**Return Values: Getting Something Back from  
Your SQL Server Stored Procedures**



*by Tom O’Neill*

You give so much to your database . . . sometimes you just want something back. What do you give your database?  Plenty. You give it new records (INSERT). You give it changes to existing records (UPDATE/DELETE). Every now and then, you just look around (SELECT). When you do these things, it is often essential that the database send back a code or value for you to pass to your front-end application. Luckily, we can do this with stored procedures in SQL Server.

Why would you want something sent back? There are plenty of reasons. Returning a value from your database action (SELECT, INSERT, etc.) can help maintain session, drive display elements (such as insert confirmations), and can even help trigger a redirect to a new web page. Essentially, something as simple as a small code—returned from the query—can drive a series of actions in an application, and provide context to a variety of user experiences.

“Return values” help get the job done. This feature provides the foundation for all the tasks described above. There are limits to using the “RETURN” function; they can’t do everything. In fact, there are a number of other ways to return data—such as “OUTPUT” and recordsets. Return values, though, are a fast, efficient way to drive information flow and context in any application that is tied to a database. A return value is a single data element that is the result of a query. Essentially, you can run a query that outputs a single value, such as:

SELECT id FROM USRS

This query will return a single value, the contents of “id”. We can use return values to store the contents of “id” in a variable, and output it to the user or application (as appropriate). The user can define what to return. This is a powerful feature that can be leveraged in a number of situations that are database-driven, including:

* Application login—Was the login successful? Username only?
* Page redirection based on database activity—Did you try to update a record that doesn’t exist?
* Alerts and notifications.

There is definitely a lot you can do with return values. Let’s take a closer look.

**Rules of the Road**

In order to get this exercise off the ground, there are a few housekeeping items for us to get out of the way. In this section, I’ll do my job for you. First, I’ll give you the DDL for the database I’ll use in this article. Check for it at the end of this article (I didn’t want to clutter everything). You will notice that it is for SQL Server. I focus on SQL Server, and I don’t apologize for it. I am focused (or lazy, as the case may be). Also, this site is focused on SQL Server performance and development. For those of you who are not “SQL Server Samurais”, don’t fret. These principles can be applied to any other database management system. If you have any thoughts or feedback, please e-mail me at [tomoneill@deloitte.com](mailto:tomoneill@deloitte.com).

**Using Return Values**

I assume that you have set up a database by now. In fact, it doesn’t even need to be the database I used to write this article (though I went to a lot of trouble). The principles you will learn in this lesson will apply to any database you decide to build and deploy. The next step is to build a simple stored procedure. It doesn’t have to be complex. We just need to build a framework to use in our exploration of stored procedures.

A basic stored procedure for the sample database will run a simple select statement, similar to what you would see when logging into an application. When you log into an application, you generally present to data elements:  a username and a password. When you log into an application, though, these data elements are generally passed to a database. To accept these data elements, you can build a stored procedure. This stored procedure (for now) will run a select to see if the user exists, and if the username and password combination is acceptable. The stored procedure will look like this (though there are a variety of ways to accomplish the same task):

CREATE PROCEDURE usp\_checkuser

            @login varchar(50),

            @pswd varchar(50)

AS

SELECT login, pswd FROM USRS

WHERE login=@login

AND pswd=@pswd

GO

For now, we are sending a username and password, and using them in a select statement against the USRS table.

The next step is to build an execution file. While you can build your execute statement every time you want to run the stored procedure, I recommend that you build it ahead of time. For a small stored procedure like this, it doesn’t add much value. But, if you get into the habit now, you’ll thank me later.

exec usp\_checkuser ‘’, ‘’

When you run this statement in the query analyzer, it will execute the stored procedure. This is the rest of our “homework.” Our next steps will be to alter the stored procedure. We’re getting ready to make return values happen!

Let’s dig into this. A quick review of what a “return value” is: it is a single data element that a stored procedure kicks back after it has been executed. It is a single data element. You can’t return a full database record. Also, you can’t return multiple rows of a single data element. One record, one element. I’m not going to negotiate this point with you!

Let’s take another look at our existing stored procedure.

CREATE PROCEDURE usp\_checkuser

            @login varchar(50),

            @pswd varchar(50)

AS

SELECT login, pswd FROM USRS

WHERE login=@login

AND pswd=@pswd

GO

The top part is not going to change much. We need to create a procedure, name it, and pass our two login data elements to the stored procedure. We will have to add one more variable, though. We are going to use this variable to store the value that the stored procedure will return (i.e. the “return value”).

CREATE PROCEDURE usp\_checkuser

            @login varchar(50),

            @pswd varchar(50),

            @usr\_id int

AS

Next, we need to change the SELECT statement. We are only going to select one data element from the USRS table. We are going to put this single value into a variable. Also, we still need the WHERE clause. Remember, we only want to return one data element. We will use the WHERE clause to ensure that the query only returns one value. Before we even work with the SELECT statement, though, we need to set the value of the @usr\_id variable.

SET @usr\_id =

We are going to store the results of the query in this variable. So, everything the SELECT statement does will have to go inside the variable itself. How do we do this?  A subquery!

SET @usr\_id = ()

The SELECT statement will go between the parentheses. The SELECT statement itself will not change much. We still want to query the USRS table, and we still want to use the same “filter” (i.e. the same WHERE clause). But, we do want to return a single value. Consequently, we will return only a single value, id.

SELECT id FROM USRS

            WHERE login=@login

            AND pswd=@pswd

We are still looking for the unique identifier that corresponds to the unique combination of username and password. That combination will yield a single “id” (the primary key of the USRS table). Now, we want to put this query between the parentheses.

SET @usr\_id = (SELECT id FROM USRS

            WHERE login=@login

            AND pswd=@pswd)

We are soooo close! The only thing we have left to do is actually return the “return value.”  To do this, use the RETURN function. This function is pretty easy to use. Simply type RETURN, followed by the variable that contains the return value.

RETURN @usr\_id

With the new return function, the total procedure will look like this:

CREATE PROCEDURE usp\_checkuser

            @login varchar(50),

            @pswd varchar(50),

            @usr\_id int

AS

SET @usr\_id = (SELECT id FROM USRS

            WHERE login=@login

            AND pswd=@pswd)

RETURN @usr\_id

GO

Executing this stored procedure will return the “id” value, based on the unique username and password combination. You will not “see” the return value. It is there, though. Programmatically, you can access this return value (using ASP, Java, Visual Basic, or whatever other programming language you use).

**Conclusion**

In this article, we have reviewed why return values are useful, and how to include them in your stored procedures. Don’t blink, you might miss it. Seriously. Return values are very easy to use, and are quite powerful. There are two basic elements to return values:  (1) putting your SELECT query results in a variable and (2) using the RETURN function to return your variable contents. Remember, use return values to have the stored procedure kick back a single value, not an entire record. If you need more output from your stored procedure than just a single value, the OUTPUT function would be more appropriate.

**Appendix:  Sample Database DDL**

IF EXISTS (SELECT name FROM master.dbo.sysdatabases WHERE name = N'ArticleTestDB')

            DROP DATABASE [ArticleTestDB]

GO

CREATE DATABASE [ArticleTestDB]  ON (NAME = N'ArticleTestDB\_Data', FILENAME = N'C:\Program Files\Microsoft SQL Server\MSSQL\data\ArticleTestDB\_Data.MDF' , SIZE = 1, FILEGROWTH = 10%) LOGON (NAME = N'ArticleTestDB\_Log', FILENAME = N'C:\Program Files\Microsoft SQL Server\MSSQL\data\ArticleTestDB\_Log.LDF' , SIZE = 1, FILEGROWTH = 10%)

 COLLATE SQL\_Latin1\_General\_CP1\_CI\_AS

GO

exec sp\_dboption N'ArticleTestDB', N'autoclose', N'false'

GO

exec sp\_dboption N'ArticleTestDB', N'bulkcopy', N'false'

GO

exec sp\_dboption N'ArticleTestDB', N'trunc. log', N'false'

GO

exec sp\_dboption N'ArticleTestDB', N'torn page detection', N'true'

GO

exec sp\_dboption N'ArticleTestDB', N'read only', N'false'

GO

exec sp\_dboption N'ArticleTestDB', N'dbo use', N'false'

GO

exec sp\_dboption N'ArticleTestDB', N'single', N'false'

GO

exec sp\_dboption N'ArticleTestDB', N'autoshrink', N'false'

GO

exec sp\_dboption N'ArticleTestDB', N'ANSI null default', N'false'

GO

exec sp\_dboption N'ArticleTestDB', N'recursive triggers', N'false'

GO

exec sp\_dboption N'ArticleTestDB', N'ANSI nulls', N'false'

GO

exec sp\_dboption N'ArticleTestDB', N'concat null yields null', N'false'

GO

exec sp\_dboption N'ArticleTestDB', N'cursor close on commit', N'false'

GO

exec sp\_dboption N'ArticleTestDB', N'default to local cursor', N'false'

GO

exec sp\_dboption N'ArticleTestDB', N'quoted identifier', N'false'

GO

exec sp\_dboption N'ArticleTestDB', N'ANSI warnings', N'false'

GO

exec sp\_dboption N'ArticleTestDB', N'auto create statistics', N'true'

GO

exec sp\_dboption N'ArticleTestDB', N'auto update statistics', N'true'

GO

if( ( (@@microsoftversion / power(2, 24) = 8) and (@@microsoftversion & 0xffff >= 724) ) or ( (@@microsoftversion / power(2, 24) = 7) and (@@microsoftversion & 0xffff >= 1082) ) )

            exec sp\_dboption N'ArticleTestDB', N'db chaining', N'false'

GO

use [ArticleTestDB]

GO