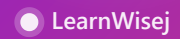


Escalado, health checks, balanceo de carga y ajuste final

Guía de la lección · Guía del laboratorio / examen



Guía de la lección

Convertir mediciones en un modelo de capacidad, umbrales de HealthCheck.json, afinidad de sesión y balanceo de carga compatible con WebSocket, contadores de producción y el informe final de rendimiento que justifica cada cambio. 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

Convertir mediciones en un modelo de capacidad, umbrales de HealthCheck.json, afinidad de sesión y balanceo de carga compatible con WebSocket, contadores de producción y el informe final de rendimiento que justifica cada cambio.

1. El problema en producción

Seis módulos de profiling han producido un WisejPerfLab más rápido y una carpeta llena de trazas. Nada de eso responde a la pregunta que hará el equipo de operaciones el día del despliegue: *¿cuántos usuarios caben en un servidor y qué pasa cuando uno se llena?* El profiling en la máquina de un desarrollador te dice el coste de un escenario; producción necesita ese coste convertido en un modelo de capacidad, unos umbrales que rechacen sesiones nuevas antes de que el proceso tenga problemas, una forma de despliegue que respete el estado de sesión del servidor y una monitorización que detecte si alguna de las seis correcciones sufre una regresión. Esa traducción es este módulo, y el informe del proyecto final es lo que permite que revise el trabajo alguien que nunca ha abierto Visual Studio.

2. Conceptos clave

Concepto	Por qué importa
modelo de capacidad	La memoria retenida por sesión inactiva y la CPU por escenario habitual, multiplicadas por la concurrencia máxima esperada y comparadas con los recursos de la instancia, reservando un margen de forma deliberada.
HealthCheck.json	Configuración mediante la cual un servidor Wisej.NET puede declararse no disponible según el número de sesiones, la memoria y la CPU, para que el balanceador de carga deje de enviarle usuarios nuevos.
503 con retry-after	La respuesta que devuelve una instancia no sana a una <i>primera</i> petición; las sesiones existentes siguen funcionando en su nodo mientras los usuarios nuevos se dirigen a otro sitio.
afinidad de sesión	Enrutamiento coherente de un cliente a la instancia que tiene su estado de sesión, normalmente implementado con cookies sticky; sin ella, una sesión de Wisej.NET no sobrevive a una segunda petición.
balanceo compatible con WebSocket	Configuración del balanceador de carga que deja pasar la conexión actualizada; cuando WebSocket no está disponible, la aplicación recurre al polling y se pierde el push en tiempo real.
contadores de producción	Métricas continuas y ligeras —CPU, heap del GC, tasa de asignación, tasa de excepciones, estado del thread pool— que muestran una regresión sin tener un profiler conectado.
informe final de rendimiento	Escenarios, entorno, evidencia de línea base, causa raíz, cambio, evidencia posterior, riesgos restantes, umbrales y plan de monitorización, en un único documento revisable.

3. Desarrollo de la lección

De una medición a un número de sesiones

El modelo de capacidad es aritmética sobre cifras que ya tienes. El Módulo 4 produjo los megabytes retenidos por sesión inactiva; los Módulos 3 y 6 produjeron el coste de CPU y de espera por escenario habitual. Multiplica la cifra de memoria por la concurrencia que esperas en el pico y compárala con la memoria que la instancia puede usar realmente, reservando margen para picos, recolección de basura y el sistema operativo en lugar de planificar hasta el último megabyte. Haz lo mismo con la CPU: coste por escenario multiplicado por la frecuencia con la que se ejecutan los escenarios en el pico.

Aquí caben dos comprobaciones de honestidad. Primera: una sesión inactiva y una activa no son lo mismo, así que indica cuál mediste y cómo generaste las sesiones. Segunda: una cifra obtenida en el portátil de un desarrollador necesita una nota sobre el hardware en el que se ejecutará de verdad. Un modelo de capacidad que declara sus supuestos se puede corregir; uno que presenta una única cifra como un hecho, no.

Health checks con umbrales justificados

Los health checks de Wisej.NET permiten a un servidor declararse no disponible según el número de sesiones, la memoria o la CPU. El mecanismo es sencillo; el valor de este módulo está en que tus umbrales salen de mediciones y no de una entrada de blog. Si una sesión retiene 22 MB y la instancia puede dar a la aplicación 4 GB con margen, un `maxSessions` de 150 es una cifra que puedes defender, y puedes mostrar el cálculo.

```
{
  "enabled": true,
  "maxMemory": 70,      // % of available memory
  "maxSessions": 150,   // from 22 MB retained/session + headroom
  "maxCPU": 85,         // % sustained
  "returnCode": 503,
  "retryAfter": 10
}
```

Merece la pena explicar con precisión lo que hace el health check en la práctica, porque a menudo se malinterpreta: una instancia por encima de sus límites devuelve 503 a una petición que *iniciaría* una sesión, así que el balanceador de carga envía a ese usuario a otro nodo, mientras las sesiones que ya están en la instancia continúan con normalidad. Es un mecanismo de degradación controlada, no un interruptor de apagado, y no sustituye a corregir la fuga que está llenando el proceso.

Desplegar algo que posee estado

Una sesión de Wisej.NET posee estado de interfaz en el servidor, y eso decide la forma del despliegue. Las peticiones de un cliente deben volver a la instancia que tiene su sesión, así que el balanceador de carga necesita afinidad de sesión: cookies sticky u otro mecanismo de enrutamiento coherente. Escalar horizontalmente sin afinidad no degrada con elegancia una aplicación Wisej.NET: la rompe.

El soporte de WebSocket es el segundo requisito. El balanceador y cualquier proxy inverso que tenga delante deben permitir la actualización de la conexión; donde WebSocket no está disponible, la aplicación puede seguir funcionando mediante polling, pero se pierde el push en tiempo real y cambia el patrón de actualizaciones. Escríbelo en las notas de despliegue en lugar de descubrirlo durante un incidente. Y mide con la configuración de producción: las opciones de depuración, la minificación y la compresión son distintas, y una medición tomada con las equivocadas no es comparable con las trazas en las que se basa el informe.

La monitorización y el informe que te sobrevive

Visual Studio sirve para investigar, no para vigilar producción. La salud continua viene de los contadores de .NET y las métricas del runtime —CPU, tamaño del heap del GC, tasa de asignación, tasa de excepciones, estado del thread pool—, más logs estructurados a nivel de aplicación alrededor de los escenarios que instrumentaste en el Módulo 1. Esa sonda sigue en el código, y sus registros `PERF end` son el detector de regresiones más barato posible: si la actualización del dashboard vuelve a registrar 1800 ms, lo sabrás antes de que un usuario lo notifique.

El informe del proyecto final lo une todo: definiciones de los escenarios, entorno, evidencia de línea base, explicaciones de las causas raíz, los cambios, la evidencia posterior, los riesgos restantes, los umbrales del

health check y la lista de comprobación del despliegue. Su prueba es una pregunta que deberías hacerte antes de entregarlo: ¿podría un compañero que nunca abrió tu profiler reproducir una medición, entender por qué se hizo cada cambio y saber qué métrica de producción le indicaría que el problema ha vuelto? Si la respuesta es sí, el trabajo de ajuste está terminado. Si no, puede que la aplicación sea más rápida, pero el conocimiento sigue estando solo en tu cabeza.



Errores habituales

Escalar horizontalmente sin afinidad de sesión. Elegir `maxSessions` sin medir la memoria por sesión. Medir con configuraciones de depuración, minificación o compresión distintas de las de producción. Tratar el balanceo de carga como sustituto de corregir fugas y escenarios lentos. Dejar trazas del profiler en marcha como si fueran monitorización.



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:

- ✓ Estimar la memoria por sesión y el coste de CPU por escenario a partir de mediciones.
- ✓ Configurar los health checks de Wisej.NET con umbrales de sesiones, memoria y CPU máximas.
- ✓ Explicar por qué los despliegues de Wisej.NET requieren afinidad de sesión y balanceo de carga compatible con WebSocket.
- ✓ Elaborar un informe final de rendimiento con línea base, correcciones, resultados de las nuevas pruebas, riesgos y plan de monitorización.

Guía del laboratorio / examen

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



Laboratorio práctico

Entrega el proyecto final de WisejPerfLab. Deriva un modelo de capacidad a partir de tus propias mediciones: memoria retenida por sesión inactiva, CPU por escenario habitual, concurrencia máxima esperada y el margen que reservas. Escribe un `HealthCheck.json` cuyos `maxSessions`, `maxMemory` y `maxCPU` salgan de esas cifras y explica cada umbral, devolviendo 503 con un `retry-after` para que el balanceador de carga dirija a los nuevos usuarios a otro sitio mientras las sesiones existentes siguen funcionando. Escribe las notas de despliegue para dos instancias detrás de un balanceador de carga con afinidad de sesión y soporte de WebSocket, y ensambla el informe final: escenarios, entorno, evidencia de línea base, causas raíz, cambios, evidencia posterior, riesgos restantes y los contadores de producción que mostrarán si vuelve una regresión.

Punto de partida

WisejPerfLab tras seis módulos: la actualización del dashboard, la búsqueda de tickets, la expansión del árbol y la exportación son todas más rápidas de forma medible, el formulario de detalle ya no tiene fugas y tienes evidencia de antes y después para cada cosa. Lo que todavía no existe es nada sobre lo que pueda actuar un equipo de operaciones: ni una cifra de capacidad, ni un health check, ni notas de despliegue, ni un plan de monitorización.

Enfoque sugerido

- ✓ Reúne los datos de entrada de tus propios informes: megabytes retenidos por sesión inactiva del Módulo 4, coste de CPU y de espera por escenario de los Módulos 3 y 6, y la concurrencia máxima esperada de la aplicación. Indica qué mediciones son de sesiones inactivas y cuáles de activas.
- ✓ Construye el modelo de capacidad como aritmética visible —memoria por sesión multiplicada por la concurrencia frente a la memoria utilizable de la instancia, CPU por escenario multiplicada por la frecuencia máxima del escenario— y reserva un margen explícito para picos, recolección de basura y el sistema operativo en lugar de planificar hasta el último megabyte.
- ✓ Escribe `HealthCheck.json` con `maxSessions`, `maxMemory`, `maxCPU`, el código de retorno 503 y un `retry-after`, y escribe una frase por umbral que explique la medición de la que procede. Verifica que una instancia por encima de sus límites rechaza sesiones nuevas mientras las existentes continúan.
- ✓ Escribe las notas de despliegue para dos instancias detrás de un balanceador de carga: la afinidad de sesión y cómo se implementa, el soporte de WebSocket a través de cada proxy del camino, qué cambia si WebSocket no está disponible y la configuración de producción que deben usar las mediciones.
- ✓ Ensambla el informe final con la evidencia de los seis módulos y termina con el plan de monitorización: qué contadores, qué umbrales, qué alertas y qué registros de log `PERF` mostrarían la vuelta de cada regresión concreta.

Entregables requeridos

- ✓ Modelo de capacidad que deriva las sesiones por servidor de la memoria medida por sesión y la CPU por escenario
- ✓ HealthCheck.json con maxSessions, maxMemory, maxCPU, código de retorno y retry-after, con cada umbral justificado
- ✓ Notas de despliegue que cubren la afinidad de sesión, el soporte de WebSocket y el fallback a polling
- ✓ Informe final de rendimiento con línea base, causa raíz, cambio y evidencia posterior para cada corrección de módulo
- ✓ Plan de monitorización que nombra los contadores, umbrales y alertas que detectarían cada regresión

Preguntas de repaso

- ✓ Tu health check rechaza sesiones nuevas a partir de 150. Un compañero pregunta por qué no 300, si el proceso no se ha caído con 300 en las pruebas. Responde con tus propias cifras.
- ✓ Se hace un despliegue detrás de un balanceador de carga sin afinidad de sesión. Describe exactamente lo que experimenta un usuario y por qué es un fallo de corrección y no una ralentización.
- ✓ Dentro de seis meses la actualización del dashboard vuelve a ser lenta. ¿Qué señal de producción te avisaría primero y qué recogerías a continuación?



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.