22/01/2015

Déborah Lurienne

Groupe eCOLE PRATIQUE

Documentation technique

GSB Hôpital

Sommaire

[1. Contexte 2](#_Toc416447559)

[2. L’environnement de développement 2](#_Toc416447560)

[3. Modélisation des exigences 2](#_Toc416447561)

[4. Persistance des données 3](#_Toc416447562)

[a. MCD 3](#_Toc416447563)

[b. Modèle relationnel 3](#_Toc416447564)

[c. Création de la base et des tables 4](#_Toc416447565)

[d. Création des contraintes 4](#_Toc416447566)

[e. Création des procédures stockées 4](#_Toc416447567)

[f. Appel des procédures stockées 5](#_Toc416447568)

[5. La chaîne de connexion 5](#_Toc416447569)

[6. IHM 7](#_Toc416447570)

[7. Code behind 8](#_Toc416447571)

[9. Architecture applicative 10](#_Toc416447572)

[10. LINQ 11](#_Toc416447573)

[a. Sugar Syntax 11](#_Toc416447574)

[b. Méthode d’extension 11](#_Toc416447575)

[11. Pattern Singleton 12](#_Toc416447576)

[a. Principe des Design Patterns 12](#_Toc416447577)

[b. Le Design Pattern Singleton 12](#_Toc416447578)

[c. Mise en œuvre 12](#_Toc416447579)

[12. Tests unitaires 13](#_Toc416447580)

[13. Conclusion 15](#_Toc416447581)

[14. Annexes : Procédures stockées 16](#_Toc416447582)

# Contexte

Le laboratoire GSB a besoin une application de gestion des commandes pour fournir en médicaments les visiteurs médicaux relevant d'une même Région et affectés à temps plein en milieu hospitalier, il est donc demandé de la développer.

Les objectifs attendus :

* Création d’une base de données
* Elaborer des procédures stockées
* Connecter la base au projet
* Concevoir l’application

# L’environnement de développement

**Langage utilisé** : C#

**Logiciels** : Visual Studio 2013, Server SQL 2014, framework .NET 4.5

**Versionning** : Github.com / Git

# Modélisation des exigences

Editer ou annuler

une commande

Ajouter une commande

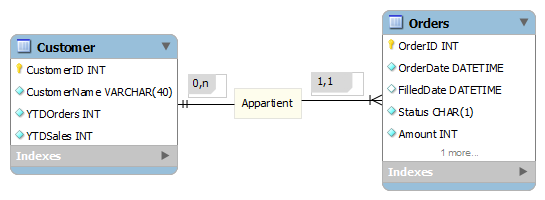
Ajouter un compte

**Application de gestion des commandes**

Visiteur

# Persistance des données

## MCD



Appartient

## Modèle relationnel

Orders

OrderID

*CustomerID*

OrderDate

FilledDate

Status

Amount

Customer

CustomerID

CustomerName

YTDOrders

YTDSales

Customer (CustomerID, CustomerName, YTDOrders, YTDSales)

* CustomerID : clé primaire

Orders (OrderID, *CustomerID*, OrderDate, FilledDate, Status, Amount)

* OrderID : clé primaire
* *CustomerID* : clé étrangère en référence à
* CUSTOMER

## Création de la base et des tables

CREATE TABLE Customer (

CustomerID INT IDENTITY(1,1) NOT NULL,

CustomerName NVARCHAR(40) NOT NULL,

YTDOrders INT NOT NULL,

YTDSales INT NOT NULL

);

CREATE TABLE Orders (

CustomerID INT NOT NULL,

OrderID INT IDENTITY(1,1) NOT NULL,

OrderDate DATETIME NOT NULL,

FilledDate DATETIME NULL,

Status CHAR(1) NOT NULL,

Amount INT NOT NULL

);

## Création des contraintes

ALTER TABLE Customer

ADD CONSTRAINT PK\_Customer PRIMARY KEY(CustomerID);

ALTER TABLE Orders

ADD CONSTRAINT PK\_Orders PRIMARY KEY(OrderID) ;

ADD CONSTRAINT FK\_Orders\_Customer FOREIGN KEY (CustomerID)

REFERENCES Customer(CustomerID)

ALTER TABLE Customer

ADD CONSTRAINT Def\_Customer\_YTDOrders DEFAULT 0 FOR YTDOrders;

ALTER TABLE Customer

ADD CONSTRAINT Def\_Customer\_YTDSales DEFAULT 0 FOR YTDSales;

ALTER TABLE Orders

## Création des procédures stockées

|  |  |
| --- | --- |
| 1. **Nom** | uspNewCustomer |
| **Rôle** | Permet d’insérer un nouveau client dans la table Customer. Renvoie l’ID du client inséré. |
| **Paramètres** | Paramètre : @CustomerName (entré, NVARCHAR(40), pas de valeur par défaut)  Paramètre : @CustomerId (entré/sortie, INT, pas de valeur par défaut) |
| CREATE PROCEDURE uspNewCustomer  @CustomerName NVARCHAR (40),  @CustomerID INT OUTPUT  AS  BEGIN  INSERT INTO Customer(CustomerName) VALUES (@CustomerName);  SET @CustomerID = SCOPE\_IDENTITY();  RETURN @@ERROR  END | |

## Appel des procédures stockées

DECLARE @ID1 INT;

exec uspNewCustomer 'Hopital St Jacques', @CustomerID = @ID1 OUTPUT;

SELECT @ID1;

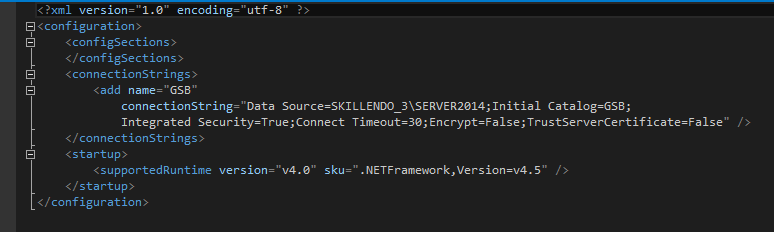
DECLARE @ID2 INT;

exec @ID2 = uspPlaceNewOrder 2, 300, '04/11/2014'

SELECT @ID2;

# La chaîne de connexion

Voici le script de « App.config » qui permet la connexion à la base de données :



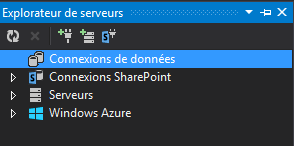
Chaine de connexion

L’intérêt d’externaliser une chaîne de connexion et d’assurer une sécurité, cela évite également de recompiler l’application en cas de changement de paramètre.

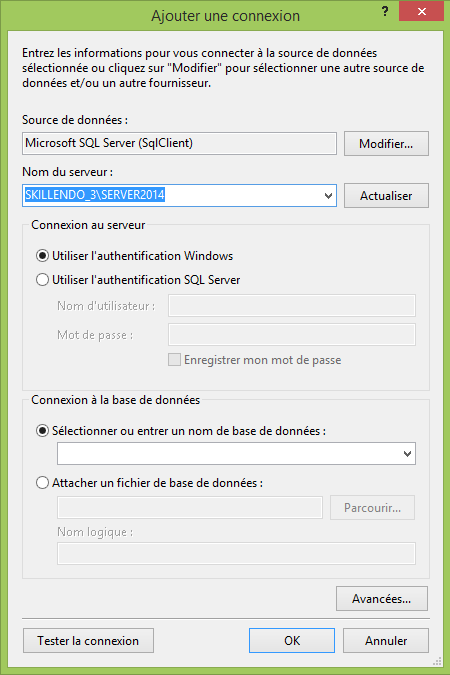
Cette chaine peut être générée par Visual Studio en allant dans :

AFFICHAGE -> EXPLORATEUR DE SERVEUR

Un volet s’affiche :



Clique droit -> Ajouter une connexion…

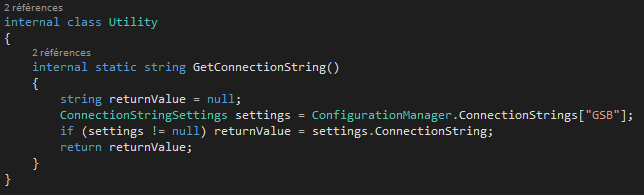


Générer la chaîne de connexion dans App.config

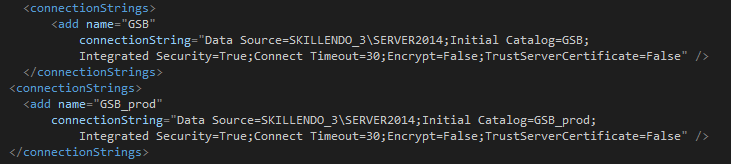
Teste la connexion

Renseigne le nom du serveur

Ensuite cette chaine de connexion peut être récupérée par une class Helper :



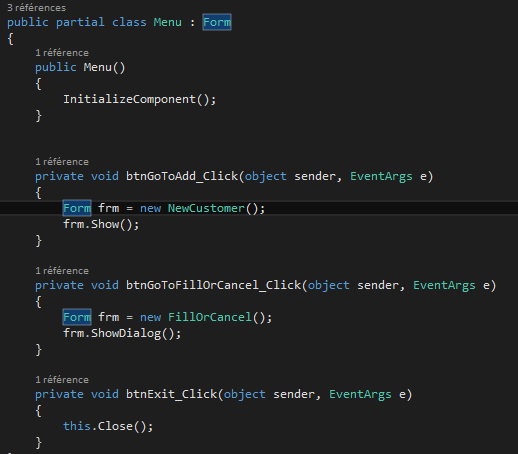
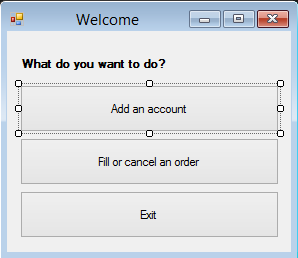
Il est possible de créer une seconde chaîne pour server SQL de production :



# IHM

Les classes partielles sont utiles dans plusieurs situations, notamment lorsqu’on travaille sur de grands projets, séparer une classe en plusieurs fichiers permet à plusieurs personnes de travailler en même temps, ainsi un designer peut interagir sur la classe gérant la vue pendant que le développeur touche le code métier.

Visual Studio génère lui-même des classes partielles pour gérer les Windows Forms, cela permet de pouvoir modifier la classe sans devoir recréer le fichier source ce qui fait un gagne un temps énorme pour les développeurs ainsi qu’un code plus organisé.



Class partielle générée par

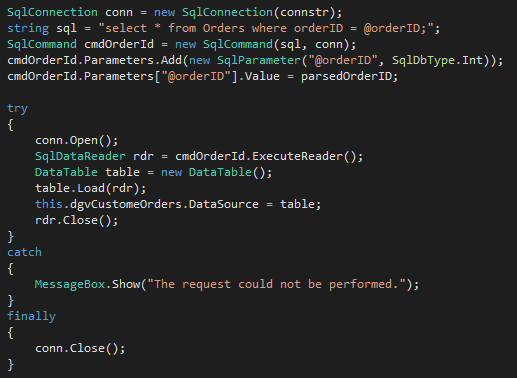
Visual Studio

Et chaque action sur un bouton du formulaire est représentée par une fonction générée par Visual Studio.

# Code behind

Pour exécuter une requête sql, cela se présente ainsi :

1



7

6

5

4

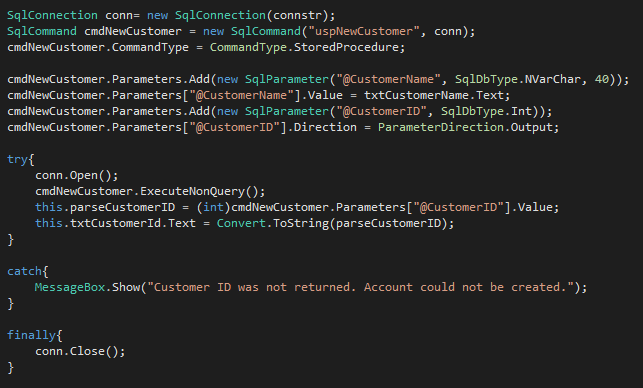
3

2

1. On donne la connexion.
2. On lui renseigne la requête sql.
3. On lui donne les valeurs des éventuels paramètres passés.
4. On ouvre la connexion.
5. On exécute la requête.
6. S’il y a des erreurs on renvoie un message.
7. Une fois finis on ferme la connexion.

On renseigne le nom de la procédure ici

Pour exécuter une procédure stockée :



On récupère ce qui est en sortie

On renseigne déclare et renseigne les paramètres d’entrés et de sorties

Lorsqu’une application doit exécuter une requête dans une base de données, il est préférable que celle-ci appelle une procédure stockée plutôt que d’envoyer une commande SQL construite dans le code de l’application.

Outre l’avantage de sécurité contre les attaques par injection et la maintenabilité du code, voici les autres avantages que cela procure :

– Dès lors qu’on appelle une procédure stockée, on a simplement besoin de spécifier son nom et de lui adjoindre la valeur de ses paramètres.

C’est donc moins coûteux en termes de quantité de données qu’une commande SQL complète à envoyer à l’instance SQL Server, si courte soit elle.

Dès lors cela réduit le trafic réseau entre les applications et le serveur.

– Lorsqu’on crée une procédure stockée, celle-ci est compilée en un plan qui demeure dans le cache de procédures, ce qui réduit considérablement le coût de calcul du plan d’une requête, gourmand en ressources processeur.

– L’exécution de requêtes par l’instance elle-même peut s’avérer moins coûteuse pour l’application mais aussi pour le serveur de bases de données.

– L’utilisation de procédure stockée permet la réutilisation de code. S’il est clair que cela n’augmente pas les performances, cela augmente la productivité des développeurs qui ont moins de code à produire, et qui passent donc moins de temps à le débugger.

– Le second plus gros avantage à mon sens après le cache de procédures est que le code qu’elles encapsulent, la logique de celles-ci, peut être maintenu sans modifier une seule ligne de code de l’application cliente, s’il n’est pas besoin de modifier les paramètres et la structure de l’ensemble de données produit.

# Architecture applicative

**Bibliothèque de classe**

Class Customer

Couche métier



*Couche présentation*

*BDD*

*Couche d’accès aux données*

*Couche logiciel métier*

**Bibliothèque de classe**

Class Utility

**Application Windows**

formulaire

**Bibliothèque de classe**

Class CustomerServices

# LINQ

LINQ signifie Language INtegrated Query. C'est un ensemble d'extensions du langage permettant de faire des requêtes sur des données en faisant abstraction de leur type. Il permet d'utiliser facilement un jeu d'instructions supplémentaires afin de filtrer des données, faire des sélections, etc. Il existe plusieurs domaines d'applications pour LINQ :

* Linq To Entities ou Linq To SQL qui utilisent ces extensions de langage sur les bases de données.
* Linq To XML qui utilise ces extensions de langage pour travailler avec les fichiers XML.
* Linq To Object qui permet de travailler avec des collections d'objets en mémoire.



De cette manière on peut créer des requêtes sur la collection listeCommandes et filtrer les données

On crée une collection d’objet

### Sugar Syntax

Cette façon d’écrire ces requêtes s’appelle en anglais la « Sugar syntax », que l’on peut traduire par « sucre syntaxique ». Il désigne de manière générale les constructions d’un langage qui facilitent la rédaction du code sans modifier l’expressivité du langage.

### Méthode d’extension

Les méthodes d'extension vous permettent d'« ajouter » des méthodes à des types existants sans créer un type dérivé, ni recompiler ou modifier le type d'origine. Les méthodes d'extension sont un type particulier de méthode statique appelées comme s'il s'agissait de méthodes d'instance sur le type étendu. Pour le code client écrit en C# et Visual Basic, il n'y a aucune différence apparente entre appeler une méthode d'extension et les méthodes qui sont réellement définies dans un type.

Les méthodes d'extension les plus courantes sont les opérateurs de requête standard LINQ qui ajoutent des fonctionnalités de requête aux types System.Collections.IEnumerable et System.Collections.Generic.IEnumerable<T> existants.

# Pattern Singleton

### Principe des Design Patterns

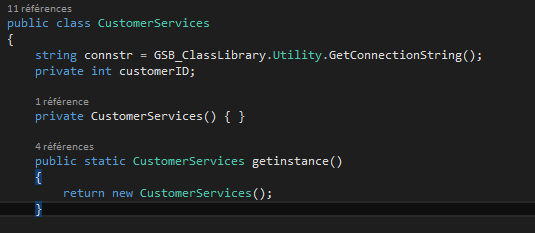
Depuis les débuts de la programmation, tout un tas de développeurs ont rencontré différents problèmes de conception. La plupart de ces problèmes étaient récurrents. Pour éviter aux autres développeurs de buter sur le même souci, certains groupes de développeurs ont développé ce qu'on appelle des design patterns (ou masques de conceptions en français). Chaque design pattern répond à un problème précis dont le pattern singleton.

### Le Design Pattern Singleton

Ce pattern permet d’obliger à une classe de se faire instancier qu’une seule fois.

### Mise en œuvre

Au sein de mon projet j’ai réalisé ce pattern sur ma class « CustomerServices » car j’ai eu besoin d’avoir qu’une seule instance d’elle pour éviter des erreurs éventuelles.



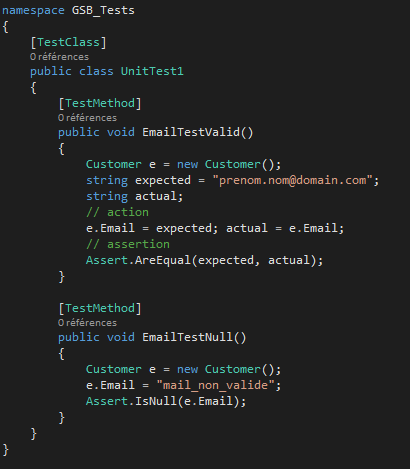
Je passe mon constructeur en private pour forcer l’instance avec ma méthode getInstance qui renverra toujours la même instance.

# Tests unitaires

Les tests unitaires, de façon concrète, il s’agit de produire du code qui va tester si notre code effectue bien ce qu’il est censé faire. Pour garantir ce code qui va tester notre code mais aussi pour gagner en productivité et efficacité, il est possible de s’appuyer sur des frameworks de tests éprouvés et outillés. Ces outils nous aideront à générer, exécuter puis valider nos séries de tests.

Visual Studio 2013 offrent un outil et une plateforme de tests unitaire : Visual Studio Unit Testing Framework.

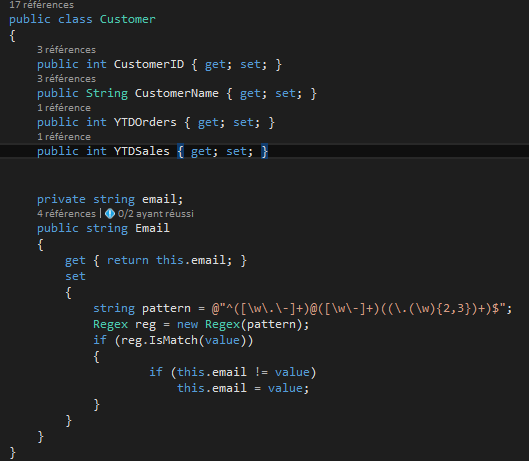
Dans mon application j’ai créé deux tests unitaires



Et une autre qui permet de vérifier un e-mail non valide

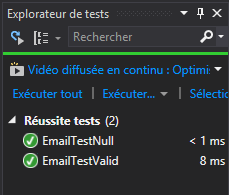
Une qui permet de vérifier un e-mail valide

Dans ma classe Customer j’ai défini une expression régulière



Cette expression régulière vérifie que l’adresse email est bien conforme.

Après avoir exécuté mes deux méthodes tests on voit bien que mes tests n’ont eu aucun problème



# Conclusion

Pour conclure, ce projet m’a permis d’améliorer mes compétences en programmation orientée objet ainsi qu’à manipuler Visual Studio, il m’a permis aussi de manipuler des procédures stockées dans un vrai contexte, de me retrouver confronté à des problèmes qui m’ont permis d’apprendre de mes erreurs.

Le projet fini, je me suis rendu compte qu’il serait intéressant de porter quelques améliorations intéressante comme :

* Séparer en deux modales l’ajout d’un client ainsi que l’ajout d’une commande
* Dans le filtre des commandes, pouvoir affiner ses recherches en filtrant aussi par client
* Pourvoir visualiser l’état de la commande (demande en cours, payé, envoyé, reçu…)
* Créer des comptes différents (Visiteurs, GSB), cela permettrai au visiteur de pouvoir suivre leurs commandes

# Annexes : Procédures stockées

|  |  |
| --- | --- |
| 1. **Nom** | uspNewCustomer |
| **Rôle** | Permet d’insérer un nouveau client dans la table Customer. Renvoie l’ID du client inséré. |
| **Paramètres** | Paramètre : @CustomerName (entré, NVARCHAR(40), pas de valeur par défaut)  Paramètre : @CustomerId (entré/sortie, INT, pas de valeur par défaut) |
| CREATE PROCEDURE uspNewCustomer  @CustomerName NVARCHAR (40),  @CustomerID INT OUTPUT  AS  BEGIN  INSERT INTO Customer(CustomerName) VALUES (@CustomerName);  SET @CustomerID = SCOPE\_IDENTITY();  RETURN @@ERROR  END | |

|  |  |
| --- | --- |
| **Nom** | uspPlaceNewOrder |
| **Rôle** | Permet d’insérer une nouvelle commande dans la table Orders, et d’incrémenter le cumul YTDOrders de la quantité commandé (Amount). |
| **Paramètres** | Paramètre : @CustomerId (entré, INT, pas de valeur par défaut)  Paramètre : @Amount (entré, INT, pas de valeur par défaut)  Paramètre : @OrderDate (entré, DATE, pas de valeur par défaut)  Paramètre : @Status (entré/sortie, CHAR(1), ‘O’ comme valeur par défaut) |
| CREATE PROCEDURE uspPlaceNewOrder  @CustomerID INT,  @Amount INT,  @OrderDate DATE,  @Status CHAR(1) = ‘O’  AS  BEGIN TRANSACTION  INSERT INTO Orders  (CustomerID, OrderDate, Status, Amount) VALUES (@CustomerID, @OrderDate, @Status, @Amount)  UPDATE Customer SET YTDOrders = YTDOrders + @Amount WHERE CustomerID = @CustomerID  IF @@ERROR = 0  COMMIT TRANSACTION  ELSE  ROLLBACK TRANSACTION | |
| **Nom** | uspShowOrderDetails |
| **Rôle** | Permet de retourner toutes les commandes d’un client. |
| **Paramètres** | Paramètre : @CustomerId (entré, INT, pas de valeur par défaut) |
| CREATE PROCEDURE uspShowOrderDetails  @CustomerID INT  AS  BEGIN  SELECT CustomerName, OrderDate, FilledDate, Status, Amount  FROM Customer  INNER JOIN Orders ON Customer.CustomerID = Orders.CustomerID  WHERE Customer.CustomerID = @CustomerID  END | |

|  |  |
| --- | --- |
| **Nom** | uspFillOrder |
| **Rôle** | Permet de valider une commande en affectant la valeur ‘F’ à la colonne Status.  Met à jour la table Customer en incrémentant YTDSales de la quantité (Amount) de la commande indentifié par OrderID. |
| **Paramètres** | Paramètre : @OrderID (entré, INT, pas de valeur par défaut)  Paramètre : @FilledDate (entré, DATE, pas de valeur par défaut) |
| CREATE PROCEDURE uspFillOrder  @OrderID INT,  @FilledDate Date  AS  BEGIN  DECLARE @Montant INT  DECLARE @CustomerID INT  BEGIN TRANSACTION  SELECT @Montant = Amount, @CustomerID = CustomerID  FROM Orders WHERE OrderID = @OrderID  UPDATE Orders SET Status = 'F' WHERE OrderID = @OrderID  UPDATE Customer SET YTDSales = YTDSales + @Montant WHERE CustomerID = @CustomerID  IF @@ERROR=0  COMMIT TRANSACTION  ELSE  ROLLBACK TRANSACTION  END | |

|  |  |
| --- | --- |
| **Nom** | uspCancelOrder |
| **Rôle** | Permet de valider une commande en affectant la valeur ‘F’ à la colonne Status.  Met à jour la table Customer en incrémentant YTDSales de la quantité (Amount) de la commande indentifié par OrderID. |
| **Paramètres** | Paramètre : @OrderID (entré, INT, pas de valeur par défaut) |
| CREATE PROCEDURE uspCancelOrder  @OrderID INT  AS  BEGIN  DECLARE @Montant INT  DECLARE @CustomerID INT  BEGIN TRANSACTION  SELECT @Montant = Amount, @CustomerID = CustomerID  FROM Orders WHERE OrderID = @OrderID  UPDATE Orders SET Status = 'X' WHERE OrderID = @OrderID  UPDATE Customer SET YTDOrders = YTDOrders - @Montant WHERE CustomerID = @CustomerID  IF @@ERROR=0  COMMIT TRANSACTION  ELSE  ROLLBACK TRANSACTION  END | |