el siguiente evento de esa sesión, así que la cantidad de lógica que desencadena el clic es el tiempo que el usuario se queda congelado.
flame graph	Los mismos datos del árbol de llamadas dibujados como barras de ancho proporcional, lo que hace evidente de un vistazo un único frame ancho: una función que se come el escenario.
CPU propia frente a número de llamadas	Una CPU propia alta con pocas llamadas significa que el cuerpo es costoso; un coste por llamada bajo con un número de llamadas enorme significa que la solución es llamarla menos, no micro-optimizarla.
view model precalculado	Un objeto pequeño que lleva cadenas y valores listos para mostrar, construido una sola vez por un servicio, para que la capa de interfaz asigne valores en lugar de formatearlos.
modificación agrupada de la interfaz	Asignar en una sola pasada, con valores preparados, las propiedades que el usuario debe ver, en lugar de modificar controles dentro de un bucle que además hace trabajo.
reconstrucción de controles	Liberar y volver a crear controles para actualizarlos; vuelve a ejecutar la construcción, el layout y el theming, y casi siempre es más costoso que actualizar los controles que ya existen.
caché de consulta inmutable	Datos de referencia compartidos por toda la aplicación y que nunca se modifican; es seguro guardarlos en caché globalmente, a diferencia del estado de cada usuario, que debe quedarse en la sesión.

3. Desarrollo de la lección

Encontrar en el flame graph el código que importa

Empieza donde te dejó el Módulo 2: una traza de CPU Usage con la línea de tiempo acotada al clic. Cambia al flame graph y busca anchura. Un único frame ancho es una función que domina el escenario; un frame ancho formado por cientos de franjas finas idénticas es la misma función llamada una y otra vez. Después usa *Show Hot Path* en el árbol de llamadas para confirmarlo, y la vista de llamadores/llamados para averiguar quién provoca las llamadas.

Las reglas de interpretación son breves. Una CPU propia alta significa que el propio cuerpo de la función es costoso: busca dentro construcción de cadenas, reflexión, serialización, conversión de imágenes o una ordenación. Una total alta con una propia baja significa que el coste está en las hijas: sigue bajando. Y si Instrumentation muestra un número de llamadas enorme con un tiempo por llamada microscópico, deja de optimizar el cuerpo: el cambio rentable está más arriba, donde está el bucle.

En WisejPerfLab la traza muestra ambos patrones en el mismo controlador, y por eso el módulo corrige los dos antes de volver a medir. Corregir solo uno y cantar victoria es la forma de acabar con una optimización imposible de demostrar.

Por qué se encarecen los eventos de interfaz

El catálogo de rutas críticas de CPU en una aplicación Wisej.NET es corto y repetitivo: formatear cada celda del grid en cada actualización; reconstruir árboles de controles enteros en lugar de actualizar valores; convertir o redimensionar imágenes una y otra vez; analizar JSON o XML en la capa de interfaz; reflexión dentro de bucles; ordenar y filtrar grandes colecciones en memoria en cada evento; recalcular valores de tema o de layout; y preparar una exportación o un informe de forma síncrona en la ruta del clic.

Todos comparten una causa. Cada uno es código que sería perfectamente aceptable una vez, colocado en un sitio que se ejecuta por fila, por celda o por tick. Un formateador que cuesta 21 microsegundos está bien; el mismo formateador invocado para cincuenta mil filas es un segundo de sesión congelada. Y como una sesión de Wisej.NET procesa sus eventos de uno en uno, ese segundo no es solo problema del dashboard: retrasa lo siguiente que haga el usuario y consume un núcleo por el que compiten las demás sesiones de la instancia.

View models en caché y actualizaciones agrupadas

La corrección del laboratorio tiene dos mitades. Primero, el formato sale de la interfaz.

`DashboardService.GetSnapshot()` devuelve un `DashboardSnapshot` cuyas propiedades ya son cadenas: `OpenTicketsText`, `OverdueTicketsText`, `AverageAgeText`, más una pequeña lista de filas para el gráfico. Todo lo costoso —la agregación, el formato sensible a la cultura, la aritmética de antigüedades— ocurre una sola vez, en un servicio que se puede probar con tests unitarios y perfilar por separado.

Segundo, el controlador se convierte en una secuencia de asignaciones de valores ya preparados.

Desactiva el botón, modifica las etiquetas y el origen de datos del gráfico una sola vez, y lo reactiva en un `finally`. No se formatea nada, no se recrea nada, y el ámbito de la sonda del Módulo 1 sigue envolviéndolo todo, de modo que el cambio aparece en la misma línea de log que vienes comparando desde la línea base.

```
private void btnRefresh_Click(object sender, EventArgs e)
{
    using (_probe.Measure("Dashboard", "Refresh"))
    {
        btnRefresh.Enabled = false;
        try
        {
            var snapshot = _dashboardService.GetSnapshot(); // formatted once
            lblOpen.Text = snapshot.OpenTicketsText; // assign, don't format
            lblOverdue.Text = snapshot.OverdueTicketsText;
            lblAvgAge.Text = snapshot.AverageAgeText;
            chartLoad.DataSource = snapshot.ChartRows;
        }
        finally { btnRefresh.Enabled = true; }
    }
}
```

El patrón viene con dos advertencias. Guarda en caché a nivel de aplicación los datos de consulta inmutables —nombres de estado, etiquetas de prioridad, tablas de SLA—, pero nunca pongas estado mutable de cada usuario en un campo estático: en un servidor con varias sesiones eso es un error de

corrección, no una simple preferencia de estilo. Y no «arregles» el trabajo de CPU marcando el método como `async`: esperar con `await` no reduce el cálculo, solo traslada la misma CPU a otro hilo del servidor.

Demostrar el cambio con una segunda traza

El entregable no es una aplicación más rápida, es una optimización defendible. Vuelve a ejecutar el mismo escenario —mismo conjunto de datos, misma compilación, mismo calentamiento— y recoge una segunda traza de CPU Usage. Lo que buscas es una ruta crítica distinta, no solo una cifra menor: `FormatRow` debería haber desaparecido de lo alto del árbol, el número de llamadas debería haberse desplomado, y lo que esté ahora arriba se convierte en el siguiente candidato o en el punto en el que paras porque se cumple el presupuesto.

Recoge ambas trazas en una tabla antes/después con las tres columnas que importan: CPU total del escenario, función principal y su número de llamadas. Si las cifras se movieron pero la ruta crítica no cambió, probablemente mediste variabilidad. Y anota con honestidad el riesgo restante: un snapshot en caché introduce una ventana de datos desactualizados, y el revisor debe ver que lo sabes.



Errores habituales

Formatear controles fila por fila en cada actualización cuando bastaría un valor en caché. Llamar a `Application.Update` dentro de un bucle de CPU intensivo. Tratar una CPU total alta en código del framework como un bug del framework sin inspeccionar las llamadas hijas. Ocultar un problema de CPU tras palabras clave `async` que siguen haciendo el mismo trabajo en hilos del servidor. Guardar estado mutable del usuario en un campo estático.



Regla de evidencia

Define el escenario antes de abrir un profiler, recoge desde una compilación Release sin el depurador, cambia una sola cosa y vuelve a ejecutar el mismo escenario para comparar. Clasifica el retraso en uno de los cinco grupos —CPU en tu código, bloqueos y esperas, asignación y retención, acceso a datos y E/S, o carga útil de actualización—, porque cada grupo tiene su propia herramienta de Visual Studio, y una optimización que no puedes defender con una traza de antes y otra de después es una conjetura.



Objetivos de aprendizaje

Al terminar este módulo serás capaz de:

- ✓ Usar CPU Usage, el flame graph y el árbol de llamadas para localizar rutas lentas de gestión de eventos.
- ✓ Reconocer formato repetido, reflexión, serialización, trabajo con imágenes y bucles síncronos dentro de los eventos de interfaz.
- ✓ Reducir el trabajo de los controladores de eventos con caché, agrupación, carga diferida y view models precalculados.
- ✓ Distinguir el trabajo limitado por CPU del tiempo de bloqueo y del tiempo de espera de la base de datos.

Guía del laboratorio / examen

Qué construirás en el laboratorio práctico, el enfoque sugerido y los entregables requeridos.



Laboratorio práctico

Arregla el botón Refresh de WisejPerfLab. Perfila el escenario de actualización con CPU Usage, abre el flame graph y Show Hot Path, e identifica a los dos culpables: un formateador que se ejecuta por celda en cada actualización y una reconstrucción de controles que vuelve a crear el panel de KPI cada vez. Sustitúyelos por un view model DashboardSnapshot cuyas cadenas de presentación se calculan una sola vez en el servicio, modifica las etiquetas y el origen de datos del gráfico una sola vez dentro de un ámbito de sonda en lugar de fila por fila, y mantén el botón desactivado en un try/finally mientras se ejecuta el trabajo. Vuelve a ejecutar el mismo escenario y captura la nueva ruta crítica.

Punto de partida

Abres WisejPerfLab con las trazas del Módulo 2 a mano: la actualización del dashboard cuesta unos 1840 ms, `TicketFormatter.FormatRow` acumula un 68,9 % de CPU propia en unas 50.000 llamadas, y `RebuildKpiPanel` libera y vuelve a crear el panel de KPI en cada clic. La capa de servicios devuelve ahora entidades en bruto y la página hace todo el formato.

Enfoque sugerido

- ✓ Reproduce primero el hallazgo: recoge tú mismo la traza de CPU Usage, abre el flame graph y confirma con la vista de Llamadores/Llamados qué bucle provoca las 50.000 llamadas al formateador. No cambies código que no hayas visto en una traza.
- ✓ Introduce un view model `DashboardSnapshot` con propiedades listas para mostrar —texto del número de abiertos, texto del número de vencidos, texto de la antigüedad media y una pequeña lista de filas para el gráfico— y constrúyelo una sola vez dentro de `DashboardService.GetSnapshot()`.
- ✓ Sacas todas las llamadas de formato de la página y llévalas al servicio, incluido el formato de números y duraciones sensible a la cultura. Guarda en caché solo los datos de consulta inmutables, como las etiquetas de estado y prioridad; mantén en la sesión todo lo que sea de cada usuario.
- ✓ Reescribe el controlador como asignaciones de valores preparados dentro del ámbito de sonda existente: desactiva el botón, asigna las etiquetas y el origen de datos del gráfico una sola vez y reactiva en un `finally`. Elimina la reconstrucción del panel de KPI y actualiza en su lugar los controles existentes.
- ✓ Vuelve a ejecutar el mismo escenario —mismo conjunto de datos, misma compilación, mismo calentamiento—, recoge una segunda traza de CPU Usage y construye la tabla antes/después con CPU total, función principal y número de llamadas.

Entregables requeridos

- ✓ Traza de CPU Usage «antes» que nombra el formateador repetido y la reconstrucción de controles con su CPU total y propia

- ✓ View model DashboardSnapshot con las cadenas de presentación precalculadas en el servicio
- ✓ Controlador de actualización agrupado que modifica los controles una sola vez dentro de un ámbito de sonda y reactiva el botón con try/finally
- ✓ Traza de CPU Usage «después» del mismo escenario con la nueva ruta crítica
- ✓ Tabla antes/después de CPU total, función principal y número de llamadas, más el riesgo restante

Preguntas de repaso

- ✓ Los dos culpables eran lentos, pero por razones opuestas. Explica cuál era un problema de número de llamadas y cuál de trabajo por llamada, y por qué la evidencia del profiler de cada uno tiene un aspecto distinto.
- ✓ Un compañero propone marcar el controlador de actualización como `async` para hacerlo más rápido. ¿Qué cambiaría realmente, qué no, y cómo lo mostraría la traza?
- ✓ Tu snapshot se construye una vez por actualización. ¿Qué riesgo de datos desactualizados o de corrección introduce, y qué tendrías que cambiar si el mismo snapshot se guardara en caché entre sesiones?



Regla de evidencia

Define el escenario antes de abrir un profiler, recoge desde una compilación Release sin el depurador, cambia una sola cosa y vuelve a ejecutar el mismo escenario para comparar. Clasifica el retraso en uno de los cinco grupos —CPU en tu código, bloqueos y esperas, asignación y retención, acceso a datos y E/S, o carga útil de actualización—, porque cada grupo tiene su propia herramienta de Visual Studio, y una optimización que no puedes defender con una traza de antes y otra de después es una conjetura.