Secteur Tertiaire Informatique

Filière « Etude et développement »

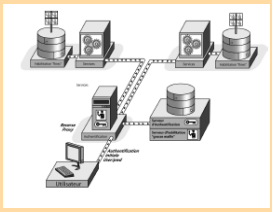
Séquence « Développer des composants d'accès aux données »

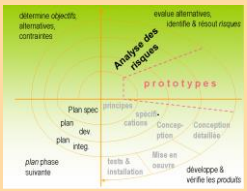
SQL Server et Transact SQL
Le langage DML partie3
Les Fonctions et Procédures Stockées

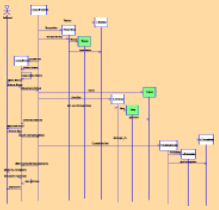
Apprentissage

Mise en pratique

Evaluation







Version	Date	Auteur(s)	Action(s)
1.0	7/10/15	B. Hézard	Création du document
1.0	16/12/15	B. Hézard	Fusion ressources fonctions et procédures stockées

TABLE DES MATIERES

Objectifs

Après étude de ce document, vous serez à même de développer des fonctions et procédures stockées dans une base de données SQL Server.

Pré requis

Connaître les bases du langage de requête SQL.

Maîtriser les bases de la programmation procédurale.

Avoir pris connaissance des instructions de programmation en langage Transact-SQL.

Outils de développement

SQL Server Management Studio.

Méthodologie

Ce document est conçu comme un aide-mémoire pour les principales techniques à utiliser dans la programmation des fonctions et procédures stockées SQL Server ; il complète sans la remplacer la documentation de référence.

Vous découvrirez tout d'abord les 2 types de fonctions offertes par SQL Server (2008 minimum), les fonctions *scalaires* et les fonctions *table*.

La partie suivante du document permet d'approfondir le sujet et de sécuriser les développements.

La dernière partie du document vous permettra de découvrir comment créer des procédures stockées dans une base de données SQL Server. Vous découvrirez tout d'abord pourquoi et comment créer des procédures stockées avec SQL Server. Vous pourrez ensuite mieux comprendre leur syntaxe et leur usage à l'aide d'exemples commentés.

Enfin, quelques compléments vous permettront d'aller un peu plus loin en terme de sécurisation des traitements et de gestion des erreurs.

Mode d'emploi

Symboles utilisés :

 Renvoie à des supports de cours, des livres ou à la documentation en ligne constructeur.

 Propose des exercices ou des mises en situation pratiques.

 Point important qui mérite d'être souligné !

Ressources

L'aide en ligne de SQL Server disponible à tout moment et indexée par rapport au contexte (touche F1).

Lectures conseillées

1. INTRODUCTION

Une fonction, dans un langage de programmation, est une **portion de code réutilisable** permettant d'effectuer un **traitement** (sur des données en entrée) **pour aboutir à une nouvelle donnée résultat**.

Le langage Transact-SQL propose deux types principaux de fonctions :

- Les **fonctions scalaires** renvoyant une valeur unique (comme la fonction système `getdate()`, ou comme un calcul entre colonnes de tables ou encore le formatage de données brutes issues de tables) ;
- Les **fonctions table**, renvoyant un ensemble de lignes et de colonnes (éventuellement formatées) que l'on manipule comme une table (dans une clause `select… from…` par exemple).

Par le biais d'une fonction, on ne peut effectuer aucune modification en table de la base de données. Seuls les objets locaux à la fonction (variables) peuvent être modifiés.

Les fonctions sont créées de la même manière qu'une requête SQL, et se retrouvent sans l'arborescence de **SQL Server Management Studio** dans le dossier de la base de données, sous-dossier **Programmabilité / Fonctions**°.

2. CREATION DE FONCTIONS SCALAIRES

Une fonction scalaire est identique à une fonction mathématique : elle peut avoir plusieurs paramètres d'entrée (jusqu'à 1024), et **renvoie une seule valeur**.

On crée une fonction grâce à la commande :

`CREATE FONCTION… RETURNS… AS BEGIN… RETURN… END`

- Chaque paramètre est défini par son nom obligatoirement préfixé du caractère `@` et son type, et séparé du suivant par une virgule
- La clause **`RETURNS`** spécifie le type de données à retourner : les données de type `timestamp`, `table`, `text`, `ntext` et `image`, et définis par l'utilisateur ne peuvent être renvoyés.
- La valeur renvoyée par la fonction est toujours définie par le mot-clé **`RETURN`**
- La fonction est définie dans un bloc **`BEGIN`** … **`END`**

Exemple: la fonction **fn_DateFormat** formate à l'aide d'un séparateur entré en paramètre, une date entrée en paramètre et la retourne sous forme de chaîne de caractères.

```
CREATE FUNCTION fn_DateFormat
(@pdate datetime, @psep char (1))
RETURNS char (10) – avec s : declaration du type de retour
AS
BEGIN
    RETURN – sans s : ce qui doit être retourné à l'exécution
    CONVERT (varchar(2), datepart (dd,@pdate))
    + @psep  + CONVERT (varchar (2), datepart (mm, @pdate))
    + @psep  + CONVERT (varchar (4), datepart (yy, @pdate))
END
```

Ce qui pourrait s'écrire aussi, en version plus 'algorithmique' :

```
create function fn_Dateformat
(@pdate datetime, @psep char(1))
returns char(10)
as
begin
     -- variables nécessaires
     declare @jj char(2);
     declare @mm char(2);
     declare @aa char(4);
     -- calcul des différentes parties de la date
     set @jj = convert(varchar(2), datepart(dd, @pdate));
     set @mm = convert(varchar(2), datepart(mm, @pdate));
     set @aa = convert(varchar(4), datepart(yy, @pdate));
     -- assemblage pour restitution
     return @jj + @psep + @mm + @psep + @aa ;
end
```

A l'utilisation, la fonction sera appelée par son nom, préfixé du nom de son propriétaire, et complété par les arguments nécessaires entre parenthèses :

```
SELECT [BaseTest.]dbo.fn_DateFormat(GETDATE(), '/')
SELECT [BaseTest.]dbo.fn_DateFormat(DATCOM) from FOURNIS
```

Cette fonction peut donc être appelée ponctuellement en lui fournissant une donnée de type date ou être appliquée à une colonne de type `datetime` dans une projection de commande `select` classique.

Une fonction peut contenir un appel à une autre fonction, y compris elle-même, à une procédure stockée, et peut contenir toutes les instructions TRANSACT-SQL.

3. CREATION DE FONCTIONS TABLE

Rappelons qu'une fonction '**scalaire'** ne peut retourner **qu'une valeur unique** ; c'est bien le cas des exemples précédents. Il est parfois nécessaire de retourner **plusieurs informations** (ex : le nom **et** le CA d'un fournisseur) ou même **plusieurs lignes d'informations** (**les** noms et CA des fournisseurs de telle région). C'est l'objet des fonctions '**Table'**.

Les fonctions table sont de deux types :

- Les fonctions table **en ligne,** qui ne contiennent qu'une seule instruction `SELECT` déterminant le format de la table renvoyée.
- Les fonctions table **multi instructions**, qui déclarent le format d'une table virtuelle, avant de la remplir par des instructions `SELECT`

Dans les deux cas, la fonction accepte des paramètres en entrée, et s'utilise comme une table dans une clause `FROM` : elles remplacent totalement les vues en y ajoutant la possibilité de passer des paramètres.

Exemple 1 :

La fonction table en ligne, **`fn_NomArticles`,** renvoie tous les articles d'une certaine couleur à partir d'une table `ARTICLES` de structure

`ARTICLES (art_num, art_nom, art_coul, art_pa, art_pv, art_qte, art_frs)`

```
CREATE FUNCTION fn_NomArticles
    ( @pcoul char(10) )
RETURNS TABLE
AS
RETURN (
    SELECT art_nom,art_frs
    FROM dbo.articles
    WHERE art_coul =@pcoul
)
```

- La clause **`RETURNS`** spécifie **`TABLE`** comme type de données à retourner
- La valeur renvoyée par la fonction est toujours définie par le mot-clé **`RETURN`** (résultat d'une instruction **`SELECT`**)
- On n'utilise pas de bloc **`BEGIN`** … **`END`** puisque qu'il s'agit d'une simple instruction `SELECT`

A l'utilisation, la fonction sera appelée **comme une table**, par son nom, préfixé du nom de son propriétaire, et complété par les arguments nécessaires entre parenthèses :

```
SELECT * from fn_NomArticles('ROUGE')
```

Exemple 2 :

La fonction table multi instructions, **fn_PrixArticles,** restitue, selon la valeur de son paramètre, soit le code article, son nom et son prix de vente, soit le code article, son nom et son prix d'achat.

```
CREATE FUNCTION fn_PrixArticles
   ( @ptypeprix char(2) )
RETURNS  @fn_PrixArticles TABLE
      ( art_num smallint PRIMARY KEY NOT NULL,
      art_nom char(20) NOT NULL,
      art_prix money)
AS
BEGIN
      IF @ptypeprix ='PV'
            INSERT @fn_PrixArticles Select art_num, art_nom, art_pv
            from dbo.articles
      else if @ptypeprix ='PA'
            INSERT @fn_PrixArticles Select art_num, art_nom, art_pa
            from dbo.articles
RETURN
END
```

- La clause `RETURNS` spécifie `TABLE` comme type de données à retourner, la structure étant définie en suivant
- La valeur renvoyée par la fonction est toujours définie par le mot-clé final `RETURN` ; cette fois c'est le résultat de l'instruction DML précédente qui a manipulé la table déclarée
- Le bloc `BEGIN … END` délimite les instructions multiples

A l'utilisation, la fonction sera appelée **comme une table**, par son nom, préfixé du nom de son propriétaire, et complété par les arguments nécessaires entre parenthèses :

```
SELECT * from dbo.fn_PrixArticles('PV')
```

4. LES OPTIONS DE FONCTION

De manière générale, chaque fonction peut être définie avec des options :

```
CREATE FUNCTION [nom_schema.] nom_fonction
( [ { @nom_parametre [ AS ][ nom_schéma. ] type_données }])
RETURNS {type_données | TABLE | @return_variable TABLE }
[WITH { ENCRYPTION | SCHEMABINDING
      | RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT ]
      | EXECUTE AS}]
AS
          Corps de la fonction tel que défini §2 et §3
```

ENCRYPTION

Indique que le moteur de base de données *chiffre* les colonnes d'affichage catalogue qui contiennent le texte de l'instruction `CREATE FUNCTION`

SCHEMABINDING

Indique que *la fonction est liée aux objets de base de données* auxquels elle fait référence. Toute modification (`ALTER`) ou suppression (`DROP`) de ces objets est vouée à l'échec.

La liaison de la fonction aux objets auxquels elle fait référence est supprimée uniquement lorsqu'une des actions suivantes se produit :

- La fonction est supprimée.
- La fonction est modifiée, avec l'instruction `ALTER`, sans spécification de l'option `SCHEMABINDING`.

Une fonction peut être liée au schéma uniquement si les conditions suivantes sont vérifiées :

- La fonction est une fonction Transact-SQL.
- Les fonctions utilisateur et vues référencées par la fonction sont également liées au schéma.
- La fonction fait référence aux objets à partir d'un nom en deux parties.
- La fonction et les objets auxquels elle fait référence appartiennent à la même base de données.

L'utilisateur qui exécute l'instruction `CREATE FUNCTION` dispose de l'autorisation `REFERENCES` pour les objets de base de données auxquels la fonction fait référence (Voir Sécurités SQL Server).

.

RETURNS NULL ON NULL INPUT | CALLED ON NULL INPUT

`CALLED ON NULL INPUT` est implicite par défaut ; le corps de la fonction est exécuté même si `NULL` est transmis comme argument.

`RETURNS NULL ON NULL INPUT` indique que SQL Server peut retourner `NULL` lorsque n'importe lequel des arguments qu'il reçoit a pour valeur `NULL`, sans réellement appeler le corps de la fonction.

La clause `EXECUTE AS` spécifie le contexte de sécurité dans lequel la fonction définie par l'utilisateur est exécutée.

(Voir Sécurités SQL Server).

5. MODIFICATION / SUPPRESSION DE FONCTION

La commande de modification **`ALTER FUNCTION`** accepte les mêmes paramètres que `CREATE FUNCTION`.

```
ALTER FUNCTION fn_DateFormat
(@pdate datetime, @psep char(1))
RETURNS char(10)
    WITH ENCRYPTION
AS
BEGIN
    RETURN
    CONVERT (varchar(10), datepart(dd,@pdate))
    + @psep  + CONVERT (varchar(10), datepart(mm, @pdate))
    + @psep  + CONVERT (varchar(10), datepart(yy, @pdate))
END
```

La commande de suppression **`DROP FUNCTION`** permet de supprimer une fonction.

```
DROP FUNCTION dbo.fn_DateFormat
```

6. DEFINITIONS ET PRINCIPES DES PROCEDURES STOCKEES

Une procédure stockée est un **ensemble nommé d'instructions Transact-SQL**, **stocké dans la base de données**, sur le serveur, en version prête à l'emploi.

Les procédures stockées permettent d'exécuter des traitements sur les données en y intégrant toute la logique de programmation nécessaire (tests, boucles…) ; a minima, une procédure stockée contient une requête SQL, paramétrée ou non.

Les procédures stockées offrent une **méthode efficace** pour encapsuler les instructions Transact SQL **en vue d'exécutions répétitives**. Elles prennent en charge des fonctionnalités de programmation puissante : variables définies par l'utilisateur, exécution conditionnelle...

SQL Server fournit déjà un grand nombre de **procédures stockées systèmes** qui facilitent l'administration du SGBD et des bases installées (modification du paramétrage, ajout de messages d'erreurs, liste des tables d'une base…).

Les **procédures stockées système** sont stockées dans la base de données `master` et identifiées par le préfixe `sp_`

Les **procédures stockées utilisateur** sont créées dans les bases de données concernées.

Des **procédures stockées temporaires** sont créées dans la base de données `tempdb` et sont supprimées automatiquement.

Enfin, les **procédures stockées distantes** sont apelées à partir d'un serveur SQL Server distant.

A la création, les **instructions** qui composent une procédure stockée sont **vérifiées syntaxiquement** afin de sécuriser et accélérer leur utilisation future. SQL Server stocke ensuite le nom de la procédure stockée dans la vue catalogue **`sys.objects`** et son texte dans la vue catalogue **`sys.sql_module`** de la base de données courante : de ce fait, elles sont **accessibles à tous** les clients autorisés.

A la première exécution (ou si elle doit être recompilée), le processeur de requête la lit à partir de la la vue catalogue **`sys.sql_module`**. Elle est alors **compilée pour la rendre plus efficace et réellement réutilisable**. L'optimiseur de requêtes l'optimise, et stocke alors un **plan de requête compilé** dans le cache de procédure (zone mémoire utilisée par SQL Server); la procédure stockée s'exécute.

Les exécutions suivantes des procédures stockées sont **plus rapides**, car elles sont exécutées directement à partir du plan de requête optimisé du cache de procédure.

De plus, avec une procédure stockée, le **trafic réseau est réduit** : le client n'a besoin de fournir que le nom de la procédure et les valeurs de ses paramètres au lieu de l'ensemble des instructions SQL qui la composent.

7. CREATION DE PROCEDURE STOCKEES

Les procédures stockées sont créées dans la base de données courante.
Elles peuvent référencer des tables, des vues, des tables temporaires et des procédures stockées.

7.1 CREATION AVEC MANAGEMENT STUDIO

Sous **SQL Server Management Studio**, on mettra au point sa requête et après un test positif, on provoquera la création de la procédure : après avoir développé le dossier **Bases de données** du serveur, et sélectionné le dossier **Procédures stockées** dans le nœud **Programmabilité** de la base de donnée sélectionnée, vous choisirez **Nouvelle procédure** dans le menu contextuel.
Vous pourrez également sélectionner l'assistant **Création de procédures stockées** pour des procédure stockées de mise à jour de données.

7.2 CREATION EN MODE COMMANDE

Les procédures stockées sont créées avec l'instruction **`create procedure`** dont la syntaxe partielle est la suivante :

```
CREATE PROC[EDURE] [nom_schema.] nom_procédure[;number]
      [(parametre1 [, parametre2] ... [parametre255])]
      [WITH {RECOMPILE | ENCRYPTION | RECOMPILE, ENCRYPTION
      | EXECUTE AS}]
AS
      bloc d'instructions_SQL : BEGIN … END
EXTERNAL NAME nom_assembly.nom_classe.nom_méthode
```

Le nom de la procédure doit être unique dans la base de données, et ne pas excéder 128 caractères de long.

Les paramètres sont optionnels et leur nombre est de 1024 au maximum.

Les paramètres sont définis en tant que paramètres d'entrée, mais peuvent également retourner de l'information au programme appelant (paramètre OUTPUT).

Un **paramètre** se déclare selon la syntaxe suivante :

```
@nom_parametre [nom_schema.]type_de_données [= valeur par défaut] [Output]
```

`nom_parametre :` Le nom du paramètre débute obligatoirement par `@`

`[nom_schema.]type_de_données` : Type du paramètre (tous les types SQL Server sont acceptés, sauf les images)

Si `nom_schéma` n'est pas précisé, le moteur de base de données de SQL Server pointe sur le type de données dans l'ordre suivant :

- types de données SQL Server fournis par le système ;
- schéma par défaut de l'utilisateur actuel dans la base de données actuelle ;
- schéma **`dbo`** dans la base de données actuelle.

valeur par défaut : Spécifie une valeur par défaut du paramètre (constante ou la valeur NULL).

output indique que le paramètre est un paramètre de sortie.

WITH RECOMPILE indique que le plan de requête de la procédure ne sera pas stocké en mémoire cache lors de son exécution ; chaque exécution de la procédure stockée donnera lieu à une recompilation. Il peut être nécessaire de recompiler explicitement une procédure stockée lorsque :

- La procédure stockée est exécutée très rarement.
- Elle sera exécutée avec des valeurs de paramètres très différentes (pas optimal d'utiliser le plan d'exécution en mémoire cache).

L'option **WITH ENCRYPTION** empêchera les utilisateurs d'afficher le texte des procédures stockées.

Créer une procédure stockée nécessite l'autorisation CREATE PROCEDURE dans la base de données, et l'autorisation ALTER sur le schéma dans lequel la procédure est créée.

La clause **EXECUTE AS** spécifie le contexte de sécurité dans lequel la fonction définie par l'utilisateur est exécutée (Voir Sécurité dans SQL Server).

Toutes ces options permettent de s'adapter à tous les besoins ; comme toujours, les valeurs par défaut sont bien souvent suffisantes.

8. EXECUTION D'UNE PROCEDURE STOCKEE

Une procédure stockée peut être exécutée en spécifiant l'instruction **EXECUTE** (ou **EXEC**) avec le nom de la procédure et ses éventuels paramètres.

La valeur d'un paramètre d'entrée peut être définie en la passant à la procédure stockée par **référence** ou **position**.

La spécification d'un paramètre dans une instruction EXECUTE en utilisant le format :

```
EXEC[UTE] nom_procédure [@parametre = valeur] [,…n]
```

s'appelle un passage **par référence** : les noms des paramètres sont spécifiés, les paramètres peuvent être spécifiées dans n'importe que ordre, les paramètres ayant une valeur par défaut peuvent être omis.

Le passage de valeurs sans référence aux noms des paramètres s'appelle le passage **par position** :

```
EXEC[UTE] nom_procédure [valeur] [,…n]
```

Les valeurs doivent alors être spécifiées **dans l'ordre dans lequel les paramètres ont été définis dans la commande CREATE PROC**.

La passation de paramètres par référence est plus verbeuse mais plus sûre, et permet d'assurer la compatibilité quand la procédure évolue.

Pour utiliser un paramètre de sortie, le mot clé OUTPUT doit être également spécifié dans l'instruction EXECUTE.

Aucune autorisation n'est requise pour exécuter l'instruction `EXECUTE`. Cependant, des autorisations sont requises sur les objets référencés dans la chaîne `EXECUTE`. Par exemple, si la procédure contient une instruction `INSERT`, l'utilisateur appelant la procédure par l'instruction `EXECUTE` doit posséder l'autorisation `INSERT` sur la table cible.

9. EXEMPLES DE PROCEDURES STOCKEES SIMPLES

9.1 EXEMPLE 1: CRÉATION D'UNE PROCÉDURE STOCKÉE SANS PARAMÈTRE

Liste des employés dont le salaire n'est pas précisé :

```
CREATE PROCEDURE Lst_SalNp
AS
BEGIN
      SELECT prenom, nom
      FROM Employes
      WHERE salaire is NULL
END
```

Cette procédure stockée pourra être exécutée, une fois créée avec l'instruction :

```
EXEC Lst_SalNp
```
ou
```
EXECUTE Lst_SalNp
```

Cette procédure n'est constituée que d'une seule requête SQL ; elle sera plus efficace que si on l'avait lancée en soumettant directement son libellé.

9.2 EXEMPLE 2: CREATION D'UNE PROCEDURE STOCKEE AVEC UN PARAMETRE D'ENTREE

Liste des employés ayant un salaire supérieur à un salaire donné

```
CREATE PROCEDURE Lst_Sal
      @vsal int
AS
BEGIN
      SELECT prenom, nom, salaire
      FROM Employes
      WHERE salaire > @vsal
END
```

Cette procédure stockée pourra être exécutée une fois créée avec l'instruction (si je veux connaître les salariés gagnant plus de 16000) :

```
EXEC Lst_Sal @vsal=16000
```

ou

```
EXEC Lst_Sal 16000
```

9.3 Exemple 3: Creation d'une procedure stockee avec deux parametres d'entree et un parametre de sortie

Addition de 2 entiers

```
CREATE PROCEDURE Addition
      @fact1 smallint,
      @fact2 smallint,
      @result smallint OUTPUT
AS
BEGIN
      SET @result = @fact1 + @fact2
END
```

Elle sera exécutée avec les instructions suivantes :

```
DECLARE @res smallint
EXECUTE Addition 5,2,@Res OUTPUT
SELECT 'L''addition est égale à :', @Res
```

Le résultat reçu sera : « `L'addition est égale à : 7` »

9.4 Exemple 4: Creation d'une procedure stockee de recherche

Requête listant tous les employés dont le nom commence par une chaîne de caractères donnée.

```
CREATE PROCEDURE Empl_Commençantpar
      @vrech varchar(23)
AS
BEGIN
      DECLARE @vrechc varchar(24)
      SET @vrechc = @vrech + '%'
      SELECT noemp, prénom, nom,
            FROM Employés
            WHERE Nom like @vRechc
END
```

9.5 EXEMPLE 5: UTILISATION DE LA CLAUSE EXECUTE AS

La clause `EXECUTE AS` Indique le contexte de sécurité dans lequel la procédure stockée doit être exécutée.

```
CREATE PROCEDURE Empl_Commençantpar
WITH EXECUTE AS 'KARINE'
      @vrech varchar(23)
AS
      ….
```

Par exemple, Karine ayant l'autorisation `select` sur la table `Employes`, lorsque n'importe quel utilisateur appelle l'exécution de cette procédure, il a accès à la table `Employes` comme si c'était Karine.

9.6 EXEMPLE 6 : MISE A JOUR D'UN EMPLOYE

On passe en paramètre le code et le nom de l'employé à mettre à jour ; dans la procédure stockée, on contrôlera que la mise à jour a bien été effectuée grâce à la fonction `@@ROWCOUNT`

```
CREATE PROCEDURE Employé_Update
    @vnum varchar(4)
    @vnom varchar(25)
AS
BEGIN
      UPDATE Employés SET Nom = @vnom
      WHERE Noemp = @vnum
      IF (@@ROWCOUNT =0)
      BEGIN
            PRINT 'Avertissement : Aucune ligne modifiée'
            RETURN
      END
END
```

10. UTILISATION DE CODE RETOUR

L'instruction **RETURN** provoque une sortie, une interruption de l'exécution. Elle peut éventuellement renvoyer une valeur d'état de type entier (code de retour).

Un code de retour égal à 0 indique un succès.

Les valeurs comprises entre –1 et –14 sont utilisées par SQL SERVER pour indiquer différentes causes d'échec.

Les valeurs comprises entre –15 et –99 sont réservées à une utilisation future.

Si aucune valeur de retour utilisateur n'est spécifiée, la valeur renvoyée est 0.

Exemple : Contrôle de l'existence d'un employé dans la table Employé : s'il est inexistant, la procédure stockée renvoie le code de retour -100, sinon 0.

```
CREATE PROCEDURE Exist_Emp
      @codemp smallint
AS
BEGIN
      IF NOT EXISTS (SELECT noemp From Employés Where noemp = @codemp)
            RETURN -100
      ELSE
            RETURN 0
END
```

Exécutée avec les instructions suivantes :

```
DECLARE @retour int
EXECUTE @retour = Exist_Emp 00121
SELECT 'Le code retour est :', @Retour
```

Le résultat sera :

```
Le code retour est : -100
```

11. LA GESTION DES ERREURS

La variable système `@@ERROR` contient le numéro de l'erreur de la dernière instruction Transact-SQL exécutée. Elle est effacée et ré-initialisée chaque fois qu'une instruction est exécutée. Si l'instruction s'est exécutée avec succès, @@ERROR renvoie la valeur 0.

Exemple : Division par un nombre pouvant être égal à 0

```
CREATE PROCEDURE Divis
      @fact1 smallint
      @fact2 smallint
AS
BEGIN
      SELECT @fact1 / @fact2
      PRINT CASE @@ERROR
            WHEN 8134 THEN 'Division par 0 impossible !'
            WHEN 0 THEN 'Aucune erreur détectée'
            ELSE 'Erreur inconnue'
END
END
```

L'instruction `RAISERROR` renvoie un message d'erreur défini par l'utilisateur et place un indicateur système pour enregistrer le fait qu'une erreur s'est produite.

Comme `RAISERROR` ne provoque pas l'interruption de la procédure en cours, le développeur devra en général ajouter une instruction `RETURN`.

RAISERROR permet à l'application de lire une entrée dans la table système **`sys.messages`** ou de construire un message dynamiquement en spécifiant sa gravité et son état. Cette instruction peut écrire des messages d'erreur dans le journal des erreurs de SQL Server et dans le journal des applications de Windows.

Exemple:

```
CREATE PROCEDURE Test
      @codemp smallint
AS
BEGIN
      IF NOT EXISTS (SELECT noemp From Employés Where noemp = @codemp)
      BEGIN /* Employé inexistant */
            RAISERROR ('N° employé incorrect, 16, 1) WITH LOG
            RETURN -100
      END
END
```

Après l'exécution de ce code, la valeur de retour contiendra la valeur `-100` si l'employé n'existe pas. Un message d'erreur sera écrit dans le journal Applications de Windows et dans le journal des erreurs de SQL SERVER.

Pour utiliser un message d'erreur préenregistré, on codera :

```
RAISERROR (n°message, 16, 1)
```

Pour inclure le code de l'employé dans le texte du message, on codera :

```
RAISERROR (n°message, 16, 1, @codemp)
```

12. VISUALISATION D'INFORMATIONS

Pour obtenir des informations supplémentaires sur tous les types de procédures stockées, vous pouvez utiliser les procédures stockées système suivantes ou manipuler SQL Server Management Studio.

Procédure stockée	Informations
sp_help nom_proc	Affiche la liste de paramètres et leur type
sp_helptext nom_proc	Affiche le texte de la procédure stockée si non cryptée
sp_depends nom_proc	Enumère les objets qui dépendent de la procédure et les objets dont dépend cette procédure
sp_stored_procedures	Renvoie la liste des procédures stockées de la base de données en cours

13. MODIFICATION/SUPPRESSION DE PROCEDURES

La commande de modification **ALTER PROCEDURE** accepte les mêmes paramètres que CREATE PROCEDURE.

La commande de suppression **DROP PROCEDURE** permet de supprimer une procédure stockée.

CREDITS

ŒUVRE COLLECTIVE DE l'AFPA
Sous le pilotage de la DIIP et du centre d'ingénierie sectoriel Tertiaire-Services

Equipe de conception (IF, formateur, mediatiseur)
E. Cattanéo - formateur
B. Hézard - Formateur
Ch. Perrachon – Ingénieure de formation

Date de mise à jour : 16/12/15