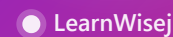


Contrôles de données, charges utiles du navigateur et grandes surfaces d'interface

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



Guide de la leçon

Grilles, listes, arborescences et répéteurs à grande échelle : mode virtuel, pagination, projection et chargement différé des nœuds, et utilisation des preuves réseau du navigateur à côté de celles de Visual Studio pour dimensionner la charge utile des mises à jour. 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

Grilles, listes, arborescences et répéteurs à grande échelle : mode virtuel, pagination, projection et chargement différé des nœuds, et utilisation des preuves réseau du navigateur à côté de celles de Visual Studio pour dimensionner la charge utile des mises à jour.

1. Le problème en production

Les grandes surfaces d'interface sont la source la plus courante de problèmes de performance dans Wisej.NET, et c'est la catégorie que le profileur ne voit pas entièrement. La grille des tickets de WisejPerfLab est liée à cinquante mille entités avec leurs propriétés de navigation ; l'arborescence des clients construit tous ses nœuds au démarrage. Sur un poste de travail, ce schéma était acceptable, car le contrôle et les données vivaient dans le même processus que l'écran. Ici, le même code paie quatre fois : la mémoire serveur pour retenir les objets, le CPU serveur pour construire et sérialiser l'état de l'interface, le réseau pour transporter la mise à jour et le temps du navigateur pour l'afficher. C'est dans ce module que la meilleure optimisation n'est généralement pas un algorithme plus rapide, mais *moins d'interface existant en même temps*.

2. Concepts clés

Concept	Pourquoi c'est important
ligne de projection	Un petit type qui ne contient que les colonnes affichées par un écran, lié à la place du graphe d'entités, pour que ni le serveur ni le navigateur ne transportent des données que personne ne regarde.
mode virtuel	La grille conserve un nombre de lignes plutôt que des objets ligne et demande les valeurs des cellules au fur et à mesure des besoins, ce qui supprime les objets par ligne et le rendu des lignes jusqu'auxquelles personne n'a fait défiler.
CellValueNeeded	L'événement qui fournit une valeur à la demande en mode virtuel ; il s'exécute très souvent, c'est donc par construction un chemin critique qui doit répondre rapidement.
pagination et filtrage côté serveur	Demander à la base de données la page que l'utilisateur regarde, au lieu de tout charger et de filtrer dans le code d'interface.
chargement différé des nœuds	Construire les enfants d'une arborescence au déploiement d'une branche, avec un espace réservé de comptage tant qu'elle est repliée, au lieu de matérialiser toute la hiérarchie au démarrage.
charge utile des mises à jour	La différence sérialisée que le serveur envoie après un gestionnaire ; des charges utiles volumineuses ou fréquentes coûtent du temps réseau et navigateur qu'aucun outil Visual Studio ne signale.
la compression ne supprime pas le travail	Wisej.NET compresse le trafic HTTP et WebSocket, ce qui réduit le nombre d'octets mais ne rend pas une mise à jour inutile peu coûteuse à produire, à transmettre ou à afficher.

3. Déroulé de la leçon

Grilles : lier moins, puis retenir moins

La correction de la grille comporte deux volets indépendants, et n'en réaliser qu'un est l'erreur habituelle. Le premier volet est la forme des données. Lier des entités entraîne les propriétés de navigation, les colonnes inutilisées et les grands champs de texte dans la mémoire serveur et dans chaque ligne que construit l'interface. Les remplacer par une projection `TicketGridRow` — numéro, nom du client, statut, texte de l'âge, et rien d'autre — réduit le graphe d'objets, les allocations mesurées au Module 4 et la mise à jour que reçoit le navigateur.

Le second volet est le nombre de lignes qui existent tout court. En mode virtuel, la grille conserve un `RowCount` et déclenche `CellValueNeeded` pour les cellules qu'elle affiche réellement, si bien que cinquante mille lignes cessent d'être cinquante mille objets ligne. Le compromis est explicite : cet événement s'exécute désormais en permanence, son gestionnaire doit donc être rapide et thread-safe, et il doit s'appuyer sur un service de requêtes paginées plutôt que sur une recherche qui interroge la base de données une fois par cellule. Précalculez les chaînes d'affichage dans la projection pour que le gestionnaire se contente de sélectionner un champ, et profilez le gestionnaire lui-même — le mode virtuel déplace le coût, il ne le supprime pas.

Arborescences, listes et répéteurs

Le même principe remodèle les autres grands contrôles. Un `TreeView` construit en entier au démarrage paie pour chaque branche que personne n'ouvre ; charger les enfants au déploiement, avec un espace réservé de comptage sur les branches repliées, rend le coût proportionnel à ce que l'utilisateur regarde réellement. Les listes et les répéteurs se prêtent aussi au traitement virtuel, et les répéteurs cachent un piège supplémentaire : un modèle de carte comportant une douzaine de contrôles imbriqués se multiplie par le nombre d'éléments ; préférez donc un résumé avec accès au détail à une riche arborescence de contrôles répétée mille fois.

Le préchargement est le contrepoids du chargement différé. Un défilement virtuel qui charge exactement ce qui est à l'écran peut sembler lent lors d'un défilement rapide ; charger un peu au-delà de la zone visible en vaut donc généralement la peine. C'est une décision de réglage à prendre à partir de mesures, pas un comportement par défaut à appliquer partout.

Les preuves que le profileur ne peut pas vous donner

Visual Studio vous dit ce qu'a fait le serveur : le CPU pour construire les lignes, les allocations pour les retenir, les requêtes pour les récupérer. Il ne dit rien de ce qui a transité sur le réseau. Pour cela, l'instrument est le panneau réseau du navigateur — la taille de la charge utile de la mise à jour après le chargement d'une grille, les frames WebSocket après le déploiement d'une arborescence, et le temps que le navigateur a passé à les afficher.

Deux mises en garde. La compression rend les nombres d'octets rassurants ; une mise à jour qui n'aurait jamais dû être envoyée reste du travail gaspillé aux deux extrémités : comparez donc le *nombre et la taille des mises à jour* avant et après, et pas seulement le total compressé. Et si désactiver la compression WebSocket peut vous aider à lire les frames pendant le débogage, ne faites pas de mesures avec des réglages de débogage que la production n'utilisera pas.

```
dataGridView1.VirtualMode = true;
dataGridView1.RowCount = _tickets.Count(CurrentFilter); // rows, not objects
dataGridView1.CellValueNeeded += (s, e) =>
{
    var row = _tickets.GetProjectedRow(CurrentFilter, e.RowIndex); // paged, cached
    e.Value = e.ColumnIndex switch
    {
        0 => row.Number, 1 => row.Customer, 2 => row.Status, 3 => row.AgeText, _ => null
    };
};
```

Règles d'optimisation à retenir

Affichez moins de lignes, moins de nœuds et moins de contrôles. Envoyez moins de mises à jour. Liez des objets plus petits. Différez le détail jusqu'à ce que l'utilisateur le demande. Mettez en cache les données de référence immuables ; ne mettez jamais en cache globalement des données mutables propres à un utilisateur. Préférez la pagination et le filtrage côté serveur au chargement complet suivi d'un filtrage dans le code d'interface. Et n'actualisez pas toute une grille quand une seule ligne ou un seul label a changé — cette seule habitude représente une part étonnante de la charge utile des tableaux de bord pilotés par des timers.

Sous toutes ces règles se trouve une question de conception plutôt que technique : à quelle question cet écran répond-il pour l'utilisateur ? Un écran conçu autour de tous les enregistrements possibles sera toujours coûteux à rendre rapide. Un écran conçu autour de la décision que l'utilisateur doit prendre est généralement peu coûteux par construction, et les chiffres avant/après du lab sont l'argument que vous apportez à cette discussion.



Pièges courants

Lier à chaque ligne des entités ORM complètes avec de grands graphes de navigation. Construire toute l'arborescence au démarrage au lieu de charger les nœuds à la demande. Actualiser des tableaux de bord entiers à chaque tick de timer. Supposer que la compression règle le problème des charges utiles surdimensionnées. Écrire un gestionnaire `CellValueNeeded` qui interroge la base de données une fois par cellule.



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 :

- ✓ Profiler des scénarios DataGridView, ListView, TreeView et DataRepeater pour le CPU, les allocations et le coût de rendu.
- ✓ Appliquer à bon escient le mode virtuel, la pagination, le filtrage, la projection et les stratégies de préchargement.
- ✓ Réduire le renouvellement des charges utiles WebSocket/HTTP en limitant les modifications de l'interface et la taille de la surface de données.
- ✓ Utiliser les outils de développement du navigateur avec Visual Studio pour diagnostiquer les problèmes de réseau et de taille des mises à jour.

Guide du lab / de l'examen

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



Lab pratique

Réduisez les deux plus grandes surfaces de WisejPerfLab. La grille des tickets est liée à 50 000 entités complètes avec leurs propriétés de navigation : remplacez cette liaison par une projection `TicketGridRow` limitée aux colonnes visibles, définissez `VirtualMode` et `RowCount` et servez les cellules depuis un gestionnaire `CellValueNeeded` adossé à un service de requêtes paginées. L'arborescence des clients construit tous les nœuds au démarrage : chargez plutôt les enfants au déploiement d'un nœud, avec un espace réservé indiquant le nombre d'enfants pour les branches repliées. Mesurez chaque écran avant et après avec CPU Usage et .NET Object Allocation côté serveur et avec le panneau réseau du navigateur pour la taille de la charge utile des mises à jour.

Point de départ

WisejPerfLab après le Module 4 : les allocations pendant la recherche ont baissé et le formulaire de détail ne fuit plus, mais la grille est toujours liée à la liste complète des entités et l'arborescence des clients est toujours construite d'un seul coup au démarrage. Le budget du Module 1 contient un objectif de déploiement de l'arborescence que l'application manque largement.

Approche suggérée

- ✓ Mesurez les deux écrans avant d'y toucher : CPU Usage et .NET Object Allocation pour le chargement de la grille et le déploiement de l'arborescence, plus le panneau réseau du navigateur pour la charge utile des mises à jour que chacun produit. Notez le nombre et la taille des mises à jour, pas seulement le total compressé.
- ✓ Définissez `TicketGridRow` avec uniquement les colonnes affichées et leurs chaînes d'affichage précalculées, et faites-le renvoyer par le service de requêtes. Rien qui comporte des propriétés de navigation ou de grandes colonnes de texte ne doit atteindre l'interface.
- ✓ Passez la grille en mode virtuel : définissez `VirtualMode`, définissez `RowCount` à partir d'une requête de comptage, et répondez à `CellValueNeeded` depuis une page paginée et mise en cache de lignes de projection. Traitez ce gestionnaire comme un chemin critique — pas d'appel à la base de données par cellule, pas de formatage, aucun verrou qui bloquerait les requêtes parallèles.
- ✓ Retravaillez l'arborescence des clients pour charger les enfants au déploiement, avec un espace réservé de comptage sur les branches repliées, et décidez à partir de mesures si un léger préchargement au-delà de la zone visible vaut les requêtes supplémentaires.
- ✓ Réexécutez les deux scénarios sur le même jeu de données et le même build, et assemblez la comparaison : CPU serveur, allocations et lignes retenues d'après Visual Studio ; nombre de mises à jour et taille de la charge utile d'après le navigateur.

Livrables attendus

- ✓ Projection TicketGridRow liée à la place des entités complètes, avec uniquement les colonnes affichées
- ✓ Grille en mode virtuel avec RowCount et un gestionnaire CellValueNeeded servi par un service de requêtes paginées
- ✓ Arborescence des clients chargeant les nœuds enfants au déploiement, avec un espace réservé de comptage pour les branches repliées
- ✓ Chiffres CPU et d'allocation avant/après pour les scénarios de chargement de la grille et de déploiement de l'arborescence
- ✓ Preuves réseau du navigateur comparant la taille de la charge utile des mises à jour avant et après, avec la réserve concernant la compression

Questions de révision

- ✓ Le mode virtuel a supprimé les objets ligne, mais votre trace CPU s'est à peine améliorée. Qu'est-ce que cela vous apprend sur l'endroit où se trouvait réellement le coût, et quelle mesure examineriez-vous ensuite ?
- ✓ Votre gestionnaire `CellValueNeeded` est correct mais lent lors d'un défilement rapide. Citez trois choses qu'il pourrait faire et qu'un gestionnaire exécuté par cellule ne doit jamais faire.
- ✓ La charge utile compressée du déploiement de l'arborescence a à peine changé après le passage au chargement différé, mais les utilisateurs trouvent l'écran plus rapide. Donnez une explication plausible que le seul nombre d'octets masquerait.



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.