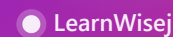


Chargements asynchrones, données liées, filtrage et performances

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



Guide de la leçon

La table de décision des données liées (projection, Include, Include filtré, chargement explicite, pas de chargement différé dans les grilles), avec ou sans suivi, la pagination et les filtres indexés, la coordination asynchrone avec états de chargement et tâches en arrière-plan, et mesurer avant d'optimiser. Ce que cela signifie pour un développeur Wisej.NET qui travaille avec de vraies bases de données — et comment aborder ce module.



Objectif production

La table de décision des données liées (projection, Include, Include filtré, chargement explicite, pas de chargement différé dans les grilles), avec ou sans suivi, la pagination et les filtres indexés, la coordination asynchrone avec états de chargement et tâches en arrière-plan, et mesurer avant d'optimiser.

1. Le problème en production

Le navigateur de tickets du Support Desk fonctionnait au Module 3 avec cinquante tickets de départ. Avec cinquante mille, il ne fonctionnera plus, à moins que la requête ne fasse moins de travail, au bon endroit. La défaillance est discrète : une grille qui met trois secondes parce que le code de mise en forme a touché une propriété de navigation par ligne, une session qui conserve des milliers d'entités suivies que personne ne modifiera, une recherche qui parcourt toute une table. Ce module consiste à faire moins de travail, en SQL, délibérément, et à le prouver avec un journal plutôt qu'avec une intuition.

2. Concepts clés

Concept	Pourquoi c'est important
projection	Un <code>Select</code> qui renvoie exactement les champs dont l'écran a besoin, comme <code>TicketListItem</code> , pour que la base joigne les noms liés en une seule instruction.
Include et Include filtré	Le chargement anticipé des entités liées avec la racine ; un <code>Include</code> filtré restreint une collection (les cinq derniers commentaires) pour qu'un éditeur charge un agrégat maîtrisé.
chargement explicite	Charger une navigation plus tard, à la demande, via <code>Collection</code> ou <code>Reference</code> sur l'entrée et <code>LoadAsync</code> , quand l'utilisateur déplie une section de détail.
chargement différé et N+1	Des propriétés de navigation chargées de façon transparente au premier accès ; dans une grille, cela donne une requête cachée par ligne ou par accès à une navigation.
AsNoTracking et résolution d'identité	Des résultats qui contournent le change tracker pour les écrans en lecture seule ; <code>AsNoTrackingWithIdentityResolution</code> renvoie tout de même une seule instance par entité quand la même ligne apparaît plusieurs fois.
pagination au niveau de la requête	<code>Where</code> , <code>OrderBy</code> , <code>Skip</code> , <code>Take</code> et <code>Select</code> composés avant <code>ToListAsync</code> , avec un <code>CountAsync</code> distinct pour le total.
journalisation EF Core	Le SQL généré par chaque commande, avec sa durée, via <code>LogTo</code> ou <code>Microsoft.Extensions.Logging</code> .

3. Le fil de la leçon

D'où vient la lenteur de la grille

Partez du navigateur de tickets volontairement lent que le lab mesure avant de le corriger. Sa méthode `SearchTicketsAsync` charge des entités `Ticket` suivies sans pagination, et `CellFormatting` lit `ticket.Customer.Name`, `ticket.Agent?.DisplayName` et `ticket.Category.Name`. Avec le chargement différé activé, chaque premier accès à une navigation non chargée exécute sa propre requête : cinquante lignes coûtent une instruction pour les tickets plus cent cinquante pour les noms. Rien ne lève d'exception ; seul le journal le révèle.

La règle qui élimine ce problème : le code d'affichage ne doit jamais provoquer de requête. Les lignes liées à un `DataGridView` Wisej.NET via la `BindingSource` sont des records `TicketListItem` qui portent déjà les noms qu'ils affichent, et `CellFormatting` se contente de mettre en forme. La même règle gouverne les écrans en arbre et en liste : un `TreeView` charge les enfants d'un nœud une seule fois, quand il est déplié, jamais en parcourant les propriétés de navigation nœud par nœud.



Erreur courante : chargement différé depuis la mise en forme des lignes ou le code d'affichage

Une propriété de navigation lue dans `CellFormatting` ou dans le dessin d'un nœud d'arbre est une requête déguisée qui s'exécute une fois par ligne. Transportez plutôt les noms dans la projection.

La table de décision des données liées

EF Core propose quatre façons d'amener des données liées dans un écran ; le choix dépend de ce que l'écran va en faire.

- Utilisez la **projection** quand l'interface a besoin d'un modèle de lecture à plat : le navigateur de tickets, les tableaux de bord, les exports.
- Utilisez **Include** quand un éditeur a besoin d'un agrégat complet dont la taille est maîtrisée : un ticket avec son client, sa catégorie et son agent.
- Utilisez l'**Include filtré** quand seul un petit sous-ensemble d'une collection est nécessaire : les cinq derniers commentaires publics.
- Utilisez le **chargement explicite** quand l'utilisateur déploie une section de détail à la demande : l'historique des commentaires derrière un panneau replié.
- Ne comptez jamais sur le **chargement différé** dans les grilles, les listes ou les arbres : il produit silencieusement une requête par ligne ou par accès à une navigation.

Dans le Support Desk, le navigateur projette vers `TicketListItem` ; la méthode `LoadEditorAsync` de l'éditeur utilise `Include` pour les trois références et un `Include` filtré pour les commentaires qu'elle affiche en aperçu ; le panneau des commentaires charge l'historique complet avec `db.Entry(ticket).Collection(t => t.Comments).LoadAsync()` dans le Context propre à l'éditeur.

Avec ou sans suivi

Le suivi est correct quand le même Context enregistrera les modifications de ces mêmes instances d'entités, comme le fait l'éditeur quand `SaveChangesAsync` écrit la différence. Le mode sans suivi est correct pour les grilles en lecture seule, les tableaux de bord, les listes de référence, les exports et les résultats d'autocomplétion. `AsNoTracking` contourne le change tracker : moins de mémoire par session, une matérialisation plus rapide, et aucune mise à jour accidentelle depuis un écran qui ne fait que lire.

`AsNoTrackingWithIdentityResolution` se situe entre les deux. Une requête sans suivi qui renvoie des entités peut rendre un objet `Customer` distinct pour chaque ticket d'un même client ; la résolution d'identité renvoie une seule instance par clé. Utilisez-la quand un écran en lecture seule construit un graphe d'objets ; passez-vous-en quand vous projetez vers des lignes à plat.



Erreur courante : utiliser des entités sans suivi et s'attendre ensuite à une détection automatique des modifications

Modifier un `Ticket` issu d'une requête `AsNoTracking`, ou un `TicketListItem` de la grille, puis appeler `SaveChangesAsync` ne change rien. L'éditeur charge une nouvelle entité suivie par sa clé dans son propre Context et y reporte les valeurs approuvées.

Pagination et filtres indexés au niveau de la requête

Une grille Wisej.NET ne doit pas charger 100 000 lignes simplement parce qu'elle sait les afficher. Composez les filtres, le tri, `Skip`, `Take` et `Select` au niveau de la requête pour que la base fasse le travail, et renvoyez un nombre total quand l'interface affiche des numéros de page, grâce à un `CountAsync` distinct sur la requête filtrée :

```

var total = await query.CountAsync(token);

var page = await query
    .AsNoTracking()
    .OrderByDescending(t => t.UpdatedAt)
    .Skip(criteria.PageIndex * 50)
    .Take(50)
    .Select(t => new TicketListItem(
        t.Id, t.Number, t.Title, t.Customer.Name,
        t.Agent == null ? null : t.Agent.DisplayName,
        t.Category.Name, t.Status, t.Priority, t.DueDate, t.UpdatedAt))
    .ToListAsync(token);

```

Les index du Module 2 sur `Status`, `DueDate`, `CustomerId` et `UpdatedAt` correspondent à ces filtres et à ce tri. Gardez la recherche textuelle compatible avec les index : un `StartsWith` sur le numéro de ticket devient un `LIKE` de préfixe qu'un index peut servir, alors qu'un `Contains` sur le titre devient un joker en tête qui parcourt la table. Quand le journal montre un filtre fréquent sans index, ajoutez-en un et mesurez à nouveau.

Coordination asynchrone avec états de chargement et tâches en arrière-plan

EF Core fournit des méthodes asynchrones pour chaque opération d'E/S. Attendez chacune avant de lancer la suivante sur le même Context, car un `DbContext` ne prend pas en charge les opérations parallèles ; les deux `await` ci-dessus s'exécutent en séquence par conception. Dans Wisej.NET, un gestionnaire d'événements asynchrone rend la main au navigateur pendant l'exécution de la requête ; la protection de chargement compte donc : affichez un état de chargement avec le `ShowLoader` de la grille ou un `Label` d'état, désactivez les boutons Search, Next et Previous qui lanceraient un travail concurrent, et réactivez-les dans un bloc `finally`.

Les tâches longues d'import, d'export et de génération de rapports n'ont pas leur place dans un gestionnaire d'événements. Lancez-les avec `Application.StartTask`, créez le `DbContext` à l'intérieur de la tâche depuis la fabrique plutôt que de capturer le Context ou les entités de la page, et ne poussez la progression que via `Application.Update`, le mécanisme de mise à jour de Wisej.NET qui tient compte du Context. Un thread brut qui touche un contrôle n'a aucun contexte de session, et le jeton d'annulation que vous transmettez permet d'arrêter le travail quand la session se termine.

Mesurer avant d'optimiser

Ne devinez pas. Activez la journalisation EF Core en développement avec `LogTo` ou `Microsoft.Extensions.Logging`, lisez le SQL et la durée de chaque commande, et n'activez `EnableSensitiveDataLogging` que sur un poste de développeur, puisqu'il affiche les valeurs des paramètres. Pour une instruction lente, examinez le plan d'exécution de la base, où un index manquant ou un parcours dû à un joker apparaît clairement. Le journal avant et après du lab est la mise en pratique concrète de cette discipline.

Ce n'est qu'une fois que le profilage a montré le goulot d'étranglement que les outils spécialisés justifient leur complexité. Les requêtes compilées aident quand le coût de compilation est mesurable sur une requête très sollicitée. Le pooling de Contexts avec `AddPooledDbContextFactory` aide quand la

construction du Context domine, et demande de la prudence quand l'état du Context varie selon l'utilisateur ou le tenant, puisqu'un Context du pool est réinitialisé puis réutilisé. Les requêtes fractionnées aident quand plusieurs `Include` de collections multiplient les lignes. Le SQL brut est le dernier recours. Chacun répond à un chiffre du journal, jamais à un réflexe par défaut.



Erreur courante : optimiser sans mesurer

Des requêtes compilées, du pooling ou du SQL brut sur un écran non profilé sacrifient la lisibilité pour une amélioration qui n'existe peut-être pas. Mesurez, changez une seule chose, mesurez à nouveau.



Réflexe de liaison de données

Séparez les deux durées de vie. Les formulaires Wisej.NET, les BindingSources et la ligne sélectionnée vivent pendant la session ; un DbContext vit le temps d'une opération. Créez-le depuis la fabrique, interrogez ou enregistrez, libérez-le, et ne le laissez jamais devenir un état de session.



Objectifs d'apprentissage

À la fin de ce module, vous saurez :

- ✓ Éviter le comportement N+1 du chargement différé dans les grilles et les écrans en arbre ou en liste.
- ✓ Utiliser délibérément Include, l'Include filtré, le chargement explicite et la projection.
- ✓ Utiliser AsNoTracking, AsNoTrackingWithIdentityResolution, la pagination, les index et des améliorations de performances mesurées.
- ✓ Coordonner les opérations asynchrones d'EF Core avec les indicateurs de chargement et les tâches en arrière-plan de Wisej.NET.

Guide du lab / de l'examen

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

Lab pratique

Optimisez le navigateur de tickets du Support Desk : activez la journalisation EF Core en développement, relevez le SQL et le temps de la recherche non optimisée, remplacez le chargement des entités par une projection, ajoutez AsNoTracking aux requêtes des grilles, ajoutez une pagination de 50 lignes par page, remplacez l'accès différé aux navigations par une projection ou un Include, et documentez les résultats mesurés avant et après dans vos notes de lab.

Point de départ

Ouvrez la solution Support Desk Data Console du Module 5 et la page du navigateur de tickets du Module 3. Pour ce lab, partez d'une branche volontairement naïve de `TicketQueryService.SearchTicketsAsync` : une version qui charge des entités `Ticket` suivies sans pagination et laisse le `CellFormatting` de la grille lire les noms du client, de l'agent et de la catégorie via les propriétés de navigation, afin d'avoir un vrai N+1 à mesurer avant de le supprimer.

Approche suggérée

- ✓ Activez la journalisation EF Core en développement avec `LogTo` ou `Microsoft.Extensions.Logging` sur `SupportDeskContext`, gardez `EnableSensitiveDataLogging` derrière une vérification de l'environnement de développement, et exécutez une fois la recherche naïve ; comptez les instructions, notez les millisecondes et collez le SQL dans vos notes de lab.
- ✓ Réécrivez la requête de la grille en projection sans suivi : `AsNoTracking`, les filtres `Where` composés, `OrderByDescending(t => t.UpdatedAt)`, puis un `Select` vers `TicketListItem` avec les noms du client, de l'agent et de la catégorie joints par la base, en terminant par `ToListAsync`.
- ✓ Ajoutez une pagination de 50 lignes par page : un `CountAsync` sur la requête filtrée pour le total, puis `Skip` et `Take` avant la projection ; assurez-vous que chaque filtre et le tri utilisent les index sur Status, DueDate, CustomerId et UpdatedAt, et gardez la recherche par numéro de ticket sous forme de correspondance de préfixe.
- ✓ Supprimez tout accès aux navigations dans le code d'affichage : `CellFormatting` ne fait que mettre en forme les champs déjà présents dans la ligne, et toute donnée liée dont l'éditeur a besoin est chargée avec `Include`, un `Include` filtré ou un chargement explicite dans le Context propre à l'éditeur.
- ✓ Gardez une protection asynchrone rigoureuse : attendez chaque opération de base de données avant la suivante sur le même Context, affichez l'état de chargement et désactivez Search, Next Page et Previous Page pendant le chargement d'une page, réactivez-les dans `finally`, puis lancez la recherche optimisée et notez les chiffres après à côté des chiffres avant.

Livrables attendus

- ✓ Journalisation EF Core activée en développement avec le SQL et le temps non optimisés relevés
- ✓ Requête de la grille réécrite en projection sans suivi
- ✓ Pagination de 50 lignes par page et filtres indexés
- ✓ Plus aucun comportement N+1 de chargement différé dans le code d'affichage
- ✓ Notes de lab documentant l'amélioration mesurée avant et après

Questions de révision

- ✓ Votre journal a montré 151 instructions pour un écran de 50 tickets. Quelle ligne de code a provoqué les 150 instructions en trop, et pourquoi rien dans l'application ne vous a-t-il averti ?
- ✓ L'éditeur a besoin du ticket, de son client et de sa catégorie, ainsi que des cinq derniers commentaires ; le panneau des commentaires ouvre l'historique complet à la demande. Quelle stratégie de chargement répond à chacun de ces trois besoins, et pourquoi la projection n'est-elle pas la bonne réponse pour l'éditeur ?
- ✓ Un collègue propose des requêtes compilées et du pooling de Contexts pour le navigateur de tickets avant même de regarder le journal. Que mesureriez-vous d'abord, et que devraient montrer les chiffres pour que l'un ou l'autre changement en vaille la peine ?



Réflexe de liaison de données

Séparez les deux durées de vie. Les formulaires Wisej.NET, les BindingSources et la ligne sélectionnée vivent pendant la session ; un DbContext vit le temps d'une opération. Créez-le depuis la fabrique, interrogez ou enregistrez, libérez-le, et ne le laissez jamais devenir un état de session.