

Objectifs : Réaliser une application WPF en appliquant les bonnes pratiques suivantes :

- **En utilisant une architecture à 3 couches**
- **Basé sur l'utilisation de vues SQL**
- **Exécutant des procédures stockées**

L'application permet de simuler un distributeur automatique de billets sur lequel on peut déposer ou retirer de l'argent :

Dépôt

Type d'opération: Dépôt

Compte: 1234567

Montant: 5000

Valider
Annuler

Retrait

Type d'opération: Retrait

Compte: 1234567

Montant: 500

Valider
Annuler

Vous n'êtes pas obligé d'utiliser material design.

Comme il y a 2 applications à réaliser (Cf. Q7), vous créerez 1 seule Window par application (pas de User control).

1- Création de la base de données

Sous PostgreSQL, créer une base de données nommée `BDComptesBancaires`.

Y exécuter le script `Script ComptesBancaires PostgreSQL.sql`.

2 tables sont créées (`Compte` et `Virement`) ainsi qu'une vue.

Compte		
<u>idCompte</u>	int	<pk>
solde	numeric(10,2)	

Virement		
<u>idTransaction</u>	int	<pk>
idCompteDebit	int	
idCompteCredit	int	
dateTransaction	datetime	
Montant	numeric(10,2)	

vComptes	
idCompte	
solde	
<input type="checkbox"/> Compte	

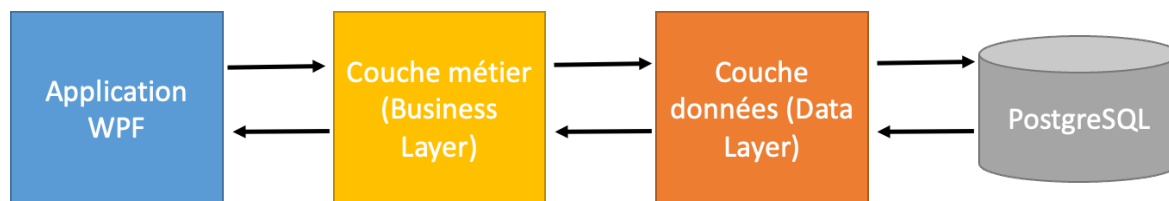
BONNE PRATIQUE : Les vues seront utilisées en lecture (`SELECT`). Comme indiqué en CM, il est préférable d'exécuter des requêtes sur des vues, plutôt que directement sur des tables (moins de problèmes de verrous => critère de performance ; découplage entre l'application et la base => critère de maintenabilité).

De même en écriture (*insert, update, delete*), il est préférable d'utiliser des procédures stockées pour éviter l'injection SQL (Cf. dernière section) ou des requêtes paramétrées (<https://webman.developpez.com/articles/aspnet/sqlparameter/csharp/>).

2- Création des projets

L'application à réaliser est structurée en 3 couches (3-tiers) :

1. `DataLayer` : classes d'accès aux données.
2. `BusinessLayer` contenant les objets métier et accédant à la couche 1.
3. `WpfComptesBancaires` représentant l'application WPF accédant à la couche 2.



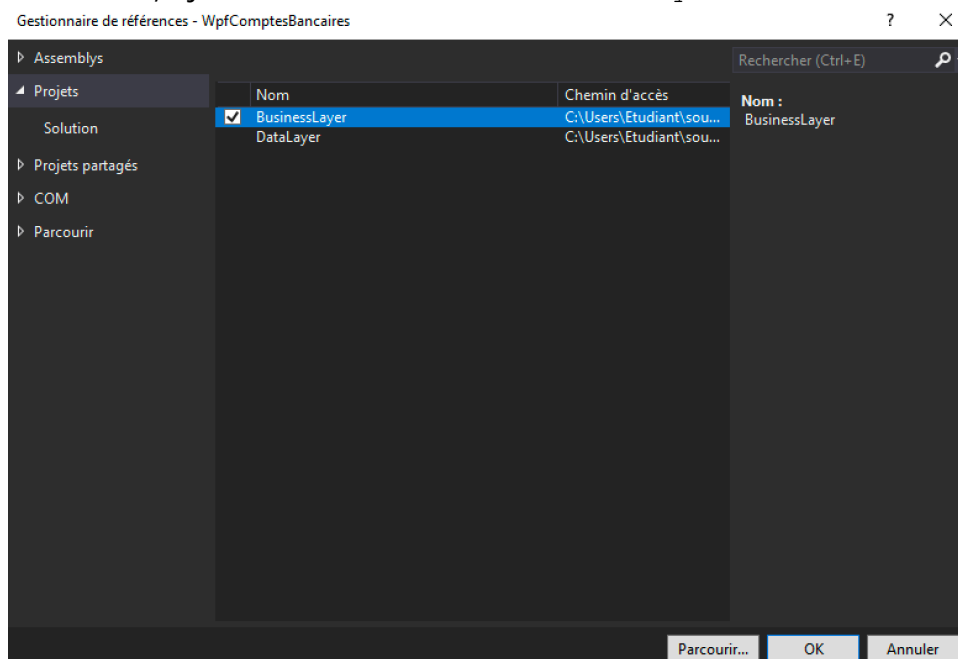
BONNE PRATIQUE : Il est préconisé d'appliquer le patron MVVM pour développer une application utilisant une base de données ou à défaut de la structurer comme ci-dessus.

Créer un projet « Application WPF » nommé `WpfComptesBancaires` et sa solution **sur le bureau**.

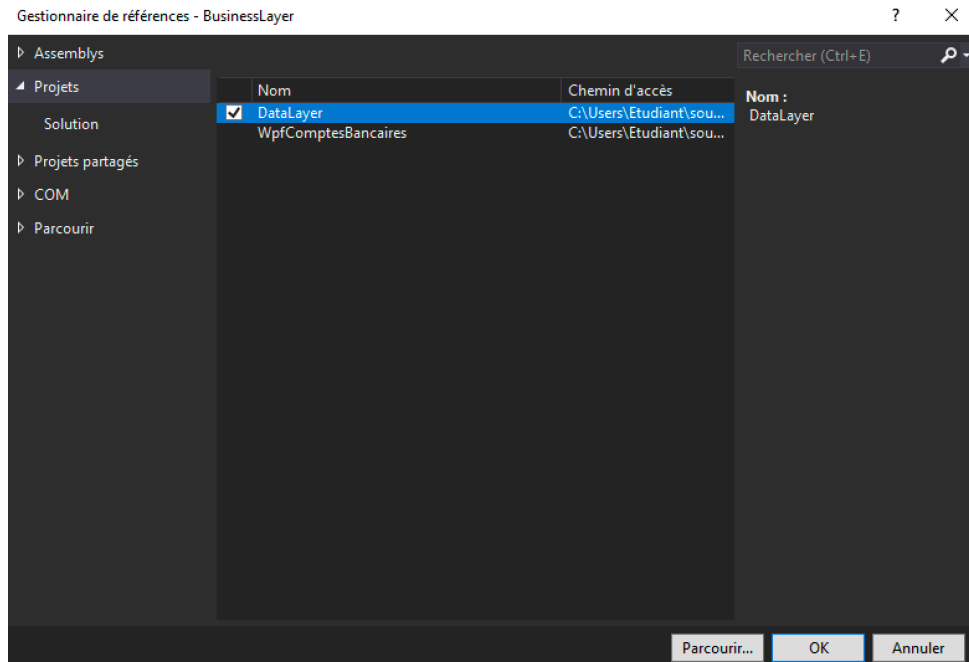
Ajouter un projet `DataLayer` de type « Bibliothèque de classes **WPF** » à la solution.

Ajouter un projet `BusinessLayer` de type « Bibliothèque de classes **WPF** » à la solution.

Dans `WpfComptesBancaires`, ajouter la référence vers `BusinessLayer`.



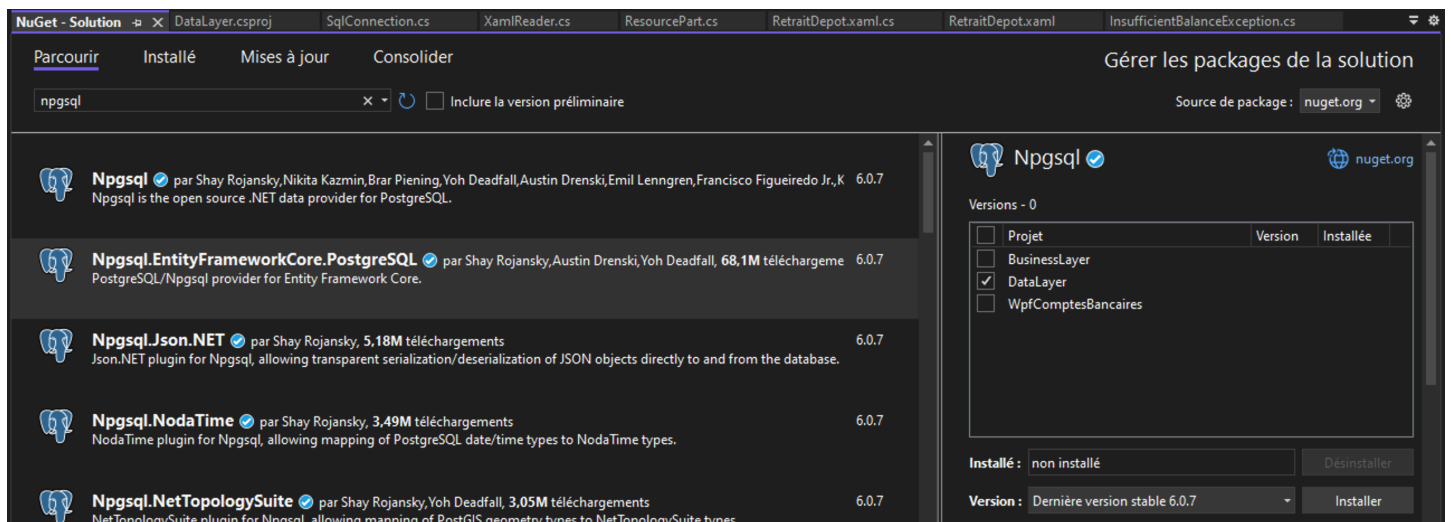
Dans `BusinessLayer`, ajouter la référence vers `DataLayer`.



Exécuter l'application.

3- Développement de la couche DataLayer

- Supprimer la classe créée.
- Installer le package Nuget Npgsql



- Cliquer avec le bouton droit de la souris sur le projet DataLayer puis *Ajouter > Élément existant* et ajouter le fichier DataAccess.cs disponible sur le serveur.

ATTENTION au namespace du fichier DataAccess, si vous n'avez pas respecté le nom du projet.

```

1  using Npgsql;
2  using System;
3  using System.Data;
4
5
6  namespace DataLayer
7  {
8      public class DataAccess // A MODIFIER SI VOTRE PROJET A UN AUTRE NOM
9      {
10         public NpgsqlConnection? NpgsqlConnect { get; set; }
11
12         /// <summary>
13         /// Connexion à la base de données
14         /// </summary>
15         /// <returns> Retourne un booléen indiquant si la connexion est ouverte ou fermée</returns>
16         private bool OpenConnection()
17         {
18             try
19             {
20                 NpgsqlConnect = new NpgsqlConnection
21                 {
22                     ConnectionString = "Server=localhost;port=5432;Database=BDComptesBancaires;uid=postgres;password=pos
23                 };
24                 NpgsqlConnect.Open();
25                 return NpgsqlConnect.State.Equals(System.Data.ConnectionState.Open);
26             }
27             catch
28             {
29
30             }
31         }
32     }
33 }

```

Plus de détails sur les classes connexion et commande PostgreSQL :

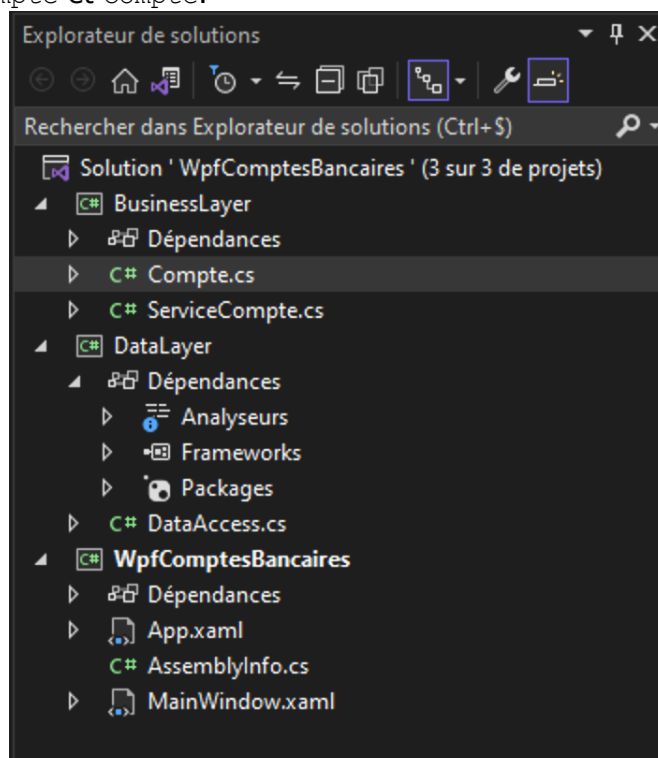
<https://learn.microsoft.com/fr-fr/azure/postgresql/single-server/connect-csharp>

Remarque : la classe d'accès aux données utilise le mode déconnecté pour récupérer les données des tables (méthode `GetData`) : on se connecte à la base de données, on récupère les données dans une table de données locale (`DataTable`) puis on se déconnecte. On pourrait aussi utiliser le mode connecté (comme l'an dernier en SAE).

4- Développement de la couche BusinessLayer

Supprimer la classe créée.

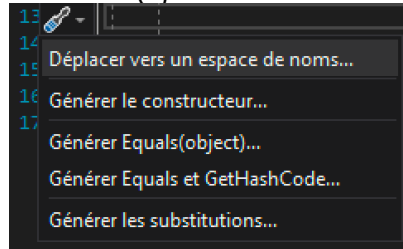
Créer les classes `ServiceCompte` et `Compte`.



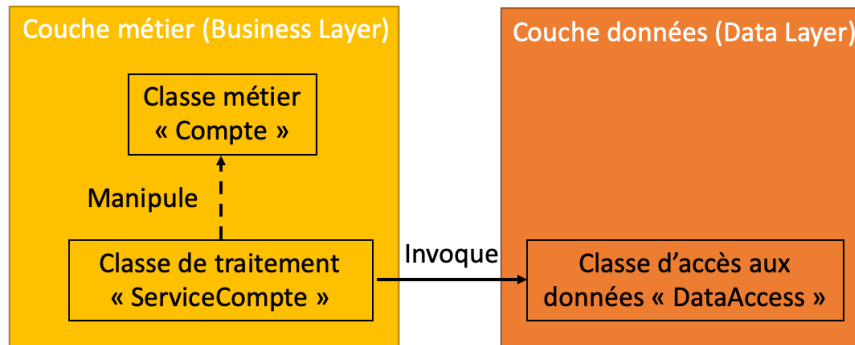
Dans la classe métier `Compte` (il s'agit d'une **classe "normale"** qui va représenter une entité), créer les *property* correspondant aux champs de la vue `vComptes` (types `int` et `double`), un constructeur sans paramètre et un constructeur paramétré (2 paramètres de type `int` et `double`).

Rappels :

- Snipet pour générer un constructeur sans paramètre : `ctor`
- Pour générer un constructeur paramétré (après avoir créé les *properties*) :



La classe `ServiceCompte` accède à la couche `DataLayer` et manipule des objets `Compte`.



Coder pour le moment la méthode `public List<Compte>? GetAllComptes()` permettant de récupérer tous les comptes de la BD :

```
/// <summary>
/// Récupère la liste de tous les comptes (id et solde) de la base de données
/// </summary>
/// <returns>Liste des comptes bancaires</returns>
public List<Compte>? GetAllComptes()
{
    ...
}
```

- Exécute la méthode `GetData()` sur un objet de la classe `DataAccess` et stocke le résultat dans un objet `DataTable` local. La requête qui sera passée en paramètre de la méthode `GetData()` portera sur la vue `vComptes` (pas de ; en fin de commande SQL).
- Une boucle récupère chaque ligne du `DataTable` (collection `monDataTable.rows`) et crée un objet `Compte` à partir de chaque champ de la ligne : [http://msdn.microsoft.com/fr-fr/library/system.data.datatable.rows\(v=vs.110\).aspx](http://msdn.microsoft.com/fr-fr/library/system.data.datatable.rows(v=vs.110).aspx). Chaque objet est ensuite ajouté à une liste d'objets `Compte`. Penser à caster les valeurs des champs en utilisant la classe statique `Convert` (ou les méthodes `Parse/TryParse` sur les classes `int` et `double`)
- La liste est retournée.
- Penser à utiliser un `try catch` et à lever une exception en cas d'erreur.

5- Développement de l'application WPF

Coder l'application WPF.

Indications :

- Pour simplifier, définir le `DataContext` sur la fenêtre (Cf. TD1 ou CM).
- Liste déroulante des types d'opération :

Type d'opération

Type d'opération
Retrait
Dépôt

- Créer une property de type `ObservableCollection<string>`.
- Dans le constructeur, initialiser la collection aux valeurs *Retrait* et *Dépôt* (utiliser la méthode `Add` de votre *ObservableCollection*)
- Binder l'attribut XAML `ItemsSource` de la `ComboBox` à la property.
- Liste déroulante des comptes :
 - Vous devez savoir faire... Cf. TD2
- Montant :
 - Vous devez savoir faire...
- Bouton « Valider » :
 - Pour le moment, coder uniquement un affichage pour tester :

Affichage pour tester



Opération : Dépôt / Compte : 2345678 / Montant : 200

OK

- **Il faudra créer autant de propriétés que de « valeurs » récupérées dans l'IHM.**

Exécuter l'application.

Pour pouvoir réaliser le débit, il faut ajouter la méthode `SetDebitCredit` dans la classe `ServiceCompte`.

```

/// <summary>
/// Met à jour le solde du compte passé en paramètre en fonction du montant passé en
paramètre. Si montant<0 => débit, sinon crédit.
/// </summary>
/// <param name="compte">Compte à débiter ou créditer</param>
/// <param name="montant">Montant du débit (si <0) ou crédit (si >0)</param>
/// <returns>Résultat de la mise à jour (update) du solde du compte : true => réussi,
false => échec</returns>
public bool SetDebitCredit(Compte compte, double montant)
{
    ...
}

```

Cette méthode appelle la méthode `SetData` de la classe `DataAccess` et lui passe en paramètre un ordre *update* permettant de mettre à jour le solde (débit ou crédit) du compte sélectionné dans la base de données. Si `montant` est positif, il s'agit d'un crédit ; un débit sinon. Si la mise à jour a eu lieu, `SetData` retourne 1 (une ligne mise à jour) et, dans ce cas, `SetDebitCredit` retourne `true` (`false` sinon).

Ajouter ensuite le code du bouton « Valider ». Afficher un message si l'opération (retrait / dépôt) a bien été effectuée.

Compte bancaire



Dépôt effectué

OK

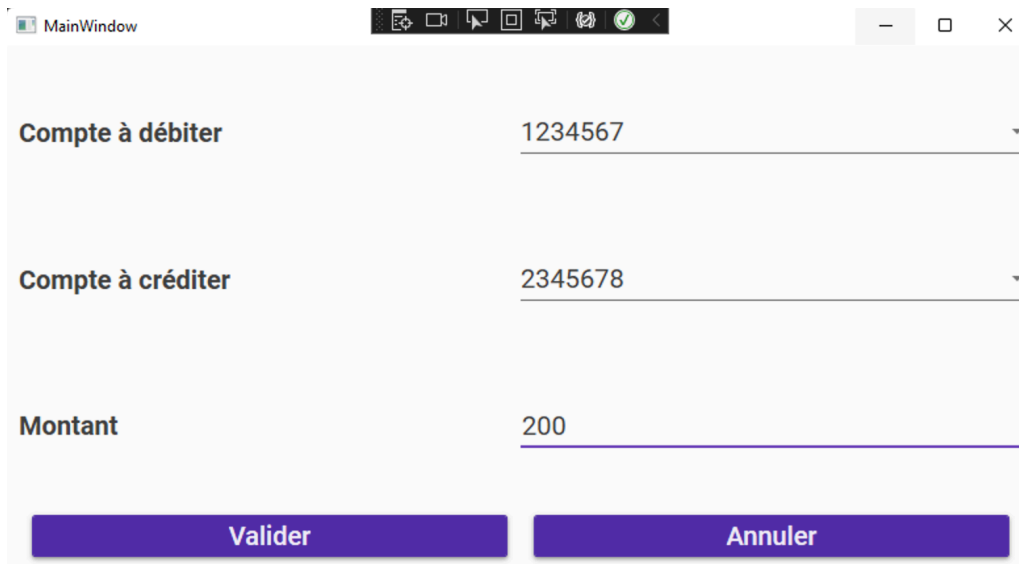
Vérifier dans la base de données la mise à jour du solde.

Si l'opération bancaire n'a pas été effectuée ou si les données (compte, type d'opération) n'ont pas été sélectionnées afficher des messages d'erreur.

6- Codage du bouton « Annuler » de l'application WPF

Le montant est remis à 0. Les combobox sont remises à « vide ».

7- Virement



Ajouter un nouveau projet WPF nommé `WpfVirement`.

Comme nous avons (bien) codé l'application précédente en couches, ce nouveau projet utilisera également les couches `BusinessLayer` et `DataLayer`.

Nous aurions également pu ajouter une nouvelle page au projet `WpfComptesBancaires`, mais il s'agit ici de démontrer que notre application a été bien architecturée...

Ajouter une référence à `BusinessLayer`.

Réaliser le code du virement dans `BusinessLayer`.

Indications :

- Squelette de la méthode de virement de `BusinessLayer` :

```
/// <summary>
/// Réalise un virement du compte en paramètre n°1 vers le compte en paramètre
n°2 d'un montant en paramètre n°3
/// </summary>
/// <param name="compteDebit">Compte à débiter</param>
/// <param name="compteCredit">Compte à créditer</param>
/// <param name="montant">Montant du virement</param>
/// <returns>Résultat de la mise à jour (update) des soldes des 2 comptes :
true => réussi (nombre de lignes modifiées par SetData=2), false => échec</returns>
public bool Virement (Compte compteDebit, Compte compteCredit, double
montant)
{
}
```

`true` n'est renvoyé que si 2 lignes ont été mises à jour.

- Exemple de commande SQL à envoyer en un seul appel au SGBD (création d'une transaction composée de 2 update) :

```
Begin; Update compte set solde = solde-100 where idcompte=1234567; Update compte
set solde = solde+100 where idcompte=2345678; commit;
```

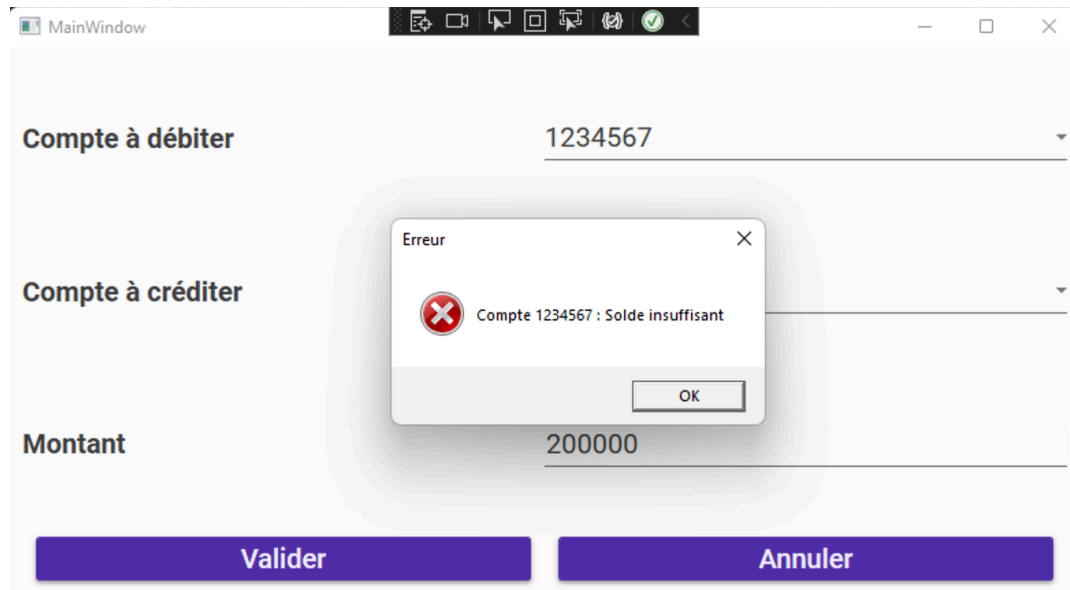
Query	Query History
1	Begin;
2	Update compte set solde = solde-100 where idcompte=1234567;
3	Update compte set solde = solde+100 where idcompte=2345678;
4	commit;

Ce code SQL a été vu l'an dernier...

8- Améliorations

1. Le client n'a pas d'autorisation de découvert (débit et virement impossibles). Modifier le code des méthodes `SetDebitCredit` et `Virement` en conséquence. Vous lèverez une exception de type `ArgumentException` ou mieux votre propre exception :
<https://learn.microsoft.com/fr-fr/dotnet/standard/exceptions/how-to-create-user-defined-exceptions>
Ajouter un bloc `try/catch` dans les applications WPF afin d'afficher un message d'erreur.

```
try
{
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message, "Erreur", MessageBoxButton.OK, MessageBoxImage.Error);
}
```



Pour les plus rapides :

A VENIR...