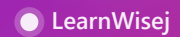


Skalierung, Health Checks, Load Balancing und abschließendes Tuning

Lektionsleitfaden · Lab- und Prüfungsleitfaden



Lektionsleitfaden

Messwerte in ein Kapazitätsmodell überführen, Schwellenwerte in HealthCheck.json, Session-Affinität und WebSocket-fähiges Load Balancing, Produktivzähler und der abschließende Performance-Bericht, der jede Änderung begründet. Was das für Wisej.NET-Entwickler bedeutet, die Engpässe in einer produktiven App diagnostizieren – und wie Sie dieses Modul am besten durcharbeiten.



Fokus: Produktivbetrieb

Messwerte in ein Kapazitätsmodell überführen, Schwellenwerte in HealthCheck.json, Session-Affinität und WebSocket-fähiges Load Balancing, Produktivzähler und der abschließende Performance-Bericht, der jede Änderung begründet.

1. Das Problem im Produktivbetrieb

Sechs Module Profiling haben ein schnelleres WisejPerfLab und einen Ordner voller Traces hervorgebracht. Nichts davon beantwortet die Frage, die das Betriebsteam am Tag des Rollouts stellen wird: *Wie viele Benutzer passen auf einen Server, und was passiert, wenn einer voll ist?* Profiling auf einem Entwicklerrechner verrät Ihnen die Kosten eines Szenarios; der Produktivbetrieb braucht diese Kosten übersetzt in ein Kapazitätsmodell, in Schwellenwerte, die neue Sessions abweisen, bevor der Prozess in Schwierigkeiten gerät, in eine Deployment-Form, die den serverseitigen Session-Zustand respektiert, und in ein Monitoring, das bemerkt, wenn einer der sechs Fixes eine Regression erleidet. Diese Übersetzung ist Gegenstand dieses Moduls, und der Bericht zum Abschlussprojekt macht die Arbeit für jemanden prüfbar, der Visual Studio nie geöffnet hat.

2. Zentrale Konzepte

Konzept	Warum es wichtig ist
Kapazitätsmodell	Gehaltener Speicher pro inaktiver Session und CPU pro häufigem Szenario, multipliziert mit der erwarteten Spitzenlast an gleichzeitigen Sessions und den Ressourcen der Instanz gegenübergestellt – mit bewusst eingeplanter Reserve.
HealthCheck.json	Konfiguration, über die sich ein Wisej.NET-Server anhand von Session-Anzahl, Speicher und CPU als nicht verfügbar melden kann, damit ein Load Balancer ihm keine neuen Benutzer mehr schickt.
503 mit retry-after	Die Antwort, die eine überlastete Instanz auf einen <i>ersten</i> Request zurückgibt; bestehende Sessions laufen auf ihrem Knoten weiter, während neue Benutzer anderswohin geleitet werden.
Session-Affinität	Konsistentes Routing eines Clients zu der Instanz, die seinen Session-Zustand hält, meist über Sticky Cookies umgesetzt; ohne sie übersteht eine Wisej.NET-Session keinen zweiten Request.
WebSocket-fähiges Load Balancing	Eine Load-Balancer-Konfiguration, die die hochgestufte Verbindung durchlässt; ist WebSocket nicht verfügbar, fällt die App auf Polling zurück, und der Echtzeit-Push geht verloren.
Produktivzähler	Schlanke, laufend erhobene Metriken – CPU, GC-Heap, Allokationsrate, Exception-Rate, Zustand des Thread-Pools –, die eine Regression ohne angehängten Profiler sichtbar machen.
abschließender Performance-Bericht	Szenarien, Umgebung, Baseline-Belege, Ursache, Änderung, Belege nachher, verbleibende Risiken, Schwellenwerte und Monitoring-Plan in einem prüfbaren Dokument.

3. Der rote Faden der Lektion

Von einer Messung zu einer Anzahl von Sessions

Das Kapazitätsmodell ist Arithmetik mit Zahlen, die Sie bereits haben. Modul 4 hat die gehaltenen Megabyte pro inaktiver Session geliefert; die Module 3 und 6 die CPU- und Wartekosten pro häufigem Szenario. Multiplizieren Sie den Speicherwert mit der zu Spitzenzeiten erwarteten Parallelität und vergleichen Sie ihn mit dem Speicher, den die Instanz tatsächlich nutzen kann – mit Reserve für Lastspitzen, Garbage Collection und Betriebssystem, statt bis zum letzten Megabyte zu planen. Verfahren Sie mit der CPU ebenso: Kosten pro Szenario mal der Rate, mit der Szenarien zu Spitzenzeiten ausgeführt werden.

Zwei Ehrlichkeitsprüfungen gehören hierher. Erstens sind eine inaktive und eine aktive Session nicht dasselbe; sagen Sie also, was Sie gemessen haben und wie Sie die Sessions erzeugt haben. Zweitens braucht eine Zahl, die auf einem Entwickler-Laptop ermittelt wurde, einen Hinweis auf die Hardware, auf der sie tatsächlich laufen wird. Ein Kapazitätsmodell, das seine Annahmen offenlegt, lässt sich korrigieren; eines, das eine einzelne Zahl als Tatsache präsentiert, nicht.

Health Checks mit begründeten Schwellenwerten

Mit Health Checks kann sich ein Wisej.NET-Server anhand von Session-Anzahl, Speicher oder CPU als nicht verfügbar melden. Der Mechanismus ist einfach; der Wert dieses Moduls liegt darin, dass Ihre Schwellenwerte aus Messungen stammen und nicht aus einem Blogbeitrag. Hält eine Session 22 MB und kann die Instanz der App mit Reserve 4 GB geben, dann ist ein `maxSessions` von 150 eine Zahl, die Sie verteidigen können – und Sie können die Rechnung vorlegen.

```
{
  "enabled": true,
  "maxMemory": 70,      // % des verfügbaren Speichers
  "maxSessions": 150,   // aus 22 MB gehalten/Session + Reserve
  "maxCPU": 85,         // % dauerhaft
  "returnCode": 503,
  "retryAfter": 10
}
```

Was der Check in der Praxis tut, sollte man genau benennen, weil es oft missverstanden wird: Eine Instanz über ihren Grenzwerten gibt 503 auf einen Request zurück, der eine Session *starten* würde, sodass der Load Balancer diesen Benutzer zu einem anderen Knoten schickt, während die bereits vorhandenen Sessions auf der Instanz normal weiterlaufen. Es ist ein Mechanismus für kontrollierte Degradation, kein Notausschalter – und kein Ersatz dafür, das Leck zu beheben, das den Prozess überhaupt erst füllt.

Etwas deployen, das Zustand besitzt

Eine Wisej.NET-Session besitzt serverseitigen UI-Zustand, und das bestimmt die Form des Deployments. Requests eines Clients müssen zu der Instanz zurückkehren, die seine Session hält; der Load Balancer braucht daher Session-Affinität – Sticky Cookies oder einen anderen konsistenten Routing-Mechanismus. Horizontal ohne Affinität zu skalieren verschlechtert eine Wisej.NET-App nicht nur allmählich; es macht sie kaputt.

WebSocket-Unterstützung ist die zweite Voraussetzung. Der Load Balancer und jeder vorgeschaltete Reverse Proxy müssen das Connection-Upgrade zulassen; wo WebSocket nicht verfügbar ist, funktioniert die App zwar weiterhin über Polling, aber der Echtzeit-Push geht verloren, und das Update-Muster ändert sich. Halten Sie das in den Deployment-Notizen fest, statt es während eines Incidents zu entdecken. Und messen Sie mit Produktiveinstellungen – Debug-Optionen, Minifizierung und Komprimierung unterscheiden sich, und eine Messung mit den falschen Einstellungen ist nicht mit den Traces vergleichbar, auf denen der Bericht aufbaut.

Monitoring – und der Bericht, der auch ohne Sie funktioniert

Visual Studio ist für Untersuchungen da, nicht zur Überwachung des Produktivbetriebs. Der laufende Gesundheitszustand ergibt sich aus .NET-Zählern und Runtime-Metriken – CPU, GC-Heap-Größe, Allokationsrate, Exception-Rate, Zustand des Thread-Pools – plus strukturierten Logs auf Anwendungsebene rund um die Szenarien, die Sie in Modul 1 instrumentiert haben. Diese Probe steckt noch im Code, und ihre `PERF end`-Einträge sind der günstigste denkbare Regressionsdetektor: Loggt die Dashboard-Aktualisierung wieder 1.800 ms, wissen Sie es, bevor ein Benutzer es meldet.

Der Bericht zum Abschlussprojekt führt alles zusammen: Szenariodefinitionen, Umgebung, Baseline-Belege, Erklärungen der Ursachen, die Änderungen, die Belege nachher, verbleibende Risiken, Health-Check-Schwellenwerte und die Deployment-Checkliste. Sein Prüfstein ist eine Frage, die Sie sich vor der Abgabe stellen sollten: Könnte ein Kollege, der Ihren Profiler nie geöffnet hat, eine Messung reproduzieren, verstehen, warum jede Änderung vorgenommen wurde, und wissen, welche Produktivmetrik ihm zeigen würde, dass das Problem zurückgekehrt ist? Wenn ja, ist die Tuning-Arbeit abgeschlossen. Wenn nicht, mag die App schneller sein, aber das Wissen existiert weiterhin nur in Ihrem Kopf.



Typische Fehler

Horizontal ohne Session-Affinität skalieren. `maxSessions` festlegen, ohne den Speicher pro Session gemessen zu haben. Mit Debug-, Minifizierungs- oder Komprimierungseinstellungen messen, die vom Produktivbetrieb abweichen. Load Balancing als Ersatz dafür betrachten, Lecks und langsame Szenarien zu beheben. Profiler-Traces weiterlaufen lassen, als wären sie Monitoring.



Gewohnheit: Belege sammeln

Definieren Sie das Szenario, bevor Sie einen Profiler öffnen, erfassen Sie aus einem Release-Build ohne Debugger, ändern Sie genau eine Sache und führen Sie dann dasselbe Szenario erneut aus und vergleichen Sie. Ordnen Sie die Verzögerung einer von fünf Kategorien zu – CPU in Ihrem Code, Blockieren und Warten, Allokation und Retention, Datenzugriff und I/O oder Update-Payload –, denn jede Kategorie hat ihr eigenes Visual Studio-Tool, und eine Optimierung, die Sie nicht mit einem Trace vorher und nachher belegen können, ist eine Vermutung.



Lernziele

Am Ende dieses Moduls können Sie:

- ✓ den Speicher pro Session und die CPU-Kosten pro Szenario aus Messungen abschätzen.
- ✓ Wisej.NET-Health-Checks mit Schwellenwerten für maximale Sessions, maximalen Speicher und maximale CPU konfigurieren.
- ✓ erklären, warum Wisej.NET-Deployments Session-Affinität und WebSocket-fähiges Load Balancing erfordern.
- ✓ einen abschließenden Performance-Bericht mit Baseline, Fixes, Ergebnissen der erneuten Tests, Risiken und Monitoring-Plan erstellen.

Lab- und Prüfungsleitfaden

Was Sie im praktischen Lab bauen, das vorgeschlagene Vorgehen und die geforderten Ergebnisse.



Praktisches Lab

Liefern Sie das WisejPerfLab-Abschlussprojekt ab. Leiten Sie aus Ihren eigenen Messungen ein Kapazitätsmodell ab: gehaltener Speicher pro inaktiver Session, CPU pro häufigem Szenario, erwartete Spitzenlast an gleichzeitigen Sessions und die Reserve, die Sie einplanen. Schreiben Sie eine HealthCheck.json, deren maxSessions, maxMemory und maxCPU aus diesen Zahlen stammen, begründen Sie jeden Schwellenwert und geben Sie 503 mit retry-after zurück, damit ein Load Balancer neue Benutzer anderswohin leitet, während bestehende Sessions weiterlaufen. Schreiben Sie die Deployment-Notizen für zwei Instanzen hinter einem Load Balancer mit Session-Affinität und WebSocket-Unterstützung und stellen Sie den Abschlussbericht zusammen: Szenarien, Umgebung, Baseline-Belege, Ursachen, Änderungen, Belege nachher, verbleibende Risiken und die Produktivzähler, die zeigen, falls eine Regression zurückkehrt.

Ausgangspunkt

WisejPerfLab nach sechs Modulen: Dashboard-Aktualisierung, Ticketsuche, Aufklappen des Baums und Export sind alle messbar schneller, das Detailformular leckt nicht mehr, und Sie haben für jeden Punkt Vorher-nachher-Belege. Was noch fehlt, ist alles, womit ein Betriebsteam arbeiten könnte – keine Kapazitätzahl, kein Health Check, keine Deployment-Notizen und kein Monitoring-Plan.

Vorgeschlagenes Vorgehen

- ✓ Tragen Sie die Eingangswerte aus Ihren eigenen Berichten zusammen: gehaltene Megabyte pro inaktiver Session aus Modul 4, CPU- und Wartekosten pro Szenario aus den Modulen 3 und 6 sowie die erwartete Spitzenlast an gleichzeitigen Sessions der App. Geben Sie an, welche Messungen inaktive und welche aktive Sessions betreffen.
- ✓ Bauen Sie das Kapazitätsmodell als nachvollziehbare Rechnung auf – Speicher pro Session mal Parallelität gegen nutzbaren Instanzspeicher, CPU pro Szenario mal Spitzenrate der Szenarien – und planen Sie ausdrücklich Reserve für Lastspitzen, Garbage Collection und Betriebssystem ein, statt bis zum letzten Megabyte zu planen.
- ✓ Schreiben Sie `HealthCheck.json` mit `maxSessions`, `maxMemory`, `maxCPU`, dem Rückgabecode 503 und einem retry-after, und schreiben Sie pro Schwellenwert einen Satz, der die zugrunde liegende Messung erklärt. Prüfen Sie, dass eine Instanz über ihren Grenzwerten neue Sessions abweist, während bestehende weiterlaufen.
- ✓ Schreiben Sie die Deployment-Notizen für zwei Instanzen hinter einem Load Balancer: Session-Affinität und wie sie umgesetzt wird, WebSocket-Unterstützung durch jeden Proxy auf dem Weg, was sich ändert, wenn WebSocket nicht verfügbar ist, und die Produktiveinstellungen, die für Messungen gelten müssen.
- ✓ Stellen Sie den Abschlussbericht aus den Belegen der sechs Module zusammen und schließen Sie mit dem Monitoring-Plan: welche Zähler, welche Schwellenwerte, welche Alarmer und welche `PERF`-Logeinträge die Rückkehr jeder einzelnen Regression zeigen würden.

Geforderte Ergebnisse

- ✓ Kapazitätsmodell, das die Sessions pro Server aus dem gemessenen Speicher pro Session und der CPU pro Szenario ableitet
- ✓ HealthCheck.json mit maxSessions, maxMemory, maxCPU, Rückgabecode und retry-after, jeder Schwellenwert begründet
- ✓ Deployment-Notizen zu Session-Affinität, WebSocket-Unterstützung und dem Polling-Fallback
- ✓ Abschließender Performance-Bericht mit Baseline, Ursache, Änderung und Belegen nachher für jeden Fix aus den Modulen
- ✓ Monitoring-Plan, der die Zähler, Schwellenwerte und Alarme nennt, die jede Regression erkennen würden

Wiederholungsfragen

- ✓ Ihr Health Check weist ab 150 neue Sessions ab. Ein Kollege fragt, warum nicht 300, da der Prozess im Test bei 300 nicht abgestürzt ist. Antworten Sie mit Ihren eigenen Zahlen.
- ✓ Ein Deployment geht hinter einem Load Balancer ohne Session-Affinität live. Beschreiben Sie genau, was ein Benutzer erlebt und warum das ein Korrektheitsfehler und keine Verlangsamung ist.
- ✓ In sechs Monaten ist die Dashboard-Aktualisierung wieder langsam. Welches einzelne Signal aus dem Produktivbetrieb würde es Ihnen zuerst zeigen, und was würden Sie als Nächstes erfassen?



Gewohnheit: Belege sammeln

Definieren Sie das Szenario, bevor Sie einen Profiler öffnen, erfassen Sie aus einem Release-Build ohne Debugger, ändern Sie genau eine Sache und führen Sie dann dasselbe Szenario erneut aus und vergleichen Sie. Ordnen Sie die Verzögerung einer von fünf Kategorien zu – CPU in Ihrem Code, Blockieren und Warten, Allokation und Retention, Datenzugriff und I/O oder Update-Payload –, denn jede Kategorie hat ihr eigenes Visual Studio-Tool, und eine Optimierung, die Sie nicht mit einem Trace vorher und nachher belegen können, ist eine Vermutung.