**Lesson 2: Designing Stored Procedures**

Estimated lesson time: 30 minutes

Views can contain only a single SELECT statement and can be used only to retrieve data. Views are not suitable for situations in which the resulting result set (and, hence, the T-SQL query to be executed) depends on external values, on data that needs to be validated, on decisions that need to be made, or on conditions that need to be checked to construct the expected result set dynamically. Views also are not suitable for situations in which the expected result is not a result set but rather a scalar value. Instead, in these cases, you would use stored procedures.

Similar to views, stored procedures do not store data. Stored procedures store the T-SQL queries they contain, so they give designers the ability to package programming logic that is executed and that queries data in real time. Unlike with views, the T-SQL code inside a stored procedure is stored in pre-compiled format. So a query plan has already been calculated for the stored procedure, which translates into better execution performance.

**IMPORTANT Reusing programming logic**

You can define as many stored procedures as you want without causing any toll on the system. Look for programming logic that can be packaged to be reused by multiple programs or by other stored procedures.

Even though stored procedures can contain any number and type of SQL statements (except some CREATE statements), in this lesson, you will focus on designing stored procedures to retrieve data. You will ignore, for now, stored procedures used to update data or for data-definition language (DDL) operations.

When designing stored procedures, you need to consider several design questions:

* What type of stored procedure do you need?
* What type of data will the stored procedure return?
* What input, output, and optional parameters do you need to define for the stored procedure?
* What status value do you need to define for the stored procedure to return?
* What error handling routines do you need to include?
* What security context should the stored procedure execute under?

**What Type of Stored Procedure Do You Need?**

The first stored procedure design decision you need to make is to choose the right technology for implementing the procedure. SQL Server 2005 supports three types of user-defined stored procedures that return data: T-SQL stored procedures, *common language runtime (CLR)* stored procedures, and extended stored procedures.

**T-SQL Stored Procedures**

This type of stored procedure is completely written in the T-SQL programming language, and as explained previously, it can contain any number and type of SQL statements. This type of stored procedure is usually used for set-based operations, for data-management language (DML) operations, and when necessary to manipulate heavy loads of data.

**CLR Stored Procedures**

This type of stored procedure is written in any Microsoft .NET programming language (usually Microsoft Visual Basic .NET or C# .NET). The main advantage of this type of procedure is that all the .NET base class libraries are available to use, so operations that are difficult to develop in T-SQL (such as XML manipulation, compression, encryption, string operations, complex mathematical calculations, and so on) are very easy to implement with .NET. This type of stored procedure is usually used to encapsulate reusable procedural code that must run inside the database server. It is not recommended for set-based operations or to manipulate heavy loads of data.

CLR stored procedures are outside the scope of this chapter; for more information about them, see the “CLR Stored Procedures” topic in SQL Server 2005 Books Online at .

**Extended Stored Procedures**

This type of stored procedure is usually written in the C programming language. The main advantage of extended stored procedures is that they are compiled routines that run natively inside the SQL Server address space. They are usually used to encapsulate complex scientific or mathematical routines. CLR stored procedures provide a more robust and secure alternative to writing extended stored procedures.

**IMPORTANT Avoid extended stored procedures**

Microsoft has announced that it will remove extended stored procedures in a future version of SQL Server and recommends that you avoid using them.

**What Type of Data Will the Stored Procedure Return?**

Stored procedures can return two types of data: tabular result sets and scalar values. Tabular result sets are returned when you include a SELECT statement inside the stored procedure code. The SELECT statement can be filtered by values coming from input parameters or from calculated local variables.

There are two ways to return scalar values out of a stored procedure: by using OUTPUT parameters and by setting the stored procedure return value. Remember that stored procedures do not return values in place of their names, and they cannot be used directly in an expression. You can run them only by using the EXECUTE statement.

**Defining Input, Output, and Optional Parameters for the Stored Procedure**

When designing the stored procedure’s input and output parameters, take special care in choosing the appropriate number of parameters, their names, their data types, their default values, and their direction.

**Choosing the Number of Parameters**

Stored procedures are like an application programming interface (API) that the database exposes to external applications and callers. Each stored procedure defines an operation contract composed by the number of parameters that it exposes and their data types. It is called an operation contract because once set, it should not be broken (unless you are using optional parameters, as explained shortly). When designing a stored procedure, carefully choose how many input, output, and optional parameters are needed. If you find yourself having to add new input parameters (required parameters) in the life cycle of a stored procedure, consider creating a new version of the stored procedure so that earlier clients can still use the previous operation contract, while newer clients will support the new parameters.

**Choosing the Name for a Parameter**

Parameter names should be self-explanatory. The parameter name must transmit to the caller the parameter’s intention. The recommendation is to avoid weird abbreviations and to be consistent. (The same word means the same thing throughout the database.) The maximum identifier size in SQL Server 2005 is 128 characters, so there is plenty of room for good naming practices!

**Choosing the Data Type for a Parameter**

The parameter’s data type constrains the type of information that can be sent into that parameter. The recommendation is to use the data type that best fits the type and size of information the parameter will contain as well as providing filters to avoid overflow problems and security issues. For example, say you are considering whether to use the *integer* data type, which accepts values up to 32,000 characters long, or the *tinyint* data type, which accepts values up to 128 characters long. If you know that the parameter value should never be more than 128 characters long, select the *tinyint* data type.

**Choosing Between Input, Output, and Optional Parameters**

For all parameters declared in a stored procedure, the external caller can provide input values. These values can be consumed inside the stored procedure, but any modification to those values inside the procedure will not be reflected to the outside caller.

To pass values from inside the stored procedure to the outside caller, the parameter must be declared as OUTPUT and called with the OUTPUT modifier. By specifying the OUTPUT modifier, the parameter is passed by reference, meaning that any changes to the parameter’s value inside the stored procedure will be copied back to the caller.

Default values can be specified for both input and output parameters. A default value indicates that the parameter is optional, so if the outside caller does not specify a value when calling the stored procedure, the default value will be used. Look at the following stored procedure parameter definition in this code example:

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_c9657f2f-3f80-4146-b319-facec5576fa6');" \o "Copy to clipboard.)

USE [AdventureWorks]

GO

CREATE PROCEDURE [Sales].[udpGetSalesByTerritory]

@TerritoryID int,

@SumTotalSubTotal money = 10000 OUTPUT,

@SumTotalDue money OUTPUT

AS

SELECT

@SumTotalSubTotal = SUM(SubTotal),

@SumTotalDue = SUM(TotalDue)

FROM Sales.SalesOrderHeader

WHERE TerritoryID = @TerritoryID) AND

(SubTotal > @SumTotalSubTotal)

The @TerritoryID parameter is an input parameter, so the caller must supply an input value of type *int*. The @SumTotalSubTotal parameter is optional, so the caller can supply a new value of type *money* or use the default value of *10,000*. The @SumTotalDue parameter is an OUTPUT parameter, so the caller is not forced to supply an input value. If the caller declares the call by specifying the OUTPUT modifier, the value is copied back to the caller after the stored procedure execution.

The following code example shows how to call the preceding stored procedure:

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_51d0a765-be91-4f03-855d-732694a101ab');)

USE [AdventureWorks]

GO

DECLARE @subTotal money,

@TotalDue money;

SET @subTotal = 1

EXEC [Sales].[udpGetSalesByTerritory] 3, @subTotal OUTPUT, @TotalDue OUTPUT PRINT @subTotal

PRINT @TotalDue

Database designers must carefully choose parameter direction to return only the necessary information. Optional parameters are more flexible in terms of maintainability when dealing with changes in the operation contract and having to maintain support to earlier calling applications.

**Defining the Status Value the Stored Procedure Returns**

Every stored procedure can return an integer value known as the execution status value or return code. By setting this return value, the stored procedure communicates any important result state to the caller.

SQL Server 2005 does not force stored procedures to return an execution status value, nor does it supply a list of possible values to return. When designing a stored procedure, the designer must decide what possible values will be returned as status by the stored procedure as well as their meaning. Outside callers must be aware of this information to understand the return codes.

**BEST PRACTICES Ensuring precision of return code**

Pre-define a closed list of possible return codes that are consistent throughout all the stored procedures in the database to indicate different statuses and to ensure that the same return code means exactly the same in every stored procedure.

**Designing Error Handling Routines**

When an error is encountered inside a stored procedure, SQL Server tries to continue execution gracefully with the next statement in the T-SQL code. Thus, an error will not stop the execution of the stored procedure. However, this does not mean that the error has been handled or resolved.

T-SQL in SQL Server 2005 implements several error handling constructs, including the new TRY and CATCH technique. Explaining the new TRY and CATCH error handling technique is outside the scope of this chapter, but the topic, “Using TRY . . . CATCH in Transact-SQL,” in SQL Server 2005 Books Online at offers a full explanation of this new feature.

When designing a stored procedure, you must always include error handling routines. The error handling must focus on errors that you expect could happen (because of data validation, for example) and errors that you do not expect to happen. (There is always a chance for something exceptional to happen, something you did not foresee or plan.)

The importance of including error handling is less focused on processing logic and more focused on making the application more manageable, reliable, and able to detect problems faster. Design an error handling strategy so that the database is capable of detecting errors, handling the errors so that it can continue execution gracefully, and logging and reporting error information to database administrators for monitoring and problem detection.

**Executing Under the Right Security Context**

The *security execution context* is the identity against which permissions to execute statements or perform actions are checked. Usually, this identity corresponds to the identity used to log on to the database.

In SQL Server 2005, the execution context of a session can be explicitly changed. This is called *impersonation*. When a user executes a stored procedure, the stored procedure can choose to impersonate a different identity to provide the user with more (or fewer) permissions than it currently holds under its own identity. This allows for the interesting scenario of denying users access to the database tables and instead granting only rights to execute stored procedures. Inside each stored procedure, the user is impersonated into a new identity with the proper rights to execute the operation and computations that the stored procedure is supposed to perform.

SQL Server 2005 introduces the EXECUTE AS clause for defining the execution context. Here is a code example of how you can apply the clause:

[Copy](javascript:if%20(window.epx.codeSnippet)window.epx.codeSnippet.copyCode('CodeSnippetContainerCode_ead68536-d4cc-4b57-90d6-3cb1d57f1e54');)

CREATE PROCEDURE

WITH EXECUTE AS ‘user1’

AS SELECT \* FROM

EXECUTE AS can be applied with four different options:

* **EXECUTE AS CALLER** Executes the stored procedure by using the security context of the outside caller. This is the default.
* **EXECUTE AS** *user\_name* Executes the stored procedure by using the security context of the specified user.
* **EXECUTE AS SELF** Executes the stored procedure by using the security context of the user that is creating or modifying the stored procedure.
* **EXECUTE AS OWNER** Executes the stored procedure by using the security context of the user that owns the stored procedure.

For more information about EXECUTE AS, see the “Using EXECUTE AS in Modules” topic in SQL Server 2005 Books Online at .

**Practice: Creating and Modifying a Stored Procedure**

In this practice, which continues with the same scenario and tables from the practice in Lesson 1, you will create a stored procedure that retrieves a result set and then modify the stored procedure to add error handling.

**IMPORTANT Practices build upon each other**

To work successfully with this practice, you need to have finished the practice from Lesson 1.

**Exercise: Create and Modify a Stored Procedure**

In this exercise, you must create a stored procedure to return the product list, so the users can filter and sort the resultset.

1. The view you designed in Lesson 1 has been in production for a while, and now the end user needs the ability to filter and sort the products to conduct the proper analysis on the data more easily. You are given the task of designing a stored procedure that returns the product category name, product subcategory name, product name, revenue (difference between product price and product cost), and the number of days required to manufacture the product. The user wants to be able to filter the results by the category and subcategory as well as by the number of days required to manufacture the product. Design your own solution before reading the suggested answer.
2. Now, your task is to modify the Get\_Products\_Estimated\_Revenue\_By\_Categories stored procedure definition to include TRY . . . CATCH error handling code. In case of an error, log an error into a Log table. Design your own solution before reading the suggested answer that follows.

**Suggested Answers**

1. The following code satisfies the requirements:

USE [AdventureWorks] GO

CREATE PROCEDURE Get\_Products\_Estimated\_Revenue\_By\_Categories (

@ProductCategoryID INT, @ProductSubcategoryID INT, @DaysToManufacture INT

) AS

SELECT Production.ProductCategory.Name AS ProductCategory,

Production.ProductSubcategory.Name AS ProductSubCategory, Production.Product.Name,

Production.Product.ListPrice - Production.Product.StandardCost AS Revenue,

Production.Product.DaysToManufacture

FROM Production.Product INNER JOIN Production.ProductSubcategory ON

Production.Product.ProductSubcategoryID = Production.ProductSubcategory.ProductSubcategoryID

INNER JOIN Production.ProductCategory ON

Production.ProductSubcategory.ProductCategoryID = Production.ProductCategory.ProductCategoryID

WHERE (Production.Product.SellEndDate IS NULL) AND

(Production.Product.DaysToManufacture < @DaysToManufacture) AND

(Production.ProductCategory.ProductCategoryID = @ProductCategoryID) AND (Production.ProductSubcategory.ProductSubcategoryID = @ProductSubcategoryID)

GROUP BY Production.ProductCategory.Name, Production.ProductSubcategory.Name, Production.Product.Name, Production.Product.ReorderPoint,

Production.Product.DaysToManufacture, Production.Product.StandardCost, Production.Product.ListPrice

ORDER BY Revenue DESC, DaysToManufacture DESC, ProductCategory, ProductSubCategory

1. Create a new table called Log that contains three columns: ID, [Error Number], and [Error Description]. Modify the stored procedure’s code to include the SELECT statement inside a TRY . . . CATCH block. When the CATCH block is reached, execute an INSERT statement to insert the required details into the newly created Log table.

**Quick Check**

1. What happens if a parameter that is not defined as an output parameter is called with the OUTPUT modifier?
2. What happens if a parameter that is defined as an output parameter is not called with the OUTPUT modifier?
3. Consider the following scenario. Mark owns the SalesOrderHeaders table in the database. He does not grant SELECT access to anybody. John creates the GetSales-Headers stored procedure that needs to read from the SalesOrderHeaders table. Mary needs to execute the GetSalesHeaders stored procedure. What is the correct setting for the EXECUTE AS clause that lets all the users perform their required tasks?

**Quick Check Answers**

1. If a parameter that is not defined as an output parameter is called with the OUTPUT modifier, the database issues an error message.
2. If a parameter that is defined as an output parameter is not called with the OUTPUT modifier, there is no error message, and the procedure is called. However, the modified value of the parameter is not copied back into the outside caller stack frame.
3. The correct setting is EXECUTE AS OWNER because only Mark has SELECT access on the table. Under this scenario, anybody can execute the stored procedure without requiring specific permissions for the SalesOrderHeaders table.