Topic: Stored Procedures

A **stored procedure** is a set of executable SQL statements saved under a **procedure name** and callable from other scripts or from an external database client (such as code on a web server). Stored procedures offer a number of benefits to database and application developers --

- code reuse
- strict calling interface
- performance optimization
- protection from SQL manipulation exploits
- try/catch bracketing
- transaction bracketing and table lock holds

Generally, if you are writing a script intended to be used over and over, you **should** code it as a stored procedure and work out a calling interface for it. If you are writing a script that will include data parameters from untrusted sources (i.e. user input) then you **must** use stored procedures to gain isolation from SQL manipulation exploits. Let's walk through a simple stored procedure example.

Creating A Stored Procedure

The recommended template for creating a stored procedure is shown below --

```
-- Create SP if it does not yet exist.
IF NOT EXISTS (SELECT * FROM sys.objects WHERE type = 'P' AND OBJECT_ID = OBJECT_ID('PMyProcedure'))
        EXEC('CREATE PROCEDURE [PMyProcedure] AS BEGIN SET NOCOUNT ON; END');
GO

-- Update SP to hold the script below.
ALTER PROCEDURE [PMyProcedure] @MyInput nvarchar(max), @Result [nvarchar](max) OUTPUT

AS

-- SET NOCOUNT ON to suppress the "number of rows retrieved" message and boost performance slightly.
-- Please see https://weblogs.sqlteam.com/dang/archive/2007/10/20/Use-Caution-with-Explicit-Transactions-in-Stored-Procedures.aspx

SET NOCOUNT ON;
SET XACT_ABORT ON;

BEGIN TRANSACTION T1            -- Delete T1 if you don't need explicit transaction bracketing.

        BEGIN TRY
```

```
                -- Do your awesome stuff in here.

                -- Commit and return.
                COMMIT TRANSACTION T1
                RETURN 28 -- or any "success" code you wish.

        END TRY

        BEGIN CATCH

                ROLLBACK TRANSACTION T1
                DECLARE @msg nvarchar(2048) = error_message()
                RAISERROR (@msg, 16, 1)
                RETURN -1

        END CATCH

GO
```

You can see it's quite straightforward. You basically just "wrap" your purpose-built script code in a CREATE PROCEDURE or ALTER PROCEDURE block, adding in transaction wrapping and try/catch if needed.

Stored procedures can take any SQL type as arguments. Any number of arguments can be specified, and arguments can be input or output. It is a good practice to provide a default value for parameters in your stored procedure call, if appropriate. Arguments are listed after the ALTER PROCEDURE statement --

```
ALTER PROCEDURE [PMyProcedure] @MyInput nvarchar(max) = '', @Result [nvarchar](max) OUTPUT
```

Calling A Stored Procedure

After you have created a stored procedure, you can call it from any SQL script (or even from another stored procedure, although this practice is somewhat discouraged). It's easy: just use the EXEC statement --

```
DECLARE @ReturnValue int, @MyInput nvarchar(max), @Result nvarchar(max)

SET @MyInput = N'Some text that is used by the procedure somehow';

EXEC @ReturnValue = [dbo].PMyProcedure @MyInput = @MyInput, @Result = @Result OUTPUT
```

Notice you need to declare the variables that will contain the data used to call the stored procedure. Notice also that parameter values are usually "named" in the calling sequence (this is optional but recommended for clarity).

Passing A Table To A Stored Procedure

You can pass any primitive type to a stored procedure (int, nvarchar, float, money, and so on). This is handy and addresses many use-cases. But what if you needed to write a stored procedure to process a set of records at once? For instance, what if your procedure was supposed to add a bunch of line items to an invoice? How would you structure this interface?

Fortunately this is no longer a problem (prior to SQL 2008 it was impossible). It is now possible to pass an entire table to a stored procedure. And it's not even difficult:

1. Create a user-defined table type to represent the data you want to send
2. Pass the table to your procedure using the table type to declare it, and also mark it READONLY
3. Use the table just like any other table inside your stored procedure

Let's take this apart:

1/ Create a user-defined table type

You can create new data types in MSSQL! They can be tables! This is a really handy language enhancement. The syntax is almost identical to the CREATE TABLE syntax you have already learned. Below is the statement to create a new table TYPE called **LineItemTableType**. Once you have created this type, you can use it in any scripts or stored procedures in your database application.

```
CREATE TYPE LineItemTableType AS TABLE
(
    InvoiceId int NOT NULL,
    LineItemId int NOT NULL,
    ItemAmount money NOT NULL,
    ItemDescription nvarchar(100) NOT NULL
)
```

2/ Pass the table to your procedure

This part is similar to passing any other parameter to a stored procedure. Don't forget to tag the table type with the READONLY keyword.

```
CREATE PROC AddLineItems
    @LineItems LineItemTableType READONLY
AS

    ...
```

3/ Use the table inside your procedure

Nothing special here either. Once inside your procedure, the table behaves just like any other table, except that you can't modify it. You can use it in SELECT statements, JOINS, subqueries, and CTEs:

```
CREATE PROC AddLineItems
```

```
    @LineItems LineItemTableType READONLY
AS

    INSERT INTO TMasterLineItems
        SELECT * FROM @LineItems
```

When you create a table type, it is persistent. This means that your table type definition is stored in your database for future use. To change your table type you'll need to remove any pre-existing definition. Also, if your table type is used in the interface to any stored procedures, you will not be able to delete it! The act of using a table type in a stored procedure creates a dependency on the type definition. You can't delete the definition without deleting the procedure first.

Here is a complete example of a user-defined table type and a stored procedure that uses it:

```
--
-- Delete any of the objects in this example if they already exist.  Note we have
-- to delete the stored procedure first, because it will "lock" the user-defined
-- table type otherwise.
--
DROP PROC IF EXISTS PAddLineItems
DROP TABLE IF EXISTS TMasterLineItems
DROP TYPE IF EXISTS LineItemTableType

--
-- Create a new user-defined table type to represent a set of line items on an order.
--
CREATE TABLE TMasterLineItems
(
    InvoiceId int NOT NULL,
    LineItemId int NOT NULL,
    ItemAmount money NOT NULL,
    ItemDescription nvarchar(100) NOT NULL
)
GO

--
-- Create a table to store line items for an order.  Notice the schema matches the
user-defined table type.
--
CREATE TYPE LineItemTableType AS TABLE
(
    InvoiceId int NOT NULL,
    LineItemId int NOT NULL,
    ItemAmount money NOT NULL,
    ItemDescription nvarchar(100) NOT NULL
)
GO

--
-- Create a stored procedure that accepts a table and adds all rows within to the
TMasterLineItems table.
--
CREATE PROC PAddLineItems
    @LineItems LineItemTableType READONLY
AS

    INSERT INTO TMasterLineItems
        SELECT * FROM @LineItems

GO

--
-- Test out this setup.  First create some sample data.
```

```
--

DECLARE @NewLineItems AS LineItemTableType;
INSERT INTO @NewLineItems VALUES (1, 1, 123.99, 'Merchandise item');
INSERT INTO @NewLineItems VALUES (1, 2, 15.99, 'Shipping charges');
INSERT INTO @NewLineItems VALUES (1, 3, 9.50, 'Tax');
INSERT INTO @NewLineItems VALUES (1, 4, 123.99+15.99+9.50, 'Total');

--
-- Run the stored procedure.
--
EXEC PAddLineItems @LineItems = @NewLineItems

--
-- Inspect the outcome.
--
SELECT * FROM @NewLineItems
SELECT * FROM TMasterLineItems
GO
```

➜ Practical Considerations

There are some useful things to know about stored procedures --

- You can't use the GO statement within a stored procedure
- You can't switch database contexts within a stored procedure (i.e. the USE statement)
- If you use table or record locks in a stored procedure, make sure you release them before you exit!
- Like all scripts, stored procedures can be difficult to debug; make sure you build your queries in a natural and logical order
- If you pass a user-defined table type to a stored procedure, the table is read-only
- If you pass a user-defined table type to a stored procedure, you cannot introduce foreign keys on the table

➜ Readings

Please review this material to explore this topic further --

- https://www.essentialsql.com/introduction-stored-procedures/
- https://www.mssqltips.com/sqlservertutorial/160/sql-server-stored-procedure-tutorial/
- https://www.youtube.com/watch?v=fOBDISV8cvY

➜ Practice

1. Create and test a stored procedure to extract matching records from your **TStudents** table (the one we made during indexing tests), where we match values based on the first characters of the student's first name. For performance reasons, limit your result set to 100 records.

2. Create and test a stored procedure to add a student to the **TStudents** table, returning the student's unique identifier on success.

3. Explain how you would create a stored procedure to insert a batch of students into the **TStudents** table, assuming you had some kind of array of new students to start with (don't implement this, just explore if it is feasible and how you would do it).

4. Explain why we have the option of bracketing everything inside a stored procedure in a **TRANSACTION**. What does that achieve (hint: think ACID)?

5. Explain how using stored procedures offers protection against SQL manipulation attacks.

6. Create and test a stored procedure to update the final grade for all students in a specified course (identified by course name) to a specified grade. Use an EXEC call to invoke the stored procedure without naming parameters explicitly (e.g. pass by position). Ensure that if no grade is specified in the EXEC call, a default grade of 100% is assumed.

7. Create and test a stored procedure to add a new student to **TStudents**, a new course to **TCourses**, and a new registration to **TRegistrations** with the new student in the new course. Make sure that if any part of this fails, the database is left in the same condition as before the SP was executed.

8. Create and test a user defined table type to represent a batch of students, and a stored procedure to add this batch to **TStudents**. Make sure your procedure fails atomically if it encounters any referential integrity issues such as a duplicate **StudentId**.

9. Create and test a user defined table type to represent a batch of student assessment updates, and a stored procedure to process this batch into **TRegistrations**. Your user defined table type should use a schema like:

   ```
   CourseId int
   StudentId int
   FinalGrade int
   ```

   Make sure your procedure fails atomically if it encounters any referential integrity issues such as unknown **StudentId** / **CourseId**. The update should only change the **FinalGrade** attribute of **TRegistrations**.

10. [100 Bt Question] If you didn't have user-defined table types available to you in MSSQL 2016, how would you design a stored procedure to accept sets of data (hint: yes, it is possible).