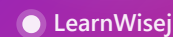


Montée en charge, health checks, équilibrage de charge et optimisation finale

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



Guide de la leçon

Transformer des mesures en modèle de capacité, les seuils de HealthCheck.json, l'affinité de session et un équilibrage de charge compatible WebSocket, les compteurs de production et le rapport de performance final qui justifie chaque modification. Ce que cela signifie pour un développeur Wisej.NET qui diagnostique les goulots d'étranglement d'une application en production — et comment aborder ce module.



Enjeux en production

Transformer des mesures en modèle de capacité, les seuils de HealthCheck.json, l'affinité de session et un équilibrage de charge compatible WebSocket, les compteurs de production et le rapport de performance final qui justifie chaque modification.

1. Le problème en production

Six modules de profilage ont produit un WisejPerfLab plus rapide et un dossier de traces. Rien de tout cela ne répond à la question que l'équipe d'exploitation posera le jour du déploiement : *combien d'utilisateurs tiennent sur un serveur, et que se passe-t-il quand il est plein ?* Le profilage sur une machine de développement vous donne le coût d'un scénario ; la production a besoin que ce coût soit converti en modèle de capacité, en seuils qui refusent les nouvelles sessions avant que le processus ne soit en difficulté, en architecture de déploiement qui respecte l'état de session côté serveur, et en supervision qui remarquera si l'une des six corrections régresse. Cette traduction est l'objet de ce module, et le rapport de synthèse est ce qui rend le travail vérifiable par quelqu'un qui n'a jamais ouvert Visual Studio.

2. Concepts clés

Concept	Pourquoi c'est important
modèle de capacité	La mémoire retenue par session inactive et le CPU par scénario courant, multipliés par la simultanéité attendue au pic et rapportés aux ressources de l'instance, avec une marge conservée délibérément.
HealthCheck.json	La configuration par laquelle un serveur Wisej.NET peut se déclarer indisponible en fonction du nombre de sessions, de la mémoire et du CPU, pour qu'un répartiteur de charge cesse de lui envoyer de nouveaux utilisateurs.
503 avec retry-after	La réponse qu'une instance en mauvaise santé renvoie à une <i>première</i> requête ; les sessions existantes continuent de tourner sur leur nœud pendant que les nouveaux utilisateurs sont orientés ailleurs.
affinité de session	L'acheminement systématique d'un client vers l'instance qui détient l'état de sa session, généralement mis en œuvre avec des cookies d'affinité ; sans elle, une session Wisej.NET ne peut pas survivre à une deuxième requête.
équilibre compatible WebSocket	Une configuration du répartiteur de charge qui laisse passer la connexion mise à niveau ; quand WebSocket n'est pas disponible, l'application se replie sur le polling et le push en temps réel est perdu.
compteurs de production	Des métriques continues et légères — CPU, tas GC, taux d'allocation, taux d'exceptions, état du pool de threads — qui révèlent une régression sans profileur attaché.
rapport de performance final	Scénarios, environnement, preuves de référence, cause racine, modification, preuves « après », risques restants, seuils et plan de supervision, réunis dans un seul document vérifiable.

3. Déroulé de la leçon

D'une mesure à un nombre de sessions

Le modèle de capacité est un calcul à partir de chiffres que vous avez déjà. Le Module 4 a produit les mégaoctets retenus par session inactive ; les Modules 3 et 6 ont produit le coût CPU et d'attente par scénario courant. Multipliez la mémoire par la simultanéité attendue au pic et comparez le résultat à la mémoire réellement utilisable par l'instance, en conservant une marge pour les pics, le garbage collector et le système d'exploitation plutôt qu'en planifiant jusqu'au dernier mégaoctet. Faites de même pour le CPU : le coût par scénario multiplié par la fréquence d'exécution des scénarios au pic.

Deux vérifications d'honnêteté ont leur place ici. D'abord, une session inactive et une session active ne sont pas la même chose : indiquez laquelle vous avez mesurée et comment vous avez piloté les sessions. Ensuite, un chiffre obtenu sur l'ordinateur portable d'un développeur doit être accompagné d'une note sur le matériel sur lequel il tournera réellement. Un modèle de capacité qui expose ses hypothèses peut être corrigé ; un modèle qui présente un chiffre unique comme un fait, non.

Des health checks aux seuils justifiés

Les health checks de Wisej.NET permettent à un serveur de se déclarer indisponible selon le nombre de sessions, la mémoire ou le CPU. Le mécanisme est simple ; la valeur de ce module est que vos seuils proviennent de mesures plutôt que d'un article de blog. Si une session retient 22 Mo et que l'instance peut donner 4 Go à l'application avec une marge, alors un `maxSessions` de 150 est un chiffre que vous pouvez défendre — et vous pouvez montrer le calcul.

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

Ce que fait concrètement le health check mérite d'être énoncé avec précision, car c'est souvent mal compris : une instance qui dépasse ses limites renvoie 503 à une requête qui *ouvrirait* une session, si bien que le répartiteur de charge envoie cet utilisateur vers un autre nœud, tandis que les sessions déjà présentes sur l'instance continuent normalement. C'est un mécanisme de dégradation maîtrisée, pas un interrupteur d'arrêt d'urgence — et il ne remplace pas la correction de la fuite qui remplit le processus au départ.

Déployer quelque chose qui détient un état

Une session Wisej.NET détient un état d'interface côté serveur, ce qui détermine l'architecture de déploiement. Les requêtes d'un client doivent revenir vers l'instance qui détient sa session : le répartiteur de charge a donc besoin de l'affinité de session — des cookies d'affinité ou un autre mécanisme de routage cohérent. Monter en charge horizontalement sans affinité ne dégrade pas une application Wisej.NET en douceur ; cela la casse.

La prise en charge de WebSocket est la seconde exigence. Le répartiteur et tout proxy inverse placé devant lui doivent autoriser la mise à niveau de la connexion ; là où WebSocket n'est pas disponible, l'application peut encore fonctionner par polling, mais le push en temps réel est perdu et le schéma des mises à jour change. Écrivez-le dans les notes de déploiement plutôt que de le découvrir pendant un incident. Et mesurez avec les réglages de production — les options de débogage, la minification et la compression diffèrent toutes, et une mesure effectuée avec les mauvais réglages n'est pas comparable aux traces sur lesquelles repose le rapport.

La supervision, et le rapport qui vous survit

Visual Studio sert aux investigations, pas à surveiller la production. La santé au quotidien se suit avec les compteurs .NET et les métriques du runtime — CPU, taille du tas GC, taux d'allocation, taux d'exceptions, état du pool de threads — ainsi qu'avec des logs structurés au niveau applicatif autour des scénarios instrumentés au Module 1. Cette sonde est toujours dans le code, et ses enregistrements `PERF end` constituent le détecteur de régression le moins coûteux possible : si l'actualisation du tableau de bord se remet à journaliser 1 800 ms, vous le saurez avant qu'un utilisateur ne le signale.

Le rapport de synthèse relie l'ensemble : définitions des scénarios, environnement, preuves de référence, explications des causes racines, modifications, preuves « après », risques restants, seuils du health check et checklist de déploiement. Son test est une question à vous poser avant de le remettre : un collègue qui n'a jamais ouvert votre profileur pourrait-il reproduire une mesure, comprendre pourquoi chaque modification a été faite, et savoir quelle métrique de production lui indiquerait le retour du problème ? Si oui, le travail d'optimisation est terminé. Sinon, l'application est peut-être plus rapide, mais la connaissance n'existe encore que dans votre tête.



Pièges courants

Monter en charge horizontalement sans affinité de session. Choisir `maxSessions` sans mesurer la mémoire par session. Faire des mesures avec des réglages de débogage, de minification ou de compression différents de la production. Considérer l'équilibrage de charge comme un substitut à la correction des fuites et des scénarios lents. Laisser tourner des traces de profilage comme s'il s'agissait de supervision.



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 :

- ✓ Estimer la mémoire par session et le coût CPU par scénario à partir de mesures.
- ✓ Configurer les health checks de Wisej.NET avec des seuils de sessions, de mémoire et de CPU maximaux.
- ✓ Expliquer pourquoi les déploiements Wisej.NET exigent l'affinité de session et un équilibrage de charge compatible WebSocket.
- ✓ Rédiger un rapport de performance final avec la référence, les corrections, les résultats des nouveaux tests, les risques et le plan de supervision.

Guide du lab / de l'examen

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



Lab pratique

Réalisez le projet de synthèse WisejPerfLab. Établissez un modèle de capacité à partir de vos propres mesures : mémoire retenue par session inactive, CPU par scénario courant, simultanéité attendue au pic et marge de sécurité que vous conservez. Rédigez un `HealthCheck.json` dont les valeurs `maxSessions`, `maxMemory` et `maxCPU` découlent de ces chiffres, en expliquant chaque seuil, et qui renvoie 503 avec un `retry-after` pour qu'un répartiteur de charge oriente les nouveaux utilisateurs ailleurs pendant que les sessions existantes continuent de fonctionner. Rédigez les notes de déploiement pour deux instances derrière un répartiteur de charge avec affinité de session et prise en charge de WebSocket, puis assemblez le rapport final : scénarios, environnement, preuves de référence, causes racines, modifications, preuves « après », risques restants et compteurs de production qui signaleront le retour d'une régression.

Point de départ

WisejPerfLab après six modules : l'actualisation du tableau de bord, la recherche de tickets, le déploiement de l'arborescence et l'export sont tous plus rapides de manière mesurable, le formulaire de détail ne fuit plus, et vous disposez de preuves avant/après pour chacun. Ce qui n'existe pas encore, c'est quoi que ce soit sur quoi une équipe d'exploitation pourrait agir — aucun chiffre de capacité, aucun health check, aucune note de déploiement et aucun plan de supervision.

Approche suggérée

- ✓ Rassemblez les données d'entrée issues de vos propres rapports : les mégaoctets retenus par session inactive du Module 4, le coût CPU et d'attente par scénario des Modules 3 et 6, et la simultanéité attendue au pic pour l'application. Indiquez quelles mesures portent sur des sessions inactives et lesquelles sur des sessions actives.
- ✓ Construisez le modèle de capacité sous forme de calcul visible — mémoire par session multipliée par la simultanéité, comparée à la mémoire utilisable de l'instance ; CPU par scénario multiplié par la fréquence de pointe des scénarios — et conservez une marge explicite pour les pics, le garbage collector et le système d'exploitation plutôt que de planifier jusqu'au dernier mégaoctet.
- ✓ Rédigez `HealthCheck.json` avec `maxSessions`, `maxMemory`, `maxCPU`, le code de retour 503 et un `retry-after`, et écrivez pour chaque seuil une phrase expliquant la mesure dont il provient. Vérifiez qu'une instance qui dépasse ses limites refuse les nouvelles sessions tandis que les existantes continuent.
- ✓ Rédigez les notes de déploiement pour deux instances derrière un répartiteur de charge : l'affinité de session et sa mise en œuvre, la prise en charge de WebSocket par chaque proxy du chemin, ce qui change si WebSocket n'est pas disponible, et les réglages de production que les mesures doivent utiliser.
- ✓ Assemblez le rapport final à partir des preuves des six modules, puis terminez par le plan de supervision : quels compteurs, quels seuils, quelles alertes et quels enregistrements de log `PERF`

montreraient le retour de chaque régression précise.

Livrables attendus

- ✓ Modèle de capacité déduisant le nombre de sessions par serveur de la mémoire mesurée par session et du CPU par scénario
- ✓ HealthCheck.json avec maxSessions, maxMemory, maxCPU, code de retour et retry-after, chaque seuil étant justifié
- ✓ Notes de déploiement couvrant l'affinité de session, la prise en charge de WebSocket et le repli sur le polling
- ✓ Rapport de performance final avec référence, cause racine, modification et preuve « après » pour chaque correction de module
- ✓ Plan de supervision indiquant les compteurs, les seuils et les alertes qui détecteraient chaque régression

Questions de révision

- ✓ Votre health check refuse les nouvelles sessions à 150. Un collègue demande pourquoi pas 300, puisque le processus n'a pas planté à 300 pendant les tests. Répondez avec vos propres chiffres.
- ✓ Un déploiement est mis en production derrière un répartiteur de charge sans affinité de session. Décrivez exactement ce que vit l'utilisateur, et pourquoi il s'agit d'une défaillance fonctionnelle plutôt que d'un ralentissement.
- ✓ Dans six mois, l'actualisation du tableau de bord est de nouveau lente. Quel signal de production vous alerterait en premier, et que collecteriez-vous ensuite ?



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.