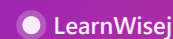


Applications temps réel avec push serveur — Recueil des leçons

L'ensemble des leçons des modules de ce cours, réunies dans un seul document.



Sommaire

01	Architecture web temps réel dans Wisej.NET	Module 1 · Guide de la leçon
02	Architecture web temps réel dans Wisej.NET	Module 1 · Guide du lab / de l'examen
03	Contexte applicatif, sessions et cycle de vie	Module 2 · Guide de la leçon
04	Contexte applicatif, sessions et cycle de vie	Module 2 · Guide du lab / de l'examen
05	Push serveur pour une session avec des tâches en arrière-plan	Module 3 · Guide de la leçon
06	Push serveur pour une session avec des tâches en arrière-plan	Module 3 · Guide du lab / de l'examen
07	Timers, polling de repli et cadence des mises à jour	Module 4 · Guide de la leçon
08	Timers, polling de repli et cadence des mises à jour	Module 4 · Guide du lab / de l'examen
09	Liaison de données en direct et grilles temps réel	Module 5 · Guide de la leçon
10	Liaison de données en direct et grilles temps réel	Module 5 · Guide du lab / de l'examen
11	Push multi-utilisateurs avec services et hubs d'événements	Module 6 · Guide de la leçon
12	Push multi-utilisateurs avec services et hubs d'événements	Module 6 · Guide du lab / de l'examen
13	Durcissement pour la production, déploiement et projet de synthèse	Module 7 · Guide de la leçon
14	Durcissement pour la production, déploiement et projet de synthèse	Module 7 · Guide du lab / de l'examen

Architecture web temps réel dans Wisej.NET

L'architecture temps réel pilotée par le serveur : push via WebSocket, mises à jour dans la requête et polling de repli dans le modèle côté serveur de Wisej.NET. Le concept, le modèle mental et les pièges en production — à lire avant le lab.



Objectif production

Architecture temps réel pilotée par le serveur : push via WebSocket, mises à jour dans la requête et polling de repli dans le modèle côté serveur de Wisej.NET.

Pourquoi ce module existe

On associe généralement les applications web temps réel au chat, aux jeux, aux cotations boursières, aux écrans de supervision et aux tableaux de bord. Dans les applications métier, le même besoin apparaît sous des formes moins spectaculaires mais plus précieuses : un superviseur voit arriver les nouveaux tickets, un opérateur de production voit une machine s'arrêter, un répartiteur voit qu'une intervention a été réaffectée, ou un employé de la facturation voit qu'un import est toujours en cours sans appuyer sur Actualiser.

Wisej.NET repose sur un modèle d'application côté serveur. Le développeur travaille en C# ou en VB.NET, utilise des contrôles, des événements, des formulaires, des pages, des timers, la liaison de données et le concepteur de Visual Studio, tandis que Wisej.NET maintient l'arbre des widgets côté navigateur et le synchronise avec l'état côté serveur. Ce modèle est particulièrement important pour le push serveur, car le serveur possède déjà l'état de l'application. Une tâche en arrière-plan peut modifier un label, ajouter une ligne à une grille, faire avancer une barre de progression ou ouvrir une notification, et Wisej.NET peut transmettre au navigateur les modifications d'interface en attente dès qu'une connexion WebSocket active existe.

Un cours pour débutants enseigne généralement à créer des pages, des fenêtres, des contrôles, des menus et des gestionnaires d'événements simples. Ce cours de niveau intermédiaire suppose ces compétences acquises et se concentre sur les questions de conception en production qui surgissent lorsque l'interface se met à jour d'elle-même. La question centrale n'est pas « Puis-je mettre à jour un label depuis un thread ? ». La question centrale est « Comment mettre à jour la session du bon utilisateur, au bon rythme, depuis le bon service, sans perdre d'état, sans fuite mémoire, sans bloquer le serveur et sans submerger le navigateur ? »

Ce que « temps réel » signifie dans ce cours

Dans ce cours, une application temps réel Wisej.NET est une application capable de recevoir les événements de l'utilisateur avec une faible latence et aussi de mettre à jour le navigateur de l'utilisateur depuis le serveur, sans attendre un nouveau clic. Le push serveur est la partie essentielle. L'application doit pouvoir modifier l'interface visible parce que quelque chose s'est produit sur le serveur : une tâche en arrière-plan a progressé, une file d'attente a changé, un message est arrivé ou un autre utilisateur a effectué une action.

Ne réduisez pas le temps réel à un « rafraîchissement rapide ». Une grille qui se rafraîchit toutes les 30 secondes est parfois acceptable, mais ce n'est pas du push. Le push est piloté par les événements : l'application se met à jour parce que le modèle a changé. Le polling demande de façon répétée si quelque chose a changé. Les deux peuvent être utiles, mais leur coût, leur latence et leur comportement en cas de défaillance diffèrent.

Le modèle mental du temps réel dans Wisej.NET

Voici un modèle mental utile :

1. Le navigateur héberge les widgets JavaScript de Wisej.NET.
2. Le serveur héberge la session de l'application Wisej.NET et les contrôles côté serveur.
3. Les actions de l'utilisateur transitent du navigateur vers le serveur sous forme d'événements.
4. Le code côté serveur modifie les contrôles et l'état de l'application.
5. Wisej.NET renvoie les modifications au navigateur sous forme de mises à jour compactes.
6. Avec WebSocket, une tâche en arrière-plan peut pousser les mises à jour en attente sans attendre la prochaine requête du navigateur.

Il ne s'agit ni de screen scraping ni d'un bureau à distance. Le navigateur affiche toujours des widgets web natifs et le serveur ne diffuse pas de pixels. Le point d'architecture important est que la logique applicative et l'état de référence de l'interface résident sur le serveur. Lorsqu'une propriété d'un contrôle Wisej.NET change, le widget côté navigateur peut être synchronisé.

Push via WebSocket et mises à jour pilotées par les requêtes

Une requête web classique commence lorsque le navigateur contacte le serveur et se termine lorsque le serveur répond. S'il n'existe aucun canal ouvert du serveur vers le navigateur, le serveur ne peut pas contacter le navigateur de sa propre initiative : il ne peut envoyer des modifications que lorsque le navigateur émet une nouvelle requête. WebSocket change la donne. Avec une connexion WebSocket, Wisej.NET peut pousser des mises à jour d'interface vers le client connecté depuis le code côté serveur.

Dans Wisej.NET, une tâche en arrière-plan peut modifier des contrôles et appeler `Application.Update()` pour pousser les modifications en attente lorsque WebSocket est disponible. Si WebSocket n'est pas disponible, `Application.Update()` ne peut rien pousser, faute de connexion active. L'application a alors besoin d'une solution de repli, comme le polling de Wisej.NET ou des requêtes déclenchées par un timer. Ce repli ne remet pas l'application en cause : il change simplement le mécanisme de livraison.

Trois mécanismes de mise à jour à bien distinguer

1. Mises à jour dans la requête

C'est le cas le plus simple. L'utilisateur clique sur un bouton, un gestionnaire d'événements s'exécute, il modifie des contrôles et Wisej.NET renvoie les modifications avec la réponse. En général, vous n'appellez pas `Application.Update()` ici, car la mise à jour est naturellement renvoyée au navigateur à la fin de la requête.

```
private void buttonRefresh_Click(object sender, EventArgs e)
{
    statusLabel.Text = "Refreshed at " + DateTime.Now.ToLongTimeString();
}
```

2. Push WebSocket hors requête

C'est le sujet principal du cours. Une tâche en arrière-plan s'exécute en dehors de l'événement d'origine du navigateur. Elle modifie l'interface côté serveur et appelle `Application.Update()` pour pousser les modifications via la connexion WebSocket.

```
private void buttonStart_Click(object sender, EventArgs e)
{
    buttonStart.Enabled = false;
    progressBar.Value = 0;

    Application.StartTask(() =>
    {
        for (int i = 1; i ≤ 100; i++)
        {
            progressBar.Value = i;
            statusLabel.Text = $"Processing {i}%";
            Application.Update(this);
            Thread.Sleep(100);
        }

        buttonStart.Enabled = true;
        statusLabel.Text = "Done";
        Application.Update(this);
    });
}
```

3. Polling ou rafraîchissement par timer

Lorsqu'un navigateur ou un serveur ne peut pas utiliser WebSocket, le serveur ne peut pas pousser de modifications hors requête. Dans ce cas, l'application peut recourir au polling : le navigateur contacte périodiquement le serveur et reçoit les modifications en attente. Un `Wisej.Web.Timer` peut aussi servir à déclencher des événements côté serveur à intervalle défini.

```
private void MainPage_Load(object sender, EventArgs e)
{
    // Ignored when WebSocket is available. Useful as fallback when it is not.
    Application.StartPolling(1000);
}
```

Le cours vous apprend à concevoir d'abord pour WebSocket, puis à dégrader le comportement de façon prévisible.

Le projet du cours : TicketOps Live

Tous les modules construisent des éléments de la même application, **TicketOps Live**. Il s'agit d'une console d'exploitation du support en temps réel, dotée des fonctionnalités suivantes :

- un bandeau d'état du système en direct ;
- un moniteur de progression des tâches serveur ;
- un tableau de tickets qui se met à jour lorsque des tickets sont créés ou modifiés ;
- un panneau de notifications pour les événements multi-utilisateurs ;
- un tableau de bord d'exploitation avec des indicateurs simples de santé du serveur ;
- une checklist de production pour le déploiement derrière IIS ou un répartiteur de charge.

Le projet est volontairement simple sur le plan métier. L'objectif est d'apprendre l'architecture temps réel, pas d'implémenter un helpdesk complet. Chaque lab doit laisser à l'apprenant du code réutilisable dans de vraies applications.

Règle de conception : d'abord le modèle, ensuite l'interface

Il est tentant de laisser les tâches en arrière-plan manipuler directement chaque contrôle à modifier. Cela fonctionne pour de petites démos, mais les vraies applications ont besoin d'une séparation entre données et présentation. La tâche en arrière-plan doit en général mettre à jour un modèle ou lever un événement métier, et l'interface propre à la session décide de la manière de l'afficher.

Pour une fenêtre de progression mono-utilisateur, mettre à jour directement les contrôles est acceptable. Pour des tableaux de bord multi-utilisateurs, un accès global direct aux contrôles est une erreur. Les contrôles appartiennent aux sessions. Un service global ne doit pas conserver indéfiniment des instances de formulaires arbitraires. Les sessions s'abonnent plutôt à un service, reçoivent des événements et mettent à jour leurs propres contrôles dans leur propre contexte d'application.

Un vocabulaire sûr pour ce cours

Session désigne une instance en cours d'exécution d'une application Wisej.NET pour un contexte d'onglet ou de fenêtre de navigateur. Ce n'est pas la même chose qu'un utilisateur connecté : un même utilisateur peut avoir plusieurs sessions.

Contexte d'application désigne le contexte Wisej.NET dont le code a besoin pour accéder aux propriétés statiques **Application** propres à la session et mettre à jour le bon navigateur.

Code dans la requête (in-bound) désigne du code exécuté dans le cadre d'une requête du navigateur ou d'un événement Wisej.NET.

Code hors requête (out-of-bound) désigne du code exécuté en dehors d'un événement du navigateur, comme un thread en arrière-plan, un événement de hosted service, un callback de file d'attente ou un callback de timer.

Push signifie que le serveur transmet les mises à jour d'interface en attente au navigateur via une connexion active.

Polling signifie que le navigateur interroge périodiquement le serveur pour obtenir les mises à jour.



Erreurs fréquentes

L'erreur la plus fréquente consiste à mettre à jour un contrôle depuis un thread en arrière-plan sans préserver le bon contexte d'application. La deuxième consiste à pousser trop souvent. Une barre de progression n'a pas besoin de 500 mises à jour par seconde. Le push est puissant, mais chaque mise à jour a un coût : sérialisation, trafic réseau, travail du navigateur et ordonnancement côté serveur.

Autre erreur courante : utiliser des champs statiques pour des données propres à la session. Les champs statiques sont partagés par tous les utilisateurs et toutes les sessions du domaine d'application. Utilisez `Application.Session`, un service par session ou `Application.GetInstance()` pour les objets propres à la session. Réservez les services statiques aux données réellement partagées, et concevez alors la concurrence avec soin.



Point d'étape

À la fin de ce module, l'apprenant doit être capable de dessiner le chemin de bout en bout d'un événement de ticket généré par le serveur : le service modifie les données, le service lève un événement, une session le reçoit, la session met à jour ses contrôles, et Wisej.NET pousse les modifications d'interface qui en résultent lorsque WebSocket est disponible.

Architecture web temps réel dans Wisej.NET

Ce que vous allez construire dans le lab pratique, les tâches, l'implémentation suggérée et les critères d'acceptation.



Lab pratique

Créez le premier écran de TicketOps Live : une page avec un bandeau d'état en direct, un label d'horloge, un indicateur de progression et un heartbeat serveur simulé qui met à jour le navigateur sans aucun clic de l'utilisateur.

Point de départ

Créez une Wisej.NET Web Page Application nommée `TicketOpsLive`. Ajoutez une page principale nommée `MainPage`. Utilisez n'importe quel thème intégré. Le lab suppose C#, mais les mêmes concepts s'appliquent à VB.NET.

Tâches

1. Ajoutez un panneau supérieur nommé `statusPanel`.
2. Ajoutez les labels `clockLabel`, `connectionLabel` et `activityLabel`.
3. Ajoutez une barre de progression nommée `serverLoadBar`.
4. Ajoutez les boutons `startButton` et `stopButton`.
5. Implémentez une tâche en arrière-plan qui simule le heartbeat lorsque l'on clique sur `startButton`.
6. Le heartbeat doit mettre à jour l'horloge, une valeur de charge fictive et un message d'activité une fois par seconde.
7. Arrêtez les mises à jour lorsque l'on clique sur `stopButton` ou lorsque la page est libérée (disposed).

Implémentation suggérée

```
private volatile bool _running;

private void startButton_Click(object sender, EventArgs e)
{
    if (_running)
        return;

    _running = true;
    startButton.Enabled = false;
    stopButton.Enabled = true;
    connectionLabel.Text = "Server heartbeat running";

    Application.StartTask(() =>
    {
```

```

var random = new Random();
var counter = 0;

while (_running && !this.IsDisposed)
{
    counter++;
    clockLabel.Text = DateTime.Now.ToString("HH:mm:ss");
    serverLoadBar.Value = random.Next(5, 95);
    activityLabel.Text = $"Heartbeat #{counter}";

    Application.Update(this);
    Thread.Sleep(1000);
}
});
}

private void stopButton_Click(object sender, EventArgs e)
{
    _running = false;
    startButton.Enabled = true;
    stopButton.Enabled = false;
    connectionLabel.Text = "Server heartbeat stopped";
}

```

Critères d'acceptation

- ✓ L'interface se met à jour une fois par seconde après que l'apprenant a cliqué sur Start.
- ✓ Le navigateur affiche les modifications sans clic supplémentaire.
- ✓ Stop désactive la boucle en moins d'une seconde.
- ✓ Démarrer deux fois ne crée pas deux boucles concurrentes.
- ✓ Le code vérifie si la page a été libérée (disposed).

Questions de réflexion

1. Pourquoi ce lab appelle-t-il `Application.Update(this)` à l'intérieur de la boucle ?
2. Qu'est-ce qui change si WebSocket n'est pas disponible ?
3. Pourquoi `_running` n'est-il pas un champ statique ?
4. Quel serait le risque d'une mise à jour toutes les 10 millisecondes ?

Contexte applicatif, sessions et cycle de vie

Sessions, contexte d'application et cycle de vie : piloter l'interface du bon utilisateur, éviter le piège des champs statiques et faire le ménage à la sortie et à l'expiration. Le concept, le modèle mental et les pièges en production — à lire avant le lab.



Objectif production

Sessions, contexte d'application et cycle de vie : piloter l'interface du bon utilisateur, éviter le piège des champs statiques et faire le ménage à la sortie et à l'expiration.

Pourquoi les sessions comptent davantage dans les applications temps réel

Dans une page requête/réponse traditionnelle, l'utilisateur clique, le serveur répond, et l'essentiel du travail côté serveur s'arrête là. Dans une application temps réel Wisej.NET, le travail peut se poursuivre après la requête d'origine. Une tâche en arrière-plan peut durer plusieurs minutes. Un tableau de bord peut s'abonner à un flux d'événements global. Un timer peut continuer à mettre à jour un graphique. Comme le travail survit au clic qui l'a lancé, le cycle de vie de la session fait partie de l'architecture de l'application.

Une session Wisej.NET ressemble à une instance d'application de bureau en cours d'exécution. Elle possède des contrôles, des fenêtres, des pages, l'état des composants et des objets de niveau application. Mais contrairement à une application de bureau, le navigateur et le serveur peuvent se déconnecter, le navigateur peut être actualisé, et il n'existe pas de thread d'interface dédié permanent. Wisej.NET restaure l'état de session correct pour chaque requête et peut conserver l'état côté serveur pour les connexions HTTP comme WebSocket.

Une session n'est pas un utilisateur

Un utilisateur connecté et une session sont liés, mais pas identiques. Un même utilisateur peut ouvrir deux onglets et créer deux sessions. Un utilisateur peut s'authentifier dans une session, se déconnecter et laisser la session active. Un utilisateur anonyme peut avoir une session avant de se connecter. Un service en arrière-plan peut exister sans appartenir à aucune session utilisateur.

Cette distinction compte pour le push. Lorsqu'un ticket change, l'application peut devoir notifier toutes les sessions d'un tenant, une seule session pour un opérateur, ou chaque tableau de bord actif. Ne partez pas du principe que « utilisateur » et « navigateur » sont la même chose.

Contexte d'application

La classe `Application` de Wisej.NET est statique, mais ses propriétés et ses opérations sont isolées par session lorsque le code s'exécute dans le bon contexte. Lorsque le code s'exécute parce que le navigateur a levé un événement, le contexte est déjà disponible. Lorsque le code s'exécute dans un thread hors requête, il peut ignorer quelle session il doit mettre à jour.

`Application.StartTask()` est le moyen le plus simple de lancer un traitement en arrière-plan depuis une session, car la tâche démarre avec le contexte de la session appelante. Lorsque le code est déjà hors requête, vous pouvez capturer `Application.Current` tant que vous êtes dans le contexte, puis utiliser

plus tard `Application.Update(context, action)` ou `Application.RunInContext(context, action)` pour exécuter du code sur la bonne session.

```
private IWisejComponent _context;

private void MainPage_Load(object sender, EventArgs e)
{
    _context = Application.Current;
}

private void OnExternalEvent(object sender, TicketChangedEventArgs e)
{
    Application.Update(_context, () =>
    {
        notificationsList.Items.Add(e.Message);
    });
}
```

Le contexte n'est pas une autorisation globale de garder des contrôles en vie indéfiniment. Il doit s'accompagner d'une stratégie claire d'abonnement et de nettoyage.

Où placer l'état

Dans une application temps réel Wisej.NET, l'état se trouve couramment à quatre endroits.

L'état des contrôles est stocké directement dans les contrôles : le texte d'un label, la ligne sélectionnée d'une grille, une fenêtre visible, l'onglet courant ou la valeur d'une barre de progression. C'est simple et souvent correct pour l'état de présentation.

Application.Session stocke des valeurs propres à la session qui ne doivent pas être visibles par les autres sessions. Utilisez-le pour des métadonnées de session légères, le tenant sélectionné, les préférences de l'utilisateur ou des objets par session lorsqu'un service formel n'est pas nécessaire.

Les services de session sont des objets créés par session. Ils sont utiles lorsque vous avez besoin d'une conception plus structurée et plus testable que des champs de session improvisés. Les patterns de Wisej.NET adaptés aux sessions aident à éviter les partages statiques accidentels.

Les services globaux sont partagés par toutes les sessions. Ils conviennent aux files d'attente, aux hubs d'événements, au cache, aux watchers en arrière-plan ou à l'état de l'ensemble du système. Ils doivent être thread-safe et ne doivent pas stocker les contrôles d'une session sans cycle de vie d'abonnement défini.

Le piège des champs statiques

Les champs statiques sont partagés. Si vous stockez `CurrentTicketId` dans un champ statique, chaque utilisateur peut écraser le ticket courant de tous les autres. Si vous stockez une référence à `MainPage` dans une liste statique sans jamais la retirer, le serveur peut garder des sessions en vie après le départ des utilisateurs. Si vous stockez un `CancellationTokenSource` dans un champ statique pour une tâche propre à un utilisateur, un utilisateur peut annuler la tâche d'un autre.

Réservez les champs statiques aux services réellement globaux. Pour les valeurs propres à la session, utilisez les contrôles, `Application.Session` ou un objet par session.

Actualisation, keep-alive et expiration

Une application Wisej.NET doit composer avec les réalités du web : les utilisateurs actualisent le navigateur, les portables se mettent en veille, les réseaux disparaissent et des onglets restent ouverts toute la nuit.

Wisej.NET s'appuie sur l'expiration de session et le keep-alive pour décider si une session doit rester active ou être terminée. Lorsqu'une session expire, elle est retirée de la mémoire. Cela ressemble davantage à la fermeture d'une application de bureau qu'au masquage d'une page web.

Pour les applications temps réel, les conséquences pratiques sont les suivantes :

- Un traitement en arrière-plan de longue durée ne doit pas supposer que l'interface existe toujours.
- Le code de mise à jour de l'interface doit vérifier si une page ou un contrôle a été libéré (disposed).
- Les abonnements à des événements globaux doivent être retirés lorsque la session se termine ou que la vue est libérée.
- Si une tâche doit se poursuivre après la fin d'une session, elle doit appartenir à un service global, et non à la page.
- Si une tâche n'a de sens que pour l'interface d'un seul utilisateur, elle doit être annulée lorsque cette interface disparaît.

Patterns de nettoyage

Une vue temps réel abonnée à un service global doit toujours se désabonner. L'événement exact dépend de la structure de votre vue, mais le pattern reste le même : s'abonner au chargement ou à la construction, se désabonner lors de la libération ou à la sortie de l'application.

```

private readonly TicketFeed _feed = TicketFeed.Instance;
private IWisejComponent _context;

private void MainPage_Load(object sender, EventArgs e)
{
    _context = Application.Current;
    _feed.TicketChanged += Feed_TicketChanged;
    Application.ApplicationExit += Application_ApplicationExit;
}

private void Application_ApplicationExit(object sender, EventArgs e)
{
    _feed.TicketChanged -= Feed_TicketChanged;
}

private void Feed_TicketChanged(object sender, TicketChangedEventArgs e)
{
    if (this.IsDisposed)
        return;

    Application.Update(_context, () =>
    {
        ticketLog.Items.Insert(0, e.Message);
    });
}

```

Le but n'est pas seulement d'éviter les exceptions, mais de rendre la propriété évidente. Le flux possède les événements de ticket globaux. La page possède ses contrôles. La session possède son contexte d'application.

Gérer les erreurs dans les tâches en arrière-plan

Une tâche en arrière-plan ne doit jamais échouer silencieusement. Dans une démo de formation, une exception peut se contenter d'arrêter la boucle. En production, les échecs silencieux font mentir les tableaux de bord. Utilisez un try/catch dans la tâche, journalisez l'exception et mettez à jour l'interface avec un message d'erreur sûr lorsque l'interface existe encore.

```
Application.StartTask() =>
{
    try
    {
        RunLongOperation();
    }
    catch (Exception ex)
    {
        Log.Error(ex);
        if (!this.IsDisposed)
        {
            labelStatus.Text = "The operation failed. See the server log.";
            Application.Update(this);
        }
    }
}
});
```

N'affichez pas le détail brut des exceptions aux utilisateurs finaux en production. Utilisez des journaux détaillés pour les développeurs et des messages sûrs pour les utilisateurs.



Point d'étape

À la fin de ce module, l'apprenant doit être capable d'expliquer pourquoi une tâche en arrière-plan lancée depuis une page peut mettre à jour cette page, pourquoi un service global ne doit pas posséder directement les contrôles d'une page, et pourquoi chaque abonnement d'une application temps réel doit prévoir son désabonnement.

Contexte applicatif, sessions et cycle de vie

Ce que vous allez construire dans le lab pratique, les tâches, l'implémentation suggérée et les critères d'acceptation.



Lab pratique

Ajoutez un panneau de diagnostic qui permet au développeur de voir la session courante, le contexte d'application, les informations sur le navigateur et les événements de cycle de vie utiles aux mises à jour temps réel.

Tâches

1. Ajoutez une group box de diagnostic à `MainPage`.
2. Affichez l'heure courante, l'identifiant client, le type d'appareil du navigateur s'il est disponible et un compteur propre à la session.
3. Stockez le compteur dans `Application.Session` ou dans un objet par session.
4. Ajoutez des boutons pour incrémenter le compteur et pour simuler une mise à jour en arrière-plan.
5. Ajoutez une journalisation du nettoyage à la sortie de l'application.
6. Actualisez le navigateur et observez ce qui persiste.

Extraits suggérés

```
private void MainPage_Load(object sender, EventArgs e)
{
    if (Application.Session.Counter == null)
        Application.Session.Counter = 0;

    clientIdLabel.Text = Application.ClientId;
    titleLabel.Text = "TicketOps Live - " + DateTime.Now.ToShortTimeString();
    Application.Title = "TicketOps Live";

    Application.ApplicationExit += Application_ApplicationExit;
}

private void incrementButton_Click(object sender, EventArgs e)
{
    Application.Session.Counter = (int)Application.Session.Counter + 1;
    counterLabel.Text = Application.Session.Counter.ToString();
}

private void backgroundButton_Click(object sender, EventArgs e)
{
    var context = Application.Current;
    Application.StartTask(() =>
    {
        Thread.Sleep(1500);
    });
}
```

```

Application.Update(context, () =>
{
    lifecycleList.Items.Add("Background update at " + DateTime.Now);
});
});
}

private void Application_ApplicationExit(object sender, EventArgs e)
{
    // Replace with your logging framework in production.
    System.Diagnostics.Debug.WriteLine("TicketOps session exited: " + Application.ClientId);
}

```

Critères d'acceptation

- ✓ Le compteur est propre à la session, et non statique.
- ✓ Une mise à jour en arrière-plan modifie la bonne session du navigateur.
- ✓ Le panneau de diagnostic indique quand la mise à jour en arrière-plan arrive.
- ✓ L'apprenant sait expliquer ce qui se passe après une actualisation et après l'expiration de la session.

Questions de réflexion

1. Pourquoi `Application.Current` est-il capturé avant de lancer l'action hors requête ?
2. Que se passerait-il si le compteur était statique ?
3. Quelle action de nettoyage ajouteriez-vous pour un abonnement à un service global ?

Push serveur pour une session avec des tâches en arrière-plan

Push mono-session : exécuter les traitements longs hors du gestionnaire de clic, signaler la progression, prendre en charge l'annulation coopérative et restaurer l'interface dans le bloc finally. Le concept, le modèle mental et les pièges en production — à lire avant le lab.



Objectif production

Push mono-session : exécuter les traitements longs hors du gestionnaire de clic, signaler la progression, prendre en charge l'annulation coopérative et restaurer l'interface dans le bloc finally.

Le scénario de push serveur le plus courant

La première utilisation du push serveur est le plus souvent une opération de longue durée. Un utilisateur lance un import, un rapport, une étape de migration, une génération de fichier, une synchronisation de données ou une routine de validation. L'opération dure plusieurs secondes ou minutes. L'interface ne doit pas se figer, l'utilisateur doit voir la progression, et l'opération doit signaler son succès ou son échec à la fin.

Dans les applications de bureau, les développeurs utilisent souvent un background worker ou une tâche et renvoient les mises à jour vers le thread de l'interface. Dans Wisej.NET, la conception équivalente est un traitement en arrière-plan dans le contexte de la session, complété par `Application.Update()` lorsque vous devez pousser des modifications d'interface vers le navigateur. L'interface côté serveur appartient toujours à la session. La tâche en arrière-plan doit respecter le cycle de vie, l'annulation et la fréquence des mises à jour.

Pattern de base

Le pattern de base comporte six étapes :

1. Désactiver ou ajuster l'interface avant de commencer le travail.
2. Créer l'état d'annulation si l'utilisateur peut annuler.
3. Lancer la tâche en arrière-plan dans le contexte de la session.
4. Exécuter le travail par petites étapes.
5. Mettre à jour l'état de l'interface à intervalles raisonnables.
6. Réactiver l'interface et afficher la fin ou l'échec.

```

private CancellationTokenSource _cts;

private void startImportButton_Click(object sender, EventArgs e)
{
    _cts = new CancellationTokenSource();
    startImportButton.Enabled = false;
    cancelImportButton.Enabled = true;
    importProgress.Value = 0;
    importStatus.Text = "Starting import ... ";

    Application.StartTask(() => RunImport(_cts.Token));
}

private void cancelImportButton_Click(object sender, EventArgs e)
{
    _cts?.Cancel();
    cancelImportButton.Enabled = false;
    importStatus.Text = "Cancelling ... ";
}

```

La méthode de la tâche doit contenir un try/catch/finally afin que l'interface ne reste pas désactivée après une exception.

```

private void RunImport(CancellationToken token)
{
    try
    {
        for (int i = 1; i ≤ 100; i++)
        {
            token.ThrowIfCancellationRequested();

            // Simulate work.
            Thread.Sleep(75);

            importProgress.Value = i;
            importStatus.Text = $"Imported {i}%";

            if (i % 5 == 0)
                Application.Update(this);
        }

        importStatus.Text = "Import complete.";
    }
    catch (OperationCanceledException)
    {
        importStatus.Text = "Import cancelled.";
    }
    catch (Exception ex)
    {
        LogError(ex);
        importStatus.Text = "Import failed. See the server log.";
    }
}

```

```

finally
{
    startImportButton.Enabled = true;
    cancelImportButton.Enabled = false;
    Application.Update(this);
}
}

```

Observez la fréquence des mises à jour. Le code modifie la barre de progression à chaque itération, mais ne pousse que toutes les cinq itérations, plus la mise à jour finale. Cela réduit le nombre de messages réseau et de rafraîchissements du navigateur tout en donnant une impression de direct.

Mise à jour directe de l'interface ou callback de contexte

Lorsqu'une tâche est lancée avec `Application.StartTask()` depuis un événement Wisej.NET, elle s'exécute avec le contexte d'application courant. Dans de nombreux exemples, vous verrez des modifications directes de contrôles suivies de `Application.Update(this)`. C'est acceptable pour une tâche mono-session qui appartient à la page.

Pour du code qui démarre en dehors de la session, utilisez la surcharge avec contexte explicite :

```

var context = Application.Current;

ExternalWorker.JobAdvanced += (s, e) =>
{
    Application.Update(context, () =>
    {
        progressBar.Value = e.Percent;
        labelStatus.Text = e.Message;
    });
};

```

Il est ainsi clair que les modifications de l'interface ont lieu dans le bon contexte de session.

Ne bloquez pas les gestionnaires d'événements

N'exécutez pas une opération longue directement dans un gestionnaire de clic. La requête du navigateur reste ouverte, l'utilisateur ne voit aucune mise à jour intermédiaire poussée, et le thread du serveur reste occupé trop longtemps. Lancez l'opération et rendez la main à l'utilisateur.

À éviter :

```

private void button_Click(object sender, EventArgs e)
{
    RunTenMinuteImport();
    labelStatus.Text = "Done";
}

```

Mieux :

```
private void button_Click(object sender, EventArgs e)
{
    Application.StartTask(() => RunTenMinuteImportWithProgress());
    labelStatus.Text = "Import started";
}
```

L'annulation est une fonctionnalité, pas un détail

Une interface de progression temps réel sans annulation peut être pire qu'une interface bloquante : elle indique à l'utilisateur que quelque chose se passe sans lui laisser aucun contrôle. L'annulation doit être intégrée aux tâches de longue durée partout où c'est sûr. Ne tuez pas les threads. Utilisez l'annulation coopérative. Vérifiez le jeton entre deux unités de travail et laissez le système dans un état cohérent.

L'annulation est aussi utile lorsque la page se ferme. Si le traitement en arrière-plan ne concerne que cette page, annulez-le lors de la libération ou à la sortie de l'application. S'il s'agit d'une tâche serveur qui doit se poursuivre, confiez-la à un service et laissez la page afficher l'état de ce service.

Signalement des erreurs

Une tâche en arrière-plan peut échouer après le retour du gestionnaire d'événements d'origine. Sans gestion explicite des erreurs, l'utilisateur verra seulement que la progression s'est arrêtée. Interceptez toujours les exceptions dans le traitement en arrière-plan. Journalisez les détails techniques. Poussez vers l'interface un état d'erreur sans risque pour l'utilisateur. Conservez suffisamment de détails pour le dépannage : identifiant de la tâche, identifiant de session/client, identifiant de l'utilisateur s'il est authentifié, nom de l'opération et heure.

Regroupement et limitation du débit

L'œil humain ne tire aucun bénéfice de mises à jour illimitées. La plupart des tableaux de bord métier paraissent en direct avec 1 à 4 mises à jour par seconde. Les barres de progression paraissent souvent fluides à 5-10 mises à jour par seconde, mais beaucoup d'opérations n'en ont pas besoin. Une télémétrie à haute fréquence doit être agrégée avant affichage.

Règles pratiques :

- Poussez l'état final immédiatement.
- Poussez la progression visible par l'utilisateur à intervalle contrôlé.
- Regroupez plusieurs modifications de labels ou de grilles dans un seul appel à `Application.Update()`.
- Évitez de pousser à l'intérieur de boucles CPU serrées.
- Préférez une file d'attente ou un timer pour lisser les rafales provenant de systèmes externes.



Point d'étape

À la fin de ce module, l'apprenant doit être capable d'implémenter une opération de longue durée qui démarre immédiatement, met à jour la progression sans bloquer le navigateur, prend en charge l'annulation, signale les erreurs et laisse l'interface dans un état cohérent.

Push serveur pour une session avec des tâches en arrière-plan

Ce que vous allez construire dans le lab pratique, les tâches, l'implémentation suggérée et les critères d'acceptation.



Lab pratique

Ajoutez à TicketOps Live un panneau d'import de longue durée réaliste. L'utilisateur peut lancer un import, suivre sa progression, l'annuler et examiner le résultat.

Exigences de l'interface

Ajoutez :

- ✓ `startImportButton`
- ✓ `cancelImportButton`
- ✓ `importProgressBar`
- ✓ `importStatusLabel`
- ✓ `importLogListBox`
- ✓ `recordsImportedLabel`

Tâches

1. Implémentez une boucle d'import sur 200 enregistrements simulés.
2. Poussez la progression tous les 10 enregistrements, et non à chaque enregistrement.
3. Ajoutez des messages de journal pour le démarrage, tous les 50 enregistrements, l'annulation, l'échec et la fin.
4. Implémentez l'annulation avec `CancellationTokenSource`.
5. Ajoutez un bouton ou une option d'échec volontaire qui lève une exception à l'enregistrement 87.
6. Assurez-vous que l'interface est réactivée quelle que soit l'issue.

Boucle d'enregistrements suggérée

```
private void RunImport(CancellationToken token, bool failAt87)
{
    try
    {
        for (int i = 1; i ≤ 200; i++)
        {
            token.ThrowIfCancellationRequested();
            Thread.Sleep(25);

            if (failAt87 && i == 87)
                throw new InvalidOperationException("Simulated malformed record.");
        }
    }
}
```

```

        recordsImportedLabel.Text = i.ToString();
        importProgressBar.Value = i / 2;

        if (i % 50 == 0)
            importLogListBox.Items.Insert(0, $"Imported {i} records");

        if (i % 10 == 0)
            Application.Update(this);
    }

    importStatusLabel.Text = "Import completed successfully.";
}
catch (OperationCanceledException)
{
    importStatusLabel.Text = "Import cancelled by user.";
    importLogListBox.Items.Insert(0, "Cancelled at " + DateTime.Now);
}
catch (Exception ex)
{
    LogError(ex);
    importStatusLabel.Text = "Import failed. Review the server log.";
    importLogListBox.Items.Insert(0, "Failed at " + DateTime.Now);
}
finally
{
    startImportButton.Enabled = true;
    cancelImportButton.Enabled = false;
    Application.Update(this);
}
}

```

Critères d'acceptation

- ✓ Les mises à jour de progression apparaissent pendant l'exécution de l'opération.
- ✓ L'annulation fonctionne et laisse l'interface utilisable.
- ✓ L'échec est signalé sans exposer de stack traces brutes.
- ✓ La mise à jour finale atteint toujours le navigateur tant que la session est active.
- ✓ Le code ne pousse pas plus que nécessaire.

Pour aller plus loin

Ajoutez un `JobId` à chaque import et incluez-le dans les messages de journal. Le projet de synthèse s'en servira pour relier les événements de l'interface aux journaux du serveur.

Timers, polling de repli et cadence des mises à jour

Cadence de mise à jour et repli : choisir entre push, timers et polling, lisser les rafales et maîtriser le cycle de vie des timers pour chaque session. Le concept, le modèle mental et les pièges en production — à lire avant le lab.



Objectif production

Cadence de mise à jour et repli : choisir entre push, timers et polling, lisser les rafales et maîtriser le cycle de vie des timers pour chaque session.

Le push est préférable, mais le repli compte

Une application en production ne peut pas supposer que chaque navigateur, proxy, serveur ou chemin réseau se comportera parfaitement. Certains environnements bloquent les WebSockets. Certains reverse proxies exigent une prise en charge explicite de WebSocket. Certains réseaux d'entreprise introduisent des timeouts. Wisej.NET prend en charge le push temps réel via WebSocket et propose aussi des solutions de repli, comme le polling, pour les systèmes qui ne peuvent pas utiliser WebSocket.

Le développeur de niveau intermédiaire doit faire la différence entre un repli et une erreur de conception. Le polling n'est pas une erreur en soi. Il le devient lorsqu'il remplace un modèle d'événements clair ou lorsque l'intervalle est choisi sans tenir compte du coût pour le serveur. Un intervalle de polling d'une seconde pour des milliers de sessions peut coûter cher. Un intervalle de dix secondes pour un tableau de bord d'exploitation critique peut paraître lent. Le bon choix dépend de la valeur pour l'utilisateur et du coût.

Wisej.Web.Timer ou tâche en arrière-plan

Un `Wisej.Web.Timer` est utile lorsque vous voulez des événements périodiques côté serveur associés à une session. Par exemple, un tableau de bord peut recalculer un KPI toutes les cinq secondes, même si aucun événement de service global n'est arrivé. Une tâche en arrière-plan convient mieux lorsqu'une opération précise démarre, s'exécute et se termine. Un service global convient mieux lorsqu'un seul watcher partagé alimente de nombreuses sessions.

Utilisez un timer pour un rafraîchissement périodique de l'interface limité à la session. Utilisez `Application.StartTask()` pour un travail de longue durée limité à la session. Utilisez un service partagé pour l'état global et les événements multi-utilisateurs.

Polling de repli avec StartPolling et EndPolling

Lorsque WebSocket est disponible, `Application.Update()` peut pousser les mises à jour en attente vers le navigateur. Lorsque WebSocket n'est pas disponible, il n'a aucune connexion active à utiliser. Wisej.NET fournit `Application.StartPolling(milliseconds)` et `Application.EndPolling()` comme mécanismes de repli. Le polling est ignoré lorsque WebSocket est actif, ce qui permet à l'application de demander un comportement de repli sans pénaliser les sessions WebSocket.

Un pattern courant est le suivant :

```
private void BeginLiveMode()
{
    Application.StartPolling(1000);

    Application.StartTask(() =>
    {
        while (_liveMode && !this.IsDisposed)
        {
            UpdateDashboardModel();
            ApplyDashboardToControls();

            // Effective when WebSocket exists. Polling handles non-WebSocket cases.
            Application.Update(this);
            Thread.Sleep(1000);
        }

        Application.EndPolling();
    });
}
```

Le code doit malgré tout être conçu avec soin. Si `_liveMode` ne passe jamais à false et que la page est libérée, la boucle devient une fuite. Prévoyez toujours une condition d'arrêt.

Choisir la cadence de mise à jour

La cadence est le rythme des mises à jour. Dans une application temps réel Wisej.NET, elle a quatre dimensions :

Cadence du modèle : la fréquence à laquelle les données sous-jacentes changent.

Cadence de l'interface : la fréquence à laquelle l'utilisateur a besoin de voir des changements visibles.

Cadence de push : la fréquence à laquelle le serveur envoie des mises à jour d'interface au navigateur.

Cadence de polling : la fréquence à laquelle le navigateur demande des mises à jour au serveur lorsque le push n'est pas disponible.

Ces cadences ne sont pas toujours identiques. Une machine peut produire de la télémétrie 100 fois par seconde alors que l'interface n'a besoin que d'une synthèse par seconde. Une file d'attente peut changer 20 fois lors d'une rafale, mais le tableau de bord peut afficher l'état final après 250 millisecondes.

Lisser les rafales

Une rafale se produit lorsque de nombreux changements arrivent en même temps. Si chaque changement provoque un push distinct, l'interface peut gaspiller des ressources et afficher des états intermédiaires incompréhensibles pour les utilisateurs. Lisser (coalescing) consiste à collecter plusieurs changements et à les appliquer ensemble.

Stratégie de lissage simple :

```

private readonly object _sync = new object();
private bool _dirty;

private void OnModelChanged()
{
    lock (_sync)
    {
        _dirty = true;
    }
}

private void refreshTimer_Tick(object sender, EventArgs e)
{
    bool shouldRefresh;

    lock (_sync)
    {
        shouldRefresh = _dirty;
        _dirty = false;
    }

    if (!shouldRefresh)
        return;

    ApplyModelToControls();
}

```

L'événement marque le modèle comme modifié. Le timer applique les mises à jour à une cadence d'interface stable.

À qui appartient le timer

Un timer appartient forcément à quelqu'un. S'il s'agit d'un composant Wisej.NET placé sur une page, la page le possède. S'il s'agit d'un timer .NET dans un service global, le service le possède. La propriété détermine le nettoyage. Un timer dupliqué peut doubler le trafic de mise à jour et donner l'impression que les bugs sont aléatoires.

Règles pratiques :

- Démarrez les timers une seule fois.
- Arrêtez les timers lorsque la vue n'est plus active.
- Ne créez pas un nouveau timer à chaque clic sur le bouton d'actualisation.
- Évitez les traitements longs dans un tick de timer ; lancez ou planifiez plutôt le travail.
- Utilisez un indicateur pour empêcher les ticks de se chevaucher si un rafraîchissement dure plus longtemps que l'intervalle.

Détecter une mauvaise cadence

Les symptômes d'un excès de mises à jour sont notamment un CPU élevé, un rendu saccadé dans le navigateur, un trafic WebSocket important, des journaux serveur bruyants et des tableaux de bord qui

semblent clignoter. Les symptômes d'un manque de mises à jour sont des utilisateurs qui appuient sur Actualiser, du travail en double, des décisions opérationnelles fondées sur des données périmées et une perte de confiance dans le tableau de bord.

Les développeurs de niveau intermédiaire doivent instrumenter les tableaux de bord importants. Comptez les événements reçus, les mises à jour d'interface appliquées, les push tentés, les erreurs et le temps moyen passé à appliquer une mise à jour.



Point d'étape

À la fin de ce module, l'apprenant doit être capable de choisir entre un timer, une tâche en arrière-plan, un polling de repli ou un événement de service en fonction du travail à accomplir, et d'expliquer le coût de la cadence retenue.

Timers, polling de repli et cadence des mises à jour

Ce que vous allez construire dans le lab pratique, les tâches, l'implémentation suggérée et les critères d'acceptation.



Lab pratique

Enrichissez TicketOps Live d'un timer de rafraîchissement du tableau de bord et de contrôles permettant d'expérimenter la cadence de mise à jour.

Exigences de l'interface

Ajoutez :

- ✓ `liveModeCheckBox`
- ✓ `cadenceComboBox` avec les valeurs `250 ms`, `1 sec`, `5 sec`
- ✓ `eventsReceivedLabel`
- ✓ `updatesAppliedLabel`
- ✓ `lastAppliedLabel`
- ✓ `refreshTimer`

Tâches

1. Démarrez le polling lorsque le mode direct est activé.
2. Arrêtez le polling lorsque le mode direct est désactivé.
3. Utilisez un timer pour appliquer les mises à jour du tableau de bord à la cadence sélectionnée.
4. Simulez des changements du modèle plus rapides que les mises à jour de l'interface.
5. Comptez séparément les événements du modèle et les mises à jour d'interface appliquées.
6. Empêchez les ticks du timer de se chevaucher.

Piste d'implémentation suggérée

```
private int _eventsReceived;
private int _updatesApplied;
private bool _refreshInProgress;
private volatile bool _dashboardDirty;

private void liveModeCheckBox_CheckedChanged(object sender, EventArgs e)
{
    if (liveModeCheckBox.Checked)
    {
        Application.StartPolling(1000);
        refreshTimer.Interval = GetSelectedInterval();
        refreshTimer.Start();
    }
    else
```

```

    {
        refreshTimer.Stop();
        Application.EndPolling();
    }
}

private void SimulatedModelChanged()
{
    _eventsReceived++;
    _dashboardDirty = true;
}

private void refreshTimer_Tick(object sender, EventArgs e)
{
    if (_refreshInProgress || !_dashboardDirty)
        return;

    try
    {
        _refreshInProgress = true;
        _dashboardDirty = false;
        _updatesApplied++;

        eventsReceivedLabel.Text = _eventsReceived.ToString();
        updatesAppliedLabel.Text = _updatesApplied.ToString();
        lastAppliedLabel.Text = DateTime.Now.ToString("HH:mm:ss.fff");
    }
    finally
    {
        _refreshInProgress = false;
    }
}

```

Critères d'acceptation

- ✓ La cadence du timer change de façon visible.
- ✓ Le nombre d'événements reçus peut augmenter plus vite que le nombre de mises à jour d'interface appliquées.
- ✓ Le polling démarre et s'arrête avec le mode direct.
- ✓ Aucun tick de timer ne se chevauche.
- ✓ L'apprenant sait expliquer pourquoi le lissage des mises à jour est utile.

Liaison de données en direct et grilles temps réel

Liaison de données en direct : mettre à jour les collections liées dans le contexte de la session sans reliaer à nouveau la grille, en préservant la sélection, le tri et le défilement. Le concept, le modèle mental et les pièges en production — à lire avant le lab.



Objectif production

Liaison de données en direct : mettre à jour les collections liées dans le contexte de la session sans reliaer à nouveau la grille, en préservant la sélection, le tri et le défilement.

Pourquoi la liaison de données se complique quand les données bougent

La liaison de données statique est simple : charger les enregistrements, lier la grille, laisser l'utilisateur modifier, enregistrer. La liaison de données en temps réel ajoute du mouvement. Des lignes arrivent, des champs changent, des statuts évoluent, et des tâches en arrière-plan mettent à jour l'écran que l'utilisateur est en train de consulter. Le défi consiste à garder l'interface synchronisée sans détruire et recréer sans cesse les bindings.

Wisej.NET prend en charge la liaison simple pour les propriétés d'un contrôle et la liaison complexe pour des contrôles comme les grilles. Le composant `BindingSource` est particulièrement utile, car il se place entre les contrôles et les sources de données, gère l'élément courant et offre un point unique pour réinitialiser ou rafraîchir les bindings.

Le modèle Ticket

Utilisez un modèle de ticket simple dans ce cours :

```
public enum TicketStatus
{
    New,
    Assigned,
    Waiting,
    Resolved,
    Escalated
}

public class Ticket : INotifyPropertyChanged
{
    private TicketStatus _status;
    private string _owner;
    private DateTime _updatedAt;

    public int Id { get; set; }
    public string Title { get; set; }
    public string Customer { get; set; }

    public string Owner
    {
```

```

        get => _owner;
        set { _owner = value; OnPropertyChanged(nameof(Owner)); }
    }

    public TicketStatus Status
    {
        get => _status;
        set { _status = value; OnPropertyChanged(nameof(Status)); }
    }

    public DateTime UpdatedAt
    {
        get => _updatedAt;
        set { _updatedAt = value; OnPropertyChanged(nameof(UpdatedAt)); }
    }

    public event PropertyChangedEventHandler PropertyChanged;

    protected void OnPropertyChanged(string propertyName)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));
    }
}

```

L'intérêt de `INotifyPropertyChanged` est de rendre les modifications de propriétés visibles par l'infrastructure de binding lorsqu'elle le prend en charge. Même lorsque vous appelez `BindingSource.ResetBindings(false)` par souci de clarté, le modèle indique toujours qu'il peut notifier ses changements.

Lier une liste en direct

Un point de départ pratique consiste à combiner `BindingList<Ticket>` et `BindingSource`.

```

private BindingList<Ticket> _tickets;
private BindingSource _ticketSource;

private void MainPage_Load(object sender, EventArgs e)
{
    _tickets = new BindingList<Ticket>();
    _ticketSource = new BindingSource();
    _ticketSource.DataSource = _tickets;

    ticketsGrid.AutoGenerateColumns = true;
    ticketsGrid.DataSource = _ticketSource;
}

```

Lorsqu'un nouveau ticket arrive, ajoutez-le à la liste dans le bon contexte de session. Lorsqu'un champ change, mettez à jour l'élément et réinitialisez les bindings si l'interface ne se rafraîchit pas automatiquement.

```
private void AddTicket(Ticket ticket)
{
    _tickets.Insert(0, ticket);
    _ticketSource.ResetBindings(false);
}

private void UpdateTicket(int id, TicketStatus status, string owner)
{
    var ticket = _tickets.FirstOrDefault(t => t.Id == id);
    if (ticket == null)
        return;

    ticket.Status = status;
    ticket.Owner = owner;
    ticket.UpdatedAt = DateTime.Now;
    _ticketSource.ResetBindings(false);
}
```

Ne reliez pas la grille à chaque mise à jour

Relier toute la grille est parfois le moyen le plus rapide de faire fonctionner une démo, mais c'est souvent une erreur en production. Cela peut réinitialiser la sélection, la position de défilement, le tri, la largeur des colonnes et le contexte de l'utilisateur. Cela peut aussi coûter cher. Préférez des modifications incrémentales de la liste liée, puis notifiez la binding source.

À éviter :

```
ticketsGrid.DataSource = null;
ticketsGrid.DataSource = LoadAllTicketsAgain();
```

Mieux :

```
ApplyTicketChange(change);
_ticketSource.ResetBindings(false);
```

Idéal pour les écrans à fort volume : regroupez plusieurs changements et rafraîchissez une seule fois par intervalle de cadence.

Thread-safety et collections liées

Une collection liée appartient à l'interface de la session. Un service global peut recevoir des mises à jour depuis un autre thread. Ne modifiez pas la collection liée depuis n'importe quel thread sans entrer dans le contexte de la session et sans réfléchir à la synchronisation.

Un pattern multi-utilisateur sûr est le suivant :

1. Le service global reçoit un événement de ticket.
2. Le gestionnaire d'événements de la session capte l'événement.
3. Le gestionnaire utilise `Application.Update(context, () => { ... })`.

4. Dans le callback, il met à jour la liste liée de la session.
5. Il réinitialise les bindings une seule fois.

```
private void Feed_TicketChanged(object sender, TicketChangedEventArgs e)
{
    Application.Update(_context, () =>
    {
        ApplyTicketChange(e.Change);
        _ticketSource.ResetBindings(false);
    });
}
```

Formatage et analyse

Les grilles temps réel ne se limitent pas au transport des données : elles doivent rester lisibles. Formatez les dates, les montants, les priorités et les statuts. Utilisez les événements Parse/Format des contrôles liés lorsque les valeurs nécessitent une conversion. Évitez de stocker des chaînes formatées dans votre modèle si la valeur réelle a un type plus précis.

Par exemple, `UpdatedAt` doit être un `DateTime`, et non une chaîne préformatée. La grille peut l'afficher en heure locale ou sous forme d'ancienneté relative.

État des lignes et repères visuels

Une grille en direct doit montrer ce qui a changé sans devenir envahissante. Patterns courants :

- afficher un marqueur temporaire « mis à jour » pendant quelques secondes ;
- trier les nouveaux tickets en haut ;
- garder stable le ticket sélectionné par l'utilisateur lorsque c'est possible ;
- éviter de déplacer une ligne pendant que l'utilisateur la modifie ;
- utiliser des colonnes de statut et de petits indicateurs plutôt que des interruptions modales.

L'interface doit aider les utilisateurs à faire confiance aux données. Trop d'animations ou de sauts de lignes peuvent leur faire croire que l'application est instable.

Validation et gestion des conflits

Des données en temps réel signifient qu'un autre utilisateur ou un service peut modifier un enregistrement pendant que l'utilisateur courant le consulte ou le modifie. Décidez de ce qui se passe lorsqu'un ticket change en cours de modification. Dans ce cours, appliquez une règle simple : si un ticket est en cours de modification dans un formulaire de détail, affichez un avertissement non bloquant indiquant que le ticket a changé et laissez l'utilisateur recharger ou continuer.

Les systèmes en production peuvent nécessiter une concurrence optimiste avec des numéros de version, des horodatages ou des versions de ligne en base de données. Wisej.NET ne supprime pas le besoin de gérer la concurrence dans la couche de données : il donne à l'interface un moyen de prévenir les utilisateurs au bon moment.

**Point d'étape**

À la fin de ce module, l'apprenant doit être capable de maintenir une grille en direct qui reçoit de nouvelles lignes et des mises à jour de lignes depuis une source en arrière-plan, tout en préservant le contexte de l'utilisateur et en évitant les rebindings inutiles.

Liaison de données en direct et grilles temps réel

Ce que vous allez construire dans le lab pratique, les tâches, l'implémentation suggérée et les critères d'acceptation.



Lab pratique

Créez une `DataGridView` en direct qui affiche les tickets de support et se met à jour à l'arrivée d'événements de ticket simulés.

Tâches

1. Ajoutez une `DataGridView` nommée `ticketsGrid`.
2. Créez le modèle `Ticket` et l'enum `TicketStatus`.
3. Liez une `BindingList<Ticket>` via une `BindingSource`.
4. Ajoutez un bouton qui crée un nouvel événement de ticket chaque seconde pendant 20 secondes.
5. Ajoutez un bouton qui modifie aléatoirement le statut des tickets.
6. Préservez la ligne sélectionnée lorsque c'est possible.
7. Affichez `UpdatedAt` dans un format lisible.

Événement de modèle suggéré

```
public class TicketChangedEventArgs : EventArgs
{
    public Ticket Ticket { get; set; }
    public string ChangeType { get; set; }
    public string Message { get; set; }
}
```

Méthode de mise à jour suggérée

```
private void ApplyTicketEvent(TicketChangedEventArgs e)
{
    var selectedId = ticketsGrid.CurrentRow?.DataBoundItem is Ticket selected
        ? selected.Id
        : (int?)null;

    var existing = _tickets.FirstOrDefault(t => t.Id == e.Ticket.Id);

    if (existing == null)
        _tickets.Insert(0, e.Ticket);
    else
    {
        existing.Status = e.Ticket.Status;
        existing.Owner = e.Ticket.Owner;
        existing.UpdatedAt = DateTime.Now;
    }

    _ticketSource.ResetBindings(false);

    if (selectedId.HasValue)
        RestoreSelection(selectedId.Value);
}
```

Critères d'acceptation

- ✓ Les nouveaux tickets apparaissent sans relia la grille entièrement.
- ✓ Les tickets existants sont mis à jour sur place.
- ✓ La ligne sélectionnée est préservée tant qu'elle existe encore.
- ✓ Le rythme des mises à jour reste maîtrisé.
- ✓ L'apprenant sait expliquer pourquoi la liste liée appartient à la session.

Pour aller plus loin

Ajoutez un filtre sur les tickets `Escalated` et vérifiez que les mises à jour en direct fonctionnent toujours lorsque le filtre est actif.

Push multi-utilisateurs avec services et hubs d'événements

Push multi-utilisateur : un hub d'événements global lève des événements métier tandis que chaque session met à jour sa propre interface dans son propre contexte. Le concept, le modèle mental et les pièges en production — à lire avant le lab.



Objectif production

Push multi-utilisateur : un hub d'événements global lève des événements métier tandis que chaque session met à jour sa propre interface dans son propre contexte.

D'une session à plusieurs sessions

Le push mono-session est utile, mais de nombreuses applications métier ont besoin d'un push multi-utilisateur. Lorsqu'un répartiteur affecte un ticket, chaque tableau de bord de superviseur doit se mettre à jour. Lorsqu'un import nocturne se termine, tous les administrateurs connectés doivent voir le résultat. Lorsqu'un contrôle de santé d'un service échoue, les consoles d'exploitation actives doivent afficher l'alerte.

Le changement d'architecture essentiel porte sur la propriété. Une page possède ses contrôles. Une session possède son contexte. Un service global possède les données et les événements partagés. Ne laissez pas le service global devenir un fourre-tout de références à des formulaires. Utilisez plutôt des événements métier et des abonnements au niveau de la session.

Le pattern du hub d'événements

Un hub d'événements est un service partagé qui reçoit ou crée des événements métier et les expose à ses abonnés. Le hub ne sait pas comment chaque interface affiche l'événement : il annonce simplement que quelque chose s'est produit.

```
public sealed class TicketHub
{
    private static readonly Lazy<TicketHub> _instance =
        new Lazy<TicketHub>(() => new TicketHub());

    public static TicketHub Instance => _instance.Value;

    public event EventHandler<TicketChangedEventArgs> TicketChanged;

    private readonly object _sync = new object();
    private readonly List<Ticket> _tickets = new List<Ticket>();

    private TicketHub() { }

    public IReadOnlyList<Ticket> GetSnapshot()
    {
        lock (_sync)
            return _tickets.Select(CloneTicket).ToList();
    }
}
```

```

public void AddOrUpdate(Ticket ticket)
{
    TicketChangedEventArgs args;

    lock (_sync)
    {
        var existing = _tickets.FirstOrDefault(t => t.Id == ticket.Id);
        if (existing == null)
            _tickets.Insert(0, ticket);
        else
            CopyTicket(ticket, existing);

        args = new TicketChangedEventArgs
        {
            Ticket = CloneTicket(ticket),
            ChangeType = existing == null ? "Added" : "Updated",
            Message = $"Ticket {ticket.Id} {ticket.Status}"
        };
    }

    TicketChanged?.Invoke(this, args);
}
}

```

Le hub garde l'état global thread-safe et lève un événement métier. Il n'appelle pas `Application.Update()`, car il ne possède aucune session.

Pattern de l'abonné de session

Chaque session s'abonne au hub et met à jour sa propre interface.

```

private readonly TicketHub _hub = TicketHub.Instance;
private IWisejComponent _context;

private void MainPage_Load(object sender, EventArgs e)
{
    _context = Application.Current;
    LoadSnapshot(_hub.GetSnapshot());
    _hub.TicketChanged += Hub_TicketChanged;
    Application.ApplicationExit += Application_ApplicationExit;
}

private void Hub_TicketChanged(object sender, TicketChangedEventArgs e)
{
    if (this.IsDisposed)
        return;

    Application.Update(_context, () =>
    {
        ApplyTicketEvent(e);
        notificationsList.Items.Insert(0, e.Message);
    });
}

private void Application_ApplicationExit(object sender, EventArgs e)
{
    _hub.TicketChanged -= Hub_TicketChanged;
}

```

Ce pattern donne à chaque session sa propre interface. Deux utilisateurs peuvent filtrer ou trier différemment tout en recevant les mêmes événements métier.

Diffusion ou mise à jour ciblée

La diffusion (broadcast) signifie que chaque abonné reçoit l'événement. La mise à jour ciblée signifie que seules certaines sessions le reçoivent. Exemples :

- Diffuser toutes les alertes système graves à toutes les consoles d'exploitation.
- N'envoyer les événements propres à un tenant qu'aux sessions de ce tenant.
- N'envoyer les notifications d'affectation qu'à l'opérateur affecté.
- N'envoyer les événements d'administration qu'aux utilisateurs ayant un rôle d'administrateur.

Le hub peut inclure suffisamment de métadonnées pour que l'abonné décide s'il doit afficher l'événement. Le hub peut aussi gérer des groupes d'abonnement. Gardez une première version simple : incluez les métadonnées de tenant, d'utilisateur et de rôle, et laissez la session filtrer.

Thread-safety et snapshots

Les services globaux maintiennent souvent des collections. Ces collections doivent être protégées par des verrous ou être des collections concurrentes. Pour exposer des données aux sessions, préférez des

snapshots à des références directes aux collections. Un snapshot évite que le code de l'interface modifie accidentellement l'état global et évite de parcourir une collection pendant que le service la modifie.

À éviter :

```
public List<Ticket> Tickets => _tickets;
```

Mieux :

```
public IReadOnlyList<Ticket> GetSnapshot()
{
    lock (_sync)
        return _tickets.Select(CloneTicket).ToList();
}
```

Le clonage n'a pas à être coûteux dans tous les cas, mais la frontière de propriété doit être claire.

Fuites mémoire via les événements

Les événements sont une source fréquente de fuites mémoire. Si un service global expose un événement et qu'une page s'y abonne, le service détient désormais une référence au gestionnaire de la page. Si la page est fermée sans jamais se désabonner, le service peut la garder en vie. Dans les applications temps réel, cette fuite est à la fois un problème de mémoire et un problème de justesse, car le service peut tenter de mettre à jour une session morte.

Chaque abonnement doit avoir son désabonnement. C'est une règle de ce cours.

Contre-pression

Si le hub émet des changements plus vite que les sessions ne peuvent les afficher, le système a besoin de contre-pression (backpressure) ou de lissage. Quelques stratégies simples :

- ignorer les événements dupliqués de faible priorité ;
- regrouper les changements et envoyer des synthèses ;
- marquer une session comme modifiée et rafraîchir via un timer ;
- ne conserver que le dernier état d'un indicateur du tableau de bord ;
- mettre les événements en file par session avec une taille maximale.

Les applications métier n'ont généralement pas besoin d'afficher chaque état intermédiaire. Elles doivent afficher l'état sur lequel les utilisateurs peuvent agir.



Point d'étape

À la fin de ce module, l'apprenant doit être capable de concevoir un service partagé qui alimente en toute sécurité de nombreuses sessions Wisej.NET sans stocker globalement les contrôles des pages, sans fuite de gestionnaires d'événements et sans modifier les collections liées à l'interface depuis le mauvais thread.

Push multi-utilisateurs avec services et hubs d'événements

Ce que vous allez construire dans le lab pratique, les tâches, l'implémentation suggérée et les critères d'acceptation.



Lab pratique

Refactorisez la simulation de tickets en un service global `TicketHub` et faites en sorte que plusieurs sessions de navigateur reçoivent les événements de ticket en direct.

Tâches

1. Créez un singleton `TicketHub` avec une liste de tickets thread-safe.
2. Ajoutez les événements `TicketChanged`.
3. Chargez un snapshot au démarrage de `MainPage`.
4. Abonnez la page à `TicketChanged`.
5. Désabonnez-la à la sortie de l'application.
6. Ouvrez deux navigateurs ou deux sessions et vérifiez que les deux reçoivent les mises à jour.
7. Ajoutez des métadonnées de tenant ou de rôle et filtrez au moins un type d'événement.

Critères d'acceptation

- ✓ Le hub ne stocke aucune référence à des pages ou à des contrôles.
- ✓ Chaque session met à jour sa propre liste liée.
- ✓ Fermer ou actualiser une session ne laisse pas d'abonné défaillant derrière elle.
- ✓ Plusieurs sessions peuvent afficher des filtres différents tout en partageant la même source d'événements métier.
- ✓ L'apprenant sait expliquer la différence entre diffusion et mise à jour ciblée.

Pour aller plus loin

Ajoutez un compteur de notifications par session. Le compteur doit être propre à la session, même si les événements proviennent d'un service global.

Durcissement pour la production, déploiement et projet de synthèse

Durcissement pour la production : vérifier tout le chemin WebSocket, les sticky sessions, les contrôles de santé, la sécurité et l'observabilité avant le déploiement. Le concept, le modèle mental et les pièges en production — à lire avant le lab.



Objectif production

Durcissement pour la production : vérifier tout le chemin WebSocket, les sticky sessions, les contrôles de santé, la sécurité et l'observabilité avant le déploiement.

Le temps réel en production, c'est de l'architecture et de l'exploitation

Une application temps réel peut fonctionner parfaitement sur le poste d'un développeur et échouer en production, parce que le chemin de déploiement est différent. WebSocket peut être bloqué par un proxy. Un répartiteur de charge peut acheminer les requêtes d'un utilisateur vers différentes instances de serveur. Une expiration de session peut interrompre un travail d'interface de longue durée. Un service global peut consommer trop de mémoire. Un paramètre de sécurité peut être sûr en développement mais inadapté derrière un reverse proxy.

Ce module transforme les patterns du cours en checklist de production.

Un déploiement compatible WebSocket

Le push temps réel de Wisej.NET fonctionne au mieux lorsque WebSocket est disponible. En production, vérifiez la prise en charge de WebSocket sur l'ensemble du chemin : navigateur, réseau, reverse proxy, répartiteur de charge, serveur web et application. Certains répartiteurs de charge exigent un routage TCP ou une prise en charge explicite de WebSocket. Les reverse proxies comme NGINX peuvent relayer WebSocket, à condition d'être correctement configurés.

Tester uniquement depuis la console du serveur ne suffit pas. Testez depuis le chemin réseau de l'utilisateur. Un VPN d'entreprise, un équipement de sécurité ou un proxy ancien peut modifier le comportement.

Répartition de charge et affinité de session

Les applications Wisej.NET conservent l'état de session côté serveur. Dans un déploiement multiserveur, les requêtes d'une session donnée doivent toujours atteindre la même instance de serveur, sauf si l'architecture prend explicitement en charge une autre stratégie. La règle pratique de répartition de charge consiste à utiliser des sticky sessions ou un autre mécanisme de routage qui garantit la cohérence entre utilisateur et serveur.

Les contrôles de santé doivent indiquer au répartiteur de charge quand un serveur doit cesser de recevoir du trafic. Wisej.NET prend en charge la configuration des contrôles de santé via un fichier `HealthCheck.json`, avec des paramètres comme la mémoire maximale, le nombre maximal de sessions et le seuil CPU. Les seuils exacts dépendent du comportement de l'application et de la taille de l'hébergement.

Zones de configuration importantes

La configuration relève à la fois de `Web.config` et des fichiers de configuration JSON de Wisej.NET. Les applications Wisej.NET utilisent des fichiers d'application JSON comme `Default.json`, et un projet peut définir plusieurs applications Wisej.NET avec leurs propres fichiers de configuration.

Les principaux paramètres temps réel et points d'attention du déploiement sont :

- `sessionTimeout` : détermine quand les sessions inactives expirent ;
- `pollingInterval` : contrôle le polling automatique pour les clients sans WebSocket ;
- `responseTimeout` : compte pour les longues réponses HTTP initiales, mais WebSocket n'expire pas de la même manière ;
- `debug=false` en production, afin que les bibliothèques JavaScript soient correctement regroupées et minifiées ;
- la clé de licence serveur et le thème par défaut ;
- les paramètres de module/handler IIS pour les applications .NET Framework ;
- le modèle d'hébergement et le middleware ASP.NET Core pour les applications .NET Core/.NET ;
- les paramètres des contrôles de santé pour les déploiements avec répartition de charge.

Exemple de fragment `Default.json` pour une application de formation :

```
{
  "mainWindow": "TicketOpsLive.MainPage, TicketOpsLive",
  "theme": "Bootstrap-4",
  "sessionTimeout": 180,
  "pollingInterval": 1000
}
```

Ne recopiez pas aveuglément les valeurs de formation en production. Appuyez-vous sur des tests de charge et sur les exigences des utilisateurs pour choisir les paramètres.

Sécurité des applications temps réel

Wisej.NET évite de nombreux problèmes classiques de concaténation de chaînes HTML, car il synchronise des contrôles serveur et des widgets du navigateur au lieu de construire des pages à partir de chaînes HTML arbitraires. Les développeurs peuvent néanmoins créer un risque XSS s'ils affichent en HTML du contenu fourni par les utilisateurs. Les contrôles et les messages qui acceptent du HTML doivent être traités avec soin. Gardez encodés les titres de tickets, commentaires, noms et messages saisis par les utilisateurs, sauf s'il existe un processus de nettoyage clairement défini.

Derrière un reverse proxy ou un répartiteur de charge, la gestion de l'adresse IP du client doit aussi être réfléchie. Les en-têtes transférés (forwarded headers) ne doivent être considérés comme fiables que si l'application se trouve derrière un proxy de confiance qui nettoie les en-têtes entrants. Sinon, un attaquant peut falsifier leurs valeurs.

Les applications temps réel doivent aussi protéger les ressources du serveur. Fixez des limites pour les sessions, les files d'attente et le débit d'événements. Ne laissez pas un utilisateur créer un nombre illimité de tâches en arrière-plan ou d'abonnements.

Observabilité

Une application temps réel a besoin de journaux et de métriques qui expliquent ce qui s'est passé après la fin de la requête d'origine de l'utilisateur. Incluez des champs de corrélation :

- identifiant de la tâche ;
- identifiant de l'utilisateur ou marqueur de session anonyme ;
- identifiant client/identifiant de session le cas échéant ;
- nom de l'opération ;
- type d'événement ;
- instance de serveur ;
- durée et résultat ;
- nombre de mises à jour d'interface poussées ou appliquées.

Dans TicketOps Live, chaque tâche d'import doit avoir un identifiant de tâche et chaque événement du hub un identifiant d'événement. L'interface peut afficher un état simplifié, mais les journaux doivent permettre au support de reconstituer la séquence.

Checklist de performance

Avant le déploiement, passez en revue ces questions :

1. Chaque boucle en arrière-plan a-t-elle une condition d'arrêt ?
2. Chaque abonnement à un événement global se désabonne-t-il ?
3. Les mises à jour de l'interface sont-elles regroupées ou limitées ?
4. Les événements de modèle à haute fréquence sont-ils lissés ?
5. Les collections liées sont-elles mises à jour dans le contexte de la session ?
6. Les champs statiques sont-ils réservés à l'état global ?
7. Les compteurs, filtres et sélections propres à la session sont-ils bien par session ?
8. Les exceptions sont-elles interceptées et journalisées dans les tâches en arrière-plan ?
9. L'application se comporte-t-elle de façon acceptable lorsque WebSocket n'est pas disponible ?
10. Le répartiteur de charge maintient-il une session sur la même instance de serveur ?

Présentation du projet de synthèse

Le projet de synthèse consiste en une revue de production et en l'achèvement de TicketOps Live. Les apprenants doivent démontrer :

- les mises à jour d'état en direct ;
- la progression de l'import en arrière-plan avec annulation et gestion des erreurs ;
- les mises à jour en direct de la grille de tickets ;
- les notifications de tickets multi-utilisateurs ;
- la stratégie de polling de repli ;
- le nettoyage des sessions ;

- la checklist de déploiement et d'observabilité.

Le projet de synthèse n'est pas évalué au nombre de contrôles à l'écran, mais sur l'architecture : propriété, contexte, cycle de vie, cadence de mise à jour et préparation à la production.



Point d'étape

À la fin de ce module, l'apprenant doit être capable d'expliquer comment une application temps réel Wisej.NET passe d'une démo à une application déployable : chemin WebSocket vérifié, routage des sessions stable, repli défini, abonnements nettoyés, mises à jour limitées, données protégées et exploitation observable.

Durcissement pour la production, déploiement et projet de synthèse

Ce que vous allez construire dans le lab pratique, les tâches, l'implémentation suggérée et les critères d'acceptation.



Lab pratique

Terminez TicketOps Live et préparez-le pour une revue d'architecture de production.

Fonctionnalités requises

1. Bandeau d'état en direct du M01.
2. Diagnostic de session du M02.
3. Moniteur d'import en arrière-plan du M03.
4. Contrôles de cadence et de polling du M04.
5. Tableau de tickets en direct du M05.
6. **TicketHub** multi-utilisateur du M06.
7. Checklist de production et revue de configuration du M07.

Livrable d'architecture

Rédigez une note d'architecture d'une page qui précise :

- ✓ quel état est propre à la session ;
- ✓ quel état est global ;
- ✓ où démarre le traitement en arrière-plan ;
- ✓ comment les mises à jour de l'interface atteignent le navigateur ;
- ✓ comment fonctionne l'annulation ;
- ✓ comment les abonnements sont retirés ;
- ✓ ce qui se passe lorsque WebSocket n'est pas disponible ;
- ✓ ce que l'environnement de déploiement doit prendre en charge.

Scénario de démonstration

1. Ouvrez TicketOps Live dans deux sessions de navigateur.
2. Démarrez le heartbeat.
3. Lancez un import et montrez la progression.
4. Annulez un import.
5. Lancez un autre import et provoquez un échec simulé.
6. Générez de nouveaux événements de ticket et montrez que les deux sessions se mettent à jour.
7. Appliquez un filtre dans une session et montrez que l'autre session est indépendante.
8. Désactivez le mode direct et expliquez le comportement du polling et de la cadence.

9. Présentez la checklist de production.

Critères d'acceptation

- ✓ L'application s'exécute sans boucles ni abonnements en double.
- ✓ L'interface reste réactive pendant le traitement en arrière-plan.
- ✓ Chaque opération de longue durée gère les états de fin, d'annulation et d'échec.
- ✓ Le hub multi-utilisateur ne conserve aucune référence directe à des pages ou à des contrôles.
- ✓ Les grilles liées se mettent à jour de façon incrémentale.
- ✓ L'apprenant sait justifier toutes les hypothèses de déploiement.