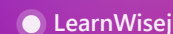


El modelo de rendimiento de Wisej.NET

Guía de la lección · Guía del laboratorio / examen



Guía de la lección

El ciclo de peticiones y eventos, el push por WebSocket, el estado de sesión como unidad de capacidad, los cinco grupos de coste y la disciplina de escenario más línea base con la que se compara cada traza posterior. 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

El ciclo de peticiones y eventos, el push por WebSocket, el estado de sesión como unidad de capacidad, los cinco grupos de coste y la disciplina de escenario más línea base con la que se compara cada traza posterior.

1. El problema en producción

Un equipo de soporte informa de que WisejPerfLab «se ha vuelto lenta». Esa frase no contiene ninguna medición, ningún escenario y ninguna capa. El dashboard podría estar esperando a SQL, el controlador de actualización podría estar formateando cincuenta mil celdas, el navegador podría estar repintando un grid que nunca debió renderizarse entero, o el servidor podría estar reteniendo simplemente los formularios de doscientas sesiones que se cerraron hace una hora. Cada una de esas causas tiene una corrección distinta y una herramienta de Visual Studio distinta, y un desarrollador que abre el profiler antes de decidir cuál es recogerá una traza que no puede responder a la pregunta. Este módulo construye primero el modelo mental: qué ocurre realmente entre un clic en el navegador y una pantalla repintada, cuánto cuesta cada uno de esos pasos, y cómo dejar por escrito un escenario y un presupuesto para que los seis módulos siguientes tengan algo con lo que medir.

2. Conceptos clave

Concepto	Por qué importa
interfaz en el servidor	Los controles, sus propiedades y su estado residen en el proceso del servidor y el navegador renderiza lo que el servidor describe, así que cada objeto de interfaz que creas es memoria del servidor y cada modificación de la interfaz es CPU del servidor más una carga útil.
viaje de ida y vuelta de un evento	El recorrido completo de una acción del usuario: evento del navegador, transporte al servidor, tu controlador, acceso a datos, modificaciones de controles, la actualización serializada de vuelta y el repintado del navegador; es la unidad que el usuario percibe de verdad.
la sesión como unidad de capacidad	Una sesión posee páginas, formularios, controles, temporizadores, selecciones, enlaces de datos y datos en caché; las preguntas de capacidad se responden por sesión, no por petición, porque esa memoria permanece residente entre clics.
push por WebSocket	El canal por el que el servidor envía actualizaciones que el usuario no pidió; abarata la entrega de datos en vivo y encarece una tarea en segundo plano sin limitación, porque cada estado intermedio se convierte en una trama.
los cinco grupos de coste	CPU en tu código, bloqueos y esperas, asignación y retención, acceso a datos y E/S, y carga útil de actualización: la clasificación que decide qué herramienta del profiler puede siquiera ver tu problema.
latencia frente a throughput	La latencia es cuánto espera un usuario; el throughput es cuántas sesiones sostiene el servidor. Una corrección que elimina una espera mejora la latencia; una que elimina trabajo o memoria mejora ambas, y algunas correcciones de latencia cuestan throughput.
escenario y línea base	Una secuencia de acciones de usuario con nombre y repetible, más el entorno, la configuración de compilación, el tamaño del conjunto de datos y las cifras medidas que produjo; sin ella, una traza es ruido y una comparación posterior no tiene sentido.

3. Desarrollo de la lección

Lo que cuesta realmente un clic

En una aplicación web sin estado, la unidad natural de medida es la petición: llega, se atiende y el proceso la olvida. Wisej.NET te da en cambio el modelo de programación de WinForms —una `Page` o un `Form` con controles cuyas propiedades asignas— y lo paga manteniendo vivo ese grafo de objetos en el servidor mientras el usuario esté conectado. El navegador tiene una representación de la interfaz; el servidor tiene la interfaz.

Sigue un clic de principio a fin. El navegador lanza un evento y lo envía por la conexión abierta. El servidor localiza la sesión, despacha a tu controlador y, a partir de ahí, todo es C# normal: consultas una base de datos, construyes cadenas, asignas `Text` a una etiqueta, asignas un `DataSource`, añades filas. Wisej.NET registra esos cambios de propiedades y, cuando el controlador termina, serializa la diferencia resultante y la envía al navegador, que repinta. En ese recorrido hay cinco costes distintos —renderizado del navegador, red y carga útil, CPU de tu controlador, espera de base de datos y E/S, y cualquier trabajo en

segundo plano que iniciara el clic—, y un síntoma que para el usuario parece idéntico («la pantalla va lenta») puede venir de cualquiera de ellos.

Hay dos consecuencias que importan de inmediato. Primero, los eventos de cliente de una sesión se procesan de uno en uno, así que un controlador que tarda cuatro segundos congela a ese usuario cuatro segundos, haga lo que haga el resto del servidor. Segundo, un controlador que quema CPU no solo ralentiza su propia sesión: compite por los mismos núcleos que todas las demás sesiones de la instancia. Por eso el curso mide el coste por escenario y después razona sobre la capacidad por sesión, en lugar de declarar «suficientemente rápido» un clic rápido con un solo usuario.

Cinco grupos de coste, cinco herramientas distintas

La clasificación no es académica: es la forma de elegir herramienta. El trabajo de CPU en tu propio código es lo que muestran CPU Usage y el flame graph. El tiempo de bloqueo y espera es en gran parte invisible para CPU Usage, y para eso están Instrumentation y .NET Async. La presión de asignación y la memoria retenida necesitan .NET Object Allocation y los snapshots de Memory Usage respectivamente, y son problemas distintos: un escenario puede asignar mucho y no retener nada, o asignar casi nada y dejar escapar un formulario entero. La latencia de acceso a datos y de E/S corresponde a las herramientas Database y File I/O. El coste de la carga útil de actualización es el único grupo que Visual Studio no ve en absoluto, y la evidencia la tienen que aportar las herramientas de desarrollo del navegador.

El hábito que se deriva es una pregunta que te haces antes de iniciar una recogida: *¿en qué grupo creo que está esto y qué demostraría que me equivoco?* Una pantalla lenta con CPU del servidor baja no es un problema de CPU por muchas trazas de CPU que recojas. El Módulo 2 convierte esto en una matriz de selección de herramientas y en un flujo de recogida repetible; este módulo es donde aprendes a formular la hipótesis de forma explícita para que la traza pueda refutarla.

Hacer medible WisejPerfLab

El laboratorio crea WisejPerfLab, el dashboard de operaciones de soporte que ajusta todo el curso, y después hace el trabajo poco vistoso de volverlo medible. El instrumento es un pequeño servicio `ScenarioProbe` registrado en el host builder e inyectado en las páginas. `Measure()` devuelve un ámbito `IDisposable` que arranca un `Stopwatch`, escribe una línea de log estructurada «PERF start» y, al liberarse, escribe «PERF end» con los milisegundos transcurridos y el número de filas. El logging estructurado importa aquí: el nombre del escenario, la acción del usuario y el número de filas son campos separados, así que los mismos tres escenarios se pueden filtrar y comparar en todas las ejecuciones posteriores.

```
using (_probe.Measure("Dashboard", "Refresh", rows.Count))
{
    // everything inside this scope is one measured scenario:
    // query, projection, control mutation and the update that follows
    RefreshKpis(rows);
    RefreshGrid(rows);
}
// PERF end Dashboard Refresh elapsedMs=1840 rows=5000
```

Tres escenarios reciben una sonda: la actualización del dashboard, la búsqueda de tickets y la expansión del árbol de clientes. Se eligieron porque caen en grupos distintos —la actualización resultará ser CPU en código de formato, la búsqueda resultará ser tiempo de base de datos y la expansión del árbol resultará ser carga útil y número de controles retenidos—, y en el Módulo 7 tendrás cifras de antes y después para los tres.

Líneas base y presupuestos, por escrito

Una línea base es el escenario más todo lo que podría cambiar su resultado: la máquina, la configuración de compilación, el tamaño del conjunto de datos, si la base de datos es local, qué navegador, cuántas sesiones había conectadas y si la aplicación se calentó antes. Si te saltas cualquiera de esos datos, la comparación que hagas en el Módulo 3 será una discusión, no una evidencia. Release importa más de lo que los desarrolladores esperan: Debug cambia la optimización del JIT, añade sobrecarga y puede producir pilas de llamadas que te lleven a perseguir la función equivocada.

Un presupuesto es la otra mitad: un umbral de aceptación por escenario, escrito antes de empezar a ajustar. «Actualización del dashboard por debajo de 800 ms de tiempo de controlador en el servidor tras el calentamiento, con menos de 40 MB retenidos por sesión inactiva» es un presupuesto. Da a cada módulo posterior un aprobado o un suspenso en lugar de una sensación, te dice cuándo *parar* de optimizar y, en el Módulo 7, se convierte en el origen de los umbrales del health check y de las alertas de producción. Escribe el presupuesto en el mismo documento que la línea base e indica junto a cada umbral qué herramienta lo demostraría.



Errores habituales

Perfilar una compilación Debug y tratar el resultado como evidencia de producción. Optimizar un método antes de definir el escenario de principio a fin. Ignorar la memoria de sesión retenida porque la CPU parece aceptable. Confundir un retraso de red o de WebSocket con CPU del servidor. Medir una sesión y suponer que la cifra se mantiene con doscientas.



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:

- ✓ Explicar el ciclo de peticiones y eventos de Wisej.NET, el push por WebSocket, el estado de sesión y el modelo de memoria del servidor.
- ✓ Separar el tiempo de renderizado del navegador, el tiempo de red y carga útil, el tiempo de los controladores de eventos del servidor, el tiempo de base de datos y el tiempo de las tareas en

segundo plano.

- ✓ Crear un escenario de rendimiento repetible con métricas de línea base y umbrales de aceptación.
- ✓ Identificar qué preguntas de rendimiento requieren las herramientas de CPU, memoria, base de datos, async, contadores o instrumentación de Visual Studio.

Guía del laboratorio / examen

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



Laboratorio práctico

Crea la solución WisejPerfLab con un dashboard de soporte, un grid de tickets y un árbol de clientes alimentados con datos de prueba, y hazla medible. Añade un servicio ScenarioProbe que envuelva un escenario en un ámbito Stopwatch y escriba logs PERF estructurados de inicio y fin con el nombre del escenario, la acción del usuario, los milisegundos transcurridos y el número de filas; regístralo en el host builder y envuelve con él tres escenarios: la actualización del dashboard, la búsqueda de tickets y la expansión del árbol de clientes. Ejecuta cada escenario una vez para calentar, luego registra la línea base y escribe un presupuesto de rendimiento que indique el entorno, la configuración de compilación, el tamaño del conjunto de datos y un umbral de aceptación por escenario.

Punto de partida

Partes de una aplicación Wisej.NET vacía y de Visual Studio 2022 con las herramientas de profiling instaladas. Todavía no existe nada: este laboratorio construye las tres pantallas que el resto del curso ajusta y las carga con datos suficientes para que sean lentas a propósito: unos 50.000 tickets y una jerarquía de clientes de varios miles de nodos. No corrigas nada de lo que notes. El objetivo del Módulo 1 es medir la aplicación tal como está construida.

Enfoque sugerido

- ✓ Crea la solución WisejPerfLab y tres pantallas: una `Page` de dashboard con etiquetas de KPI, un botón de actualización y un gráfico; un grid de tickets enlazado a los datos de prueba de los tickets; y un `TreeView` de clientes. Carga los datos desde un script o un `IHostedService` para que cada ejecución parta del mismo conjunto de datos.
- ✓ Escribe `ScenarioProbe` como un servicio pequeño: `Measure(scenario, userAction, rowCount)` devuelve un ámbito `IDisposable` que arranca un `Stopwatch` y escribe «PERF start», y en `Dispose` escribe «PERF end» con `elapsedMs` y `rows` como campos estructurados, no mediante concatenación de cadenas.
- ✓ Regístralo como singleton en el host builder junto al proveedor de logging de consola, y haz que el contenedor sea accesible desde tus páginas Wisej.NET para que cada pantalla pueda obtener la sonda que necesita.
- ✓ Envuelve los tres escenarios: actualización del dashboard, búsqueda de tickets y expansión del árbol de clientes. Coloca el ámbito de la sonda alrededor de toda la acción del usuario — consulta, proyección y modificación de controles—, no solo de la consulta, porque el trabajo de interfaz forma parte de lo que espera el usuario.
- ✓ Cambia a Release, ejecuta cada escenario una vez para absorber el ruido del arranque y del JIT, luego ejecuta cada uno tres veces y anota la mediana. Escribe juntas la nota de línea base y el presupuesto, e indica junto a cada umbral la herramienta de Visual Studio que lo demostraría o lo refutaría.

Entregables requeridos

- ✓ Solución WisejPerfLab con dashboard, grid de tickets y árbol de clientes sobre datos de prueba
- ✓ Servicio ScenarioProbe que escribe logs PERF estructurados de inicio y fin con milisegundos transcurridos y número de filas
- ✓ Tres escenarios instrumentados: actualización del dashboard, búsqueda de tickets y expansión del árbol de clientes
- ✓ Nota de línea base que registra entorno, configuración de compilación, tamaño del conjunto de datos y procedimiento de calentamiento
- ✓ Presupuesto de rendimiento con un umbral de aceptación por escenario y la herramienta que demostraría cada uno

Preguntas de repaso

- ✓ La actualización de tu dashboard registra 1900 ms, pero la CPU del servidor apenas se mueve durante el escenario. ¿Qué grupo de coste descarta eso, a cuál apunta y cuál es la siguiente medición más barata?
- ✓ La sonda mide el controlador. Nombra dos costes que el usuario sufre en el mismo clic y que tu cifra `eLapsedMs` no incluye, y explica cómo medirías cada uno.
- ✓ ¿Por qué la nota de línea base tiene que registrar el tamaño del conjunto de datos y la configuración de compilación, si el nombre del escenario y el tiempo transcurrido parecen la parte interesante?



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.