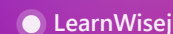


Le modèle de performance de Wisej.NET

Guide de la leçon · Guide du lab / de l'examen



Guide de la leçon

Le cycle des requêtes et des événements, le push WebSocket, l'état de session comme unité de capacité, les cinq catégories de coût et la discipline scénario-plus-référence à laquelle chaque trace ultérieure est comparée. Ce que cela signifie pour un développeur Wisej.NET qui diagnostique les goulets d'étranglement d'une application en production — et comment aborder ce module.



Enjeux en production

Le cycle des requêtes et des événements, le push WebSocket, l'état de session comme unité de capacité, les cinq catégories de coût et la discipline scénario-plus-référence à laquelle chaque trace ultérieure est comparée.

1. Le problème en production

Une équipe de support signale que WisejPerfLab « est devenu lent ». Cette phrase ne contient aucune mesure, aucun scénario et aucune couche. Le tableau de bord attend peut-être SQL, le gestionnaire d'actualisation formate peut-être cinquante mille cellules, le navigateur redessine peut-être une grille qui n'aurait jamais dû être rendue en entier, ou le serveur retient tout simplement les formulaires de deux cents sessions fermés il y a une heure. Chacun de ces cas appelle une correction différente et un outil Visual Studio différent, et le développeur qui ouvre le profileur avant d'avoir décidé duquel il s'agit collectera une trace incapable de répondre à la question. Ce module construit d'abord le modèle mental : ce qui se passe réellement entre un clic dans le navigateur et un écran redessiné, ce que coûte chacune de ces étapes, et comment consigner un scénario et un budget pour que les six modules suivants disposent d'un point de comparaison.

2. Concepts clés

Concept	Pourquoi c'est important
interface côté serveur	Les contrôles, leurs propriétés et leur état résident dans le processus serveur et le navigateur affiche ce que le serveur décrit : chaque objet d'interface que vous créez est donc de la mémoire serveur, et chaque modification de l'interface est du CPU serveur plus une charge utile.
aller-retour d'un événement	Le chemin complet d'une action utilisateur : événement dans le navigateur, transport vers le serveur, votre gestionnaire, accès aux données, modifications des contrôles, mise à jour sérialisée en retour, nouveau rendu dans le navigateur — l'unité que l'utilisateur ressent réellement.
la session comme unité de capacité	Une session possède des pages, des formulaires, des contrôles, des timers, des sélections, des bindings et des données en cache ; les questions de capacité se traitent par session, et non par requête, car cette mémoire reste résidente entre les clics.
push WebSocket	Le canal par lequel le serveur envoie des mises à jour que l'utilisateur n'a pas demandées ; il rend peu coûteuse la diffusion de données en direct et rend coûteuse une tâche en arrière-plan non limitée, car chaque état intermédiaire devient une frame.
les cinq catégories	Le CPU dans votre code, le blocage et l'attente, l'allocation et la rétention, l'accès aux données et les E/S, et la charge utile des mises à jour — la classification qui détermine quel outil du profileur peut même voir votre problème.
latence ou débit	La latence est le temps d'attente d'un utilisateur ; le débit est le nombre de sessions que le serveur tient. Une correction qui supprime une attente améliore la latence ; une correction qui supprime du travail ou de la mémoire améliore les deux, et certaines corrections de latence se paient en débit.
scénario et référence	Une séquence nommée et reproductible d'actions utilisateur, avec l'environnement, la configuration de build, la taille du jeu de données et les chiffres mesurés qu'elle a produits — sans cela, une trace n'est que du bruit et toute comparaison ultérieure est dénuée de sens.

3. Déroulé de la leçon

Ce que coûte réellement un clic

Dans une application web sans état, l'unité de mesure naturelle est la requête : elle arrive, elle est servie, le processus l'oublie. Wisej.NET vous offre à la place le modèle de programmation WinForms — une **Page** ou un **Form** avec des contrôles dont vous définissez les propriétés — et le paie en maintenant ce graphe d'objets en vie sur le serveur tant que l'utilisateur est connecté. Le navigateur détient un rendu de l'interface ; le serveur détient l'interface.

Suivez un clic de bout en bout. Le navigateur déclenche un événement et l'envoie sur la connexion ouverte. Le serveur localise la session, le transmet à votre gestionnaire, et à partir de là tout est du C# ordinaire : vous interrogez une base de données, construisez des chaînes, définissez **Text** sur un label, affectez un **DataSource** , ajoutez des lignes. Wisej.NET enregistre ces modifications de propriétés et, au

retour du gestionnaire, sérialise la différence obtenue et l'envoie au navigateur, qui redessine. Cinq coûts distincts jalonnent ce chemin — le rendu dans le navigateur, le réseau et la charge utile, le CPU de votre gestionnaire, l'attente de la base de données et des E/S, et tout traitement en arrière-plan lancé par le clic — et un symptôme identique pour l'utilisateur (« l'écran est lent ») peut provenir de n'importe lequel d'entre eux.

Deux conséquences comptent immédiatement. D'abord, les événements client d'une session sont traités un par un : un gestionnaire qui prend quatre secondes fige cet utilisateur pendant quatre secondes, quoi que fasse le serveur par ailleurs. Ensuite, un gestionnaire qui brûle du CPU ne ralentit pas seulement sa propre session ; il se dispute les mêmes cœurs que toutes les autres sessions de l'instance. C'est pourquoi le cours mesure le coût par scénario puis raisonne sur la capacité par session, au lieu de déclarer « assez rapide » un clic rapide pour un seul utilisateur.

Cinq catégories, cinq outils différents

La classification n'a rien d'académique — c'est ainsi que vous choisissez un outil. Le travail CPU dans votre propre code est ce que montrent CPU Usage et le flame graph. Le temps de blocage et d'attente est en grande partie invisible pour CPU Usage, et c'est à cela que servent Instrumentation et .NET Async. La pression d'allocation et la mémoire retenue nécessitent respectivement .NET Object Allocation et les instantanés Memory Usage, et ce sont deux problèmes différents : un scénario peut allouer beaucoup et ne rien retenir, ou n'allouer presque rien et faire fuir un formulaire entier. La latence d'accès aux données et d'E/S relève des outils Database et File I/O. Le coût de la charge utile des mises à jour est la seule catégorie que Visual Studio ne voit pas du tout, et ce sont les outils de développement du navigateur qui doivent en fournir la preuve.

L'habitude qui en découle est une question à vous poser avant de lancer une collecte : *dans quelle catégorie je pense que se trouve le problème, et qu'est-ce qui me prouverait le contraire ?* Un écran lent avec un CPU serveur faible n'est pas un problème de CPU, quel que soit le nombre de traces CPU que vous collectez. Le Module 2 transforme cela en matrice de choix des outils et en démarche de collecte reproductible ; ce module est celui où vous apprenez à formuler explicitement l'hypothèse pour que la trace puisse la réfuter.

Rendre WisejPerfLab mesurable

Le lab crée WisejPerfLab, le tableau de bord d'opérations de support que tout le cours optimise, puis accomplit le travail ingrat de le rendre mesurable. L'instrument est un petit service `ScenarioProbe` enregistré dans le host builder et injecté dans les pages. `Measure()` renvoie une portée `IDisposable` qui démarre un `Stopwatch`, écrit une ligne de log structurée « PERF start » et, à sa libération, écrit « PERF end » avec les millisecondes écoulées et le nombre de lignes. La journalisation structurée est essentielle ici : le nom du scénario, l'action de l'utilisateur et le nombre de lignes sont des champs distincts, si bien que les trois mêmes scénarios peuvent être filtrés et comparés sur toutes les exécutions ultérieures.

```
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
```

Trois scénarios reçoivent une sonde : l'actualisation du tableau de bord, la recherche de tickets et le déploiement de l'arborescence des clients. Ils ont été choisis parce qu'ils relèvent de catégories différentes — l'actualisation se révélera être du CPU dans du code de formatage, la recherche du temps de base de données, le déploiement de l'arborescence de la charge utile et un nombre de contrôles retenus — et d'ici le Module 7 vous disposerez de chiffres avant/après pour les trois.

Références et budgets, par écrit

Une référence, c'est le scénario plus tout ce qui pourrait modifier son résultat : la machine, la configuration de build, la taille du jeu de données, le fait que la base de données soit locale ou non, le navigateur, le nombre de sessions connectées et le fait que l'application ait été mise en température au préalable. Omettez l'un de ces éléments et la comparaison que vous ferez au Module 3 sera un débat, pas une preuve. Le mode Release compte plus que les développeurs ne le pensent — le mode Debug modifie l'optimisation JIT, ajoute une surcharge et peut produire des piles d'appels qui vous lancent sur la piste d'une tout autre fonction.

Le budget est l'autre moitié : un seuil d'acceptation par scénario, rédigé avant le début de l'optimisation. « Actualisation du tableau de bord sous 800 ms de temps de gestionnaire serveur après mise en température, avec moins de 40 Mo retenus par session inactive » est un budget. Il donne à chaque module suivant un verdict de réussite ou d'échec au lieu d'une impression, il vous dit quand *arrêter* d'optimiser et, au Module 7, il devient la source des seuils du health check et des alertes de production. Rédigez le budget dans le même document que la référence, et notez en regard de chaque seuil l'outil qui permettrait de le prouver.



Pièges courants

Profiler un build Debug et traiter le résultat comme une preuve valable en production. Optimiser une méthode avant d'avoir défini le scénario de bout en bout. Ignorer la mémoire de session retenue parce que le CPU paraît acceptable. Confondre un délai réseau ou WebSocket avec du CPU serveur. Mesurer une seule session et supposer que le chiffre tient à deux cents.



Réflexe de preuve

Définissez le scénario avant d'ouvrir un profileur, collectez à partir d'un build Release sans débogueur, changez une seule chose, puis réexécutez le même scénario et comparez. Classez le délai dans l'une des cinq catégories — CPU dans votre code, blocage et attente, allocation et rétention, accès aux données et E/S, ou charge utile des mises à jour — car chaque catégorie a son propre outil Visual Studio, et une optimisation que vous ne pouvez pas défendre avec une trace avant et après n'est qu'une supposition.



Objectifs d'apprentissage

À la fin de ce module, vous serez capable de :

- ✓ Expliquer le cycle requêtes/événements de Wisej.NET, le push WebSocket, l'état de session et le modèle de mémoire serveur.
- ✓ Distinguer le temps de rendu du navigateur, le temps réseau/charge utile, le temps des gestionnaires d'événements serveur, le temps de base de données et le temps des tâches en arrière-plan.
- ✓ Créer un scénario de performance reproductible avec des métriques de référence et des seuils d'acceptation.
- ✓ Identifier quelles questions de performance nécessitent les outils Visual Studio de CPU, de mémoire, de base de données, d'asynchrone, de compteurs ou d'instrumentation.

Guide du lab / de l'examen

Ce que vous allez construire dans le lab pratique, l'approche suggérée et les livrables attendus.



Lab pratique

Créez la solution WisejPerfLab avec un tableau de bord de support, une grille de tickets et une arborescence de clients alimentés par des données de départ, puis rendez-la mesurable. Ajoutez un service ScenarioProbe qui encadre un scénario dans une portée Stopwatch et écrit des logs PERF structurés de début et de fin avec le nom du scénario, l'action de l'utilisateur, les millisecondes écoulées et le nombre de lignes, enregistrez-le dans le host builder et encadrez-y trois scénarios : l'actualisation du tableau de bord, la recherche de tickets et le déploiement d'un nœud de l'arborescence des clients. Exécutez chaque scénario une fois pour la mise en température, puis enregistrez la référence et rédigez un budget de performance qui précise l'environnement, la configuration de build, la taille du jeu de données et un seuil d'acceptation par scénario.

Point de départ

Vous partez d'une application Wisej.NET vide et de Visual Studio 2022 avec les outils de profilage installés. Rien n'existe encore : ce lab construit les trois écrans que le reste du cours optimise, et les alimente avec assez de données pour qu'ils soient lents à dessein — environ 50 000 tickets et une hiérarchie de clients de plusieurs milliers de nœuds. Ne corrigez rien de ce que vous remarquez. L'objet du Module 1 est de mesurer l'application telle qu'elle est construite.

Approche suggérée

- ✓ Créez la solution WisejPerfLab et trois écrans : une `Page` de tableau de bord avec des labels de KPI, un bouton d'actualisation et un graphique ; une grille de tickets liée aux données de départ ; et un `TreeView` de clients. Injectez les données depuis un script ou un `IHostedService` pour que chaque exécution parte du même jeu de données.
- ✓ Écrivez `ScenarioProbe` comme un petit service : `Measure(scenario, userAction, rowCount)` renvoie une portée `IDisposable` qui démarre un `Stopwatch` et journalise « PERF start », puis, dans `Dispose`, journalise « PERF end » avec `elapsedMs` et `rows` comme champs structurés, et non par concaténation de chaînes.
- ✓ Enregistrez-le comme singleton dans le host builder à côté du fournisseur de journalisation console, et rendez le conteneur accessible depuis vos pages Wisej.NET pour que chaque écran puisse obtenir la sonde dont il a besoin.
- ✓ Encadrez les trois scénarios : actualisation du tableau de bord, recherche de tickets et déploiement de l'arborescence des clients. Placez la portée de la sonde autour de toute l'action de l'utilisateur — requête, projection et modification des contrôles — et pas seulement autour de la requête, car le travail d'interface fait partie de ce que l'utilisateur attend.
- ✓ Passez en Release, exécutez chaque scénario une fois pour absorber le bruit du démarrage et du JIT, puis exécutez-les chacun trois fois et consignez la médiane. Rédigez ensemble la note de référence et le budget, et indiquez en regard de chaque seuil l'outil Visual Studio qui permettrait de le prouver ou de l'infirmer.

Livrables attendus

- ✓ Solution WisejPerfLab avec un tableau de bord, une grille de tickets et une arborescence de clients sur des données de départ
- ✓ Service ScenarioProbe écrivant des logs PERF structurés de début et de fin avec les millisecondes écoulées et le nombre de lignes
- ✓ Trois scénarios instrumentés : actualisation du tableau de bord, recherche de tickets et déploiement de l'arborescence des clients
- ✓ Note de référence consignant l'environnement, la configuration de build, la taille du jeu de données et la procédure de mise en température
- ✓ Budget de performance avec un seuil d'acceptation par scénario et l'outil qui permettrait de prouver chacun

Questions de révision

- ✓ L'actualisation de votre tableau de bord journalise 1 900 ms, mais le CPU du serveur bouge à peine pendant le scénario. Quelle catégorie cela exclut-il, vers laquelle cela pointe-t-il, et quelle est la mesure suivante la moins coûteuse ?
- ✓ La sonde mesure le gestionnaire. Citez deux coûts que l'utilisateur subit sur le même clic et que votre chiffre `elapsedMs` n'inclut pas, et indiquez comment vous mesureriez chacun.
- ✓ Pourquoi la note de référence doit-elle consigner la taille du jeu de données et la configuration de build, alors que le nom du scénario et le temps écoulé semblent être la partie intéressante ?



Réflexe de preuve

Définissez le scénario avant d'ouvrir un profileur, collectez à partir d'un build Release sans débogueur, changez une seule chose, puis réexécutez le même scénario et comparez. Classez le délai dans l'une des cinq catégories — CPU dans votre code, blocage et attente, allocation et rétention, accès aux données et E/S, ou charge utile des mises à jour — car chaque catégorie a son propre outil Visual Studio, et une optimisation que vous ne pouvez pas défendre avec une trace avant et après n'est qu'une supposition.