Creating Your Own Functions

Although SQL Server's built-in functions are powerful and useful, they're not always sufficient to meet your customized needs. Fortunately, SQL Server allows you to create your own user-defined functions using the `CREATE FUNCTION` command.

The structure of the `CREATE FUNCTION` statement is as follows:

```
CREATE FUNCTION <owner>.<function_name> (<parameters>)
RETURNS <type>
AS
BEGIN
          <SQL code>
END
```

The following list describes the user-defined elements in this statement.

- *owner* is the SQL Server account that owns the function. In many cases, this will be the database owner (dbo) account.
- *function_name* is the name you select for your function.
- *parameters* consist of zero, one, or more input parameters that must be supplied when the function is executed. You provide them in the form `@parameter_name datatype`, and you separate multiple parameters with commas.
- *type* is the datatype of the function's output value.
- *SQL code* is the "meat" of the function, where you perform whatever actions are necessary to create the output value. Here are some tips:
    - Separate multiple SQL statements by ending each one with a semicolon.
    - Create working variables (used within the function or returned as output) using the `DECLARE <variable_name> <datatype>` SQL statement.
    - Set variable values using the `SET <variable_name> = <value>` SQL statement.
    - When finished, provide the return value using the `RETURN(<value>)` SQL command.

Suppose you wanted to create a function for use in your business that takes a wholesale price as input and computes the sales price based upon two business rules:

- All wholesale prices are marked up 20 percent to cover the business' operating expenses and profit margin.
- The business is required to collect 6.5 percent sales tax on all purchases.

You could create a function called `GetSalesPrice` to perform this computation for you, as follows:

```
CREATE FUNCTION dbo.GetSalesPrice (@wholesale_price smallmoney)
RETURNS smallmoney
AS
BEGIN
 -- Declare a temporary value to hold our sales price
    DECLARE @salesprice smallmoney;

 -- Add on a 20% markup to the wholesale price
    SET @salesprice = @wholesale_price + @wholesale_price * 0.2;

 -- Add on 6.5% sales tax
    SET @salesprice = @salesprice + @salesprice * .065;

    RETURN (@salesprice);
END;
```

TIP

You should be able to correlate each part of the preceding `CREATE FUNCTION` statement with the general syntax I provide earlier in this section.

After you create the `GetSalesPrice` function, you can call it from within any other SQL statement. Here's a simple example in which I ask SQL Server to tell me the sales price corresponding to a wholesale price of $1:

```
SELECT dbo.GetSalesPrice(1.00) AS 'Sales Price'
```

SQL Server responds with the new value, including the 20 percent markup and 6.5 percent sales tax:

```
Sales Price
---------------------
1.278

(1 row(s) affected)
```

You can also use the function within the context of a more complicated statement. Suppose you wanted to retrieve the wholesale and selling price for each item in your database. You could use this statement:

Comments

Note that in the preceding statement, I do introduce one new item, however: the SQL comment. You use a comment when you want to add some explanatory text so that people can understand the purpose of each SQL statement in the function, but you don't want SQL Server to think that it's part of the function itself.

Comments are a critical part of any type of programming, whether you're using SQL or any other programming language. They allow you to leave notes within your code explaining how it works so that when another person comes across your work, he or she can easily interpret your syntax. In fact, I've found myself grateful that I left comments in my own SQL statements when I've needed to look back at them years later!

You can make an entire line a comment by beginning the line with two dashes (`--`) or comment out multiple lines of the statement by inserting a line with the text `/*` before the first line and ending it with a last line of `*/`. This comment syntax works in any SQL statement, not just function definitions.

```
SELECT item, warehouse, wholesale_price, dbo.
          GetSalesPrice(wholesale_price) AS 'sales price'
FROM stock
```

which provides the output

```
item             warehouse        wholesale_price      sales price
---------------- ---------------- -------------------- ---------------------
Apples           New York         0.12                 0.1534
Apples           Seattle          0.13                 0.1661
Limes            Seattle          0.33                 0.4217
Oranges          New York         0.55                 0.7029
Oranges          Tampa            0.52                 0.6646
Pears            New York         0.39                 0.4984

(6 row(s) affected)
```

Reusing SQL Code with Stored Procedures

SQL Server stored procedures are precompiled bundles of SQL statements that are stored within a SQL Server database. Stored procedures may have zero, one, or more input parameters and may return a scalar value, a table, or nothing at all.

Why use stored procedures? There are two great reasons to include them in your SQL Server repertoire:

- Stored procedures offer the same code reuse benefits provided by functions.
- Stored procedures allow you to enhance the security of your database. You may grant users permission to execute a stored procedure (which in turn inserts, updates, retrieves, or removes data from your tables) without granting them full access to the underlying table.

At this point, you may be asking yourself, "Gee, stored procedures sure sound a lot like functions. What's the difference?" There are actually two significant differences between stored procedures and functions:

- Functions must always return a value to the caller. Stored procedures do not have this requirement. They may simply execute and complete silently.
- You commonly use functions within another expression, whereas you often use stored procedures independently.

As I do with functions earlier in this chapter, in this section I first explain the system stored procedures included with SQL Server 2008 and then cover how you can create your own stored procedures.

Saving time with system stored procedures

SQL Server offers dozens of built-in system stored procedures. Most of these allow you to obtain or modify information about SQL Server or your database. One very helpful system stored procedure is `sp_helptext`, which retrieves the SQL statement associated with a function, stored procedure, trigger, `CHECK` constraint, or database view (among other SQL Server objects). This ability to retrieve a statement is very useful when you want to verify or modify the functionality of one of these objects.

You can execute a system stored procedure (or any stored procedure, for that matter) using the `EXEC` command. If you wanted to use `sp_helptext` to retrieve the text of the `GetSalesPrice` function I describe earlier in the chapter, you would use the following SQL statement:

```
EXEC sp_helptext GetSalesPrice
```

SQL Server then provides the statement used to create the function. SQL Server will include comments and formatting, as shown in the following code:

```
Text
-----------------------------------------------------------------------------
CREATE FUNCTION dbo.GetSalesPrice (@wholesale_price smallmoney)
RETURNS smallmoney
AS
BEGIN
-- Declare a temporary value to hold our sales price
              DECLARE @salesprice smallmoney;

-- Add on a 20% markup to the wholesale price
              SET @salesprice = @wholesale_price + @wholesale_price * 0.2;

-- Add on 6.5% sales tax
              SET @salesprice = @salesprice + @salesprice * .065;

              RETURN (@salesprice);
END;
```

Notice that SQL Server provides the text of the function in `CREATE FUNCTION` format. You could recreate this function by simply cutting and pasting the text into SSMS and executing it. Similarly, you could change the words `CREATE FUNCTION` to `ALTER FUNCTION` and use this SQL statement to modify the function's behavior. (I discuss `ALTER FUNCTION` more at the end of this chapter.)

TIP

You can obtain information about system stored procedures using the same process I describe in the "Obtaining a list of built-in functions" section of this chapter. However, rather than expand the Functions and System Functions folders, you expand the Stored Procedures and System Stored Procedures folders.

Writing your own stored procedures

It's very likely that at some point in your SQL Server career, you'll want to create your own stored procedure. I do this constantly and I encourage you to embrace the reusability and security benefits of stored procedures in your own databases.

Creating your stored procedures

You can create stored procedures using a syntax very similar to that used to create a function. Simply change the `CREATE FUNCTION` statement to `CREATE PROCEDURE`. You don't need to include the `RETURNS` clause if your stored procedure has no output.

Suppose you wanted to write a stored procedure that removes an item from your inventory. Specifically, you want to

- Delete the item from the stock table.
- Send an e-mail to the supervisor alerting him or her of the change.

You can accomplish these tasks with the following stored procedure:

```
CREATE PROCEDURE dbo.RemoveProduct(@item varchar(16), @warehouse varchar(16))
AS
BEGIN
-- Delete the item from the stock table
   DELETE
   FROM stock
   WHERE item = @item AND warehouse = @warehouse;

-- Send an e-mail to the supervisor
   EXEC msdb.dbo.sp_send_dbmail
    @profile_name = 'Inventory Mail',
    @recipients = 'supervisor@foo.com',
    @body = 'Stored procedure RemoveProduct altered the inventory.',
    @subject = 'Inventory Deleted' ;
END
```

Executing your stored procedures

after you create the stored procedure, you execute it using the same syntax used for a system stored procedure, except that you must also include the name of the stored procedure's owner (dbo, in this case):

```
EXEC dbo.RemoveProduct 'Pears', 'New York'
```

My stored procedure doesn't include any return value, so the output is quite simple:

```
(1 row(s) affected)
Mail queued.
```

The "1 row(s) affected" statement is the result of the DELETE SQL statement, and the "Mail queued" statement is the result of sending the message to the supervisor.

Notice that I'm calling a system stored procedure (msdb.dbo.sp_send_dbmail()) from within my own stored procedure. Calling one stored procedure from within another is known as "nesting" stored procedures, and it's perfectly acceptable.

TIP

SQL Server allows you to have up to 32 levels of nesting.

REMEMBER

The send_dbmail() stored procedure uses SQL Server's Database Mail functionality, which I discuss in Chapter 2.

TECHNICAL STUFF

Performing complex database interactions with SQLCLR technology

Functions and stored procedures provide a sophisticated way to hide the complexity of your SQL statements and improve security. However, they're not always the best way to achieve your goal. If you need to perform very complex operations, you can improve their performance by using Microsoft's SQL Common Language Runtime (SQLCLR).

SQLCLR allows programmers to use advanced programming languages to create SQL Server objects, including:

- User-defined functions
- Stored procedures
- Triggers
- User-defined types
- Aggregates

You can use any of the following Microsoft .NET programming languages with SQLCLR:

- Microsoft Visual Basic
- Microsoft Visual C++
- Microsoft Visual C#

Creating SQLCLR objects requires programming skills in one of these languages and is beyond the scope of this book.

Updating Data Automatically with Triggers

Triggers are actions that take place when a series of conditions are met. You see them in everyday life all the time. Consider the dreaded Internal Revenue Service (IRS). It depends on a complex series of triggers to help it in collecting taxes and keeping us honest. Here are some examples:

- When it receives a W-2 from an employer stating your annual wages, it checks to make sure that you reported that income on your 1040 form. If you didn't, the IRS sends you a notice that you must correct your taxes.
- When you claim a dependent on your tax return, the IRS checks the Social Security number of that dependent against other forms in its database to ensure that only one taxpayer claims each dependent. If it detects duplication, it opens an investigation.
- When you file a form with itemized deductions, it compares your deductions to those of similar taxpayers and flags your return for an audit if your deductions seem excessive.

Each of these triggers consists of a condition (“if there is duplication of SSNs”) and an action (“open an investigation”). SQL Server provides similar functionality for database users, allowing you to automatically take specified actions when certain conditions are met.

Creating a trigger

SQL Server triggers consist of four major components:

- Trigger name
- Trigger scope (the database, server, table, or view affected by the trigger)
- Trigger timing (determining whether the trigger should fire after [using the `AFTER` function] or instead of [using the `INSTEAD OF` function] the triggering action)
- Trigger condition (the conditions that cause the trigger to fire)
- Trigger action (the action SQL Server should take when the trigger condition is met)

The basic syntax used for creating a trigger is as follows:

```
CREATE TRIGGER <trigger_name>
ON <scope>
<trigger timing> <trigger condition>
AS
BEGIN
<trigger action>
END
```

Suppose you wanted to create a trigger that automatically notifies the supervisor whenever anyone changes your stock table. You would want this trigger to fire whenever an `INSERT` or `UPDATE` statement occurs on the table. Here’s the way to make that happen:

```
CREATE TRIGGER inventory_minimum
ON stock
AFTER INSERT, UPDATE
AS
BEGIN
   EXEC msdb.dbo.sp_send_dbmail
    @profile_name = 'Inventory Mail',
    @recipients = 'supervisor@foo.com',
    @body = 'Someone changed the inventory.',
    @subject = 'Change Notification' ;
END
```

After you create the trigger, SQL Server automatically monitors the database every time an `INSERT` or `UPDATE` statement modifies the stock table.

Disabling a trigger

You may want to temporarily disable a trigger in certain circumstances using the `DISABLE TRIGGER` statement. For example, if you plan to make numerous changes to your inventory and don't want to clutter your e-mail with notifications from the `inventory_minimum` trigger, you can disable it with the following statement:

```
DISABLE TRIGGER inventory_minimum
ON stock
```

Re-enabling the trigger uses a similar statement:

```
ENABLE TRIGGER inventory_minimumON stock
```

Modifying and Deleting Functions, Stored Procedures, and Triggers

Throughout this chapter, I show you how to create programmable SQL Server objects: functions, stored procedures, and triggers. It's also sometimes necessary to change or remove those objects after you create them. The syntax for doing this is very similar for all three types of programmable objects.

Modifying objects

If you want to modify a function, stored procedure, or trigger, simply write a `CREATE` statement that contains the modified SQL and change the keyword `CREATE` to `ALTER`.

For example, to modify the `GetSalesPrice` stored procedure to charge a higher markup of 25 percent, use this SQL statement:

```
ALTER FUNCTION dbo.GetSalesPrice (@wholesale_price smallmoney)
RETURNS smallmoney
AS
BEGIN
 -- Declare a temporary value to hold our sales price
    DECLARE @salesprice smallmoney;
 -- Add on a 20% markup to the wholesale price
    SET @salesprice = @wholesale_price + @wholesale_price * 0.25;

 -- Add on 6.5% sales tax
    SET @salesprice = @salesprice + @salesprice * .065;

    RETURN (@salesprice);
END;
```

The `sp_helptext` command described earlier in this chapter comes in quite handy when you need to modify a function, stored procedure, or trigger. You can use `sp_helptext` to retrieve the `CREATE` command used to create the object and simply change the keyword `CREATE` to `ALTER`, modify the logic, and execute the statement to update your database.

Deleting objects

Deleting programmable objects is simple. Use one of the following DROP commands:

```
DROP FUNCTION <function_name>
```

```
DROP PROCEDURE <procedure_name>
```

```
DROP TRIGGER <trigger_name>
```

For example, you could delete the GetSalesPrice function using the following SQL statement:

```
DROP FUNCTION GetSalesPrice;
```