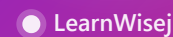


Architecture : EF Core dans une application Wisej.NET

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



Guide de la leçon

Où se place EF Core dans une application Wisej.NET avec état : les durées de vie de la session, de l'interface, de la requête et de l'unité de travail, une DbContextFactory avec un Context par opération, la passerelle vers l'injection de dépendances de Microsoft et les règles de sécurité entre sessions. Ce que cela signifie pour un développeur Wisej.NET qui travaille avec de vraies bases de données — et comment aborder ce module.



Objectif production

Où se place EF Core dans une application Wisej.NET avec état : les durées de vie de la session, de l'interface, de la requête et de l'unité de travail, une DbContextFactory avec un Context par opération, la passerelle vers l'injection de dépendances de Microsoft et les règles de sécurité entre sessions.

1. Le problème en production

Une application Wisej.NET ressemble à WinForms parce que les formulaires, les pages, les contrôles, les composants `BindingSource` et les gestionnaires d'événements vivent tous sur le serveur, et cette familiarité invite à un raccourci dangereux : traiter un `DbContext` comme un objet serveur de longue durée de plus. EF Core s'exécute lui aussi côté serveur, mais ce n'est pas un conteneur d'état de l'interface ; c'est une courte unité de travail dont le change tracker n'est ni thread-safe ni fait pour être partagé. De nombreuses sessions de navigateur ouvriront la Support Desk Data Console en même temps ; la première compétence de ce cours consiste donc à séparer les durées de vie de l'interface de celle de la base de données, avant même de lier la moindre grille.

2. Concepts clés

Concept	Pourquoi c'est important
état de session	L'état serveur propre à chaque session de navigateur que Wisej.NET maintient en vie : pages ouvertes, ligne sélectionnée, filtres, messages de validation et modifications en attente, accessible via <code>Application.Session</code> et l'arbre des contrôles actifs.
durée de vie des objets d'interface	Le temps d'existence d'une <code>Page</code> , d'un <code>Form</code> ou d'une <code>BindingSource</code> , de sa construction jusqu'à sa fermeture par l'utilisateur ou la fin de la session ; souvent des heures, bien plus que n'importe quel appel à la base de données.
durée de vie de la requête/du thread	Un événement du navigateur ou une continuation d' <code>await</code> sur le serveur ; le thread qui a lancé un gestionnaire de clic n'est pas forcément celui qui le termine.
unité de travail	La durée de vie d'un <code>DbContext</code> : le créer, interroger ou modifier des entités, appeler <code>SaveChangesAsync</code> , le libérer ; quelques secondes, pas des sessions.
DbContextFactory	Le service <code>IDbContextFactory<T></code> enregistré par <code>AddDbContextFactory</code> , qui fournit un Context neuf et correctement configuré chaque fois qu'une opération en a besoin.
pont vers la DI de Microsoft	L'enregistrement de l' <code>IServiceProvider</code> d'ASP.NET Core auprès d' <code>Application.Services</code> , pour que les pages Wisej.NET puissent résoudre les mêmes services que ceux configurés au démarrage.
fuite entre sessions	Un objet modifiable statique ou partagé (un Context, un change tracker, une liste liée, une entité sélectionnée) que la session d'un utilisateur peut lire après que la session d'un autre utilisateur l'a modifié.

3. Le fil de la leçon

Quatre durées de vie sur un même serveur

Tout ce qui se trouve dans un processus Wisej.NET appartient à l'une de quatre durées de vie, et la plupart des bugs de liaison de données viennent d'un objet placé dans la mauvaise. L'état de session vit aussi longtemps que la session du navigateur : l'identité de l'utilisateur, le navigateur de tickets ouvert, les filtres de recherche en cours. La durée de vie des objets d'interface appartient à une `Page` ou à un `Form` particulier et aux composants qu'il possède, comme la `BindingSource` derrière la grille des tickets et la ligne sélectionnée ; un éditeur modal n'existe que tant que la boîte de dialogue est ouverte. La durée de vie de la requête correspond à un aller-retour : le navigateur signale un clic, le serveur exécute le gestionnaire, et tout `await` à l'intérieur peut reprendre sur un autre thread du pool, si bien qu'un gestionnaire ne peut pas supposer d'affinité de thread. La durée de vie de l'unité de travail est celle d'EF Core : un `DbContext` est créé, interroge ou modifie quelques entités, enregistre, puis est libéré.

Elles s'emboîtent mal. Une session survit à des milliers de requêtes, un formulaire survit à de nombreux appels à la base, et un Context ne devrait survivre à aucun des deux. Un Context conservé dans une session ou un formulaire suit chaque entité qu'il a jamais chargée, grossit jusqu'à la fin de la session et devient une seconde copie périmée de la base que l'utilisateur croit à jour. Deux opérations asynchrones sur ce même Context font lever une exception à EF Core, car un Context n'est pas thread-safe. Ce sont ces

modes de défaillance qui répondent à la première question du module : pourquoi un `DbContext` statique ou partagé est dangereux.

Enregistrer EF Core et relier les deux conteneurs

Wisej.NET possède son propre système de services et peut aussi travailler avec l' `IServiceProvider` de Microsoft. Pour EF Core, l'injection de dépendances de Microsoft est la place naturelle :

`AddDbContextFactory<SupportDeskContext>` (ou `AddPooledDbContextFactory` quand le coût de création des connexions compte) enregistre une fabrique qui connaît le fournisseur, la chaîne de connexion issue de la configuration et les diagnostics réservés au développement. Les services qui utilisent la fabrique, comme le `TicketQueryService` que le lab résout depuis une page, vont dans le même conteneur. Après `builder.Build()`, une seule ligne transmet le fournisseur de Microsoft à Wisej.NET pour que les conteneurs d'interface de premier niveau puissent résoudre ces services, avec `Application.Services.GetService<T>()` ou par injection de propriété avec l'attribut `[Inject]` sur une `Page` ou un `Form`.

```
var builder = WebApplication.CreateBuilder(args);
var cs = builder.Configuration.GetConnectionString("SupportDesk")
    ?? throw new InvalidOperationException("Missing connection string 'SupportDesk'.");

builder.Services.AddDbContextFactory<SupportDeskContext>(options =>
{
    options.UseSqlServer(cs, sql => sql.EnableRetryOnFailure());
    if (builder.Environment.IsDevelopment())
        options.EnableDetailedErrors(); // sensitive-data logging: local development only
});
builder.Services.AddTransient<TicketQueryService>();

var app = builder.Build();
Application.Services.AddService<IServiceProvider>(app.Services); // the Wisej.NET bridge
```

Une fois la passerelle en place, une seule règle s'applique : enregistrez chaque service applicatif via l'injection de dépendances de Microsoft et ne mélangez pas les conteneurs, sinon vous obtenez deux objets là où vous en attendiez un. La chaîne de connexion vient de la configuration, jamais d'une constante dans un écran, et `EnableSensitiveDataLogging` est un interrupteur de développement local qui ne doit jamais atteindre les journaux de production. La fabrique de conception du projet de données n'existe que pour que les outils `dotnet ef` puissent construire un Context sans démarrer l'application web.

Un Context par opération, par formulaire d'édition ou derrière un service

Le choix par défaut le plus sûr pour les écrans de données Wisej.NET est la fabrique avec un Context par opération. Une recherche dans la grille, un chargement de liste de référence, une commande d'enregistrement et une commande de suppression créent et libèrent chacune leur propre Context. C'est conforme aux recommandations d'EF Core pour les frameworks d'interface côté serveur avec état, où les composants vivent longtemps et où un Context limité à la requête ne correspond pas à la durée de vie de l'interface. Dans la Support Desk Data Console, cela prend la forme d'une méthode de service : `await` `using var db = await _factory.CreateDbContextAsync();` suivi de `await db.Tickets.CountAsync()`,

c'est-à-dire exactement la première requête asynchrone que le lab exécute derrière un indicateur de chargement et un message d'erreur, sans rien lier pour l'instant.

Un Context à durée de vie plus longue est acceptable à un seul endroit : dans un éditeur modal, quand l'utilisateur ouvre un ticket, modifie des entités suivies, parcourt quelques petites collections liées comme ses commentaires, puis enregistre ou annule en une seule unité. Même dans ce cas, il est propre à cette instance de l'éditeur, protégé pour que deux opérations ne s'y exécutent jamais simultanément, et libéré lors de la libération du formulaire ; jamais dans un service partagé, un champ statique ou un cache global à l'application. La troisième option est la frontière repository ou service : les pages dialoguent avec `TicketQueryService` et `TicketCommandService`, qui possèdent la fabrique, si bien que l'interface ne voit jamais de Context. Cette frontière s'étoffe tout au long des labs, et c'est pourquoi le lab résout depuis la page un service de requête, et non un Context.

Sécurité des sessions : ce qui peut être partagé et ce qui ne doit jamais l'être

Chaque session de navigateur a ses propres objets d'interface, sa ligne sélectionnée, ses messages de validation, ses filtres et ses modifications en attente ; ce sont des préoccupations de session ou de formulaire. Les entités EF Core, les instances de `DbContext` et les change trackers ne doivent pas franchir les frontières des sessions. Un cache statique peut contenir des données de référence immuables, comme les catégories de tickets une fois chargées, ou des délégués de requêtes compilées, car personne ne les modifie. Il ne doit jamais contenir un `DbContext` modifiable, une `BindingSource` ou une entité sélectionnée, car un membre statique est partagé par toutes les sessions du serveur et le dernier à écrire l'emporte silencieusement.

La même règle s'applique au traitement en arrière-plan : une tâche lancée avec `Application.StartTask` crée son propre Context dans l'opération et met à jour l'interface via les mécanismes de Wisej.NET qui tiennent compte de la session, au lieu d'emprunter un Context créé par une page. Dans un gestionnaire, chaque appel EF Core est attendu avant que le même Context soit réutilisé, les boutons sont désactivés pendant un chargement ou un enregistrement, et un message d'état compréhensible remplace l'exception brute ; l'indicateur de chargement du lab est la plus petite version de cette discipline.



Pièges courants

Enregistrer le `DbContext` comme service Wisej.NET global partagé, si bien que toutes les sessions suivent leurs entités dans le même objet. Partager des entités ou une `BindingSource` entre sessions via un champ statique ou un cache. Lancer des opérations EF Core en parallèle sur le même Context, par exemple deux requêtes démarrées sans attendre la première. Garder un Context vivant dans une page parce que cela semble plus rapide, jusqu'à ce que son change tracker contienne chaque ticket jamais ouvert. Mélanger les services entre le registre de Wisej.NET et l'injection de dépendances de Microsoft une fois la passerelle installée.



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 :

- ✓ Expliquer la différence entre l'état de session Wisej.NET, la durée de vie des objets d'interface, la durée de vie de la requête/du thread et la durée de vie de l'unité de travail EF Core.
- ✓ Enregistrer EF Core dans l'injection de dépendances de Microsoft et exposer le fournisseur de services à Wisej.NET.
- ✓ Choisir entre un DbContext par opération, un DbContext par formulaire d'édition et une frontière repository/service.
- ✓ Éviter les instances de DbContext statiques partagées et les fuites de données accidentelles entre sessions.

Guide du lab / de l'examen

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



Lab pratique

Créez la solution Support Desk Data Console avec les projets Web, Data, Services et Tests, ajoutez les packages du fournisseur EF Core au projet de données, créez SupportDeskContext avec une fabrique de conception, enregistrez AddDbContextFactory dans le démarrage ASP.NET Core et exposez l'IServiceProvider de Microsoft à Wisej.NET, puis résolvez un service de requête depuis une Page Wisej.NET et exécutez une première requête de comptage asynchrone protégée par un indicateur de chargement et un message d'erreur, sans rien lier pour l'instant.

Point de départ

Ouvrez Visual Studio et créez une nouvelle solution nommée `SupportDesk` avec une application web Wisej.NET (`SupportDesk.Web`), les bibliothèques de classes vides `SupportDesk.Data` et `SupportDesk.Services` et un projet de test `SupportDesk.Tests` ; une base locale SQL Server LocalDB ou SQLite est le seul autre prérequis. Le modèle web vous fournit un `Program.cs` et une `Page` principale vide ; aucune entité, aucun Context et aucun service n'existe encore, et le modèle complet arrive au Module 2.

Approche suggérée

- ✓ Ajoutez à `SupportDesk.Data` le package du fournisseur EF Core (SQL Server ou SQLite) et le package Design, puis créez `SupportDeskContext` avec une entité `Ticket` minimale et `DbSet<Ticket> Tickets`, afin que la requête de comptage ait quelque chose à compter.
- ✓ Ajoutez `SupportDeskDesignTimeFactory`, qui implémente `IDesignTimeDbContextFactory<SupportDeskContext>`, pour que les outils `dotnet ef` puissent construire le Context sans démarrer l'application web.
- ✓ Dans `Program.cs`, lisez la chaîne de connexion depuis la configuration, appelez `AddDbContextFactory<SupportDeskContext>` avec des diagnostics réservés au développement, enregistrez `TicketQueryService` et, après `Build()`, appelez `Application.Services.AddService<IServiceProvider>(app.Services)`.
- ✓ Écrivez `TicketQueryService` dans `SupportDesk.Services` avec une dépendance de constructeur `IDbContextFactory<SupportDeskContext>` et une méthode `CountTicketsAsync` qui crée, utilise et libère un seul Context.
- ✓ Dans `TicketBrowserPage`, résolvez le service avec une propriété `[Inject]`, puis écrivez `countButton_Click` comme gestionnaire asynchrone avec un indicateur `_loading`, un bouton désactivé, un libellé d'état, un try / catch qui affiche un message compréhensible et un bloc finally qui rétablit le bouton.

Livrables attendus

- ✓ Solution avec les projets Web, Data, Services et Tests et les packages du fournisseur EF Core

- ✓ SupportDeskContext enregistré via AddDbContextFactory avec une fabrique de conception
- ✓ IServiceProvider de Microsoft exposé à Wisej.NET et un service de requête résolu depuis une Page
- ✓ Première requête de comptage asynchrone avec une protection de chargement et un message d'erreur
- ✓ Aucun DbContext statique dans toute la solution

Questions de révision

- ✓ Quelle durée de vie avez-vous choisie pour le DbContext derrière `CountTicketsAsync`, et que se passerait-il si la page gardait ce Context vivant entre deux clics ?
- ✓ Deux utilisateurs ouvrent la console Support Desk en même temps et cliquent tous deux sur Count. Quels objets sont distincts pour chaque session, lesquels sont partagés, et pourquoi cette répartition est-elle sûre ?
- ✓ Si la base de données est injoignable, que voit l'utilisateur, que fait le bouton, et comment l'indicateur de chargement revient-il à false ?



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.