

État, sessions, tâches en arrière-plan et erreurs

État, sessions et tâches en arrière-plan · Gestion sûre des erreurs et journalisation

État, sessions et tâches en arrière-plan

État de session par utilisateur et état partagé (et le piège des champs statiques), plus l'exécution d'un travail long dans une tâche en arrière-plan réactive.



Objectif de la leçon

Comprendre l'**état de session propre à chaque utilisateur** — et en quoi il diffère de l'état partagé — puis exécuter un travail long dans une **tâche en arrière-plan** qui garde la page réactive et met à jour la progression de façon délibérée.

1. L'état de session par utilisateur

Une application Wisej.NET s'exécute sur le serveur, mais chaque utilisateur connecté doit avoir l'impression de disposer de sa propre copie de l'application. L'**état de session** désigne les données qui appartiennent à une seule session utilisateur connectée — le ticket sélectionné, le filtre courant, la tâche en arrière-plan en cours et les messages de progression de cet utilisateur.

Type d'état	Ce que cela signifie	Exemple pour débutant
État de l'interface	Des informations servant uniquement à afficher l'écran courant.	Ligne sélectionnée, panneau visible, texte du libellé d'état.
État de session	Des informations qui appartiennent à une session utilisateur.	Tâche d'import en cours, tâches terminées pour cet utilisateur connecté.
État métier	Les vraies données de l'application, à enregistrer ou à partager délibérément.	Tickets stockés par un service / repository.
État statique / partagé	Un état partagé entre les utilisateurs, sauf conception soignée.	Une liste statique de tâches que chaque utilisateur peut modifier par accident.



Attention : état statique / partagé

Ne stockez pas des informations ordinaires propres à un utilisateur dans des champs `static`, sauf si vous voulez délibérément les partager. Une liste statique peut permettre à un utilisateur de voir les données ou la progression des tâches d'un autre — l'une des erreurs de débutant les plus fréquentes dans les applications côté serveur.

2. Travail en arrière-plan et pages réactives

Certains traitements sont trop longs pour s'exécuter directement dans un clic de bouton — exporter des enregistrements, importer des fichiers, envoyer des notifications, produire des rapports. Une bonne application lance la tâche, désactive les boutons à risque, affiche la progression et laisse la page utilisable pendant que la tâche se poursuit.

```
Ouverture de page → une nouvelle session voit son propre état
Clic sur un bouton → l'événement d'interface lance la logique serveur
Tâche de fond      → le travail long s'exécute sans figer l'interface
Progression        → état, barre et journal se mettent à jour sans risque
Fin / erreur       → l'utilisateur voit un message final clair
```

3. Mauvais pattern et meilleur pattern

Mauvais pattern	Meilleur pattern
Mettre tout le travail directement dans l'événement de clic et figer la page.	Lancer une tâche en arrière-plan et mettre à jour la progression / l'état de façon délibérée.
Laisser les utilisateurs cliquer plusieurs fois sur Start et créer des tâches en double.	Désactiver Start pendant qu'une tâche s'exécute ; activer Cancel quand c'est pertinent.
N'afficher qu'une erreur générique ou un plantage.	Journaliser les détails pour les développeurs, mais montrer un message court et aimable à l'utilisateur.



Vous êtes prêt pour la leçon suivante

Vous savez distinguer l'état de session de l'état partagé, repérer le piège des champs statiques et décrire le workflow d'une tâche en arrière-plan réactive. Ensuite : gérer les erreurs en toute sécurité et journaliser des détails utiles.

Gestion sûre des erreurs et journalisation

Interceptez les exceptions et affichez des messages sans risque, journalisez des détails utiles et gardez le gestionnaire de la tâche lisible avec try/catch/finally.



Objectif de la leçon

Intercepter les exceptions et afficher des messages utiles **sans** exposer de détails internes, ajouter une journalisation simple et garder lisible le gestionnaire d'événements de la tâche en arrière-plan grâce à un pattern try / catch / finally clair.

1. Gestion sûre des erreurs

La gestion des erreurs doit protéger à la fois l'utilisateur et le système. L'utilisateur a besoin d'un message clair sur ce qui s'est passé et sur la marche à suivre ; le développeur a besoin de suffisamment de détails dans le journal pour diagnostiquer. N'exposez jamais de chaînes de connexion, de chemins de fichiers, de traces de pile, de secrets ou de détails internes du serveur dans le message destiné à l'utilisateur.

Détail de l'erreur	À montrer à l'utilisateur ?	Pourquoi
« Import failed. Please try again or contact support. »	Oui	Utile et sans risque.
<code>NullPointerException</code> à la ligne 52 de <code>TicketService.cs</code>	Non	Détail d'implémentation interne — à mettre plutôt dans un journal.
Mot de passe de la base de données ou chaîne de connexion	Jamais	Information sensible.
ID de la tâche et horodatage	Parfois	Peut aider le support à retrouver le journal, à condition de ne révéler aucun secret.

2. Une liste de contrôle simple pour la journalisation

- Journalisez le démarrage de la tâche, chaque étape importante, la fin, l'annulation et l'échec.
- Incluez des horodatages et des noms de tâches pour que le journal soit utile plus tard.
- Gardez les messages destinés à l'utilisateur courts ; conservez les détails techniques dans le journal.
- Ne journalisez pas de mots de passe, de clés d'API, de numéros de carte complets ni de données privées sans raison sérieuse et sans protection.
- Lors du dépannage, vérifiez : l'état des boutons, l'état de la tâche en cours, la valeur de progression, le message de l'exception et la dernière entrée du journal.

3. Le pattern à retenir

Le clic sur le bouton lance le workflow. La tâche en arrière-plan effectue le travail long. L'interface met à jour la progression et l'état. Le service ou la méthode utilitaire porte la logique réutilisable. Le logger enregistre les détails. L'utilisateur voit des messages sans risque.

```
private async void btnStartImport_Click(object sender, EventArgs e)
{
    try
    {
        SetJobRunning(true);
        await RunImportJobAsync();
        lblStatus.Text = "Import completed successfully.";
    }
    catch (Exception ex)
    {
        LogError(ex);
        lblStatus.Text = "Import failed. Please try again or contact support.";
    }
    finally
    {
        SetJobRunning(false);
    }
}
```



Pourquoi finally compte

finally s'exécute que la tâche réussisse ou lève une exception : c'est donc le bon endroit pour remettre les boutons et l'état de l'interface dans une situation sûre — la page ne reste jamais bloquée en mode « en cours ».



Vous êtes prêt pour le lab

Ensuite : regardez la vidéo pas à pas sur le traitement en arrière-plan, réussissez la vérification des connaissances, puis construisez un exécuteur de tâches — boutons Start/Cancel, une barre de progression et un journal, un try/catch/finally qui journalise les détails mais affiche un message sans risque, et un état d'interface qui se réinitialise toujours.