Database Management
Scripts

By: Meyer Tanuan (2022), Rick Kozak (2019), Glenn Paulley (2015), John Mckay (2011)

DB Objects

- Script
- User-Defined Function (UDF)
- Stored Procedure (SP)

Script

- Script is a file stored on disk containing one or more SQL Statements
- Scripts do not accept parameters
- The groups of the statements comprising the script can be further divided into batches.
- To indicate the end of batch you use ‘GO’ command.

Batch Commands

Require Batch End

- Create Schema
- Create Trigger
- Create Procedure
- Create Function
- Create Database

Do not Require Batch End

- Create Table command
- Create View Command
- Any DML command, performing INSERT, UPDATE, DELETE and SELECT

Batch Command Example

```
CREATE DATABASE DbTest
GO
USE DbTest
CREATE TABLE t1 (f1 int)
CREATE TABLE t2 (f1 int)
```

Statements

- **USE** - Changes DB context
- **PRINT** – prints a message in the batch output channel
- **DECLARE** - declares a local variable
- **SET** - sets the value of the local variable
- **EXEC** - executes a stored procedure

USE Statement

- Use statement changes the database in which the statements will be executed
- It is especially important when the batch is executed as script outside of the interactive query window
- In the offline (batch) execution the default database is always (or almost always) **master.** Users should not create objects in this database, hence the actual target database must be explicitly specified.

USE Statement Example

```
USE SIS;
DECLARE @totalPaid MONEY;
SET @totalPaid =
      (SELECT SUM(amount)
       FROM Payment)
PRINT @totalPaid
```

Variables

- Variables, which are also called scalar variables, can hold a single value
- Variables can be used in the SELECT statements as holders of the result or as holders of the condition values
- Variables cannot substitute object names (table, view or procedure names)

Variables

- Scalar Variables are defined with a data type and designated to hold a singular scalar value
- Name of the variable always starts with @. Use long descriptive names.
- The scope of the variable is the batch in which it defined. The variable cannot be referred outside of the batch.
- Use SET or SELECT to assign value to a variable.
- Variables can be used in expressions

Table Variables

- Table variables are declared in the same manner as the local variables using TABLE data type

```
DECLARE @TableVar TABLE (f1 INT)

INSERT INTO @TableVar VALUES (1)

SELECT *
FROM @TableVar
```

Table Variables

- Table variable can be filled as a regular table
- The same SELECT operations can be performed

```
INSERT @TableVar
SELECT id FROM Payment


SELECT * FROM @TableVar
```

Temporary Table

- Temporary tables can be created within a script
- There are two types of tables:
  - Local - denoted by # sign
  - Global - denoted by ## sign
- The scope of local temporary tables is a database session. They are very useful in complex scripts
- Global temporary tables are visible to all sessions
- Temporary table name is limited to 116 characters

Temporary Table Example

```
SELECT TOP 1 StudentNumber
INTO #Student
FROM Payment

SELECT * FROM #Student
```

Flow of Control

- IF..ELSE – Branches flow based on condition
- CASE..END – flow based on condition
- BEGIN..END – defines a statement block
- WHILE – defines beginning of a loop
- BREAK – exits innermost WHILE loop
- CONTINUE – returns to the beginning of loop
- TRY..CATCH – exception handling

IF..ELSE

- Works similarly to the same statement in most procedural languages
- If branch contains more than one statement BEGIN..END block is required in either IF or ELSE branch

```
IF OBJECT_ID('Payment') IS NULL
  CREATE TABLE Payment (f1 int)
ELSE
  PRINT 'Payment table exists'
```

CASE..END

- Works similarly to the switch/case statements in most procedural languages

```
SELECT name, object_id, type,
  CASE type
    WHEN 'C' THEN 'Check Constraint'
…
    WHEN 'V' THEN 'View'
    ELSE 'Other object type'
  END AS [Object Type Name]
FROM sys.objects
WHERE LEN(type) = 1
ORDER BY type;
```

TRY..CATCH

- The intention of using a try-catch block is the same as in any procedural language
- The actual statements incorporate BEGIN and END in order to show the blocks of error handling
- Functions ERROR_NUMBER() and ERROR_MESSAGE() can be used to show which database or script error occurred

TRY..CATCH Example

```
BEGIN TRY
   CREATE TABLE Payment (f1 int)
END TRY
BEGIN CATCH
   PRINT 'Payment table exists'
   PRINT ERROR_MESSAGE()
END CATCH
```

System Functions

- @@IDENTITY – shows last generated number
- @@ROWCOUNT – shows the number of rows affected by the statement
- @@ERROR – shows the last error number
- @@SERVERNAME – shows local name of the server
- SYSTEM_USER – shows username of the current user

Session Settings

- SET DATEFORMAT mdy – sets date format.
- SET NOCOUNT {ON|OFF}
  - controls whether statement returns the number of the affected rows
- SET ANSI_NULLS {ON|OFF}
  - ON requires to write WHERE X IS NULL
  - OFF allows "= NULL" to return the same number of rows
- SET ROWCOUNT nn
  - where nn is the number of rows to be processed.
  - The default is zero, which specifies that all rows are processed

User Defined Functions (UDF)

- User defined functions (UDF) always has a return data type
- UDF can return a scalar or a table value
- Use SELECT to perform a UDF
- UDF can also be called in a stored procedure

UDF Example (udf1)

```
CREATE FUNCTION dbo.ToFahrenheit
   (@Celsius decimal(10,2))
   RETURNS decimal(10,2)
AS
BEGIN
   DECLARE @Fahrenheit
      decimal(10,2);
   SET @Fahrenheit =
      (@Celsius * 1.8 + 32);
   RETURN @Fahrenheit
END
GO

SELECT dbo.ToFahrenheit(100)
```

UDF Example (udf2)

```
CREATE FUNCTION
  dbo.getSumPayment()
  RETURNS MONEY
BEGIN
  RETURN (
    SELECT SUM(amount)
    FROM Payment
  )
END
GO

SELECT dbo.getSumPayment()
  AS [Total Payment Collected]
```

SQLCMD

- SQLCMD is a power shell, which allows executing SQL statements outside of sql server management studio
- **sqlcmd -?**
  - Provides list of command line options
- This utility allows executing batch scripts
  - E.g., to create RPT file from the command line
  - **sqlcmd -S localhost\SQLEXPRESS19 -i ex1.sql -o ex1.rpt**

Stored Procedure (SP)

- A script which is stored within the database
- When executed for the first time, the code of the procedure is ‘compiled’ and *execution plan* is created
- The subsequent executions do not require the compilation (unless the code is changed), which makes stored procedure executing faster than any script
- Accepts parameters, unlike file based scripts
- Only one batch allowed

Stored Procedure

- Convention: starts with “usp” or “p”
- Executing a stored procedure can be performed from a script or from another stored procedure
- Stored procedures are permanent. They are stored within the database used when procedure was created
- Stored Procedures can also be local starting with # and global starting with ##. These stored procedures are stored in database temporary storage

Stored Procedure Options

- **WITH RECOMPILE** - makes the database recompile the stored procedure every time it is executed. Required for stored procedures which use volatile data
- **WITH ENCRYPTION** - makes database encrypt the stored procedure instructions. If this option is chosen, database will warn the user instead of showing the code

Stored Procedure (sp1)

- Stored Procedure is created by using CREATE PROC statement
  - executed by using EXEC statement

```
CREATE PROC pGetAllEmployees AS
   SELECT e.number, p.firstName
      , p.lastName
   FROM Employee e JOIN Person p
      ON e.number = p.number
   ORDER BY p.lastName
GO

EXEC pGetAllEmployees
```

Stored Procedure Parameters

- Parameters of the stored procedure must start with @
- There are two types of parameters:
  - Input parameters, used to convey data to the stored procedure
  - Output parameters, used to get the data from the stored procedure

Stored Procedure with Parameters (sp2)

```
CREATE PROC pGetEmployee
  @School CHAR(3),
  @Name VARCHAR(20)
AS
  SELECT e.number,
    p.firstName, p.lastName
  FROM Employee e JOIN Person p
    ON e.number = p.number
  WHERE schoolCode = @School
    AND lastName LIKE @Name
GO

EXEC pGetEmployee
'EIT','YUROVIC'
```

Optional Parameter Example (sp3)

```
CREATE PROC pGetEmployeeBySchool
   @School CHAR(3),
   @Name VARCHAR(20) = '%'
AS
   SELECT e.number, p.firstName
      , p.lastName
   FROM Employee e JOIN Person p
      ON e.number = p.number
   WHERE schoolCode = @School
      AND lastName LIKE @Name
GO

EXEC pGetEmployeeBySchool 'EIT'
```

With Output Parameter Example (sp4a)

```
CREATE PROC pGetAmount
  @studentNum VARCHAR(10),
  @total MONEY OUTPUT
AS
  SELECT @total =
    (SELECT SUM(amount)
     FROM Payment
     WHERE studentNumber
       LIKE @studentNum)
GO

DECLARE @totalAll MONEY
EXEC pGetAmount '%',
  @totalAll OUTPUT
PRINT @totalAll
```

Call Parameters By Name Example (sp4b)

- Call parameters by name and order is not important

```
DECLARE @totalByName MONEY
EXEC pGetAmount
   @studentNum = '%',
   @total = @totalByName OUTPUT

PRINT @totalByName
```

Returning a value (sp5)

```
CREATE PROC pGetAmountR
  @studentNum VARCHAR(10)
AS
  DECLARE @total MONEY
  SELECT @total =
    (SELECT SUM(amount)
     FROM Payment
     WHERE studentNumber
       LIKE @studentNum)
  RETURN @total
GO

DECLARE @totalR MONEY
EXEC
  @totalR = pGetAmountR '%'
PRINT @totalR
```

System Stored Procedures

- Starts with “sp”
- SP_HELP – returns information about database objects
- SP_HELPTEXT – shows the text of any stored procedure, user defined function or trigger
- SP_HELPDB – returns information about the database
- SP_WHO – returns information about system users
- SP_COLUMNS – returns information on the columns of table or view

View All Stored Procedures

- view all stored procedures in current database that starts with "p"

```
SELECT SCHEMA_NAME(schema_id)
    AS SchemaName
  , name AS ProcedureName
FROM sys.procedures
WHERE name LIKE 'p%'
ORDER BY SchemaName
```