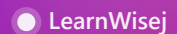


Das Performance-Modell von Wisej.NET

Lektionsleitfaden · Lab- und Prüfungsleitfaden



Lektionsleitfaden

Der Request- und Event-Zyklus, WebSocket-Push, der Session-Zustand als Kapazitätseinheit, die fünf Kostenkategorien und die Disziplin aus Szenario plus Baseline, an der jeder spätere Trace gemessen wird. Was das für Wisej.NET-Entwickler bedeutet, die Engpässe in einer produktiven App diagnostizieren – und wie Sie dieses Modul am besten durcharbeiten.



Fokus: Produktivbetrieb

Der Request- und Event-Zyklus, WebSocket-Push, der Session-Zustand als Kapazitätseinheit, die fünf Kostenkategorien und die Disziplin aus Szenario plus Baseline, an der jeder spätere Trace gemessen wird.

1. Das Problem im Produktivbetrieb

Ein Support-Team meldet, WisejPerfLab sei „langsam geworden“. Dieser Satz enthält keine Messung, kein Szenario und keine Schicht. Das Dashboard könnte auf SQL warten, der Aktualisierungs-Handler könnte fünfzigtausend Zellen formatieren, der Browser könnte ein Grid neu zeichnen, das nie vollständig hätte gerendert werden dürfen, oder der Server hält schlicht die Formulare von zweihundert Sessions, die vor einer Stunde geschlossen wurden. Jede dieser Ursachen hat einen anderen Fix und ein anderes Visual Studio-Tool, und wer den Profiler öffnet, bevor er entschieden hat, um welche es sich handelt, erfasst einen Trace, der die Frage nicht beantworten kann. Dieses Modul baut deshalb zuerst das mentale Modell auf: was zwischen einem Klick im Browser und einer neu gezeichneten Maske tatsächlich passiert, welcher dieser Schritte was kostet und wie Sie ein Szenario und ein Budget schriftlich festhalten, damit die nächsten sechs Module einen Maßstab haben.

2. Zentrale Konzepte

Konzept	Warum es wichtig ist
serverseitige UI	Controls, ihre Eigenschaften und ihr Zustand leben im Serverprozess, und der Browser rendert, was der Server beschreibt. Jedes UI-Objekt, das Sie erzeugen, ist also Serverspeicher, und jede Änderung an der UI ist Server-CPU plus Payload.
Event-Roundtrip	Der vollständige Weg einer Benutzeraktion: Browser-Event, Transport zum Server, Ihr Handler, Datenzugriff, Änderungen an Controls, das serialisierte Update zurück, Neuzeichnen im Browser – die Einheit, die ein Benutzer tatsächlich spürt.
Session als Kapazitätseinheit	Eine Session besitzt Pages, Formulare, Controls, Timer, Auswahlen, Bindings und gecachte Daten. Kapazitätsfragen werden pro Session beantwortet, nicht pro Request, weil dieser Speicher zwischen den Klicks belegt bleibt.
WebSocket-Push	Der Kanal, über den der Server Updates sendet, um die der Benutzer nicht gebeten hat. Er macht die Auslieferung von Live-Daten günstig und einen ungedrosselten Hintergrund-Task teuer, weil jeder Zwischenzustand zu einem Frame wird.
die fünf Kostenkategorien	CPU in Ihrem Code, Blockieren und Warten, Allokation und Retention, Datenzugriff und I/O sowie Update-Payload – die Einordnung, die entscheidet, welches Profiler-Tool Ihr Problem überhaupt sehen kann.
Latenz vs. Durchsatz	Latenz ist, wie lange ein Benutzer wartet; Durchsatz ist, wie viele Sessions der Server tragen kann. Ein Fix, der eine Wartezeit beseitigt, hilft der Latenz; ein Fix, der Arbeit oder Speicher einspart, hilft beidem – und manche Latenz-Fixes kosten Durchsatz.
Szenario und Baseline	Eine benannte, wiederholbare Folge von Benutzeraktionen plus Umgebung, Build-Konfiguration, Datenmenge und die damit gemessenen Werte. Ohne sie ist ein Trace Rauschen und jeder spätere Vergleich bedeutungslos.

3. Der rote Faden der Lektion

Was ein einzelner Klick wirklich kostet

In einer zustandslosen Webanwendung ist die natürliche Messeinheit der Request: Er kommt an, wird bedient, und der Prozess vergisst ihn. Wisej.NET bietet Ihnen stattdessen das Programmiermodell von WinForms – eine **Page** oder ein **Form** mit Controls, deren Eigenschaften Sie setzen – und bezahlt dafür, indem dieser Objektgraph auf dem Server lebendig bleibt, solange der Benutzer verbunden ist. Der Browser hält eine Darstellung der UI; der Server hält die UI selbst.

Verfolgen Sie einen Klick. Der Browser löst ein Event aus und sendet es über die offene Verbindung. Der Server findet die Session, ruft Ihren Handler auf, und ab dort ist alles gewöhnliches C#: Sie fragen eine Datenbank ab, bauen Strings, setzen **Text** auf einem Label, weisen eine **DataSource** zu, fügen Zeilen hinzu. Wisej.NET zeichnet diese Eigenschaftsänderungen auf, serialisiert nach dem Ende des Handlers die resultierende Differenz und sendet sie an den Browser, der neu zeichnet. Auf diesem Weg liegen fünf unterschiedliche Kosten – Rendering im Browser, Netzwerk und Payload, die CPU Ihres Handlers, Warten auf Datenbank und I/O sowie jede Hintergrundverarbeitung, die der Klick angestoßen hat –, und ein

Symptom, das für den Benutzer identisch aussieht („die Maske ist langsam“), kann aus jeder davon stammen.

Zwei Konsequenzen sind sofort relevant. Erstens werden Client-Events einer Session nacheinander verarbeitet: Ein Handler, der vier Sekunden braucht, friert diesen Benutzer für vier Sekunden ein, egal was der Server sonst tut. Zweitens bremst ein Handler, der CPU verbrennt, nicht nur seine eigene Session; er konkurriert um dieselben Kerne wie jede andere Session auf der Instanz. Deshalb misst dieser Kurs die Kosten pro Szenario und leitet daraus die Kapazität pro Session ab, statt einen schnellen Klick mit einem einzigen Benutzer für „schnell genug“ zu erklären.

Fünf Kategorien, fünf verschiedene Tools

Die Einordnung ist keine akademische Übung – sie bestimmt, welches Tool Sie wählen. CPU-Arbeit in Ihrem eigenen Code zeigen CPU Usage und das Flammendiagramm. Blockier- und Wartezeit ist für CPU Usage weitgehend unsichtbar; dafür gibt es Instrumentation und .NET Async. Allokationsdruck und gehaltener Speicher brauchen .NET Object Allocation bzw. Memory Usage-Snapshots, und es sind unterschiedliche Probleme: Ein Szenario kann stark allokalieren und nichts halten oder kaum etwas allokalieren und ein ganzes Formular lecken. Latenz bei Datenzugriff und I/O gehört zu den Tools Database und File I/O. Die Kosten des Update-Payloads sind die eine Kategorie, die Visual Studio überhaupt nicht sieht; hier müssen die Entwicklertools des Browsers die Belege liefern.

Daraus folgt eine Gewohnheit, eine Frage, die Sie sich vor jeder Erfassung stellen: *In welcher Kategorie vermute ich das Problem, und was würde mich widerlegen?* Eine langsame Maske bei niedriger Server-CPU ist kein CPU-Problem, ganz gleich, wie viele CPU-Traces Sie erfassen. Modul 2 macht daraus eine Matrix zur Tool-Auswahl und einen wiederholbaren Erfassungs-Workflow; in diesem Modul lernen Sie, Ihre Vermutung ausdrücklich zu formulieren, damit der Trace sie widerlegen kann.

WisejPerfLab messbar machen

Das Lab legt WisejPerfLab an, das Dashboard für den Support-Betrieb, das der gesamte Kurs optimiert, und erledigt dann die unspektakuläre Arbeit, es messbar zu machen. Das Messinstrument ist ein kleiner `ScenarioProbe` -Service, der im Host-Builder registriert und in die Pages injiziert wird. `Measure()` gibt einen `IDisposable` -Scope zurück, der eine `Stopwatch` startet, eine strukturierte Logzeile „PERF start“ schreibt und beim Verwerfen „PERF end“ mit den verstrichenen Millisekunden und der Zeilenanzahl ausgibt. Strukturiertes Logging ist hier entscheidend: Szenarioname, Benutzeraktion und Zeilenanzahl sind getrennte Felder, sodass sich dieselben drei Szenarien in jedem späteren Lauf filtern und vergleichen lassen.

```
using (_probe.Measure("Dashboard", "Refresh", rows.Count))
{
    // alles innerhalb dieses Scopes ist ein gemessenes Szenario:
    // Abfrage, Projektion, Änderung der Controls und das folgende Update
    RefreshKpis(rows);
    RefreshGrid(rows);
}
// PERF end Dashboard Refresh elapsedMs=1840 rows=5000
```

Drei Szenarien erhalten eine Probe: Dashboard aktualisieren, Tickets suchen und Kundenbaum aufklappen. Sie wurden ausgewählt, weil sie in unterschiedliche Kategorien fallen – die Aktualisierung wird sich als CPU in Formatierungscode herausstellen, die Suche als Datenbankzeit, das Aufklappen des Baums als Payload und Anzahl gehaltener Controls –, und bis Modul 7 haben Sie für alle drei Vorher-nachher-Werte.

Baselines und Budgets, schriftlich festgehalten

Eine Baseline ist das Szenario plus alles, was sein Ergebnis verändern könnte: die Maschine, die Build-Konfiguration, die Datenmenge, ob die Datenbank lokal läuft, welcher Browser, wie viele Sessions verbunden waren und ob die App vorher aufgewärmt wurde. Lassen Sie etwas davon weg, wird der Vergleich in Modul 3 zu einer Diskussion statt zu einem Beleg. Release ist wichtiger, als Entwickler erwarten – Debug verändert die JIT-Optimierung, fügt Overhead hinzu und kann Call Stacks erzeugen, die Sie auf eine völlig falsche Funktion ansetzen.

Ein Budget ist die andere Hälfte: ein Abnahme-Schwellenwert pro Szenario, festgelegt, bevor das Tuning beginnt. „Dashboard-Aktualisierung unter 800 ms Handler-Zeit auf dem Server nach dem Aufwärmen, bei weniger als 40 MB gehaltenem Speicher pro inaktiver Session“ ist ein Budget. Es gibt jedem späteren Modul ein Bestanden oder Nicht bestanden statt eines Gefühls, es sagt Ihnen, wann Sie mit dem Optimieren *aufhören* können, und in Modul 7 wird es zur Quelle der Health-Check-Schwellenwerte und der Produktiv-Alerts. Schreiben Sie das Budget in dasselbe Dokument wie die Baseline und notieren Sie neben jedem Schwellenwert, welches Tool ihn belegen würde.



Typische Fehler

Einen Debug-Build profilieren und das Ergebnis als Beleg für den Produktivbetrieb behandeln. Eine Methode optimieren, bevor das End-to-End-Szenario definiert ist. Gehaltenen Session-Speicher ignorieren, weil die CPU akzeptabel aussieht. Netzwerk- oder WebSocket-Verzögerung für Server-CPU halten. Eine Session messen und annehmen, dass der Wert auch bei zweihundert gilt.



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 Request- und Event-Zyklus von Wisej.NET, WebSocket-Push, den Session-Zustand und das Modell des Serverspeichers erklären.
- ✓ Rendering-Zeit im Browser, Netzwerk- und Payload-Zeit, Zeit in serverseitigen Event-Handlern, Datenbankzeit und Zeit in Hintergrund-Tasks voneinander trennen.

- ✓ ein wiederholbares Performance-Szenario mit Baseline-Metriken und Abnahme-Schwellenwerten erstellen.
- ✓ erkennen, welche Performance-Fragen die Visual Studio-Tools für CPU, Speicher, Datenbank, Async, Zähler oder Instrumentation erfordern.

Lab- und Prüfungsleitfaden

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



Praktisches Lab

Legen Sie die WisejPerfLab-Solution mit einem Support-Dashboard, einem Ticket-Grid und einem Kundenbaum an, die aus Seed-Daten gespeist werden, und machen Sie sie anschließend messbar. Fügen Sie einen ScenarioProbe-Service hinzu, der ein Szenario in einen Stopwatch-Scope einschließt und strukturierte PERF-Start/Ende-Logs mit dem Szenarionamen, der Benutzeraktion, den verstrichenen Millisekunden und der Zeilenanzahl schreibt, registrieren Sie ihn im Host-Builder und umschließen Sie damit drei Szenarien: Dashboard aktualisieren, Tickets suchen und Kundenbaum aufklappen. Führen Sie jedes Szenario einmal zum Aufwärmen aus, halten Sie dann die Baseline fest und schreiben Sie ein Performance-Budget, das Umgebung, Build-Konfiguration, Datenmenge und einen Abnahme-Schwellenwert pro Szenario nennt.

Ausgangspunkt

Sie beginnen mit einer leeren Wisej.NET-Anwendung und Visual Studio 2022 mit installierten Profiling-Tools. Noch existiert nichts: Dieses Lab baut die drei Masken, die der Rest des Kurses optimiert, und füllt sie mit so vielen Daten, dass sie absichtlich langsam sind – rund 50.000 Tickets und eine Kundenhierarchie mit mehreren Tausend Knoten. Beheben Sie nichts, was Ihnen auffällt. In Modul 1 geht es darum, die App so zu messen, wie sie gebaut wurde.

Vorgeschlagenes Vorgehen

- ✓ Legen Sie die WisejPerfLab-Solution und drei Masken an: eine Dashboard- `Page` mit KPI-Labels, einem Aktualisieren-Button und einem Diagramm; ein Ticket-Grid, das an die Seed-Ticketdaten gebunden ist; und einen Kunden- `TreeView`. Erzeugen Sie die Seed-Daten per Skript oder über einen `IHostedService`, damit jeder Lauf mit derselben Datenmenge beginnt.
- ✓ Schreiben Sie `ScenarioProbe` als kleinen Service: `Measure(scenario, userAction, rowCount)` gibt einen `IDisposable`-Scope zurück, der eine `Stopwatch` startet und „PERF start“ loggt und bei `Dispose` „PERF end“ mit `elapsedMs` und `rows` als strukturierten Feldern schreibt, nicht per String-Verkettung.
- ✓ Registrieren Sie ihn als Singleton im Host-Builder neben dem Console-Logging-Provider und machen Sie den Container aus Ihren Wisej.NET-Pages erreichbar, damit jede Maske die benötigte Probe erhalten kann.
- ✓ Umschließen Sie die drei Szenarien: Dashboard aktualisieren, Tickets suchen und Kundenbaum aufklappen. Legen Sie den Probe-Scope um die gesamte Benutzeraktion – Abfrage, Projektion und Änderung der Controls –, nicht nur um die Abfrage, denn die UI-Arbeit gehört zu dem, worauf der Benutzer wartet.
- ✓ Wechseln Sie zu Release, führen Sie jedes Szenario einmal aus, um Start- und JIT-Rauschen abzufangen, dann jeweils dreimal, und halten Sie den Median fest. Schreiben Sie Baseline-Notiz und Budget zusammen und nennen Sie zu jedem Schwellenwert das Visual Studio-Tool, das ihn belegen oder widerlegen würde.

Geforderte Ergebnisse

- ✓ WisejPerfLab-Solution mit Dashboard, Ticket-Grid und Kundenbaum auf Seed-Daten
- ✓ ScenarioProbe-Service, der strukturierte PERF-Start/Ende-Logs mit verstrichenen Millisekunden und Zeilenanzahl schreibt
- ✓ Drei instrumentierte Szenarien: Dashboard aktualisieren, Tickets suchen und Kundenbaum aufklappen
- ✓ Baseline-Notiz mit Umgebung, Build-Konfiguration, Datenmenge und Aufwärmverfahren
- ✓ Performance-Budget mit einem Abnahme-Schwellenwert pro Szenario und dem Tool, das ihn jeweils belegen würde

Wiederholungsfragen

- ✓ Ihre Dashboard-Aktualisierung loggt 1.900 ms, aber die CPU des Servers bewegt sich während des Szenarios kaum. Welche der fünf Kategorien schließt das aus, auf welche deutet es hin, und was ist die günstigste nächste Messung?
- ✓ Die Probe misst den Handler. Nennen Sie zwei Kosten, die der Benutzer beim selben Klick erlebt und die Ihr `eLapsedMs`-Wert nicht enthält, und sagen Sie, wie Sie jede davon messen würden.
- ✓ Warum muss die Baseline-Notiz Datenmenge und Build-Konfiguration festhalten, obwohl Szenarioname und verstrichene Zeit der interessante Teil zu sein scheinen?



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.