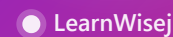


Controles de datos, cargas útiles del navegador y grandes superficies de interfaz

Guía de la lección · Guía del laboratorio / examen



Guía de la lección

Grids, listas, árboles y repetidores a escala: modo virtual, paginación, proyección y carga diferida de nodos, y el uso de la evidencia de red del navegador junto a la de Visual Studio para dimensionar la carga útil de actualización. 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

Grids, listas, árboles y repetidores a escala: modo virtual, paginación, proyección y carga diferida de nodos, y el uso de la evidencia de red del navegador junto a la de Visual Studio para dimensionar la carga útil de actualización.

1. El problema en producción

Las grandes superficies de interfaz son la fuente más habitual de problemas de rendimiento en Wisej.NET, y son el grupo de coste que el profiler no puede ver del todo. El grid de tickets de WisejPerfLab enlaza cincuenta mil entidades con sus propiedades de navegación; el árbol de clientes construye todos los nodos al arrancar. En escritorio ese patrón era aceptable, porque el control y los datos vivían en el mismo proceso que la pantalla. Aquí el mismo código paga cuatro veces: memoria del servidor para retener los objetos, CPU del servidor para construir y serializar el estado de la interfaz, red para transportar la actualización y tiempo del navegador para renderizarla. En este módulo la mejor optimización no suele ser un algoritmo más rápido, sino *que exista menos interfaz a la vez*.

2. Conceptos clave

Concepto	Por qué importa
fila de proyección	Un tipo pequeño que contiene solo las columnas que muestra una pantalla, enlazado en lugar del grafo de entidades, para que ni el servidor ni el navegador carguen con datos que nadie mira.
modo virtual	El grid guarda un número de filas en lugar de objetos de fila y pide los valores de las celdas a medida que los necesita, lo que elimina los objetos por fila y el renderizado de filas a las que nadie ha llegado.
CellValueNeeded	El evento que proporciona un valor bajo demanda en modo virtual; se ejecuta muy a menudo, así que es una ruta crítica por construcción y debe devolver el valor rápido.
paginación y filtrado en el servidor	Pedir a la base de datos la página que el usuario está mirando, en lugar de cargarlo todo y filtrar en el código de interfaz.
carga diferida de nodos	Construir los hijos del árbol cuando se expande una rama, con un marcador de recuento mientras está contraída, en lugar de materializar toda la jerarquía al arrancar.
carga útil de actualización	La diferencia serializada que el servidor envía tras un controlador; las cargas grandes o frecuentes cuestan tiempo de red y de navegador que ninguna herramienta de Visual Studio muestra.
la compresión no ahorra el trabajo	Wisej.NET comprime el tráfico HTTP y WebSocket, lo que reduce los bytes pero no abarata producir, transmitir o renderizar una actualización innecesaria.

3. Desarrollo de la lección

Grids: enlaza menos y luego retén menos

La corrección del grid tiene dos mitades independientes, y hacer solo una es el error habitual. La primera mitad es la forma de los datos. Enlazar entidades arrastra propiedades de navegación, columnas no usadas y campos de texto grandes a la memoria del servidor y a cada fila que construye la interfaz. Sustituirlo por una proyección `TicketGridRow` —número, nombre del cliente, estado, texto de antigüedad y nada más— reduce el grafo de objetos, las asignaciones que mediste en el Módulo 4 y la actualización que recibe el navegador.

La segunda mitad es cuántas filas existen. En modo virtual el grid guarda un `RowCount` y lanza `CellValueNeeded` para las celdas que realmente muestra, así que cincuenta mil filas dejan de ser cincuenta mil objetos de fila. El compromiso es explícito: ese evento se ejecuta ahora constantemente, así que su controlador tiene que ser rápido y seguro para hilos, y debe apoyarse en un servicio de consultas paginadas y no en una búsqueda que acceda a la base de datos una vez por celda. Precalcula las cadenas de presentación en la proyección para que el controlador solo seleccione un campo, y perfila el propio controlador: el modo virtual traslada el coste, no lo elimina.

Árboles, listas y repetidores

El mismo principio da nueva forma a los demás controles grandes. Un `TreeView` construido entero al arrancar paga por cada rama que nadie abre; cargar los hijos al expandir, con un marcador de recuento en las ramas contraídas, hace que el coste sea proporcional a lo que el usuario mira de verdad. Las listas y los repetidores admiten también el tratamiento virtual, y los repetidores tienen una trampa adicional: una plantilla de tarjeta con una docena de controles anidados se multiplica por el número de elementos, así que es preferible un resumen con navegación al detalle que un árbol de controles rico repetido mil veces.

La precarga es el contrapeso de la carga diferida. Un desplazamiento virtual que pide exactamente lo que hay en pantalla puede resultar lento en un desplazamiento rápido, así que pedir un poco por delante de la zona visible suele compensar. Es una decisión de ajuste que se toma a partir de mediciones, no un valor predeterminado que se aplica en todas partes.

La evidencia que el profiler no te puede dar

Visual Studio te dice lo que hizo el servidor: CPU para construir las filas, asignaciones para retenerlas, consultas para obtenerlas. No dice nada de lo que viajó por la red. Para eso, el instrumento es el panel de red del navegador: el tamaño de la carga útil de la actualización tras cargar un grid, las tramas WebSocket tras expandir un árbol y el tiempo que el navegador dedicó a renderizarlas.

Dos advertencias. La compresión hace que los recuentos de bytes parezcan tranquilizadores; una actualización que nunca debió enviarse sigue siendo trabajo desperdiciado en ambos extremos, así que compara el *número y el tamaño de las actualizaciones* antes y después, no solo el total comprimido. Y aunque desactivar la compresión de WebSocket puede ayudarte a leer las tramas al depurar, no midas el rendimiento con configuraciones de depuración que producción no va a usar.

```
dataGridView1.VirtualMode = true;
dataGridView1.RowCount = _tickets.Count(CurrentFilter); // rows, not objects
dataGridView1.CellValueNeeded += (s, e) =>
{
    var row = _tickets.GetProjectedRow(CurrentFilter, e.RowIndex); // paged, cached
    e.Value = e.ColumnIndex switch
    {
        0 => row.Number, 1 => row.Customer, 2 => row.Status, 3 => row.AgeText, _ => null
    };
};
```

Reglas de ajuste que conviene recordar

Renderiza menos filas, menos nodos y menos controles. Envía menos actualizaciones. Enlaza objetos más pequeños. Aplaza el detalle hasta que el usuario lo pida. Guarda en caché los datos de consulta inmutables; nunca guardes en caché globalmente datos mutables de cada usuario. Prefiere paginar y filtrar en el servidor a cargarlo todo y filtrar en el código de interfaz. Y no actualices un grid entero cuando ha cambiado una fila o una etiqueta: esa sola costumbre explica una parte sorprendente de la carga útil en los dashboards movidos por temporizadores.

Debajo de todas ellas hay una pregunta de diseño más que técnica: ¿qué pregunta responde esta pantalla para el usuario? Una pantalla diseñada en torno a todos los registros posibles siempre será costosa de hacer rápida. Una pantalla diseñada en torno a la decisión que tiene que tomar el usuario suele ser barata

por construcción, y las cifras de antes y después del laboratorio son el argumento que llevas a esa conversación.



Errores habituales

Enlazar a cada fila entidades ORM completas con grandes grafos de navegación. Construir el árbol entero al arrancar en lugar de cargar los nodos bajo demanda. Actualizar dashboards completos en cada tick del temporizador. Suponer que la compresión arregla las cargas útiles sobredimensionadas. Escribir un controlador `CellValueNeeded` que consulta la base de datos una vez por celda.



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:

- ✓ Perfilar escenarios de DataGridView, ListView, TreeView y DataRepeater en cuanto a CPU, asignaciones y coste de renderizado.
- ✓ Aplicar de forma adecuada las estrategias de modo virtual, paginación, filtrado, proyección y precarga.
- ✓ Reducir la rotación de cargas útiles WebSocket/HTTP limitando las modificaciones de la interfaz y el tamaño de la superficie de datos.
- ✓ Usar las herramientas de desarrollo del navegador junto con Visual Studio para diagnosticar problemas de red y de tamaño de las actualizaciones.

Guía del laboratorio / examen

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



Laboratorio práctico

Reduce las dos superficies más grandes de WisejPerfLab. El grid de tickets enlaza 50.000 entidades completas con propiedades de navegación: sustituye el enlace por una proyección `TicketGridRow` con solo las columnas visibles, establece `VirtualMode` y `RowCount` y sirve las celdas desde un controlador `CellValueNeeded` respaldado por un servicio de consultas paginadas. El árbol de clientes construye todos los nodos al arrancar: carga los hijos al expandir, usando un marcador de recuento para las ramas contraídas. Mide cada pantalla antes y después con CPU Usage y .NET Object Allocation en el servidor, y con el panel de red del navegador para el tamaño de la carga útil de actualización.

Punto de partida

WisejPerfLab tras el Módulo 4: las asignaciones durante la búsqueda han bajado y el formulario de detalle ya no tiene fugas, pero el grid sigue enlazando la lista completa de entidades y el árbol de clientes sigue construyéndose de una pasada al arrancar. El presupuesto del Módulo 1 tiene un objetivo de expansión del árbol que la aplicación incumple por mucho.

Enfoque sugerido

- ✓ Mide las dos pantallas antes de tocarlas: CPU Usage y .NET Object Allocation para la carga del grid y la expansión del árbol, más el panel de red del navegador para la carga útil de actualización que produce cada una. Anota el número y el tamaño de las actualizaciones, no solo el total comprimido.
- ✓ Define `TicketGridRow` solo con las columnas mostradas y sus cadenas de presentación precalculadas, y haz que el servicio de consultas lo devuelva. Nada con propiedades de navegación o grandes columnas de texto debería llegar a la interfaz.
- ✓ Pasa el grid a modo virtual: establece `VirtualMode`, fija `RowCount` a partir de una consulta de recuento y responde a `CellValueNeeded` desde una página paginada y en caché de filas de proyección. Trata ese controlador como una ruta crítica: ninguna llamada a la base de datos por celda, ningún formato y ningún bloqueo que frene las peticiones paralelas.
- ✓ Rehaz el árbol de clientes para que cargue los hijos al expandir, mostrando un marcador de recuento en las ramas contraídas, y decide a partir de mediciones si una pequeña precarga por delante de la zona visible compensa las peticiones adicionales.
- ✓ Vuelve a ejecutar ambos escenarios con el mismo conjunto de datos y la misma compilación, y monta la comparación: CPU del servidor, asignaciones y filas retenidas según Visual Studio; número de actualizaciones y tamaño de la carga útil según el navegador.

Entregables requeridos

- ✓ Proyección `TicketGridRow` enlazada en lugar de entidades completas, solo con las columnas mostradas

- ✓ Grid en modo virtual con RowCount y un controlador CellValueNeeded servido desde un servicio de consultas paginadas
- ✓ Árbol de clientes que carga los nodos hijos al expandir con un marcador de recuento para las ramas contraídas
- ✓ Cifras de CPU y asignaciones antes/después para los escenarios de carga del grid y expansión del árbol
- ✓ Evidencia de red del navegador que compara el tamaño de la carga útil de actualización antes y después, con la advertencia sobre la compresión

Preguntas de repaso

- ✓ El modo virtual eliminó los objetos de fila, pero tu traza de CPU apenas mejoró. ¿Qué te dice eso sobre dónde estaba realmente el coste y qué medición mirarías a continuación?
- ✓ Tu controlador `CellValueNeeded` es correcto pero lento con un desplazamiento rápido. Enumera tres cosas que podría estar haciendo y que un controlador por celda nunca debe hacer.
- ✓ La carga útil comprimida de la expansión del árbol apenas cambió tras la carga diferida, pero los usuarios dicen que la pantalla parece más rápida. Da una explicación plausible que el recuento de bytes por sí solo ocultaría.



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.