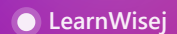


Profiling-Workflow in Visual Studio für Wisej.NET

Lektionsleitfaden · Lab- und Prüfungsleitfaden



Lektionsleitfaden

Den Performance Profiler richtig gegen eine Wisej.NET-App einsetzen: Release-Builds ohne Debugger, die Matrix zur Tool-Auswahl, das Anhängen an dotnet oder w3wp und das Lesen von Zusammenfassungs-Timeline, Aufrufstruktur, Hot Path sowie Caller/Callee-Ansichten. Was das für Wisej.NET-Entwickler bedeutet, die Engpässe in einer produktiven App diagnostizieren – und wie Sie dieses Modul am besten durcharbeiten.



Fokus: Produktivbetrieb

Den Performance Profiler richtig gegen eine Wisej.NET-App einsetzen: Release-Builds ohne Debugger, die Matrix zur Tool-Auswahl, das Anhängen an dotnet oder w3wp und das Lesen von Zusammenfassungs-Timeline, Aufrufstruktur, Hot Path sowie Caller/Callee-Ansichten.

1. Das Problem im Produktivbetrieb

Modul 1 hat WisejPerfLab mit einer Zahl, aber ohne Erklärung zurückgelassen: Die Dashboard-Aktualisierung dauert 1.840 ms. Der naheliegende nächste Schritt ist, sie zu profilieren – und genau an diesem Schritt scheitern die meisten Untersuchungen. Ein Trace, der unter dem Debugger erfasst wurde, misst den Debugger. Ein Trace, der gestartet wird, bevor die App aufgewärmt ist, misst den JIT. Ein Trace über zwei Minuten Klickerei misst alles und beweist nichts. Ein Trace mit vier gleichzeitig aktivierten Tools misst den Overhead des Profilers selbst. Dieses Modul vermittelt die handwerkliche Disziplin, die einen Trace zu einem verwertbaren Beleg macht: der richtige Build, das richtige Tool für die vermutete Kategorie, der richtige Prozess, wenn die App nicht von Visual Studio gestartet wird, ein Szenario pro Erfassung und ein Bericht, den Sie zu lesen wissen.

2. Zentrale Konzepte

Konzept	Warum es wichtig ist
Performance Profiler (Alt+F2)	Der Weg der nachträglichen Auswertung: Er kann das Projekt ohne Debugger starten, ein ganzes Szenario erfassen und einen gespeicherten Bericht erzeugen – daher stammen echte Belege.
Fenster „Diagnosetools“	Die Live-Ansicht mit angehängtem Debugger; nützlich für einen schnellen Blick beim schrittweisen Durchlaufen des Codes, aber irreführend als Quelle für Zeitmessungen, weil der Debugger das Verhalten der Runtime verändert.
Matrix zur Tool-Auswahl	CPU Usage für Berechnungen, Instrumentation für Aufrufanzahlen und Wall-Clock-Zeit, Object Allocation für Speicherumschlag, Memory Usage für Retention, Database für Abfragen, File I/O für Dateien, .NET Async für await-Ketten, .NET Counters für den Live-Zustand.
.diagsession	Die gespeicherte Trace-Datei; erst ein Name mit Modul, Szenario, Build, Datenmenge und Zeitstempel macht einen Vorher-nachher-Vergleich auch Wochen später möglich.
Zusammenfassungs-Timeline	Der CPU- oder Speichergraph über die gesamte Erfassung; die Auswahl auf genau die Benutzeraktion einzugrenzen ist der Schritt, der Start und Aufwärmphase aus der Aufrufstruktur entfernt.
Hot Path	Die teuerste Aufrufkette im ausgewählten Bereich; Show Hot Path klappt sie auf, damit Sie Ihren eigenen Code unter den Framework-Frames finden.
eigene vs. gesamte CPU-Zeit	Die gesamte Zeit schließt aufgerufene Funktionen ein, die eigene nur den Funktionsrumpf: Hohe eigene Zeit bedeutet, dass die Methode selbst teuer ist; hohe Gesamtzeit bei niedriger eigener Zeit bedeutet, dass die Kosten in dem liegen, was sie aufruft.

3. Der rote Faden der Lektion

Release, kein Debugger, ein Szenario

Der Standard-Workflow hat vier feste Schritte, bevor irgendetwas Interessantes passiert. Stellen Sie die Konfiguration auf Release um. Starten Sie die App und führen Sie das Szenario einmal aus, damit Start und JIT-Kompilierung bereits bezahlt sind. Öffnen Sie *Debug > Performance Profiler* mit Alt+F2 und aktivieren Sie genau ein Tool. Starten Sie dann die Erfassung, führen Sie nur das zu testende Szenario aus und stoppen Sie umgehend.

Jede Regel hat ihren Grund. Debug-Builds deaktivieren Optimierungen, führen kein Inlining durch und können Call Stacks erzeugen, in denen ein trivialer Helper wie ein Hot Path aussieht. Der Debugger bringt eigenen Overhead mit und verändert das Timing. Das Aufwärmen ist wichtig, weil die erste Ausführung jeder Methode JIT-Zeit enthält – real, aber nicht die Kosten, die der Benutzer bei der hundertsten Aktualisierung spürt. Und ein Tool nach dem anderen ist wichtig, weil die Erfassung nicht kostenlos ist: Object Allocation und Instrumentation zusammen erzeugen genug Overhead, um den Hot Path zu verschieben.

Die letzte Regel wird am häufigsten gebrochen: *ein Szenario pro Trace*. Eine Erfassung mit Anmeldung, drei Aktualisierungen, einer Suche und etwas Scrollen liefert eine Aufrufstruktur, in der jede interessante Zahl ein Mittelwert aus unzusammenhängender Arbeit ist. Schließen Sie die übrigen Browser-Tabs, klicken Sie einmal, stoppen Sie.

Das Tool anhand der Kategorie wählen

Die fünf Kategorien aus Modul 1 lassen sich fast eins zu eins auf die Tool-Liste des Performance Profilers abbilden, und genau diese Zuordnung ist der Kern dieses Moduls. Setzen Sie zuerst *CPU Usage* ein, wenn das Symptom hohe CPU-Last oder langsame Berechnung ist – es tastet Stacks günstig ab und zeigt, wohin die Prozessorzeit ging. Greifen Sie zu *Instrumentation*, wenn Sie exakte Aufrufanzahlen oder Wall-Clock-Zeit brauchen oder vermuten, dass Zeit mit Warten statt mit Rechnen verbracht wird: Es misst Ein- und Austritt jeder instrumentierten Funktion und sieht daher blockierte Zeit, die dem Sampling entgeht.

Verwenden Sie *.NET Object Allocation*, wenn ein Szenario viel Speicher umzuschlagen scheint, und *Memory Usage*-Snapshots, wenn Sie vermuten, dass etwas gehalten wird, nachdem der Benutzer es geschlossen hat. Verwenden Sie *Database*, wenn EF Core- oder ADO.NET-Abfragen dominieren könnten, *File I/O* für Importe, Exporte und Berichtserzeugung und *.NET Async*, wenn eine Async-Kette langsamer ist, als die Summe ihrer awaits vermuten lässt. *.NET Counters* fällt aus der Reihe: Es ist eine Live-Ansicht von CPU, GC-Heap, Allokationsrate, Exception-Rate und Zustand des Thread-Pools und damit das richtige Tool, um eine App unter Last zu beobachten, statt einen einzelnen Klick zu zerlegen.

In WisejPerfLab setzt das Lab die ersten beiden ein. CPU Usage beantwortet „Wohin ging die Prozessorzeit während der Aktualisierung?“; Instrumentation beantwortet „Wie oft wurde das aufgerufen, und wie viel der 1.840 ms wurde tatsächlich mit Rechnen verbracht?“. Zwei Traces, ein Szenario und eine Schlussfolgerung, die einer Prüfung standhält.

Einen Prozess profilieren, den Visual Studio nicht gestartet hat

Ist das Startprojekt die Wisej.NET-App, startet der Profiler sie für Sie, und es gibt nichts auszuwählen. Produktivnahes Hosting ist anders. Unter Kestrel ist die App ein **dotnet**-Prozess; unter IIS oder IIS Express ist sie **w3wp** bzw. **iisexpress**, einer pro Anwendungspool. Der Performance Profiler kann sich an einen laufenden Prozess anhängen, und die einzige echte Falle ist der falsche Prozess: Prüfen Sie auf einer Maschine mit mehreren Sites Prozess-ID und Anwendungspool, senden Sie dann einen Request und kontrollieren Sie, dass die Erfassung tatsächlich Aktivität zeigt, bevor Sie das Szenario ernsthaft ausführen.

WebSocket-Szenarien erfordern einen zusätzlichen Gedanken. Verbindet sich der Browser, während der Trace noch startet, landen der Verbindungs-Handshake und das erste Rendering in Ihrer Messung. Starten Sie den Browser entweder erst, wenn die Erfassung läuft, oder erfassen Sie einen separaten Aufwärm-Trace und vergleichen Sie Gleiches mit Gleichem.

Den Bericht lesen, ohne sich täuschen zu lassen

Jeder Bericht öffnet sich mit der Zusammenfassungs-Timeline. Bevor Sie auch nur einen Funktionsnamen ansehen, ziehen Sie eine Auswahl um genau das Intervall der Benutzeraktion und lassen die Ansichten neu berechnen; das entfernt Start, Seitenaufbau und Leerlaufzeit aus den Zahlen. Arbeiten Sie sich dann von oben nach unten vor: die Aufrufstruktur mit *Show Hot Path* für die dominante Kette, das Flammendiagramm für dieselbe Information als Form, die Funktionsansicht sortiert nach eigener CPU-Zeit für teure Funktionsrümpfe und Caller/Callee, um zu erfahren, wer eine teure Funktion wie oft aufruft.

In einem Wisej.NET-Trace liegt Ihr eigener Code ein gutes Stück tiefer im Stack, unterhalb des Event-Dispatchings von Wisej.NET und der Hosting-Frames von ASP.NET Core. Das ist normal und kein Framework-Bug. Die Regel lautet: an der Infrastruktur vorbeiscrollen bis zum ersten Frame in Ihrem eigenen Namespace und dort mit der Analyse beginnen. Ist der Framework-Frame über Ihrem Code teuer, fragen Sie, was Ihr Code von ihm verlangt – meist stellt sich eine tausendfach wiederholte Änderung heraus statt eines langsamen Frameworks.

```
// Trace-Notiz, neben der .diagsession gespeichert
// Modul:      02
// Szenario: DashboardRefresh_5000Tickets
// Build:      Release, x64, kein Debugger, aufgewärmt
// Hosting:    Kestrel, lokaler SQL Server  Browser: Edge
// Tool:       CPU Usage, danach Instrumentation zur Vertiefung
// Budget:     < 800 ms Handler-Zeit auf dem Server
```

Respektieren Sie schließlich das Grundrauschen. Eine Differenz von dreißig Millisekunden zwischen zwei Läufen eines Zwei-Sekunden-Szenarios ist keine Verbesserung, sondern Streuung. Wiederholen Sie das Szenario, nehmen Sie den Median und beanspruchen Sie nur einen Erfolg, den Sie reproduzieren können.



Typische Fehler

Den Start profilieren, wenn die Beschwerde ein langsamer Button-Klick ist. Alle Tools gleichzeitig aktivieren und das Szenario im Erfassungs-Overhead ertränken. Einen Debug-Trace mit einem Release-Trace vergleichen. Den Profiler eine lange Session hindurch laufen lassen, bis niemand mehr die Interaktion findet. Sich an den falschen `w3wp` anhängen und einen untätigen Anwendungspool profilieren.



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:

- ✓ Release-Builds mit Debug > Performance Profiler / Alt+F2 profilieren.
- ✓ Traces mit CPU Usage, Instrumentation, .NET Object Allocation, Memory Usage, .NET Counters, Database, File I/O und .NET Async erfassen.
- ✓ sich bei Bedarf an einen laufenden Wisej.NET-Prozess anhängen und den richtigen Prozess identifizieren.

- ✓ Aufrufstruktur, Hot Path, Caller/Callee, Funktionsliste, Flammendiagramm, Allokationsbaum und Snapshot-Vergleiche interpretieren.

Lab- und Prüfungsleitfaden

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



Praktisches Lab

Profilieren Sie die langsame Dashboard-Aktualisierung in WisejPerfLab richtig. Wechseln Sie zu Release, wärmen Sie die App einmal auf, erfassen Sie dann mit Alt+F2 einen CPU Usage-Trace genau dieses einen Szenarios, grenzen Sie die Zusammenfassungs-Timeline auf den Klick selbst ein und finden Sie mit Show Hot Path den obersten Pfad unterhalb der Dispatch-Frames von Wisej.NET. Erfassen Sie einen zweiten Instrumentation-Trace desselben Szenarios, um Aufrufanzahlen und Wall-Clock-Zeit zu erhalten, entscheiden Sie anhand der beiden Traces, ob die Kosten CPU-Zeit in Ihrem Code oder Wartezeit sind, und speichern Sie beide .diagsession-Dateien unter einem Namen, der Modul, Szenario, Build, Datenmenge und Zeitstempel enthält.

Ausgangspunkt

Sie öffnen WisejPerfLab aus Modul 1 mit seinen drei instrumentierten Szenarien und seiner **Baseline.md**. Die Dashboard-Aktualisierung loggt rund 1.840 ms für 5.000 Zeilen, und noch weiß niemand, warum. An der Anwendung ändert sich in diesem Lab nichts: Es geht ausschließlich darum, Belege zu sammeln, auf deren Grundlage das nächste Modul handeln kann.

Vorgeschlagenes Vorgehen

- ✓ Stellen Sie die Solution auf Release um, schließen Sie alle Browser-Tabs außer der App, setzen Sie die Seed-Daten auf die Baseline-Datenmenge zurück und führen Sie die Aktualisierung einmal ohne Aufzeichnung aus, damit JIT und Start bereits bezahlt sind.
- ✓ Öffnen Sie *Debug > Performance Profiler* mit Alt+F2, aktivieren Sie *CPU Usage* und sonst nichts, starten Sie die Erfassung, klicken Sie genau einmal auf Aktualisieren und stoppen Sie, sobald die Maske zur Ruhe kommt.
- ✓ Ziehen Sie im Bericht auf der Zusammenfassungs-Timeline eine Auswahl um den Klick selbst, öffnen Sie dann die Aufrufstruktur, aktivieren Sie *Show Hot Path* und scrollen Sie an den Dispatch-Frames von ASP.NET Core und Wisej.NET vorbei bis zur ersten Funktion in Ihrem eigenen Namespace. Notieren Sie deren gesamte und eigene CPU-Zeit und ermitteln Sie über Caller/Callee, wer sie wie oft aufruft.
- ✓ Erfassen Sie einen zweiten Trace des identischen Szenarios mit *Instrumentation*. Vergleichen Sie Aufrufanzahlen und Wall-Clock-Zeit mit dem abgetasteten CPU-Bild: Eine Funktion mit hoher Aufrufanzahl und winzigen Kosten pro Aufruf ist ein anderes Problem als eine Funktion mit langem Rumpf.
- ✓ Entscheiden Sie, welche Kategorie die Belege stützen, schreiben Sie die Trace-Notiz neben jede **.diagsession** und benennen Sie beide Dateien so, dass Modul, Szenario, Build, Datenmenge und Zeitstempel ohne Öffnen lesbar sind.

Geforderte Ergebnisse

- ✓ Release-CPU Usage-Trace des Szenarios Dashboard-Aktualisierung, Timeline auf den Klick eingegrenzt
- ✓ Hot-Path-Screenshot, der die oberste Anwendungsfunktion und ihre eigene gegenüber der gesamten CPU-Zeit nennt
- ✓ Instrumentation-Trace desselben Szenarios mit Aufrufanzahlen und Wall-Clock-Zeit
- ✓ Trace-Notiz mit Kurs, Modul, Szenario, Build, Hosting, Tool, Browser und erwartetem Budget
- ✓ Ein Absatz zur vermuteten Ursache, der angibt, auf welche der fünf Kategorien die Belege hinweisen

Wiederholungsfragen

- ✓ Ihr CPU Usage-Trace erklärt nur 300 ms der 1.840 ms, die die Probe gemessen hat. Was bedeutet diese Lücke, und welches Tool würden Sie als Nächstes einsetzen?
- ✓ Eine Framework-Methode steht ganz oben im Hot Path, mit sehr hoher gesamter und fast keiner eigenen CPU-Zeit. Erklären Sie, was das bedeutet und was Sie sich ansehen würden, statt einen Framework-Bug zu melden.
- ✓ Warum braucht eine zweite Person die Trace-Notiz, um Ihr Ergebnis zu reproduzieren, obwohl die `.diagsession` bereits den gesamten Trace enthält?



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.