M16:
SGBDR II

Zakariaa
Oulhafiane

Table des matières

I. INTRODUCTION

Objectif du module :

Etre capable de développer :

Des procédures, fonctions avec le T-SQL

Des triggers

II. Traduction des instructions de base

1) Déclaration :

Syntaxe :

```
DECLARE @numVariable type [=valeur d''initialisation];
```

Les types disponibles sont ceux de SQL :

bit, int, nchar, nvarchar, ntext, binary, varbinary,

timestamp, datetime, real, float, money…

2) Affectation :

Syntaxe :

```
SET @variable = expression;
```

3) Affichage :

Syntaxe :

```
PRINT 'msg'
```

4) Test Binaire :

Syntaxe :

```
If(condition)
        BEGIN
                --Instructions
        END
ELSE
        BEGIN
                --Instructions
        END
```

5) Test multiple :

Syntaxe :

```
SELECT Colonne = CASE
                        WHEN NomColonne = Valeur1 THEN Instruction1
                        WHEN NomColonne = Valeur2 THEN Instruction2
                        ELSE Instruction_Default
                END
FROM Table
```

6) Boucle WHILE :

Syntaxe :

```
WHILE(condition)
BEGIN
        <Instructions>
END
```

III. Les Procédures Stockés

Une procédure est un objet stocké qui fait partie de la structure logique d'une BD SQL SERVER

Une procédure est précompilée ce qui améliore la performance

Création :

```
CREATE PROCEDURE NomProcédure (@nompara1 type, @nompara2 type,
@nompara3 type, @nomVar type OUTPUT)
AS
        BEGIN
                Instructions
        END
```

Modification :

```
ALTER PREOCEDURE NomProcédure (@nompara1 type, @nompara2 type,
@nompara3 type)
AS
        BEGIN
                Instructions
        END
```

Suppression :

```
DROP PROCEDURE NomProcédure
```

Exemple :

Ecrire la procédure qui permet de calculer la somme de deux nombre avec ou sans paramètre de sortie

```
--Sans paramètre de sortie
ALTER PROCEDURE CalculeSomme (@nbr1 int, @nbr2 int)
AS
        BEGIN
                DECLARE @somme int;
                SET @somme = @nbr1+@nbr2;
                PRINT 'La somme est : '+CAST(@somme AS VARCHAR);
        END
--Appel de la procédure CalculeSomme sans paramètre de sortie
EXEC CalculeSomme 5,2
```

```
--Avec paramètre de sortie
ALTER PROCEDURE CalculeSomme (@nbr1 int, @nbr2 int, @somme int OUTPUT)
AS
        BEGIN
                SET @somme = @nbr1+@nbr2;
        END

--Appel de la procédure CalculeSomme avec paramètre de sortie
DECLARE @s int = 0
EXEC CalculeSomme 5, 2, @s out
PRINT 'La somme est : '+@s
```

Exercice :

1)Ecrire une procédure qui permet de calculer le total des quantités approvisionnées aux usines d'une ville donnée aux paramètres

Si cette quantité est inférieur ou égale à 500 Afficher seuil non atteint
Si entre 800 et 1000 afficher seuil atteint.
Si Supérieur ou égale à 1000 afficher réussite totale.

```
--Création de la procedure CalculeQteUsineParVille
ALTER PROCEDURE CalculeQteUsineParVille(@ville varchar(30),@message varchar(MAX) OUTPUT)
AS
BEGIN
        set @message = 'Ville Inexistant';
        declare @qte int;
        set @qte = (    SELECT SUM(Quantite)
                        FROM PUF JOIN U ON PUF.NU = U.NU
                        WHERE Ville = @ville
                    )
        set @message = CASE
                                WHEN @qte <= 50 THEN 'Seuil non atteint'
                                WHEN @qte BETWEEN 50 AND 100 THEN 'Seuil atteint'
                                WHEN @qte > 100 THEN 'Réussite totale'
                                WHEN @qte IS NULL THEN 'Il y a pas de quantité'
                        END
END
```

```
--Appel de la procedure CalculeQteUsineParVille
declare @msg varchar(MAX);
exec CalculeQteUsineParVille 'CASABLANCA', @msg OUT;
PRINT @msg;
```

IV. Les Fonctions

Création :

```
CREATE Function NomFonction (@nompara1 type, @nompara2 type, @nompara3 type)
RETURNS TypeDeRetour
AS
        BEGIN
                <Instructions>
                RETURN @variableDeRetour
        END
```

Modification :

```
ALTER Function NomFonction (@nompara1 type, @nompara2 type, @nompara3 type)
RETURNS TypeDeRetour
AS
        BEGIN
                <Instructions>
                RETURN @variableDeRetour
        END
```

Suppression :

```
DROP FUNCTION NomFonction
```

Exemple :

```
CREATE FUNCTION CalculeSommeF (@nbr1 int, @nbr2 int) RETURNS int
AS
        BEGIN
                DECLARE @somme int;
                SET @somme = @nbr1+@nbr2;
                RETURN @somme
        END

--Appel de la fonction CalculeSommeF
DECLARE @s int = 0
SET @s=dbo.CalculeSommeF(5,6)
PRINT @s
```

Note : **dbo** est obligatoire avant le nom de la fonction.

V. Fonction qui retourne une table

Syntaxe (1er Méthode) :

```
CREATE FUNCTION NomFonction() RETURNS TABLE
AS
        RETURN <<Requête SQL>>
```

Note : Dans cette syntaxe il ya pas de BEGIN et END.

Exemple :

```
--Une fonction qui retourne la liste des produits de couleur rouge.
--Création de la fonction
CREATE FUNCTION FlistePdtRouge() RETURNS TABLE
AS
        RETURN SELECT *
                FROM P
                WHERE UPPER(Couleur)='ROUGE';
```

```
--Appel de fonction
SELECT *
FROM dbo.FlistePdtRouge()

--Création d'une table nommé PdtRouge à partir de la fonction
SELECT *
INTO PdtRouge
FROM dbo.FlistePdtRouge()
```

Syntaxe (2éme Méthode) :

```
CREATE FUNCTION NomFonction()
RETURNS @NomTable TABLE(
                        NomColonne1 TypeColonne,
                        NomColonne2 TypeColonne
                        )
AS
    BEGIN
        INSERT INTO @NomTable SELECT <<Requête SQL>>
        RETURN;
    END
```

Exemple :

```
CREATE FUNCTION ListePdtRouge()
RETURNS @PdtRouge TABLE(
                        ID int PRIMARY KEY,
                        Nom varchar(20)
                        )
AS
    BEGIN
        INSERT INTO @PdtRouge SELECT NP,NomP
                              FROM P
                              WHERE UPPER(Couleur)='ROUGE'
        RETURN
    END
```

VI. Gestion des erreurs

1) Personnalisé :

```
--Exemple : (Gérer l'erreur de la clé primaire)
IF(@NumUsine NOT IN(SELECT NU
                    FROM U))
BEGIN
        <<Traitement Insertion>>
END
```

2) Exploitation de la réponse du serveur :

```
BEGIN TRY
        INSERT INTO U VALUES(12,'Hiho', 'CASA')
        PRINT 'Insertion Effectué'
END TRY
BEGIN CATCH
        IF(ERROR_NUMBER() = 2627)
        BEGIN
                PRINT 'Erreur de clé primaire : '+ERROR_MESSAGE()
        END
END CATCH
```

3) Levée des exceptions :

Syntaxe :

RAISERROR('Message Ou Bien CodeMessage', NiveauErreur, NumState, ...)

Note : Niveau d'erreur entre 11-16 peut être corrigées par l'utilisateur

Exemple :

```
--Stocker un message erreur
sp_addmessage 50003, 11, 'Erreur! Numéro d''usine déjà existe.', 'English', true, 'REPLACE'
```

```
--Gérer notre erreur
BEGIN TRY
        IF(33 IN (SELECT NU
                  FROM U))
          RAISERROR(50003, 11, 1)
END TRY
BEGIN CATCH
        IF(ERROR_NUMBER() = 50003)
          PRINT 'Message d''erreur : '+ERROR_MESSAGE()
END CATCH
```

VII. Les Curseurs

Définition :

Cache mémoire représentant une table mémoire.

Son intérêt : Traitement de ligne par ligne.

C'est aussi un type de données T-SQL.

Syntaxe :

```
DECLARE NomCurseur [INSENSITIVE] [SCROLL] CURSOR
FOR <<Requete_Select>>
```

Ouverture :

```
Open NomCurseur
```

Fermeture :

```
Close NumCurseur
```

Destruction :

```
DEALLOCATE NomCurseur
```

Chargement :

```
FETCH Mode FROM NomCurseur INTO @Var1, @Var2
Mode :
        NEXT : Lit la ligne suivante. C'est la seule option possible pour un INSENSITIVE CURSOR.
        PRIOR : Lit la ligne précédente. (Il faut déclarer le curseur avec le mode SCROLL)
        FIRST : Lit la première ligne.
        LAST : Lit la dernière ligne.
        ABSOLUTE n : Lit la nième ligne de l'ensemble.
        RELATIVE n : Lit la nième ligne à partir de la ligne courante.
```

Variable système intéressantes :

```
@@Fetch_Status : Renvoie une valeur différente de 0 si on n'est pas à la fin du curseur.
@@Cursor_rows : Renvoie le Nombre de lignes affectées dans le dernier curseur ouvert
        Remarque : Pour utiliser @@Cursor_rows il faut déclarer le curseur en SCROLL
@@Cursor_Status : Permet à l''appelant d''une procédure stockée de déterminer si la procédure
a retourné un curseur et un ensemble de résultats pour un paramètre donné.
```

Exemple :

```
--Déclaration du curseur
DECLARE DescriptionUsine SCROLL CURSOR
FOR     SELECT *
        FROM U
--Ouverture du curseur
Open DescriptionUsine

--Déclaration des variables
DECLARE @NumU int, @NomU VARCHAR(20), @Ville VARCHAR(20),@Description VARCHAR(50)
```

```
--Chargement du curseur
FETCH FIRST
FROM DescriptionUsine
INTO @NumU, @NomU, @Ville
--Faire une boucle pour exploiter le curseur
WHILE (@@FETCH_STATUS = 0)
        BEGIN
                SET @Description =  'Usine N°'+CAST(@NumU AS VARCHAR(5))+' est situé à '+@Ville
                PRINT @Description
                FETCH NEXT
                FROM DescriptionUsine
                INTO @NumU, @NomU, @Ville
        END
--Fermeture du curseur
Close DescriptionUsine
--Destruction du curseur depuis la mémoire
DEALLOCATE DescriptionUsine
```

Exercice :

```
1)Afficher la liste des fournisseurs pour chaque produit
DECLARE C SCROLL CURSOR
FOR SELECT NP
    FROM P

OPEN C
--Empêche le message indiquant le nombre de lignes concernées par une instruction ou une procédure stockée Transact-SQL
SET NOCOUNT ON
DECLARE @NP int
FETCH FIRST FROM C INTO @NP
WHILE(@@FETCH_STATUS=0)
BEGIN
        PRINT 'Produit N° '+CAST(@NP AS VARCHAR(5))
        DECLARE CF SCROLL CURSOR
        FOR SELECT PUF.NF,NomF,Ville
                FROM PUF JOIN F ON PUF.NF=F.NF
                WHERE NP=@NP
        OPEN CF
        DECLARE @NF int, @NomF varchar(30), @Ville varchar(30)
        FETCH FIRST FROM CF INTO @NF,@NomF,@Ville
        WHILE(@@FETCH_STATUS=0)
        BEGIN
                PRINT char(9)+CAST(@NF AS VARCHAR(5))+'|'+@NomF+'|'+@Ville
                FETCH NEXT FROM CF INTO @NF,@NomF,@Ville
        END
        CLOSE CF
```

```
        DEALLOCATE CF
        FETCH NEXT FROM C INTO @NP
END
CLOSE C
DEALLOCATE C

2)Faire une mise à jour à travers un curseur
DECLARE C SCROLL CURSOR
FOR SELECT NP
    FROM P

OPEN C

FETCH ABSOLUTE 3 FROM C

--Faire la mise à jour
UPDATE P
SET NomP = 'Apple'
WHERE CURRENT OF C

CLOSE C
DEALLOCATE C
```

Note : On peut aussi supprimer une ligne à travers le curseur.

```
3)Ecrire une procedure permettant de renvoyer la liste des fournisseurs d''une ville donnée
--Création du procedure
CREATE PROCEDURE FournisseurVille(@Ville varchar(30), @CF CURSOR VARYING OUTPUT)
AS
BEGIN
        SET @CF = CURSOR FOR SELECT NF,NomF
                             FROM F
                             WHERE UPPER(Ville)=@Ville
        OPEN @CF;
END

DECLARE @NF int, @NomF varchar(30)
DECLARE @C CURSOR

--Appel de procédure
EXEC FournisseurVille 'CASABLANCA', @C Out

--Chargement et exploitation du curseur
FETCH FROM @C INTO @NF,@NomF
WHILE(@@FETCH_STATUS=0)
BEGIN
        PRINT 'NF: '+CAST(@NF AS VARCHAR(5))+' | Nom Fournisseur : '+@NomF
        FETCH FROM @C INTO @NF,@NomF
END

CLOSE @C
DEALLOCATE @C
```

VIII. Les triggers (Déclencheurs)

Définition :

C'est un bloc d'instructions mais qui se déclenche automatiquement lors des événements suivants :

INSERTION dans une table / **INSERT**
Modification dans une table / **UPDATE**
Suppression dans une table / **DELETE**

Les déclencheurs permettent de faire des audits/surveillances au niveau d''une table
Le déclencheur est un objet stocké

Remarque :

Le déclencheur est lié à une table

Mode de déclenchement :

-**AFTER**
-**INSTEAD OF** (Il remplace l'action standard de l'instruction)
-**FOR** (C'est la même chose que AFTER)

Création :

```
CREATE TRIGGER NomTrigger
ON NomTable/NomVue
[MODE Déclenchement] [Evénement]
AS
        BEGIN
                <<Instructions>>
        END
```

Modification :

```
ALTER TRIGGER NomTrigger
ON NomTable/NomVue
[MODE Déclenchement] [Evénement]
AS
        BEGIN
                <<Instructions>>
        END
```

Suppression :

```
DROP TRIGGER NomTrigger
```

Les Pseudo-Tables : (Inserted / Deleted) :

Les instructions de déclenchement DML utilisent deux tables spéciales : la table deleted et la table inserted.

SQL Server crée et gère automatiquement ces tables. Ces tables temporaires résidant en mémoire servent à tester

Les effets de certaines modifications de données et à définir des conditions pour les actions de déclencheur DML.

Vous ne pouvez pas modifier directement les données contenues dans les tables ou effectuer des opérations DDL (Data Définition Langage) sur les tables

Exemple :

```
--Créer un trigger qui permet de vérifier et insérer des villes ligne par ligne
CREATE TRIGGER VerifVille
ON F
INSTEAD OF INSERT
AS
BEGIN
	DECLARE C SCROLL CURSOR
	FOR SELECT *
		FROM INSERTED

	OPEN C

	DECLARE @NF int, @NomF varchar(30), @Status varchar(30), @Ville varchar(30)
	FETCH FIRST FROM C INTO @NF,@NomF,@Status,@Ville
	WHILE(@@FETCH_STATUS = 0)
	BEGIN
		IF(UPPER(@Ville) IN ('RABAT','CASA'))
		BEGIN
			INSERT INTO F VALUES(@NF, @NomF, @Status, @Ville)
		END
		ELSE
			PRINT 'Le Fournisseur n° '+ CAST(@NF AS VARCHAR(5)) + ' N''est pas inseré ,la ville est incorrect'
		FETCH NEXT FROM C INTO @NF, @NomF, @Status, @Ville
	END
	CLOSE C
	DEALLOCATE C
END
```

Autre Exemple :

```
--Créer un trigger qui permet de faire l'insertion et si une ligne n'est pas correcte il va insérer rien
CREATE TRIGGER VerifVille
ON F
INSTEAD OF INSERT
AS
BEGIN
        IF NOT EXISTS (  SELECT *
                         FROM inserted
                         WHERE Ville NOT IN ('CASA','RABAT'))
                INSERT INTO F SELECT *
                FROM inserted
```

Autre Exemple (AFTER INSERT) :

```
--Créer un trigger qui permet de vérifier les valeurs après l'insertion et si une ligne n'est pas correcte il va annuler tout
CREATE TRIGGER VerifVille
ON F
AFTER INSERT
AS
BEGIN
        IF EXISTS(SELECT *
                  FROM inserted
                  WHERE Ville NOT IN('CASA','RABAT'))
        BEGIN
                ROLLBACK;
                PRINT 'La transaction est annulé'
        END
END
```

Autre Exemple (AFTER DELETE) :

```
--Créer un trigger qui permet de vérifier les lignes supprimées et si une ligne a le statut ETRANGER il va annuler tout
CREATE TRIGGER DeleteF
ON F
AFTER DELETE
AS
BEGIN
	IF EXISTS(SELECT *
			FROM deleted
			WHERE UPPER(Statut)='ETRANGER')
	BEGIN
		ROLLBACK;
		PRINT 'Votre suppression est Annulé'
	END
END
```

Autre Exemple (INSTEAD OF DELETE) :

```
--Dès qu'on trouve un étranger on supprime aucun
CREATE TRIGGER InsteadOFDeleteF
ON F
INSTEAD OF DELETE
AS
BEGIN
	SET NOCOUNT ON
	DELETE F
	WHERE NF IN (SELECT NF
			FROM deleted
			WHERE Statut<>'ETRANGER')
	PRINT 'Les fournisseur non supprimé est : '
	DECLARE C SCROLL CURSOR
	FOR	SELECT NF
		FROM deleted
		WHERE Statut='ETRANGER'
	OPEN C
	DECLARE @NF int
	FETCH FIRST FROM C INTO @NF
	WHILE(@@FETCH_STATUS=0)
	BEGIN
		PRINT 'Le fournisseur n° '+CAST(@NF AS VARCHAR(5))
		FETCH NEXT FROM C INTO @NF
	END
	CLOSE C
	DEALLOCATE C
END
```

Note : Cas de modification (UPDATE)

Dans la table **deleted** il y a les lignes à modifier
Dans la table **inserted** il y a les lignes modifié (Avec nouvelles valeurs)

Exemple :

On a les tables suivantes :

Employe(EmpNo, Name, Salaire, DeptNo*)
Departement(DeptNo, NomDept, Manager*)

Question :

1)Ecrire un trigger qui permet de vérifier la contrainte suivante :
Le salaire d'un employé ne peut être supérieur à son manager.

Stratégie :

Dès que le salaire d'un manager modifiée est < à au moins d'un employé, on annule la transaction

```
CREATE TRIGGER ModifManagerDept
ON Departement
AFTER UPDATE
AS
BEGIN
        IF(UPDATE(Manager))
        BEGIN
                IF(EXISTS(SELECT *
                        FROM inserted I JOIN Employe E ON I.Manager=E.EmpNo
                        WHERE Salaire < ANY (SELECT Salaire
                                             FROM Employe
                                             WHERE DeptNo=I.DeptNo)
                        )
                  )
                 ROLLBACK;
                 PRINT 'Modif Annulé';
        END
END
```

Autre Exemple pour le même trigger précédent avec (INSTEAD OF UPDATE) :

```
--On Tient compte aussi du cas du changement du numéro du département
--En cas de problème de la mise à jour on annule toute la transaction
--Remarque : Attention au Commit est nécessaire
ALTER TRIGGER ModifDept
ON Departement
INSTEAD OF UPDATE
AS
BEGIN
        IF(NOT EXISTS(SELECT *
                        FROM inserted I JOIN Employe E ON I.Manager=E.EmpNo
                        WHERE Salaire <ANY (SELECT Salaire
                                            FROM Employe
                                            WHERE DeptNo=I.DeptNo))
           AND NOT EXISTS(SELECT *
                          FROM inserted I LEFT Join Employe E ON I.Manager=E.EmpNo
                          WHERE E.EmpNo IS NULL))
        BEGIN
                --Curseur pour la table inserted
                DECLARE CI CURSOR
                FOR SELECT *
                    FROM inserted
                --Curseur pour la table deleted
                DECLARE CD CURSOR
                FOR SELECT DeptNo
                    FROM deleted
                OPEN CI
                OPEN CD
                --Déclaration des variables
                DECLARE @DeptNoD int, @DeptNo int, @NomDept varchar(30), @Manager int
                DECLARE @EtatI int, @EtatD int
                --Commencer une transaction
                BEGIN TRANSACTION
                        --Parcourir les tables inserted et deleted pour faire une update correcte
                        FETCH FROM CI INTO @DeptNo, @NomDept, @Manager
                        SET @EtatI = @@FETCH_STATUS
                        FETCH FROM CD INTO @DeptNoD
                        SET @EtatD = @@FETCH_STATUS
                        WHILE(@EtatI=0 AND @EtatD=0)
                        BEGIN
                                BEGIN TRY
                                        UPDATE Departement
                                        SET DeptNo=@DeptNo, NomDept=@NomDept, Manager=@Manager
                                        WHERE DeptNo=@DeptNoD
                                END TRY
                                BEGIN CATCH
```

```
                                    PRINT 'Error du Traitement';
                                    ROLLBACK;
                                    CLOSE CI
                                    CLOSE CD
                                    DEALLOCATE CI
                                    DEALLOCATE CD
                                    return;
                                END CATCH

                                --Charger les valeurs suivantes
                                FETCH FROM CI INTO @DeptNo, @NomDept, @Manager
                                SET @EtatI = @@FETCH_STATUS

                                FETCH FROM CD INTO @DeptNoD
                                SET @EtatD = @@FETCH_STATUS
                            END
                    CLOSE CI
                    CLOSE CD
                    DEALLOCATE CI
                    DEALLOCATE CD
                    COMMIT;
            END
            ELSE
                    PRINT 'Opération non complète';
END
```

IX. Les triggers DDL

1) Définition :

Même chose que les triggers LMD sauf qu'ils sont liés à des clauses DDL.
Ils sont attachés à une base de données ou bien Serveur(s).

2) Evénement Déclencheurs :

Mode :

AFTER
FOR

Note : **INSTEAD OF** n'est pas disponible dans ces triggers.

Opération :

Toute Opération DDL (**CREATE**, **DROP**, **ALTER**, **GRANT**, **REVOKE**, ...)

3) Intérêt :

Contrôle au niveau de la base ou du serveur.

Audite ou suivi des opérations/mouvements au niveau de la base ou du serveur.

4) Syntaxe :

Création :

```
CREATE TRIGGER NomTrigger
ON (DATABASE/SERVER/ALL)
(FOR/AFTER) NomOperation
AS
        BEGIN

        END
```

Modification :

```
ALTER TRIGGER NomTrigger
ON (DATABASE/SERVER/ALL)
(FOR/AFTER) NomOperation
AS
        BEGIN

        END
```

Suppression :

```
DROP TRIGGER NomTrigger
```

5) Les informations sur l'événement via XML :

C'est un langage qui permet de faire une description des données.
Représentation hiérarchique de donné dans des fichier type texte _.xml
Une norme pour le transfert des données.

Exemple :

```
<Inscription>
        <Stagiaire code=...>
                <Nom>.....</Nom>
                <Adresse>.....</Adresse>
                <Filiere>
                        <CodeFiliere>...</CodeFiliere>
                        <Intitulé>...</Intitulé>
                </Filiere>
        </Stagiaire>
</Inscription>
```

Exercice :

1) Créer un trigger DDL qui permet de contrôler la création de tables dans la base PUF. NomTable, NomUtilisateur, DateCréation.

```
CREATE TABLE AuditPUF
(
        NomTable SYSNAME,
        NomUtilisateur SYSNAME,
        DateCreation datetime
)

CREATE TRIGGER OnCreateTable
ON DATABASE
FOR CREATE_TABLE
AS
BEGIN
        DECLARE @eventdata XML;
        SET @eventdata = EVENTDATA();
        INSERT INTO AuditPUF VALUES(CONVERT(SYSNAME, @eventdata.query('data(/EVENT_INSTANCE/ObjectName)')),
                                    CONVERT(SYSNAME, @eventdata.query('data(/EVENT_INSTANCE/UserName)')),
                                    GETDATE())
END
```

2) Créer un trigger qui contrôle/suivi des événements qui se produisent (au niveau du serveur).

```
CREATE TABLE AuditServer
(
        NomEvenement varchar(50),
        TextSQL varchar(max),
        Utilisateur varchar(50),
        DateEvenement datetime
)


CREATE TRIGGER AuditServeur
ON ALL SERVER
FOR CREATE_TABLE, ALTER_TABLE, DROP_TABLE
AS
        BEGIN
                DECLARE @eventinfo XML;
                SET @eventinfo = EVENTDATA();

                INSERT INTO AuditServer
                        VALUES(CAST(@eventinfo.query('data(/EVENT_INSTANCE/ObjectType)') AS VARCHAR(50)),
                                CAST(@eventinfo.query('data(/EVENT_INSTANCE/TSQLCommand/CommandText)') AS VARCHAR(MAX)),
                                CAST(CURRENT_USER AS VARCHAR(50)),
                                GETDATE());
        END
```

Exercice (SQL/XML) :

1) Transformer le résultat d'une requête sélection en fichier XML en 2 variantes.

```
SELECT *
FROM U
FOR Xml Auto,Type

SELECT *
FROM U
FOR Xml Path('Usine'), root('ListeUsines'),Type
```

2) Lecture d''un fichier XML et le transformer en table Sql (IMPORT).

1er méthode :

```
declare @idDoc int
declare @contentDoc varchar(5000)=
'<Entreprise>
<F nf="1" nomF="F1" status="FIDELE" villeF="NICE" />
<F nf="2" nomF="F2" status="NOUVEAU" />
<F nf="3" nomF="F3" status="FIDELE" villeF="NICE" />
</Entreprise>'

exec sp_xml_preparedocument @idDoc out, @contentDoc

SELECT *
FROM openxml(@idDoc, '/Entreprise/*') with (nf int, nomF varchar(20), status varchar(20), villeF varchar(20))
```

2éme Methode :

```
DECLARE @iddoc int;
DECLARE @content varchar(max) =
        '<ListeUsines>
                <Usine>
                        <NU>11</NU>
                        <NomU>ISTA</NomU>
                        <Ville>CASABLANCA</Ville>
                </Usine>
                <Usine>
                        <NU>12</NU>
                        <NomU>Hiho</NomU>
                        <Ville>CASA</Ville>
                </Usine>
        </ListeUsines>';
exec sp_xml_preparedocument @iddoc out, @content

--Les alias des champs est obligatoire dans ce type xm
SELECT *
FROM OPENXML(@iddoc, '/ListeUsines/Usine') with(NumUsine int 'NU', NomUsine varchar(20) 'NomU', Ville varchar(20) 'Ville')
--On peut préciser quel usine
--  OPENXML(@iddoc, '/ListeUsines/Usine[3]')
--Ou bien
--  OPENXML(@iddoc, '/ListeUsines/Usine[NU=12]')
```