# La gestion d'erreur sous SQL Server

**La gestion des erreurs sous SQL Server**

Avec la nouvelle version de SQL Server, une gestion améliorée des exceptions a été mise en place.
Nous verrons donc les méthodes utilisées pour SQL Server 2000 et SQL Server 2005.

### Introduction

Dans tout développement, il est impératif d'avoir une gestion d'erreur. Avec la nouvelle version de SQL Server, le langage TSQL a été amélioré en intégrant cette gestion d'erreur.

Nous verrons donc simplement comment utiliser cela dans les scripts SQL.

---

### Présentation

Lors de chaque erreur rencontrée, le moteur associe celle-ci avec le message associé. Les messages d'erreur sont stockés dans la base Master. On peut lister le contenu de cette table avec la commande :

```
SELECT * FROM sys.messages;
```

Si l'erreur rencontrée lors de l'exécution n'existe pas dans cette table, elle obtient le code 50000.

Le moteur ajoute aussi une notion de "sévérité" qui est notée par une valeur :

- Severity 0 - Information (T-SQL PRINT)
- Severity 1-9 - Information (not used by SQL Server)
- Severity 10 - Converted to severity 0
- Severity 11-16 - Programming Errors
- Severity 17-25 - Resource Error
- Severity 19 and above - must be sysadmin
- Severity 20-25 - Terminate the connection

Voyons comment on gère ces erreurs.

### Gestion sous SQL Server 2000

Avec la précédente version de SQL Server (2000), on ne pouvait tester la présence d'une erreur que par la variable système **@@ERROR**. Ainsi une requête provoquant une erreur charge la variable @@ERROR avec le code erreur correspondant.

On peut donc voir que la requête :

```
SELECT * FROM TABLEINEXISTANTE;
```

Provoque une erreur :

```
Msg 208, Niveau 16, État 1, Ligne 1 Nom d'objet 'TABLEINEXISTANTE' non valide.
```

Et charge la variable @@ERROR avec la valeur 208 :

```
SELECT @@ERROR
```

De ce fait, dans les scripts TSQL ou dans les procédures stockées, nous devions faire un test de validation avec cette variable système pour valider une transaction par un COMMIT ou l'annuler avec un ROLLBACK.

```
------------ DEBUT DE TRANSACTION -----------
BEGIN TRAN
  SELECT * FROM pubs.dbo.authors
  IF (@@ERROR = 0)
    BEGIN
      COMMIT
      PRINT 'EXECUTION OK : '+ CAST(@@ERROR AS VARCHAR(10))
    END
  ELSE
    BEGIN
      ROLLBACK
      PRINT 'ERREUR : '+ CAST(@@ERROR AS VARCHAR(10))
    END
--------------- FIN DE TRANSACTION -----------
```

Cette gestion pouvait être très lourde, car la variable @@ERROR ne conserve qu'un code d'erreur, et dans certains scripts ceci est trop limité.

Dans ces situations, la commande GOTO pouvait nous aider en demandant au moteur d'aller à un point donné du script TSQL, cela permet de faire une gestion globale de ces erreurs.

```
------------ DEBUT DE TRANSACTION -----------
BEGIN TRAN
  SELECT * FROM pubs.dbo.authors
  IF (@@ERROR = 0)
    BEGIN
      COMMIT
      GOTO AFFICHAGEOK
    END
  ELSE
    BEGIN
      ROLLBACK
      GOTO AFFICHAGEERREUR
    END

AFFICHAGEERREUR:
  PRINT 'ERREUR : '+ CAST(@@ERROR AS VARCHAR(10))
  SELECT @@ERROR
AFFICHAGEOK:
  PRINT 'EXECUTION OK : '+ CAST(@@ERROR AS VARCHAR(10))

--------------- FIN DE TRANSACTION -----------
```

Malgré tout, on sent bien que cette gestion devient vite complexe à gérer et à suivre. De plus, certaines erreurs ne sont pas captées par la variable, comme le cas des objets inexistants qui bloque directement le script.

Sous le langage TSQL de SQL Server 2005, ceci a été revu et une nouvelle gestion est apparue qui rapproche le TSQL d'un réel langage de développement.

**Gestion sous SQL Server 2005**

Avec SQL Server 2005, on conserve la variable @@ERROR (pour des raisons de compatibilité ascendante et de migration), mais une nouvelle notion a été ajoutée qui rapproche le TSQL des langages .NET :

- Le Try ... Catch

Le Try ... Catch permet de capter les erreurs dont la sévérité est supérieure à 10 afin de passer le script dans une partie prévue à cet effet.

Cette gestion d'erreur se fait avec les mots clé (il ne doit rien y avoir entre le END TRY et le BEGIN CATCH, sinon vous aurez une erreur de syntaxe) :

```
.....
BEGIN TRY
  .....
END TRY
BEGIN CATCH
  .....
END CATCH
.....
```

On retrouve en plus de cette nouvelle organisation des fonctions système qui permettent d'obtenir plus d'informations sur l'erreur éventuelle :

- ERROR_NUMBER() : renvoie le numéro de l'erreur
- ERROR_SEVERITY() : renvoie la gravité de l'erreur
- ERROR_STATE() : renvoie le numéro d'état de l'erreur
- ERROR_PROCEDURE() : renvoie le nom de la procédure stockée ou du déclencheur dans lequel s'est produit l'erreur
- ERROR_LINE() : renvoie le numéro de ligne au sein de la routine qui a entraîné l'erreur
- ERROR_MESSAGE() : renvoie le texte complet du message d'erreur. Ce texte comprend les valeurs fournies pour tous les paramètres substituables, tels que les longueurs, les noms d'objet ou les heures

Voila un exemple d'utilisation de Try ... Catch :

```
BEGIN TRY
   -- Création d'une erreur de division par 0
   SELECT 1/0;
END TRY
BEGIN CATCH
   SELECT
      ERROR_NUMBER() AS NumeroErreur,
      ERROR_SEVERITY() AS SeveriteErreur,
      ERROR_STATE() AS EtatErreur,
      ERROR_PROCEDURE() AS ProcedureErreur,
      ERROR_LINE() AS LigneErreur,
      ERROR_MESSAGE() AS MessageErreur;
END CATCH;
GO
```

**Attention :**
Lors du passage par un Try ... Catch, vous ne transférez pas l'erreur rencontrée au programme ayant appelé le code (TSQL ou Procédure Stockée). De ce fait, le "client", ne verra pas qu'une erreur a été rencontrée (sauf si le code dans le Catch le lui signale).

Le Try ... Catch ne capte pas toutes les erreurs, notamment :

- Les messages d'information (Sévérité d'erreur inférieur à 10)
- Les messages avec une sévérité supérieure à 20 (perte de connexion par exemple)
- Erreur sur la communication entre le client et le serveur SQL
- Coupure de la session par un KILL d'un administrateur
- Erreur de syntaxe TSQL qui entraine une erreur de compilation

- Problème de résolution de nom d'objet

Dans les cas cités ci-dessus, la solution est l'encadrement par deux Try ... Catch. Ainsi le premier bloque ne passe pas dans le catch et renvoie le message directement au second test qui lui saura le gérer.

Ces deux boucles ne doivent pas être directement imbriquées car le compilateur bloque avant le passage dans le CATCH.

Nous sommes dans ce cas obligé de passer par une procédure stockée intermédiaire dont voila un exemple.

```
-------------------------------------------------
-- On teste l'existence de la procédure de test
-------------------------------------------------
IF OBJECT_ID ( N'usp_ExampleProc', N'P' ) IS NOT NULL
   DROP PROCEDURE usp_ExampleProc;
GO

-------------------------------------------------
-- Création de la procédure qui provoque une
-- erreur de compilation
-------------------------------------------------
CREATE PROCEDURE usp_ExampleProc
AS
   SELECT * FROM NonexistentTable;
GO

-------------------------------------------------
-- Script d'exécution du test d'erreur
-------------------------------------------------
BEGIN TRY
   EXECUTE usp_ExampleProc
END TRY
BEGIN CATCH
   SELECT
      ERROR_NUMBER() as ErrorNumber,
      ERROR_MESSAGE() as ErrorMessage;
END CATCH;
-------------------------------------------------
```

**La gestion des transactions sous SQL Server 2005**

En plus de cette partie, nous avons un contrôle de la validité d'une transaction. Une fonction est fournie afin de tester cette validité :

- XACT_STATE

Celle-ci peut avoir 3 valeurs :

- -1 : La transaction est considérée comme valide et peut être "Commitée"
- 0 : Etat de base, il n'y a aucune transaction à valider
- 1 : La transaction est invalide et doit être "Roll Backée"

Associée à cette fonction, nous avons une variable système qui compte le nombre de transactions en cours :

- @@TRANCOUNT

Afin d'utiliser XACT_STATE, il nous faut activer :

- SET XACT_ABORT ON;

Ainsi le script d'exemple suivant (issu de la MSDN) nous montre comment utiliser ces deux tests pour que la base conserve son intégrité lorsqu'une erreur de violation de contrainte apparaît sur une suppression.

```
USE AdventureWorks;
GO

-- Check to see if this stored procedure exists.
IF OBJECT_ID (N'usp_GetErrorInfo', N'P') IS NOT NULL
   DROP PROCEDURE usp_GetErrorInfo;
GO

-- Create procedure to retrieve error information.
CREATE PROCEDURE usp_GetErrorInfo
AS
   SELECT
      ERROR_NUMBER() AS ErrorNumber,
      ERROR_SEVERITY() AS ErrorSeverity,
      ERROR_STATE() as ErrorState,
      ERROR_LINE () as ErrorLine,
      ERROR_PROCEDURE() as ErrorProcedure,
      ERROR_MESSAGE() as ErrorMessage;
GO

-- SET XACT_ABORT ON will render the transaction uncommittable
-- when the constraint violation occurs.
SET XACT_ABORT ON;

BEGIN TRY
   BEGIN TRANSACTION;
      -- A foreign key constrain exists on this table. This
      -- statement will generate a constraint violation error.
      DELETE FROM Production.Product
         WHERE ProductID = 980;

   -- If the DELETE statement succeeds, commit the transaction.
   COMMIT TRANSACTION;
END TRY
BEGIN CATCH
   -- Execute error retrieval routine.
   EXECUTE usp_GetErrorInfo;

   -- Test XACT_STATE:
      -- If 1, the transaction is committable.
      -- If -1, the transaction is uncommittable and should
      --     be rolled back.
      -- XACT_STATE = 0 means that there is no transaction and
      --     a COMMIT or ROLLBACK would generate an error.

   -- Test if the transaction is uncommittable.
   IF (XACT_STATE()) = -1
   BEGIN
      PRINT
         N'The transaction is in an uncommittable state.' +
         'Rolling back transaction.'
      ROLLBACK TRANSACTION;
   END;

   -- Test if the transaction is committable
   IF (XACT_STATE()) = 1
   BEGIN
      PRINT
         N'The transaction is committable.' +
         'Committing transaction.'
      COMMIT TRANSACTION;
   END;
END CATCH;
GO
```

## Conclusion

Avec ces nouveautés, nous voyons bien que le langage TSQL se rapproche de plus en plus d'un réel langage de développement tel que le C#. Nous pouvons alors créer des scripts plus évolués.

Attention toutefois à bien garder à l'esprit que SQL Server est et doit rester le moteur de stockage des données, il ne faut pas considérer que la couche métier d'une application doit se déplacer dans la base de données (aussi puissant soit le langage utilisé).

---

Voici quelques liens utiles si cet article vous a intéressé :

- **Le Try ... Catch (TSQL)**
- **GOTO (TSQL)**
- **@@ERROR (TSQL)**
- **XACT_STATE**
- **SQL Skills**
- **Blog de Bob Beauchemin**
- **SQL SERVER 2005 - CHANGER LE SCHEMA DES OBJETS DANS LES OBJETS APPELANT**

---

En vous souhaitant de bons projets de développement.

Romelard Fabrice (alias F___)

Consultant Technique **ilem SA**