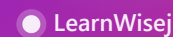


Modélisation, configuration du DbContext et migrations

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



Guide de la leçon

Des entités conçues pour les écrans liés aux données, la Fluent API pour les clés, les longueurs, les index, le comportement de suppression et RowVersion, les migrations traitées comme du code source relu avec scripts et bundles, et la fabrique de conception. Ce que cela signifie pour un développeur Wisej.NET qui travaille avec de vraies bases de données — et comment aborder ce module.



Objectif production

Des entités conçues pour les écrans liés aux données, la Fluent API pour les clés, les longueurs, les index, le comportement de suppression et RowVersion, les migrations traitées comme du code source relu avec scripts et bundles, et la fabrique de conception.

1. Le problème en production

La Support Desk Data Console affichera les tickets dans une grille paginable, les modifiera dans un formulaire modal et laissera plusieurs agents travailler sur la même file d'attente en même temps, et chaque écran ne vaut que ce que vaut le modèle en dessous. Un filtre sans index correspondant devient un parcours de table à cinquante mille tickets, deux agents qui enregistrent le même ticket perdent silencieusement le travail de l'un d'eux si la ligne ne porte pas de jeton de concurrence, et un schéma corrigé à la main casse le déploiement suivant. Le modèle est le contrat entre l'interface et la base de données ; ce module le construit donc explicitement et le versionne comme du code source.

2. Concepts clés

Concept	Pourquoi c'est important
entité adaptée à la liaison	Une entité avec une clé prévisible, des champs d'affichage stables, des propriétés de navigation explicites et de petites tables de référence, pour qu'une grille, une <code>ComboBox</code> et un éditeur puissent tous s'y lier.
modèle de vue <code>TicketListItem</code>	Une projection à plat des seules colonnes affichées par la grille, pour que la grille ne reçoive jamais tout le graphe de <code>Ticket</code> et reste indépendante de l'unité de travail d'édition.
configuration par la Fluent API	Des règles de mappage écrites dans <code>OnModelCreating</code> (chaînes obligatoires, longueurs, index, comportement de suppression, précision, valeurs par défaut) pour que les règles de persistance vivent à côté du <code>DbContext</code> .
jeton de concurrence <code>RowVersion</code>	Un tableau d'octets généré par la base et marqué <code>IsRowVersion</code> ; EF Core ajoute sa valeur d'origine à la clause WHERE de l'UPDATE, si bien qu'un enregistrement obsolète ne met à jour aucune ligne et lève <code>DbUpdateConcurrencyException</code> .
comportement de suppression	La règle <code>OnDelete</code> d'une relation (Restrict, SetNull, Cascade) qui décide si la suppression d'un client est refusée, désaffecte des tickets ou supprime aussi les commentaires.
migration	Une classe C# accompagnée du snapshot du modèle, qui décrit un changement de schéma relu, commité dans le contrôle de source et appliqué dans l'ordre pour amener n'importe quelle base au niveau du modèle.
fabrique de conception	Une <code>IDesignTimeDbContextFactory</code> qui indique aux outils <code>dotnet ef</code> comment construire le Context sans exécuter le démarrage de l'application Wisej.NET.

3. Le fil de la leçon

Des entités conçues pour les écrans maître-détail

Les interfaces liées aux données fonctionnent mieux quand le modèle a des clés prévisibles, des champs d'affichage stables, des relations explicites et de petites tables de référence. Dans la Support Desk Data Console, l'entité `Ticket` porte `Customer`, `Agent`, `Category`, `Status`, `Priority`, `DueDate`, `Title`, `Description`, `RowVersion`, `CreatedAt` et `UpdatedAt`. `Customer`, `Agent` et `Category` sont de petites entités de référence, chacune avec une collection `Tickets`, et `TicketComment` est la collection de détail d'un ticket.

`CustomerId` et `CategoryId` sont des entiers obligatoires, car un ticket sans eux n'a de sens sur aucun écran ; `AgentId` est un `int?` nullable, car un nouveau ticket existe avant que quiconque lui soit affecté, ce qui permet à la `ComboBox` des agents d'afficher « Unassigned » sans ligne sentinelle. Une grille n'a pas besoin de toutes les propriétés ; le cours définit donc aussi des modèles de vue `TicketListItem` (numéro, titre, client, statut, échéance, agent, dernière mise à jour) pour la grille en lecture seule, tandis qu'un éditeur se lie directement à un `Ticket` suivi ou à un modèle d'édition mappé en retour vers l'entité.

La Fluent API d'abord pour les modèles de production

Les conventions d'EF Core découvrent les clés et les clés étrangères à partir des noms, mais une application de production doit configurer explicitement tout ce qui influe sur le schéma et le comportement : les chaînes obligatoires, les longueurs maximales, les index utilisés par les filtres de recherche, le comportement de suppression, la précision décimale, les valeurs par défaut et les jetons de concurrence. Les annotations de données conviennent aux projets simples ; la Fluent API garde les règles de persistance près du `DbContext` et évite de mélanger la validation de l'interface et le mappage de la base. Le cœur de `SupportDeskContext.OnModelCreating` :

```
modelBuilder.Entity<Ticket>(b =>
{
    b.Property(x => x.Title).HasMaxLength(180).IsRequired();
    b.Property(x => x.Status).HasMaxLength(30).IsRequired();
    b.Property(x => x.RowVersion).IsRowVersion();
    b.HasIndex(x => new { x.Status, x.DueDate });
    b.HasIndex(x => x.CustomerId);
    b.HasIndex(x => x.UpdatedAt);
    b.HasOne(x => x.Customer).WithMany(c => c.Tickets)
        .HasForeignKey(x => x.CustomerId).OnDelete(DeleteBehavior.Restrict);
    b.HasOne(x => x.Agent).WithMany(a => a.Tickets)
        .HasForeignKey(x => x.AgentId).OnDelete(DeleteBehavior.SetNull);
});
```

Chaque ligne est une décision dont dépend le navigateur de tickets. L'index composite sur `Status` et `DueDate` sert la file d'attente qui filtre par statut et trie par échéance, `CustomerId` sert le filtre par client et `UpdatedAt` le tri « modifiés récemment ». `Restrict` signifie qu'un client ou une catégorie ayant des tickets ne peut pas être supprimé par accident, `SetNull` signifie qu'un agent qui part désaffecte simplement ses tickets, et les commentaires utilisent `Cascade`. Une colonne de type monétaire recevrait ici `HasPrecision(18, 2)`, et un statut par défaut « Open » se définit ici plutôt que dans le code du formulaire.

Le jeton de concurrence est ce qui compte le plus dans une application Wisej.NET multi-utilisateur. Un `byte[] RowVersion` configuré avec `IsRowVersion` devient une colonne SQL Server `rowversion` que la base modifie à chaque écriture, et EF Core inclut la valeur d'origine dans la condition de mise à jour : `UPDATE ... WHERE Id = @p0 AND RowVersion = @p1`. Si un autre utilisateur a modifié la ligne après son chargement, la mise à jour ne touche aucune ligne et `SaveChangesAsync` lève `DbUpdateConcurrencyException` ; l'interface peut alors afficher les valeurs actuelles de la base et proposer de recharger, d'écraser ou de fusionner. SQLite n'a pas de type `rowversion` : la propriété y reste un jeton de concurrence, mais un trigger ou le code doit l'incrémenter.



Erreur courante : des annotations pour tout

Décorer chaque entité avec `[Required]`, `[MaxLength]` et `[Timestamp]` alors que la Fluent API serait plus claire, si bien que l'entité est à la fois le mappage de persistance et le contrat de validation de l'interface.



Erreur courante : aucun jeton de concurrence sur les écrans multi-utilisateurs

Laisser `RowVersion` de côté parce que « la grille est en lecture seule ». L'éditeur, lui, ne l'est pas, et sans jeton le second agent qui enregistre écrase silencieusement le premier.

Les migrations, du code source relu

Les migrations sont du code source. Pendant le développement, ajoutez-en une à chaque évolution du modèle avec `dotnet ef migrations add InitialCreate --project SupportDesk.Data --startup-project SupportDesk.Web`, relisez les opérations `Up` et `Down` générées, puis appliquez-la avec `dotnet ef database update`. La relecture repère un renommage généré sous forme de suppression puis ajout de colonne qui perd des données, un `Cascade` là où vous vouliez `Restrict`, un index manquant ou une colonne `rowversion` qui n'est jamais apparue. La migration et le snapshot du modèle sont commités ensemble.

La production, c'est autre chose. Évitez les changements de schéma automatiques et non relus au démarrage de l'application dans les environnements à plusieurs instances : deux instances qui démarrent en même temps se disputent l'application de la même migration. Préférez un script SQL relu produit par `dotnet ef migrations script --idempotent` quand une approbation de la base et un plan de retour arrière sont exigés, ou un bundle de migrations produit par `dotnet ef migrations bundle`, un exécutable autonome qu'un pipeline de livraison lance avant de basculer le trafic.

- Chaînes de connexion externalisées, jamais dans le code source.
- Chaque migration relue ; script ou bundle généré et conservé avec la livraison.
- Risque de migration au démarrage en multi-instance traité par écrit.

Les outils `dotnet ef` doivent pouvoir construire `SupportDeskContext` au moment de la conception, et une fabrique de conception est le moyen le plus explicite de le leur permettre.

`SupportDeskDesignTimeFactory` implémente `IDesignTimeDbContextFactory<SupportDeskContext>` ; sa méthode `CreateDbContext` construit les options avec le fournisseur de l'application et une connexion locale sûre (`SupportDeskDev` sur LocalDB, connexion approuvée, sans mot de passe) et renvoie un nouveau Context. C'est important, car le démarrage Wisej.NET inclut une configuration du framework d'interface qui ne doit pas s'exécuter pendant la génération des migrations ; avec la fabrique, les outils n'y touchent jamais.



Erreur courante : considérer les migrations comme des fichiers générés qui n'ont jamais besoin d'être relus

Traiter le dossier Migrations comme une sortie de build. Une migration est un changement de schéma que quelqu'un exécutera en production ; relisez-la comme n'importe quel autre commit.

Les contraintes du schéma ne sont pas de la validation d'interface

`HasMaxLength(180).IsRequired()` sur `Title` produit une colonne `NOT NULL nvarchar(180)`, et la violer produit une `DbUpdateException` après un aller-retour, avec un message du fournisseur qu'aucun utilisateur ne devrait lire. C'est le dernier filet de sécurité, pas le premier. Le message compréhensible « Enter a title » revient à l' `ErrorProvider` de l'éditeur et au `TicketValidator` qu'ajoute le Module 5,

tout comme « closed tickets cannot have a future due date », qu'aucune contrainte de colonne ne peut exprimer. Gardez volontairement les niveaux séparés : la Fluent API déclare ce que la base doit garantir (nullabilité, longueur, index unique sur le numéro de ticket, clés étrangères, jeton de concurrence) ; les `DataAnnotations` et les validateurs du modèle d'édition déclarent ce qu'il faut dire à l'utilisateur.

Les données initiales suivent la même discipline. Les lignes de référence fixes comme les catégories peuvent passer par `HasData` pour voyager avec la migration ; les cinquante tickets sont mieux produits par une méthode d'insertion réservée au développement qui crée un Context depuis la fabrique, vérifie si des tickets existent déjà, ajoute les lignes, appelle `SaveChangesAsync` et libère le Context. Placée derrière un bouton « Seed development data » avec une protection de chargement et un `try / catch`, c'est elle qui fait renvoyer pour la première fois plus de zéro à la requête de comptage du Module 1.



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 :

- ✓ Modéliser des entités et des relations pour une interface maître-détail.
- ✓ Utiliser la Fluent API pour les clés, les index, les champs obligatoires, la longueur, la précision décimale, le comportement de suppression et la version de ligne.
- ✓ Créer des migrations et comprendre quand utiliser des scripts, des bundles et des fabriques de conception.
- ✓ Séparer les contraintes du schéma de base de données de la validation de l'interface et du domaine.

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 les entités Customer, Agent, Category, Ticket et TicketComment de la Support Desk Data Console, configurez les relations et les comportements de suppression avec la Fluent API, configurez RowVersion avec IsRowVersion, ajoutez des index pour Status, DueDate, CustomerId et UpdatedAt, ajoutez une fabrique de conception, créez la migration InitialCreate et insérez cinq clients, trois agents, des catégories et au moins cinquante tickets dans une base locale SQLite ou SQL Server LocalDB.

Point de départ

Ouvrez la solution Support Desk Data Console du Module 1 : `SupportDesk.Web` (le projet Wisej.NET), `SupportDesk.Data` avec un `SupportDeskContext` vide enregistré via `AddDbContextFactory`, `SupportDesk.Services` et `SupportDesk.Tests`. La page du Module 1 exécute toujours sa requête de comptage asynchrone sur une base sans tables ; la première chose que vous verrez une fois ce lab terminé, c'est ce comptage qui devient un vrai nombre.

Approche suggérée

- ✓ Écrivez d'abord les cinq classes d'entités dans `SupportDesk.Data`, avec des clés entières, les propriétés de navigation dans les deux sens, un `AgentId` nullable et un `byte[] RowVersion` sur `Ticket` ; laissez-les sans attributs de mappage.
- ✓ Configurez tout ce qui touche au schéma dans `OnModelCreating` : chaînes obligatoires et longueurs, `IsRowVersion`, les quatre index prévus et `OnDelete` pour chaque relation (Restrict pour Customer et Category, SetNull pour Agent, Cascade pour les commentaires).
- ✓ Ajoutez `SupportDeskDesignTimeFactory` avec le même fournisseur que l'application et une connexion locale sans mot de passe, puis exécutez `dotnet ef migrations add InitialCreate` depuis la racine de la solution.
- ✓ Lisez la migration générée avant de l'appliquer : vérifiez la colonne `rowversion`, les noms des index, les clauses `ON DELETE` et le snapshot, puis exécutez `dotnet ef database update`.
- ✓ Écrivez une méthode d'insertion réservée au développement qui crée un Context depuis la fabrique, vérifie si des tickets existent déjà, ajoute les clients, les agents, les catégories et les tickets, enregistre et libère le Context, puis appelez-la depuis un bouton Seed protégé par un indicateur de chargement et un `try / catch`.

Livrables attendus

- ✓ Entités Customer, Agent, Category, Ticket et TicketComment avec leurs relations et comportements de suppression
- ✓ RowVersion configuré avec IsRowVersion
- ✓ Index pour Status, DueDate, CustomerId et UpdatedAt
- ✓ Fabrique de conception et migration InitialCreate

- ✓ Données initiales : cinq clients, trois agents, des catégories et au moins cinquante tickets

Questions de révision

- ✓ Vous supprimez un client qui a encore des tickets ouverts. Que se passe-t-il avec le comportement de suppression que vous avez configuré, et que se serait-il passé avec Cascade ?
- ✓ Deux agents chargent le ticket SD-1042 et cliquent tous deux sur Save à une seconde d'intervalle. Quel enregistrement lève une exception, que contient la clause WHERE de l'instruction UPDATE, et que voit l'utilisateur ?
- ✓ Pourquoi `IsRequired()` sur `Title` ne donne-t-il pas à l'utilisateur un message « Enter a title », et d'où viendra ce message plus loin dans le cours ?



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.