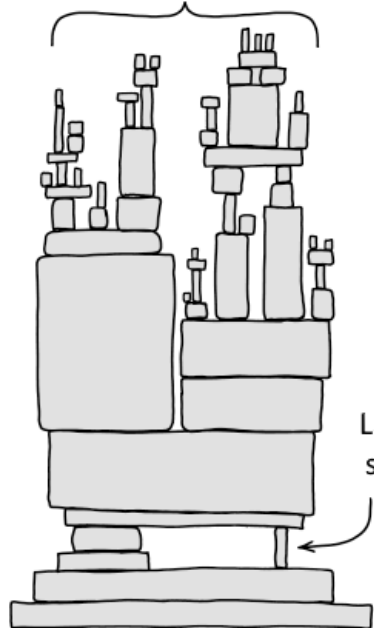


Semaine 09

Évolution de la BD

Bases de données et programmation Web

15 projets qui utilisent
une même BD






La BD qu'on
s'apprête à
modifier



- ❖ Le but
- ❖ Évolution de la BD
- ❖ Évolution de l'application Web



❖ Maintenance d'une base de données

- ◆ Toute modification sur la BD risque de demander des changements dans l'application Web
 - Ex : Nouvelle table -> Ajout d'un **DbSet**, ajout d'un **Model**, ajout de [**DataAnnotations**] et d'instructions **FluentAPI**, ajout et modifications de **contrôleurs** et de **vues**, etc.
 - Idéalement, la représentation de la BD dans le **DbContext** doit continuer de correspondre exactement à la BD pour que toutes les opérations sur les **DbSet** fonctionnent.
- ◆ S'il y a **déjà des données présentes** dans la BD, **protéger leur intégrité** est la **priorité** ! 
 - Avant même de s'inquiéter pour notre application Web, il faut songer à la BD elle-même !
- ◆ Dans ce cours, nous aborderons brièvement comment faire l'évolution **d'une BD** et d'une application Web reliés  
 - Certains changements peuvent être **automatisés**
 - Certaines **librairies** et **techniques** peuvent **simplifier** ou **guider** les tâches liées à la maintenance.
 - D'autres tâches nécessiteront du **bidouillage** plus « manuel ».



❖ Maintenance d'une base de données

◆ Développement Agile

- Comme les pratiques Agile amènent à **incrémenter** la taille d'un projet (et donc de sa BD) **progressivement**, cela signifie une **évolution continue** de la BD. (Avec possiblement déjà des données si l'application est en production)

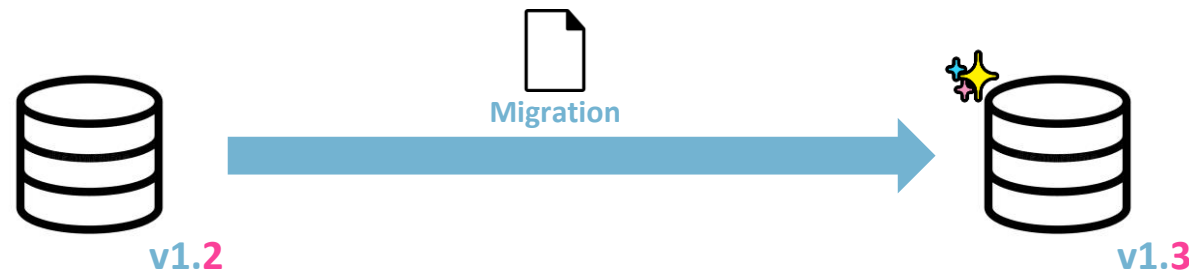
◆ Plus simple en Code-First ?

- En Prog Web orientée services, on tape **deux commandes** pour modifier la structure de la BD après avoir modifié nos Models !
 - S'il y a des **données à conserver** dans la BD, ça devient délicat.
 - S'il y a d'autres projets Code-First (ou DB-First) qui **dépendent de la même BD**, ça devient délicat.
 - Ça nous amène quand même à modifier nos contrôleurs et nos vues.
- Donc non ! La maintenance d'une BD est rarement une *partie de plaisir*.
 - Les notions de cette semaine pourraient vous offrir une perspective intéressante même pour des projets Code-First.



❖ Migrations

- ◆ On appelle « **migrations** » les **scripts SQL** qui transfèrent des données ou changent la structure d'une base de données.
 - Une migration permet à une base de données de passer d'un **état** à un autre. (Donc de changer de « version »)
 - Le script de la version initiale de la BD peut également être considéré comme une migration, mais on y réfère parfois comme le « seed » ou « l'initialisation ».





<https://evolve-db.netlify.app/>

❖ Évolution de la BD

- ◆ Dans cette section nous aborderons certains types de changements typiques sur la BD et des exemples de solutions pour les mettre en œuvre proprement.
 - ... mais avant !
- ◆ **Evolve*** (Outil open source inspiré de **Flyway**)
 - Flyway et Evolve DB sont des exemples d'outils pour encadrer le versionnage de bases de données.
 - Avantages :
 - Encadre notre historique d'évolution de la BD. (Plus facile de retracer les versions et les évolutions qui ont été faites)

*Evolve n'est pas un outil ultra populaire, mais il est facile à utiliser, fonctionne avec .NET et ses principes sont réutilisables avec d'autres outils plus répandus.



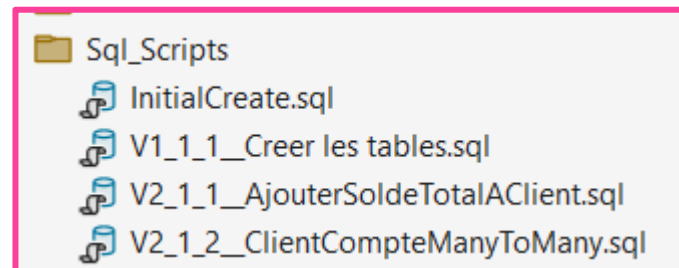
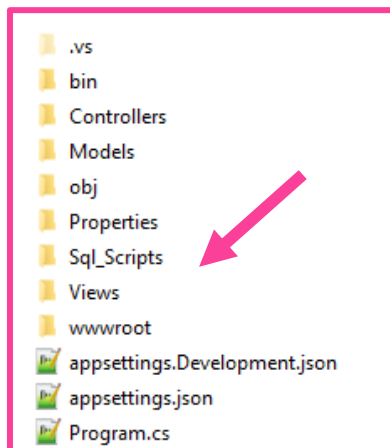
❖ Evolve

◆ Préparation des migrations

- Les fichiers **.sql** de migration doivent être situés dans le dossier **Sql_Scripts** du projet ASP.NET Core. (Créez le dossier dans le projet)
- Les fichiers de migrations doivent suivre la convention de nommage

VX_X_X_X__Descriptif.sql.

- Vous pouvez spécifier autant de sous-versions que vous voulez. Dans l'exemple ci-dessous il n'y a que des **VX_X_X**.
- Attention au **double __ trait de soulignement** qui précède le descriptif !
- Tout autre fichier (Par exemple ici, **InitialCreate.sql**) sera ignoré par **Evolve**.



Sauter des versions dans les numéros ne cause pas de problèmes. (C'est simplement moins **cohérent** en termes de versionnage, mais si vous décidez d'éliminer une migration pour une raison quelconque, pas besoin de tout renommer non plus)



❖ Evolve

◆ Préparation de la BD

- **Prérequis supplémentaire : Créez votre BD dans SSMS ! (Sans schéma, sans table, sans rien d'autre !)**
- Comme **Evolve** enrobe toutes les migrations dans des **transactions** (ce qui lui permet de **rollback** une migration entière si jamais il y a une erreur), une migration ne peut pas contenir l'instruction **CREATE DATABASE**, qui ne fonctionne pas dans une **transaction**.

```
InitialCreate.sql - L...hantal.vallieres (60))
IF EXISTS(SELECT * FROM sys.databases WHERE name='S09_Theorie')
BEGIN
    DROP DATABASE S09_Theorie
END
CREATE DATABASE S09_Theorie
GO
```

- De plus, comme d'habitude, obtenez votre **string de connexion** à cette BD vide. Attention le serveur change à chaque fois que vous changez d'ordi. Mettez un **.** après **Data Source=**

Cộng nghệ tiên tiến

S. Ewölwé Dátjā Sōusçê İnîtiāl Căţălōg S. Thēōsiē İnţēgřsăţfēd Şēçusîţy Tsuē Rēsşîştş Şēçusîţy İnğō Găłşē Rōđlîñğ Găłşē Nul'tiřlê Açtîwê Rēşul'tş Şēţş Găłşē Ençsŷřţ Găłşē Tsuştş Şēsŵēs Cēsţîğîçăţē Găłşē Cōññăñđ Tîñēōuţ .



❖ Evolve

◆ Installation

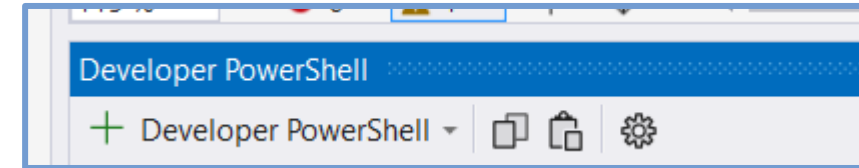
- À l'école, on a installé l'outil Evolve mais cet outil veut utiliser .Net 7.0 qui n'est plus supporté par Microsoft.
- Il faut exécuter une commande pour le mettre à jour pour la version prerelease qui fonctionne avec .Net 8.0

dotnet tool update evolve.tool -g --prerelease



❖ Evolve

◆ Exécution des migrations



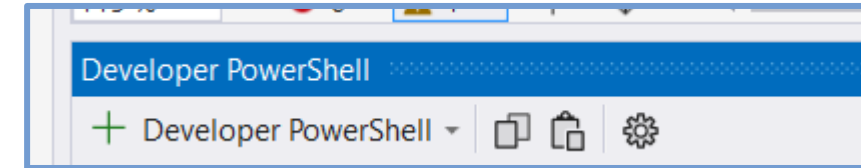
```
evolve migrate sqlserver -c "Server=.;Initial Catalog=S09_Theorie;Integrated Security=True;Persist Security Info=False;Pooling=False;MultipleActiveResultSets=False;Encrypt=False;TrustServerCertificate=False" -s Clients -s Comptes --target-version 1.1.1
```

- On utilise la commande **evolve migrate** dans la fenêtre **Powershell** du projet de Visual Studio.
 - **sqlserver** : Evolve est compatible avec plusieurs **SGBD**, alors il faut préciser **SQL Server**.
 - **-c** : L'option **-c** est suivie de notre **string de connexion**. Attention ! Le nom du serveur doit absolument être précédé d'un **.** (et non de DESKTOP32C0330 par exemple)
 - **-s** : L'option **-s** est suivi du nom d'un **schéma** qui sera créé / modifié lors de l'exécution de la migration. (Evolve créera les schémas de la BD pour nous s'ils n'existent pas déjà) Généralement on spécifie tous les schémas de la BD ici.
 - **--target-version** : Cette option est suivie de la **version de la BD** qu'on veut atteindre avec les migrations. Ici, on a choisi **1.1.1**. Si on avait choisi 1.5, cela exécuterait, par exemple, les fichiers **V1_1**, **V1_2**, **V1_3**, **V1_4** et **V1_5** (Dans l'ordre). Ainsi, pas besoin d'exécuter chaque script un à la fois.



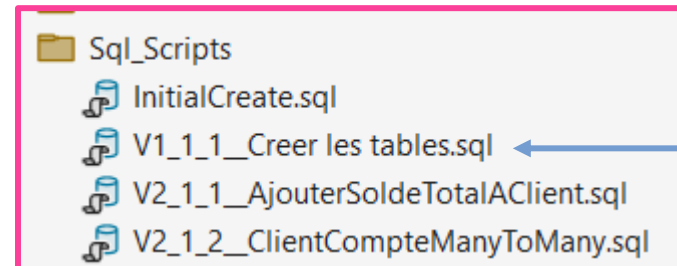
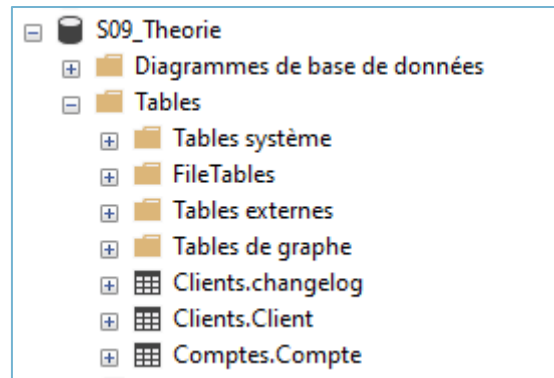
❖ Evolve

◆ Exécution des migrations



```
evolve migrate sqlserver -c "Server=.;Initial Catalog=S09_Theorie;Integrated Security=True;Persist Security Info=False;Pooling=False;MultipleActiveResultSets=False;Encrypt=False;TrustServerCertificate=False" -s Clients -s Comptes --target-version 1.1.1
```

- L'exécution de cette commande cause la création des schémas Clients et Comptes ainsi que l'exécution du script V1_1_1__Créer les tables (Clients.Client et Comptes.Compte)



- Il nous faut donc régénérer les modèles de la BD, avec --force

```
dotnet ef dbcontext scaffold Name=S09_Evolve Microsoft.EntityFrameworkCore.SqlServer -o Models --context-dir Data --data-annotations --force
```



❖ SANS Evolve

◆ Exécution des migrations

- On va exécuter chacun des scripts un à la fois, simulant des changements incrémentaux à la BD suite au travail fait dans un sprint par exemple.
- **À chaque fois qu'un script SQL fait des changements dans les tables ou les vues, il nous faut régénérer les modèles de l'application, avec `--force`**

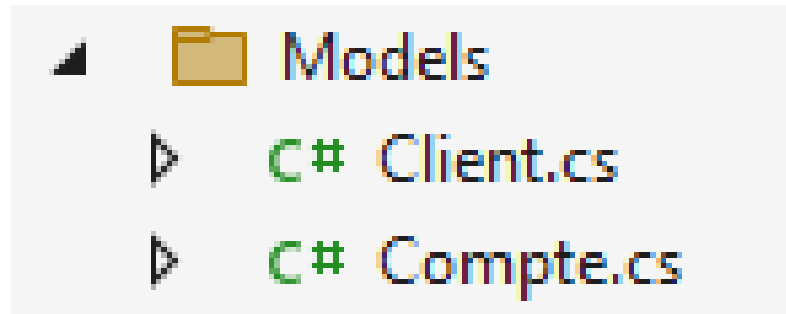
```
dotnet ef dbcontext scaffold Name=S09_Evolve Microsoft.EntityFrameworkCore.SqlServer -o Models --context-dir Data --data-annotations --force
```



❖ SANS Evolve

- ◆ Ainsi, après avoir exécuté `V1_1_1__Créer les tables.sql` on a 2 tables de créées dans sql_Server.
- ◆ On exécuter la commande pour forcer les modèles à s'adapter aux changements de la structure de la BD

```
dotnet ef dbcontext scaffold Name=S09_Evolve Microsoft.EntityFrameworkCore.SqlServer -o Models --context-dir Data --data-annotations --force
```





❖ L'intérêt d'un outil de gestion comme Evolve

◆ Exécution des migrations

- Dans le cas d'un **échec** d'une des migrations (Message invalid checksum), la migration échouée est **rollback** et les suivantes **ne sont pas exécutées**.
 - Il ne reste plus qu'à corriger l'erreur dans la migration et réexécuter la même commande.

◆ Rembobiner la version de la BD

- La commande **evolve erase** permet de rembobiner des migrations. Cela dit, il y a quelques contraintes.
 - Les schémas modifiés (rembobinés) doivent **avoir été créés par Evolve**.
 - Les contraintes habituelles (Clés étrangères, clés primaires, etc.) s'appliquent et vont souvent **empêcher d'annuler certaines migrations**.
- Par souci de simplicité, nous nous contenterons de **supprimer la BD manuellement**, puis de réutiliser la commande **migrate** avec la version de notre choix. (Pour les rares cas où on souhaite rembobiner la BD)
 - Pour recréer la BD rapidement, utilisez le petit script proposé dans la prochaine diapo.





❖ Evolve

-- Code du script InitialCreate.sql pour la demo du cours.

```
IF EXISTS(SELECT * FROM sys.databases WHERE name='S09_Theorie')
BEGIN
    DROP DATABASE S09_Theorie
END

CREATE DATABASE S09_Theorie
GO
```

  dbo.changelog

Attention ! Lorsqu'on utilise Evolve, une table nommée changelog est créée automatiquement pour permettre à Evolve de garder des traces des opérations qui ont été faites sur la BD. Ne pas supprimer.



❖ Maintenance de la BD

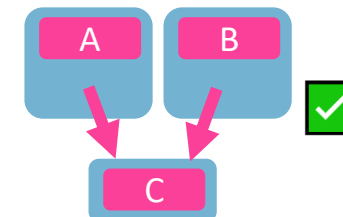
- ◆ Cette prochaine section aborde, concrètement, la mise en œuvre de certains changements dans la structure, la qualité ou l'intégrité d'une BD.
- ◆ Pour le moment, **on ne se soucie pas des applications qui utilisent la BD.**
 - Principes de bases
 - Exemples de changements dans la structure
 - Exemples de changements dans la qualité
 - Exemples de changements à l'intégrité référentielle



❖ Maintenance de la BD

◆ Principes de base

- 🖐️ **Petits changements** progressifs : Chaque changement risque de briser des choses. Généralement préférable de découper en petits morceaux les ajustements et les correctifs nécessaires. (N'ayez pas peur de faire beaucoup de sous-versions !)
- 🏷️ **Identifier** (versionner) les changements : Evolve nous « oblige » à le faire !
- 📄 Avoir une **table qui spécifie la configuration** / version de la BD : Par exemple, une table nommée « changelog » comme Evolve le fait et spécifie que la BD est à la version 2.4.1 !
- 🔄 **Éviter de dupliquer du SQL** : Comme pour n'importe quel langage, on veut éviter de dupliquer du code.
 - **Exemple** : Deux procédures ont une opération en commun (**C**). Cette opération commune pourrait être encapsulée dans une troisième procédure, qui serait appelée par les deux premières. Si jamais l'opération encapsulée dans la nouvelle procédure doit être modifiée (à cause de changements dans une table), elle n'aura qu'à être modifiée à un endroit plutôt que deux !





❖ Maintenance de la BD

◆ Principes de base

- Cela peut sembler évident, mais lorsque des changements sont faits dans une BD, il faut **vérifier que tout le reste autour fonctionne encore** ! Cela inclut :
 - **Vues** : Des vues peuvent être brisées si des colonnes ont été renommées, déplacées, retirées.
 - **Déclencheurs** : Des changements sur la table liée au déclencheur ou sur d'autres tables qui interviennent dans le code du déclencheur peuvent l'affecter.
 - **Procédures** : Des procédures peuvent être brisées si des colonnes ont été renommées, déplacées, retirées.
 - **Tables** : Les renommages de clés primaires peuvent amener à renommer des clés étrangères associées.
 - **Cycles référentiels** : Nous en avons parlé pour les contraintes FK qui génèrent des cycles. (Empêchant ainsi les ON CASCADE ...) Dès que des tables forment un cycle référentiel, il faut retirer les ON CASCADE sur certaines contraintes FK et créer / modifier des déclencheurs pour compenser.
- Encore une fois, faire des **petits changements à la fois** simplifie grandement le démêlage de tous ces objets à corriger.



❖ Maintenance de la BD

◆ Exemples de changements de la structure

- Supprimer une colonne, table ou vue
- Colonne calculée (V2_1_1__AjouterSoldeTotalAClient.sql)
- Clé artificielle
- Fusion de colonnes ou de tables
- Séparation de colonnes ou de tables
- Déplacer une colonne
- Renommer une colonne, table ou vue
- De 1-N à N-M (V2_1_1__ClientCompteManyToMany.sql)

Les changements de structure en rouge sont ceux qu'on va expliquer en détail et que vous pourrez pratiquer à partir de la démo AVANT de faire vos exercices.



❖ Maintenance de la BD

◆ Supprimer une colonne, une table ou une vue

○ Colonne :

- Si elle faisait partie de la **clé primaire**, il faut remplacer la clé primaire d'abord. (Et les clés étrangères associées)
- Si la donnée perdue pourrait être utile situationnellement, l'archiver dans une autre table pendant une période raisonnable.
- Modifier les vues, procédures, déclencheurs, etc. qui utilisaient la colonne.

○ Table :

- Archiver les données si nécessaire.
- Attention aux conflits avec des FK -> Il faut d'abord supprimer les FK.
- Modifier les vues, procédures, déclencheurs, etc. qui utilisaient la table.

○ Vue :

- Modifier les autres vues (oui), les procédures, déclencheurs, etc. qui l'utilisaient.
- Attention, un déclencheur de type INSTEAD OF (Pas AFTER) peut être créé pour une vue. Dans ce cas, le trigger devrait être supprimé.

Client	
PK ClientID	
Prénom	
Nom	
Courriel	✗
NoTel	

```
ALTER TABLE schema.table  
DROP COLUMN colonne;
```

Client	
PK ClientID	
Prénom	
Nom	
Courriel	
NoTel	

```
DROP TABLE schema.table;
```

```
DROP VIEW schema.vue;
```



❖ Maintenance de la BD

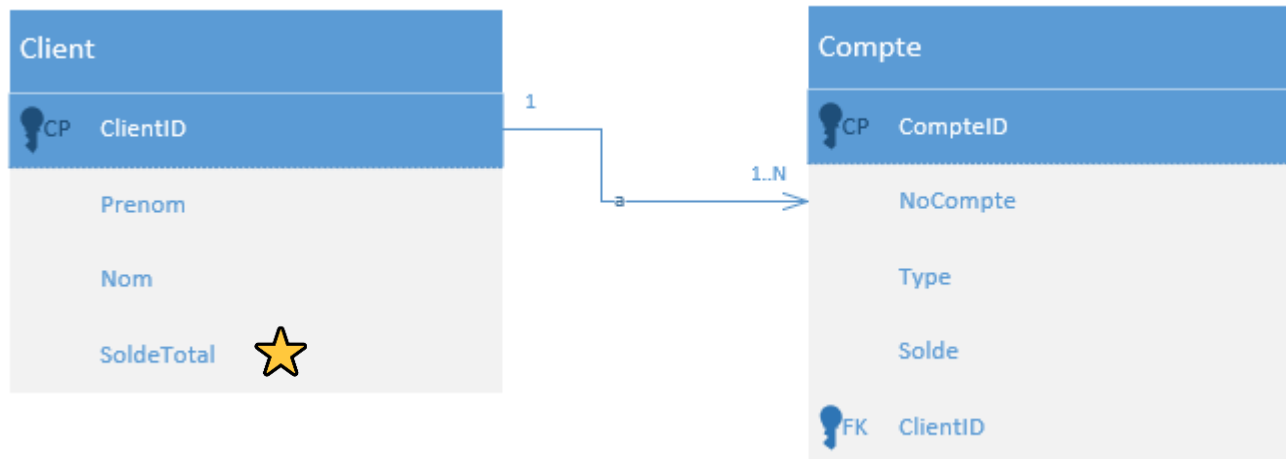
◆ Colonne calculée

- Créer la colonne et la remplir à l'aide du calcul approprié n'est pas suffisant. Il faut établir une **stratégie de synchronisation**.
 - Ça peut être un **déclencheur** sur la table compte qui met à jour SoldeTotal en temps réel. Cette stratégie est la plus « lourde ».
 - Ça peut être une « **batch job** » (Lot d'instructions répété à intervalle prédéterminé) si la donnée n'a pas à être calculée systématiquement.
 - Ça peut être une **procédure** qui remplace un SELECT et en profite pour mettre à jour la donnée pour la rangée des clients sélectionnés.
- Il est préférable de s'assurer que les avantages d'une **colonne calculée** l'emportent sur les désavantages introduits.



❖ Maintenance de la BD

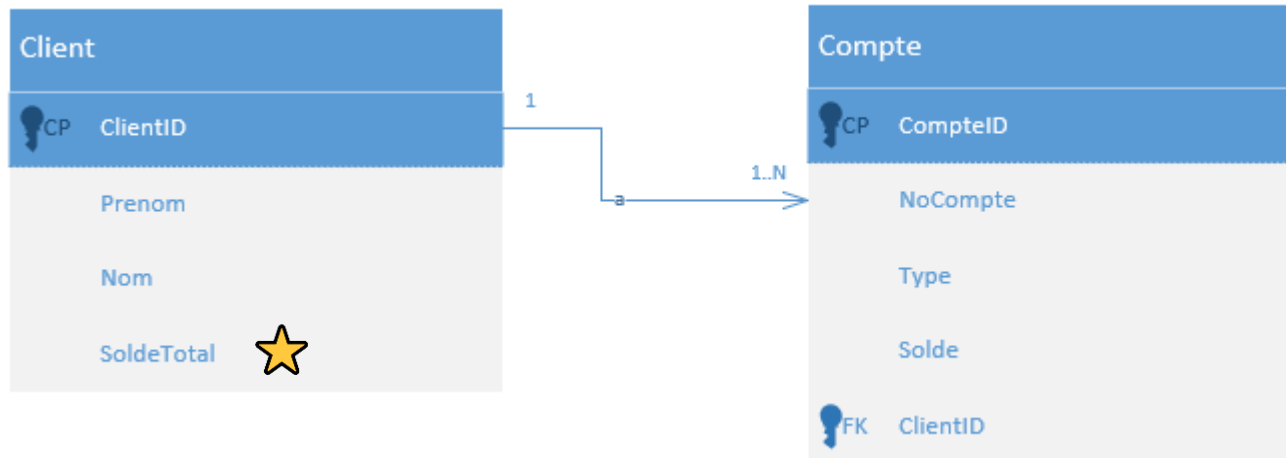
- ◆ Quand on a commencé l'application, on n'a pas pensé que nos clients aimeraient savoir en tout temps combien d'argent ils ont globalement, dans tous leurs comptes de courtage.
- ◆ On veut maintenant ajouter une colonne calculée dans Client pour avoir cette information disponible en tout temps.





❖ Maintenance de la BD

- ◆ Ajouter la **Colonne calculée**,
- ◆ **Fonction** pour calculer sa valeur,
- ◆ **Update** utilisant la fonction pour mettre le champ à jour,
- ◆ **Déclencheur** pour quand le solde d'un compte change, on utilise la fonction pour mettre à jour la valeur du champ.



V2_1_1__AjouterSoldeTotalAClient.sql

```
USE S09_Theorie
GO

ALTER TABLE Clients.Client
ADD SoldeTotal money NOT NULL;
GO

CREATE OR ALTER FUNCTION Clients.CalculerSoldeTotal
(@ClientID int)
RETURNS money
AS
BEGIN
    Return (
        SELECT ISNULL(SUM(Solde),0)
        FROM Comptes.Compte
        WHERE ClientID = @ClientID;
    )
END
GO

UPDATE Clients.Client
SET SoldeTotal = Clients.CalculerSoldeTotal(ClientID)
GO

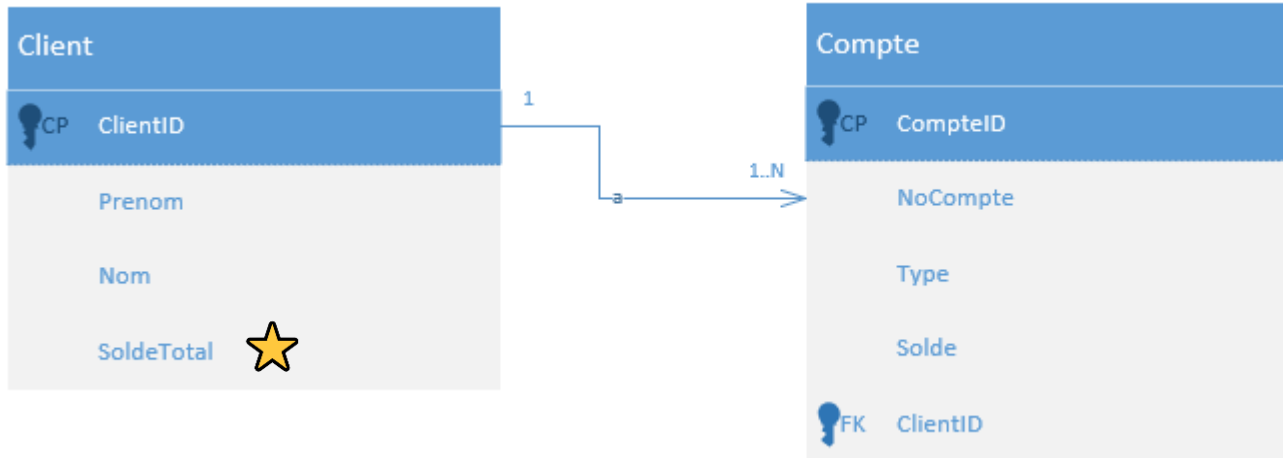
CREATE OR ALTER TRIGGER Comptes.trg_uSoldeCompte
ON Comptes.Compte
AFTER UPDATE
AS
BEGIN
    IF (UPDATE(Solde))
    BEGIN
        DECLARE @ClientID INT
        SELECT @ClientID = ClientID FROM inserted
        UPDATE Clients.Client
        SET SoldeTotal = Clients.CalculerSoldeTotal(@ClientID) ;
    END
END
GO
```

Observez que cette migration change une table de la BD.



❖ Exécution de la migration:

```
evolve migrate sqlserver -c "Server=.;Initial Catalog=S09_Theorie;Integrated Security=True;Persist Security Info=False;Pooling=False;MultipleActiveResultSets=False;Encrypt=False;TrustServerCertificate=False" -s Clients -s Comptes --target-version 2.1.1
```



❖ Régénération des modèles puisqu'une table de la BD a changé:

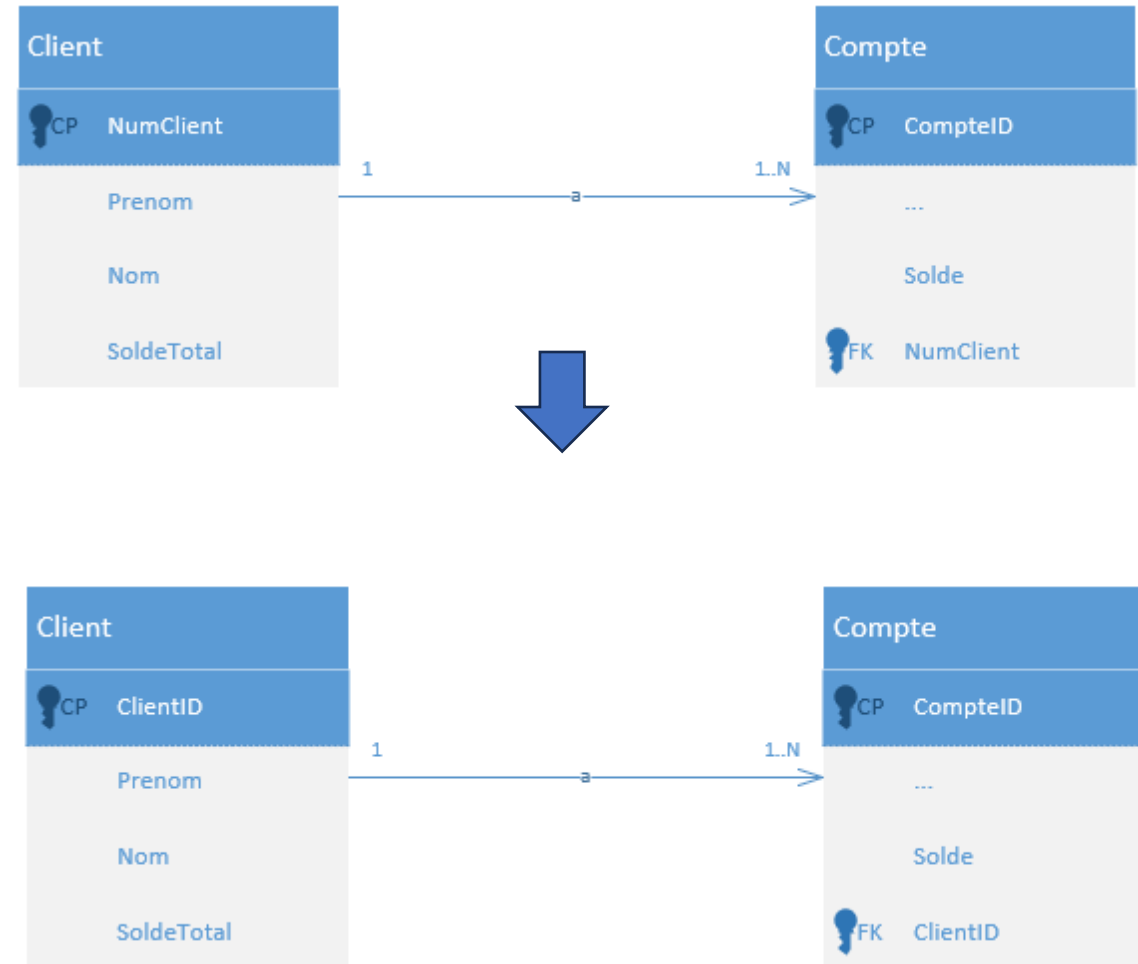
```
dotnet ef dbcontext scaffold Name=S09_Evolve Microsoft.EntityFrameworkCore.SqlServer -o Models --context-dir Data --data-annotations --force
```




❖ Maintenance de la BD

◆ Introduction d'une **clé artificielle**

- Intéressant si la clé **naturelle** est moins performante (ex : des nvarchar) ou si la clé **naturelle** utilisée change ou devient obsolète.

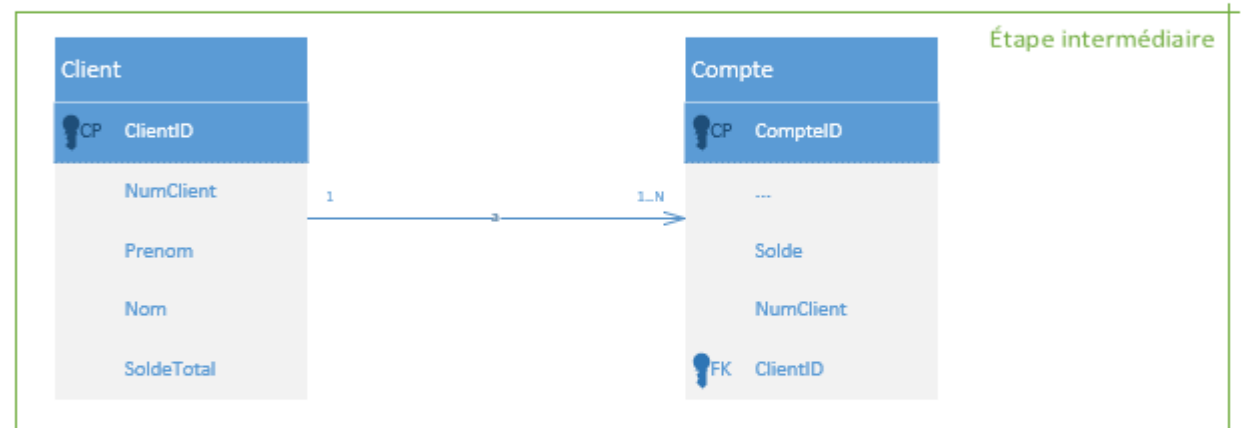
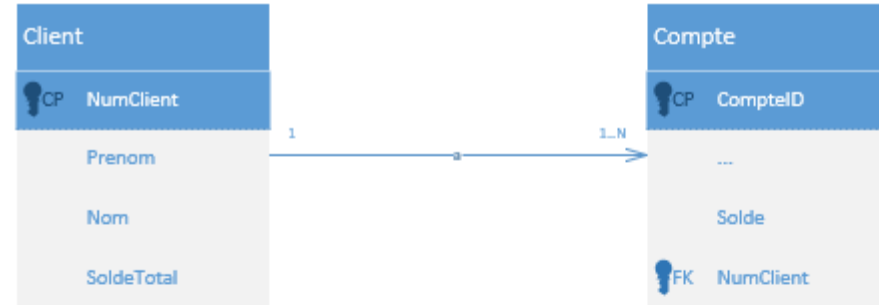




❖ Maintenance de la BD

◆ Introduction d'une clé artificielle

L'étape intermédiaire sera utilisée pour faire le lien entre les deux tables sur la valeur de NumClient afin d'identifier quelle valeur de ClientID de la table Client sera mis dans la clé étrangère ClientID de la table Compte.





❖ Maintenance de la BD

◆ Introduction d'une clé artificielle

○ Généralement, les étapes de la transformation ressembleront à ceci :

- Ajouter la colonne **IDENTITY(1,1)** dans la table et ajouter une colonne **int null** dans les tables avec la FK. La colonne **IDENTITY** se remplira **immédiatement et automatiquement**.
- Supprimer l'ancienne **contrainte FK**, créer la nouvelle **contrainte FK**, supprimer l'ancienne **contrainte PK**, créer la nouvelle **contrainte PK**.
- Remplir la nouvelle **colonne FK** dans les tables associées. (Et rendre la FK **not null**, si cela s'applique) UPDATE sur les numClient
- Supprimer les anciennes colonnes. (Et archiver les données supprimées si nécessaire)



❖ Maintenance de la BD

◆ Introduction d'une clé artificielle

- Vous allez le faire dans le labo! On ne vous donne pas la solution ici. On vient juste de vous expliquer le processus.



❖ Maintenance de la BD

◆ Fusion de colonnes ou de tables

- **Colonnes** : Rarement intéressant et risque de faire perdre de la précision aux données. Seulement pertinent si les deux valeurs sont toujours utilisées ensemble et ne sont jamais manipulées séparément.
 - Si une des colonnes était une clé, la manipulation doit être plus délicate.
- **Tables** : Si deux tables ont une relation One-To-One, les fusionner peut éviter des jointures répétitives et coûteuses. Dans les autres cas, c'est rarement intéressant.
 - Dans le cas de 2 tables avec une relation One-To-One, il y a une FK à éliminer.
 - Dans tous les cas, les vues, procédures et déclencheurs concernés devront être modifiés.
 - Mode opératoire : On choisit une table **hôte**, puis crée des nouvelles colonnes qu'on meuble avec un **INSERT SELECT** ensuite. On supprime l'autre table en dernier.



❖ Maintenance de la BD

◆ Séparation de colonnes ou de tables

- **Colonnes** : Améliorer la **granularité** / **précision** de certaines données. (Ex : découper une adresse en plusieurs colonnes au lieu d'un gros nvarchar mal standardisé)
 - Attention à ne pas **dupliquer** des données !
 - Mode opératoire : Potentiellement un **cauchemar**, surtout si la donnée était **composée** et **mal standardisée**. (Comme une adresse) Un script ou un logiciel tier peut devenir nécessaire pour restructurer les données et dans les pires cas, une intervention humaine, rangée par rangée.
 - COALESCE est une commande souvent utilisée ici dans un tel cas.
- **Tables** : Pour **normaliser** la BD, pour séparer des données avec des **niveaux de sécurité** différents ou pour séparer des données lourdes (ex : image) moins souvent utilisées que d'autres données simples pour améliorer la **performance**.
 - Mode opératoire : implique de créer une nouvelle table, de créer une clé étrangère, de meubler la nouvelle table avec **INSERT... SELECT** ... et de supprimer les anciennes colonnes de la table initiale.



❖ Maintenance de la BD

◆ Déplacer une colonne

- Généralement à des fins de normalisation. (Ou de dénormalisation pour réduire le nombre de jointures nécessaires, pour des cas très pointus)
 - Généralement simple si ce n'est pas une clé ! On crée la nouvelle colonne, on la meuble avec INSERT SELECT, puis on supprime l'ancienne colonne.

◆ Renommer une table, colonne ou une vue

- Peut sembler banal, mais si d'autres équipes ont du mal à identifier la nature de certaines tables, colonnes ou vues, les renommer et rendre leur usage plus intuitif est important.
 - Les correctifs à apporter sont généralement simples de toute façon.



❖ Maintenance de la BD

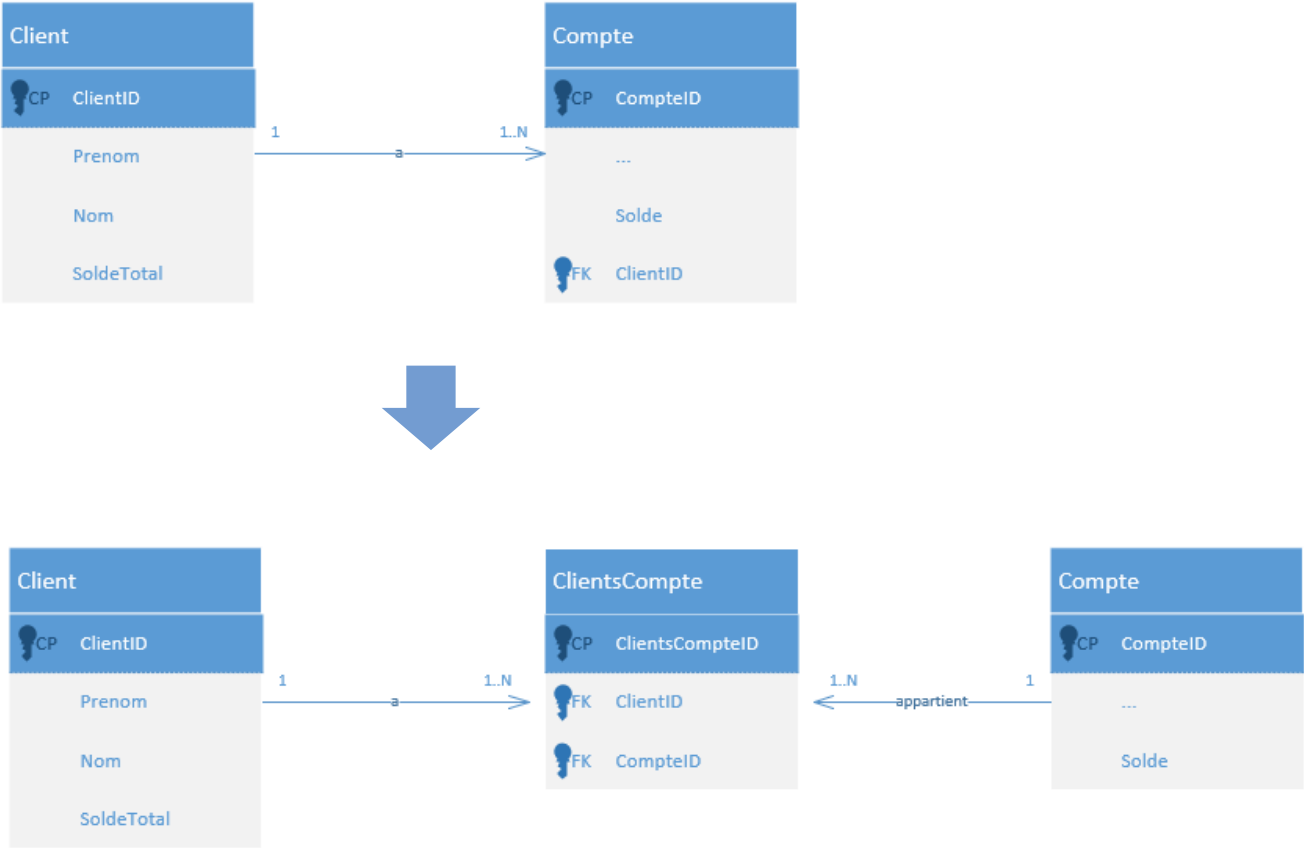
◆ De 1-N à N-M

- Modification plus fréquente qu'on pourrait croire. Parfois les besoins évoluent et un compte bancaire peut soudainement appartenir à plusieurs clients, par exemple.
 - Mode opératoire : Bien entendu, il faut ajouter une table associative et remanier les clés étrangères en faisant bien attention de maintenir les relations existantes. (**Exemple complet dans quelques diapos**)
 - Devrait-on toujours préconiser une table associative, même pour des relations **One-To-Many** pour ne pas avoir à faire cette modification plus tard ? **Non**. Si on est sûr que la relation restera toujours **One-To-Many**, éviter les tables associatives **prévient des jointures coûteuses**. Au pire, on fait la modification quand cela devient nécessaire.



❖ Maintenance de la BD

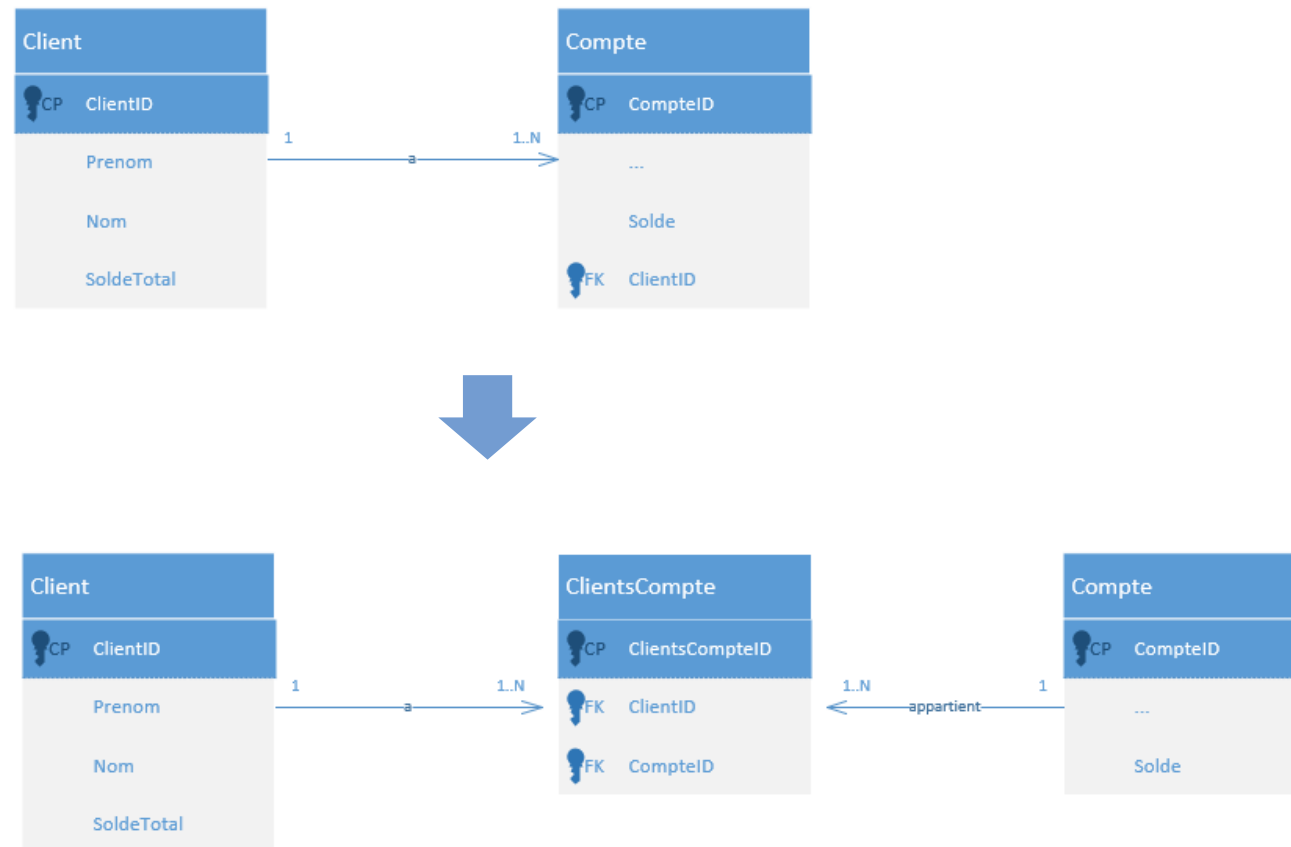
◆ De 1-N à N-M





❖ Maintenance de la BD

◆ Migration V2_1_2__ClientCompteManyToMany.sql





❖ Migrations

- ◆ Exemple : une migration pour créer une **table de liaison** et passer d'une relation **1-N** à **N-M**.
- ◆ N'oubliez pas de tester votre migration !
 - Testez vos vues, vos procédures, vos déclencheurs, etc.
 - Si tout fonctionne, la migration peut être officiellement ajoutée dans le dossier **Sql_Scripts** !

```
USE S09_Theorie
GO
-- Nouvelle table de liaison
CREATE TABLE Comptes.ClientsCompte (
    ClientsCompteID int IDENTITY NOT NULL,
    ClientID int NOT NULL,
    CompteID int NOT NULL,
    CONSTRAINT PK_ClientsCompte_ClientsCompteID PRIMARY KEY (ClientsCompteID)
);
GO

-- Nouvelles contraintes FK
ALTER TABLE Comptes.ClientsCompte ADD CONSTRAINT FK_ClientsCompte_ClientID
FOREIGN KEY (ClientID) REFERENCES Clients.Client(ClientID);
GO
ALTER TABLE Comptes.ClientsCompte ADD CONSTRAINT FK_ClientsCompte_CompteID
FOREIGN KEY (CompteID) REFERENCES Comptes.Compte(CompteID);
GO

-- Remplir la table avec les relations Existantes
INSERT INTO Comptes.ClientsCompte (ClientID, CompteID)
SELECT CL.ClientID, CO.CompteID
FROM Clients.Client CL
INNER JOIN Comptes.Compte CO
ON CL.ClientID = CO.ClientID

-- Enlever l'ancienne FK ClientID de la table Comptes.Compte
ALTER TABLE Comptes.Compte
DROP CONSTRAINT FK_Compte_ClientID
GO
-- Enlever le champs qui était l'ancienne FK ClientID de la table Comptes.Compte
ALTER TABLE Comptes.Compte
DROP COLUMN ClientID
```

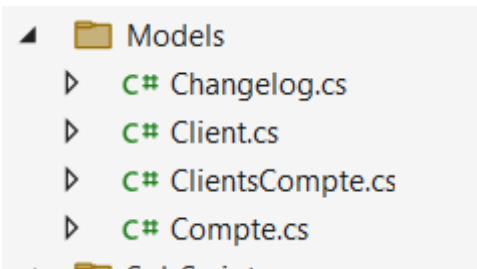


❖ Exécution de la migration:

```
evolve migrate sqlserver -c "Server=.;Initial Catalog=S09_Theorie;Integrated Security=True;Persist Security Info=False;Pooling=False;MultipleActiveResultSets=False;Encrypt=False;TrustServerCertificate=False" -s Clients -s Comptes --target-version 2.1.2
```

❖ Régénération des modèles puisque des tables de la BD ont changé:

```
dotnet ef dbcontext scaffold Name=S09_Theorie Microsoft.EntityFrameworkCore.SqlServer -o Models --context-dir Data --data-annotations --force
```





❖ Maintenance de la BD

◆ Changements de **qualité**

- Ex : changement de **type** d'une colonne, ajout d'une **contrainte**, retrait d'une contrainte, valeurs par **défaut**, valeurs **null** ou **non null**, etc.
- Ces changements ont des impacts plus subtils, mais pas inexistants ! Des exemples :
 - L'ajout ou le retrait d'une contrainte peut rendre des données **invalides** ou au contraire **élargir les cas à gérer** pour une vue, une procédure ou un déclencheur. Certaines valeurs dans les tables pourraient devoir être modifiées. Certaines valeurs *hardcodées* dans des clauses **WHERE** aussi.
 - Si une colonne obtient la possibilité de devenir **null**, des vues, procédures ou déclencheurs pourraient devoir être modifiées si certaines opérations **prenaient pour acquis que la colonne contenait une valeur**.



❖ Maintenance de la BD

◆ Changements à l'intégrité référentielle

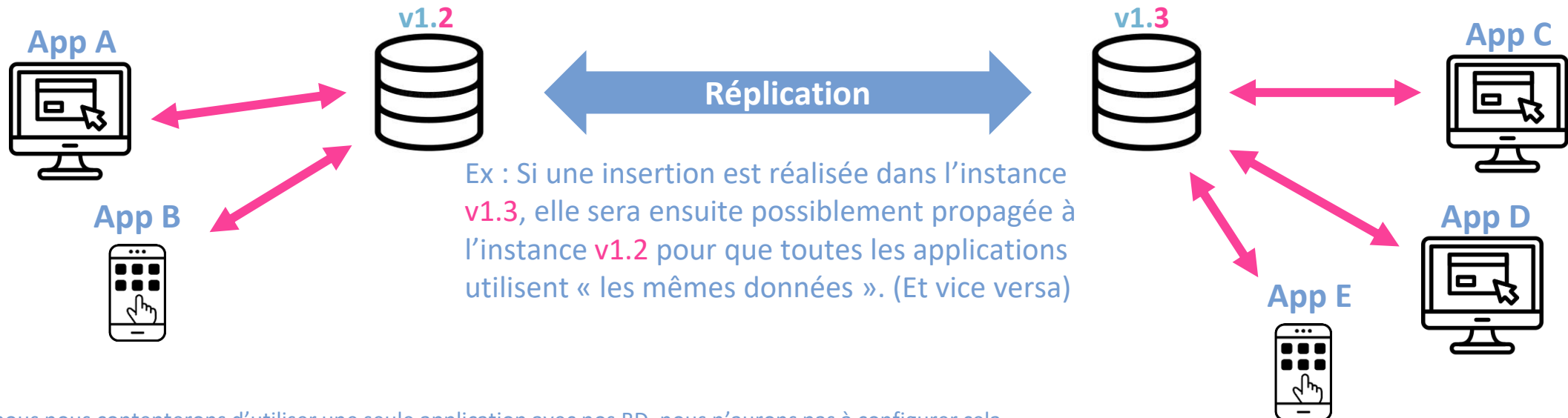
- Ex : Ajout ou retrait d'une contrainte FK, introduction d'un *soft delete*, introduction d'un DELETE / UPDATE en cascade, etc.
 - Nouvelle **contrainte FK** : Signifie que toutes les données existantes dans la table doivent au préalable avoir été liées avec une **PK**. Est-ce bel et bien le cas ? Sinon que faire des rangées **orphelines** ? La **FK** peut-elle être **NULL** sans problème ?
 - Introduction d'un **soft delete** : (C'est-à-dire l'ajout d'une colonne **EstSuppr bit NOT NULL**, par exemple) Peut être nécessaire s'il faut archiver certaines données. Cela dit, les vues, procédures et déclencheurs qui utilisent la table risquent d'avoir besoin de la clause **WHERE EstSuppr = 0** pour être sûrs de ne manipuler que les données « non supprimées ».
 - **DELETE en cascade** : Peut être **impossible** à cause de relations cycliques. (Implique de créer un déclencheur INSTEAD OF) Peut causer des suppressions en chaîne. (Ex : Une table qui possède la FK d'une autre table, qui possède la FK d'une autre table, qui possède la FK d'une autre table, etc.) Il faut donc s'assurer d'identifier où on souhaite que la cascade commence et s'arrête lorsqu'on remanie des contraintes FK.



❖ Maintenance de la BD

◆ Se soucier des applications qui communiquent avec la BD

- Inévitablement, lorsqu'on modifie la BD, les applications qui communiquent avec elle devront **s'adapter** également.
- L'équipe qui fait la maintenance de la BD a le **devoir** d'offrir un délai raisonnable aux autres équipes pour s'adapter à la BD. Pendant ce délai, la BD doit être disponible dans son ancienne version et dans sa nouvelle version.
 - Une **réplication des données** peut être mise en place entre les deux instances pour qu'elles partagent tout changement réalisé sur les données.

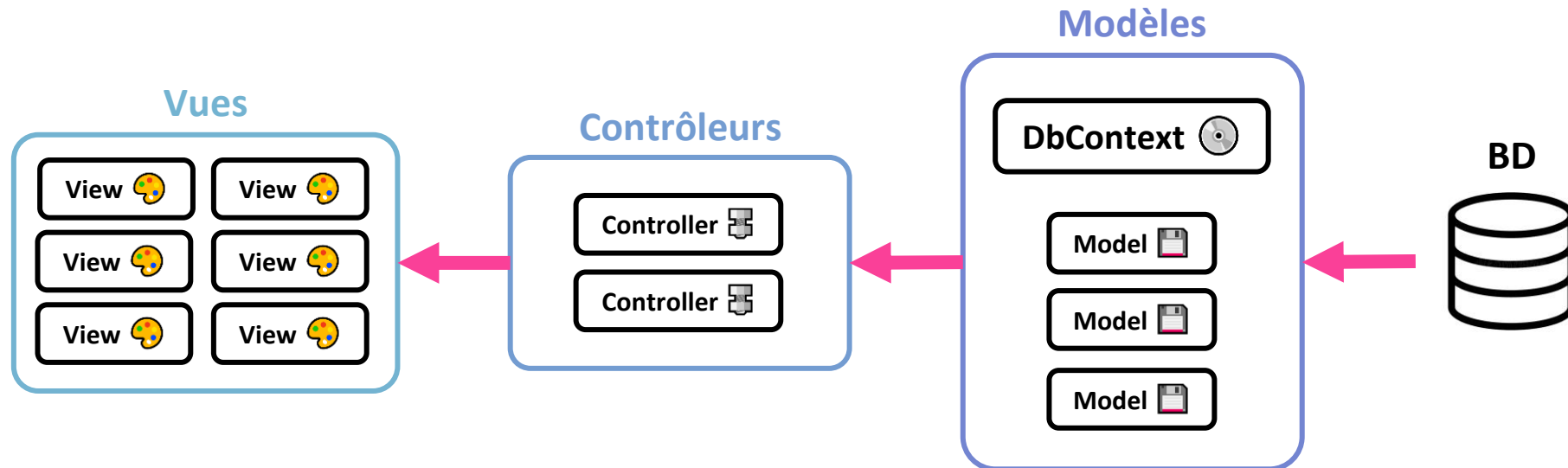




❖ Maintenance de l'application Web

◆ Intéressons-nous maintenant à l'application Web : lorsque la BD change, comment notre application ASP.NET Core MVC doit s'adapter ?

- Qu'est-ce qui peut **s'adapter automatiquement** ?
- Quelle évolution peut être **simplifiée** ?
- Qu'est-ce qui doit être **adapté manuellement** ?

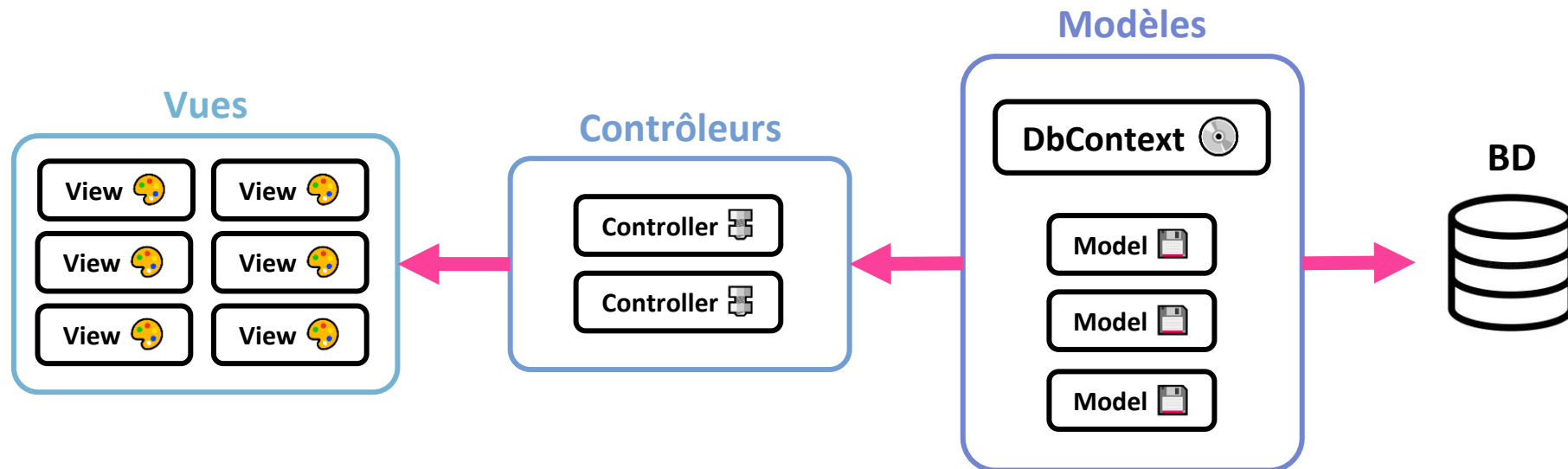


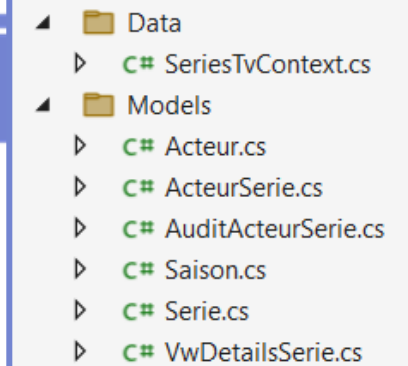


❖ Maintenance de l'application Web

◆ Perspective Code-First

- (Flèche inversée entre **Modèles** et **BD**) En **Code-First**, nous aurions modifié les **modèles** et nous aurions ensuite simplement généré de nouvelles **migrations** pour la BD à l'aide de quelques commandes. Les **contrôleurs** et les **vues** auraient ensuite été re-générés à l'aide d'assistants (wizards) ou modifiés manuellement, au besoin.
 - Précision : Les **migrations** sont parfois retravaillées à la main malgré tout pour mieux contrôler certaines évolutions. (Ou par exemple si d'autres applications dépendent de la même BD et que des problèmes sont anticipés)



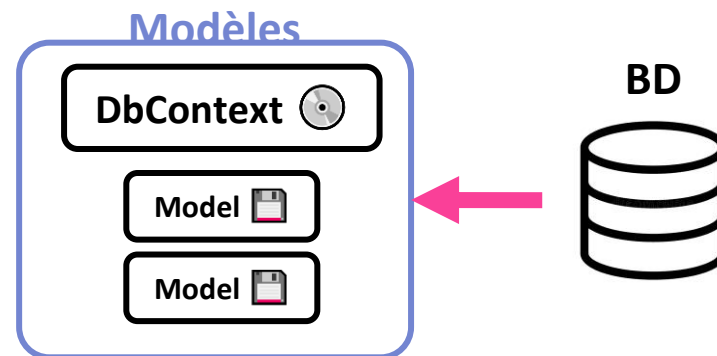


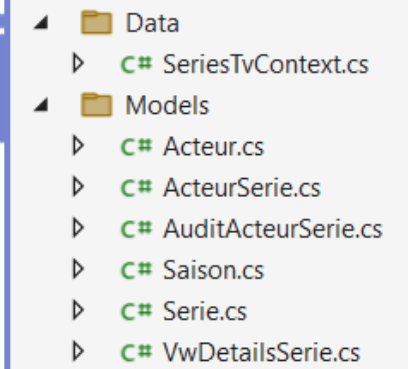
❖ Maintenance de l'application Web

◆ Perspective DB-First : Les modèles et le DbContext

- **Option 1** : Si nous n'avons fait aucune modification manuelle dans les **Models** et le **DbContext**, on peut tout simplement **réutiliser** la commande **dotnet ef dbcontext scaffold** pour tout re-générer les **Models** et le **DbContext** à l'image de la BD. Avec cette option, on a la certitude que les **Models** et la **BD** seront toujours parfaitement compatibles.

- On ajoute **-- force** pour forcer la reconstruction





❖ Maintenance de l'application Web

◆ Perspective DB-First : Les modèles et le DbContext

- **Option 2** : Si on désire conserver certains ajustements qui ont été faits manuellement dans les **Models** ou le **DbContext**, la *boîte de Pandore* est ouverte : on **peut** utiliser la commande, mais on doit d'abord **sauvegarder** les anciens Models pour pouvoir répliquer les ajustements qu'on a faits. On peut aussi utiliser les options **--context** et **--output-dir** pour changer le **nom** du **DbContext** et la **destination** des **Models** pour ne pas écraser les anciens. Il faut aussi s'assurer que nos ajustements personnalisés **respectent le nouvel état** de la **BD** et des **Models**.



❖ Maintenance de l'application Web

◆ Modifier les modèles : pourquoi ?

- Nous favoriserons l'**option 1** dans le cours : **on ne touche pas aux Models et au DbContext !** 🚫 De cette manière, on peut au moins garantir de ne jamais avoir à nous occuper nous-mêmes de cette partie.
- Voici un exemple de changement manuel qui pourrait nous pousser à utiliser l'**option 2** :
 - Entity Framework n'est pas parfait : ce ne sont pas toutes les contraintes SQL qui sont répliquées dans les Models. La contrainte CHECK ci-dessous ne sera pas répliquée dans le projet ASP.NET Core :

```
CHECK (TypePersonnalite IN ('A', 'B'));
```

On a donc deux opportunités : L'ajouter nous-mêmes en **DataAnnotation** ou utiliser **Fluent API** dans le DbContext. Dans les deux cas, les modifications se feront écraser au prochain dotnet ef dbcontext scaffold.

```
[Required]
[StringLength(1)]
[RegularExpression("[AB]")]
public string Type { get; set; }
```

DataAnnotation

```
modelBuilder.Entity<User>()
    .Property(e => e.Type).IsRequired()
    .HasMaxLength(1).IsFixedLength()
    .HasConversion(v => v.ToString(), v => (v == "A" || v == "B") ?
v : throw new ArgumentException("Le type doit être A ou B"));
```

Fluent API



❖ Maintenance de l'application Web

◆ Modifier les modèles : pourquoi ?

```
CHECK (TypePersonnalite IN ('A', 'B'));
```

- Et si on ne réplique pas la contrainte CHECK dans ASP.NET Core ? **Pas la fin du monde**, mais il faut se préparer à ce que les opérations sur la BD génèrent des **exceptions SQL**.

- Seulement utiliser de la **validation côté BD** (et donc pas **côté Serveur** ni **Client**) permet tout de même d'assurer l'**intégrité des données**.

- Cela dit, si l'application Web n'est pas capable d'**identifier clairement quelle contrainte n'est pas respectée** et que l'utilisateur ne sait pas ce qu'il doit changer dans le formulaire, c'est problématique.

- En résumé : dans une situation optimale, on ajouterait des **[DataAnnotation]** ou des instructions avec **Fluent API** pour compenser les **contraintes manquantes**. Dans ce cours, nous ne nous attarderons pas sur cela.

```
try
{
    // INSERTION dans un DbSet
    // _context.Client.Add( ... );

    await _context.SaveChangesAsync();
}
catch (DbUpdateException e)
{
    // Vérifier si c'est une exception SQL
    if (e.InnerException is SqlException)
    {
        // Message d'erreur
        ModelState.AddModelError(string.Empty, "Une contrainte SQL n'a pas été respectée");

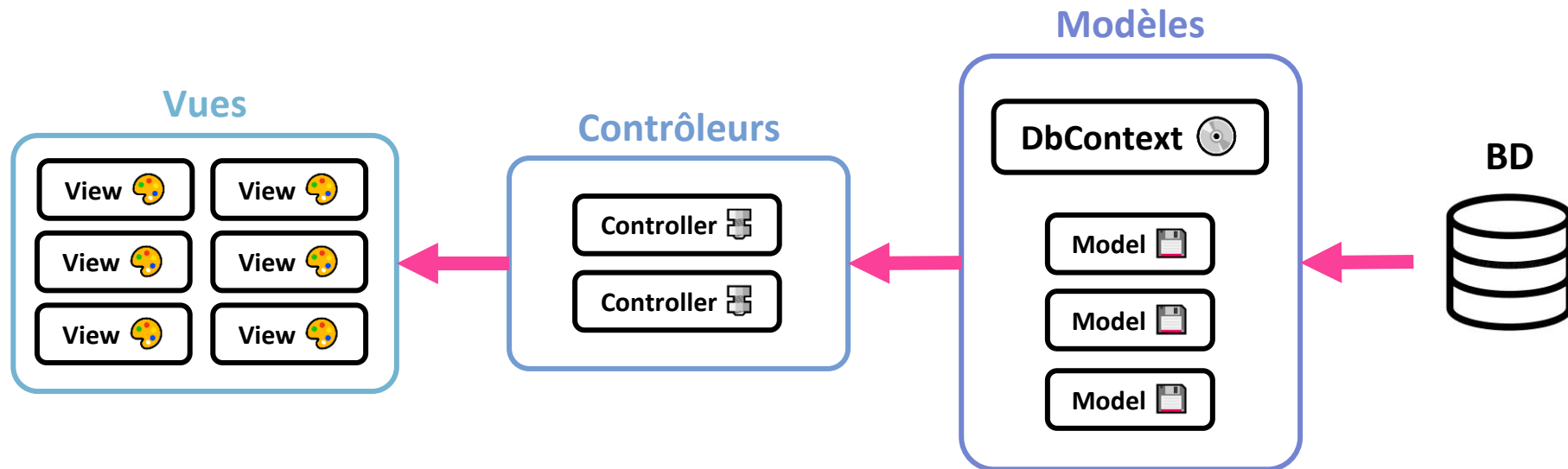
        // Retourner la vue avec les erreurs de validation
        return View(model);
    }
    else
    {
        // Autres types d'exception
        throw;
    }
}
```



❖ Maintenance de l'application Web

◆ Perspective DB-First : Contrôleurs et vues

- Généralement, on doit faire des modifications **à la main**. Pas le choix, comme en **Code-First** !
- Si de tout nouveaux Models ont été ajoutés, on peut aussi profiter de la génération de contrôleurs et de vues avec les assistants de Visual Studio.



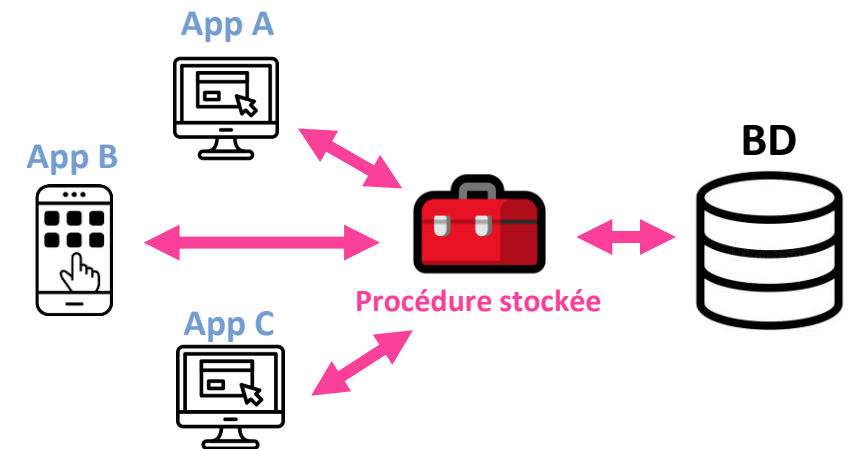


❖ Maintenance de l'application Web

◆ Perspective DB-First : **Procédures stockées** et **vues**

- Avec les procédures stockées et les vues, on a un avantage par rapport au Code-First !
- Si une action du contrôleur appelle une **procédure stockée** (au lieu de manipuler un **DbSet** manuellement) et qu'on a déjà **corrigé** / **modifié** cette procédure en faisant les migrations... on risque d'avoir beaucoup moins de travail pour mettre à jour le code !
 - Ce n'est pas tout : imaginez que plusieurs applications utilisent la même **BD** et particulièrement la même **procédure**... Mettre à jour cette procédure **simplifie la maintenance** de TOUTES les applications.

- Okay. Est-ce qu'on remplace tout le code par des appels de **procédures stockées** ?
-> **Non**, bien entendu. Une opération **simple** / **atomique** (ex : Un INSERT d'une seule rangée, un DELETE d'une seule rangée, etc.) ne mérite pas vraiment une **procédure** stockée et, étant une opération simple, cela sera déjà très simple à corriger lors de la maintenance de l'application.





❖ Maintenance de l'application Web

◆ Parenthèse : Les **services**

- En temps normal, on ajouterait une couche entre les **contrôleurs** et le **DbContext** : des classes « **Services** » qui **encapsulent les opérations sur la BD**.
- Ce sont dans les **services** que nous devrions retrouver les opérations sur les **DbSet** et les appels de **procédures stockées**.
 - Toutefois, comme vous pratiquerez amplement les **services** dans les autres cours de programmation Web, nous nous concentrerons sur autre chose dans ce cours.
 - On **injecte** donc toujours directement le **DbContext** dans les **contrôleurs**.
 - Rien ne vous empêche d'utiliser des **services** si vous préférez.

