22/01/2015

Déborah Luenne

Groupe eCOLE PRATIQUE

Documentation technique

Activité 3.1 - GSB Hôpital

Sommaire

[1. Contexte 2](#_Toc409640108)

[2. L’environnement de développement 2](#_Toc409640109)

[3. Modélisation des exigences 2](#_Toc409640110)

[4. Persistance des données 3](#_Toc409640111)

[a. MCD 3](#_Toc409640112)

[b. Modèle relationnel 3](#_Toc409640113)

[c. Création de la base et des tables 4](#_Toc409640114)

[d. Création des contraintes 4](#_Toc409640115)

[e. Création des procédures stockées 4](#_Toc409640116)

[f. Appel des procédures stockées 5](#_Toc409640117)

[5. La chaîne de connexion 6](#_Toc409640118)

[6. IHM 8](#_Toc409640119)

[7. Code behind 9](#_Toc409640120)

[8. Conclusion 11](#_Toc409640121)

J’ia

# 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#

**Outils IDE** : Visual Studio

**Versionning** : Github.com

# Modélisation des exigences

GSB Hôpital

Editer ou annuler

une commande

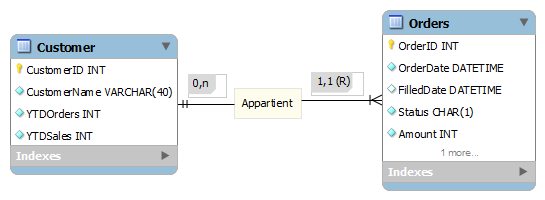
Ajouter une commande

Ajouter un compte

**Application de gestion des commandes**

# Persistance des données

## MCD



## 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 vers table 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

CREATE PROCEDURE uspCancelOrder

@OrderID INT

AS

BEGIN

DECLARE @Delta INT, @CustomerID INT

BEGIN TRANSACTION

SELECT @Delta = Amount, @CustomerID = CustomerID

FROM Orders WHERE OrderID = @OrderID;

UPDATE Orders

SET Status = 'X'

WHERE OrderID = @OrderID;

UPDATE Customer

SET

YTDOrders = YTDOrders = @Delta

WHERE CustomerID = @CustomerID

COMMIT TRANSACTION

END

CREATE PROCEDURE uspFillOrder

@OrderID INT, @FilledDate DATE

AS

BEGIN

DECLARE @Delta INT, @CustomerID INT

BEGIN TRANSACTION

SELECT @Delta = Amount, @CustomerID = CustomerID

FROM Orders WHERE OrderID = @OrderID;

UPDATE Orders

SET Status = 'F',

FilledDate = @FilledDate

WHERE OrderID = @OrderID;

UPDATE Customer

SET YTDSales = YTDSales + @Delta

WHERE CustomerID = @CustomerID

COMMIT TRANSACTION

END

CREATE PROCEDURE upsNewCustomer

@CustomerName NVARCHAR(40),

@CustomerID INT OUTPUT

AS

BEGIN

INSERT INTO Customer(CustomerName) VALUES (@CustomerName);

SET @CustomerID = SCOPE\_IDENTITY().

RETURN @@ERROR

END

CREATE PROCEDURE uspPlaceNewOrder

@CustomerID INT, @Amount INT, @OrderDate DATE, @Status CHAR(1) = 'O'

AS

BEGIN

DECLARE @RC INT

BEGIN TRANSACTION

INSERT INTO Order (CustomerID, OrderDate, FilledDate, Status, Amount)

VALUES (@CustomerID, @OrderDate, NULL, @Status, @Amount)

SELECT @RC = SCOPE\_IDENTITY();

UPDATE Customer

SET

YTDOrders = YTDOrders + @Amount

WHERE CustomerID = @CustomerID

COMMIT TRANSACTION

RETURN @RC

END

CREATE PROCEDURE uspShowOrderDetails

@CustomerID INT = 0

AS

BEGIN

SELECT C.CustomerName, O.OrderDate, O.FilledDate, O.Status, O.Amount

FROM Customer C

INNER JOIN Orders O

On O.CustomerID = C.CustomerID

WHERE C.CustomerID = @CustomerID

END

## Appel des procédures stockées

DECLARE @ID1 INT;

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

SELECT @ID1;

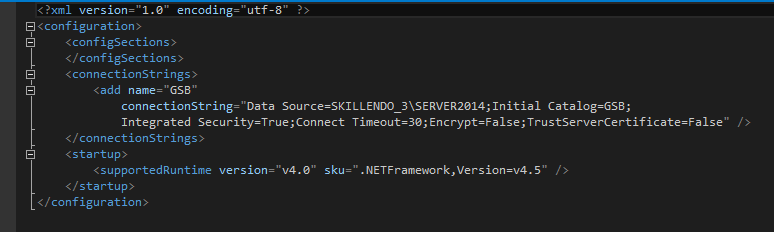
DECLARE @ID2 INT;

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 :

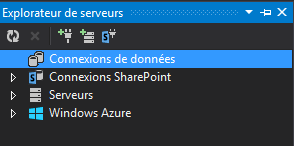


Chaine de connexion

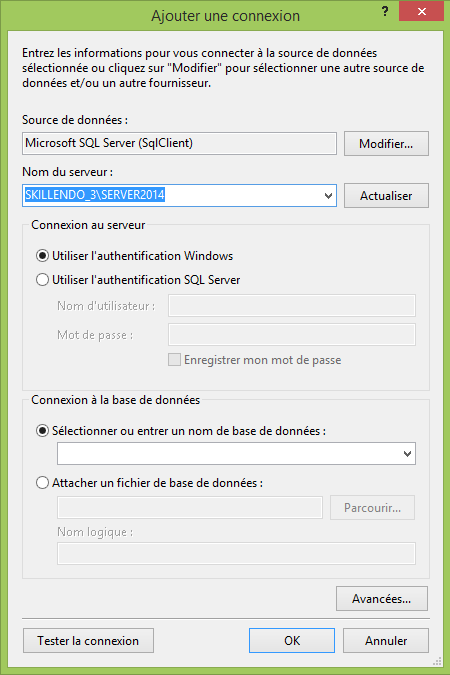
Cette chaine peut être générée par Visual Studio en allant dans :

AFFICHAGE -> EXPLORATEUR DE SERVEUR

Un volet s’affiche :



Clique droit -> Ajouter une connexion…

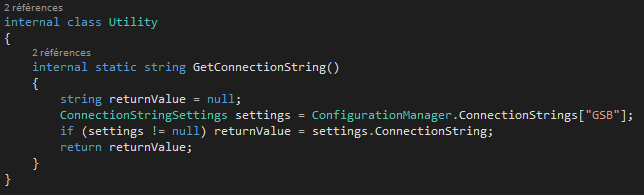


Générer la chaîne de connexion dans App.config

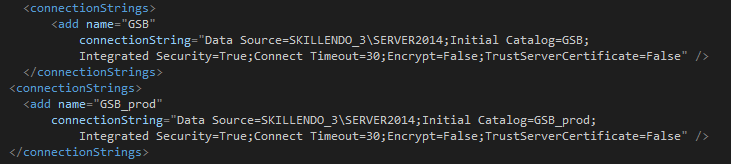
Teste la connexion

Renseigne le nom du serveur

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



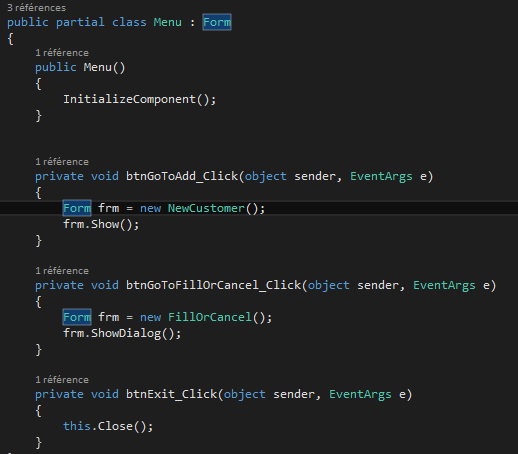
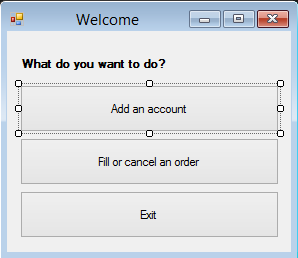
Il est possible de créer une seconde chaîne pour server SQL de production :



# IHM

Les classes partielles sont utiles dans plusieurs situations. Par exemple, lorsqu’on travaille sur de grands projets, séparer une classe en plusieurs fichiers permet à plusieurs programmeurs de travailler en même temps.

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

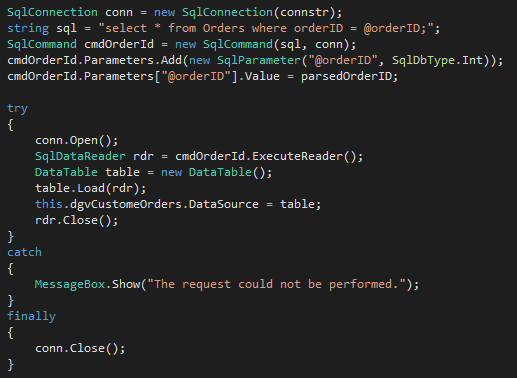
Visual Studio

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

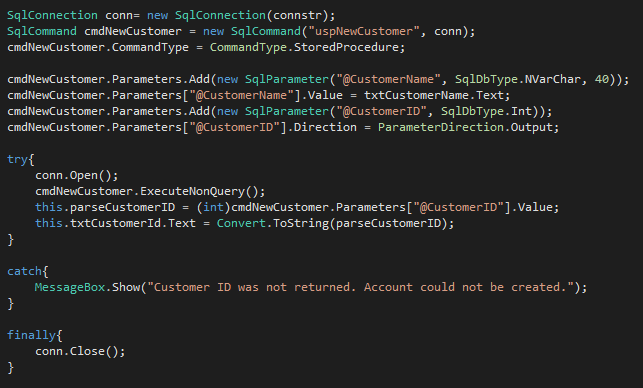
3

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.

# Conclusion

Pour conclure, ce projet m’a permis d’améliorer mes compétences en C# 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