Writing a Stored Procedure

By Nathan Pond

http://www.4guysfromrolla.com/webtech/111499-1.shtml

If you're anything like me, you don't easily pick up on development techniques just by hearing about them. When I first installed my MS SQL server on my computer, it opened up a whole new world of features that I had never used. Among these were Stored Procedures. This article is designed to tell you how to begin writing stored procedures. I am using Microsoft SQL Server 7.0, but these examples should work in any SQL version.

Writing Stored Procedures doesn't have to be hard. When I first dove into the technology, I went to every newsgroup, web board, and IRC channel that I knew looking for answers. Through all the complicated examples I found in web tutorials, it was a week before I finally got a working stored procedure. I'll stop rambling now and show you what I mean:

Normally, you would call a database with a query like:

```
Select column1, column2 From Table1
```

To make this into a stored procedure, you simple execute this code:

```
CREATE PROCEDURE sp_myStoredProcedure
AS
Select column1, column2 From Table1
Go
```

That's it, now all you have to do to get the recordset returned to you is execute the stored procedure. You can simply call it by name like this:

```
sp_myStoredProcedure
```

Note: You can name a stored procedure anything you want, provided that a stored procedure with that name doesn't already exist. Names do not nee to be prefixed with sp_ but that is something I choose to do just as a naming convention. It is also somewhat a standard in the business world to use it, but SQL server does not require it.

Now, I realize you aren't gaining much in this example. I tried to make it simple to make it easy to understand. In part II of this article, we'll look at how it can be useful, for now let's look at how you can call a Stored Procedure with parameters.

Let's say that we want to expand on our previous query and add a WHERE clause. So we would have:

```
Select column1, column2 From Table1
Where column1 = 0
```

Well, I know we could hard code the `Where column1 = 0` into the previous stored procedure. But wouldn't it be neat if the number that 0 represents could be passed in as an input parameter? That way it wouldn't have to be 0, it could be 1, 2, 3, 4, etc. and you wouldn't have to change the stored procedure. Let's start out by deleting the stored procedure we already created. Don't worry, we'll recreate it with the added feature of an input parameter. There isn't a way that I'm aware of to simply over-write a stored procedure. You must drop the current one and re-create it with the changes. We will drop it like this:

```
DROP PROCEDURE sp_myStoredProcedure
```

Now we can recreate it with the input parameter built in:

```
CREATE PROCEDURE sp_myStoredProcedure
```

```
   @myInput  int
AS
Select column1, column2 From Table1
Where column1 = @myInput
Go
```

Ok, why don't we pause here and I'll explain in more detail what is going on. First off, the parameter: you can have as many parameters as you want, or none at all. Parameters are set when the stored procedure is called, and the stored procedure receives it into a variable. @myInput is a variable. All variables in a stored procedure have a @ symbol preceding it. A name preceded with @@ are global variables. Other than that, you can name a variable anything you want. When you declare a variable, you must specify its datatype. In this case the datatype is of type Int (Integer). Now, before I forget, here's how to call the stored procedure with a parameter:

```
sp_myStoredProcedure 0
```

If you want more than one parameter, you separate them with commas in both the stored procedure and the procedure call. Like so:

```
CREATE PROCEDURE sp_myStoredProcedure
   @myInput  int,
   @myString varchar(100),
   @myFloat
AS
.....
Go
```

And you would call it like this:

```
sp_myStoredProcedure 0, 'This is my string', 3.45
```

Note: The varchar datatype is used to hold strings. You must specify the length of the string when you declare it. In this case, the variable is assigned to allow for 100 characters to be held in it.

Now, I'm sure some of you are wondering if there is a difference for SQL calls coming from ASP. There really isn't, let's take our first stored procedure example and I'll show how it is called from ASP. If it wasn't a stored procedure, you would call it something like this:

```
<%

        dim dataConn, sSql, rs
        set dataConn = Server.CreateObject("ADODB.Connection")
        dataConn.Open  "DSN=webData;uid=user;pwd=password" 'make connection
        sSql = "Select column1, column2 From Table1"
        Set rs = dataConn.Execute(sSql) 'execute sql call
%>
```

Now let's see how we call the stored procedure.

```
<%
        dim dataConn, sSql, rs
        set dataConn = Server.CreateObject("ADODB.Connection")
        dataConn.Open  "DSN=webData;uid=user;pwd=password" 'make connection
        sSql = "sp_myStoredProcedure"
        Set rs = dataConn.Execute(sSql) 'execute sql call
%>
```

As you can see, the only difference is the query that is to be executed, which is stored in the sSql command. Instead of being the actual query, it is simply the name of the stored procedure. Now let's take a quick look at how you would call it with parameters. In our second example, we created the stored procedure to accept one integer parameter. Here's the code:

```
<%
        dim dataConn, sSql, rs, myInt
        myInt = 1 'set myInt to the number we want to pass to the stored procedure
        set dataConn = Server.CreateObject("ADODB.Connection")
        dataConn.Open  "DSN=webData;uid=user;pwd=password" 'make connection
        sSql = "sp_myStoredProcedure " & myInt
        Set rs = dataConn.Execute(sSql) 'execute sql call
%>
```

Well, that's all for this article. Sometime in the near future I plan on writing a second part that really dives into more specifics about stored procedures. I hope this is enough to get you started though. Feel free to e-mail me with any questions or comments about the article!

Let me start out by first correcting (or rather updating) something I said in my first article. I said there that I wasn't aware of a way to update a stored procedure without deleting it and recreating it. Well now I am. :-) There is an ALTER command you can use, like this:

```
ALTER PROCEDURE sp_myStoredProcedure
AS
......
Go
```

This will overwrite the stored procedure that was there with the new set of commands, but will keep permissions, so it is better than dropping and recreating the procedure. Many thanks to Pedro Vera-Perez for e-mailing me with this info.

As promised I am going to dive into more detail about stored procedures. Let me start out by answering a common question I received via e-mail. Many people wrote asking if it was possible, and if so how to do it, to use stored procedures do to more than select statements. Absolutely!!! Anything that you can accomplish in a sql statement can be accomplished in a stored procedure, simply because a stored procedure can execute sql statements. Let's look at a simple INSERT example.

```
CREATE PROCEDURE sp_myInsert
  @FirstName  varchar(20),
  @LastName   varchar(30)
As
INSERT INTO Names(FirstName, LastName)
values(@FirstName, @LastName)
Go
```

Now, call this procedure with the parameters and it will insert a new row into the Names table with the FirstName and LastName columns approximately assigned. And here is an example of how to call this procedure with parameters from an ASP page:

```
<%
dim dataConn, sSql
dim FirstName, LastName

FirstName = "Nathan"
LastName = "Pond"

set dataConn = Server.CreateObject("ADODB.Connection")
dataConn.Open  "DSN=webData;uid=user;pwd=password" 'make connection
```

```
sSql = "sp_myInsert '" & FirstName & "', '" & LastName & "'"

dataConn.Execute(sSql) 'execute sql call
%>
```

Remember, you can use stored procedures for anything, including UPDATE and DELETE calls. Just embed a sql statement into the procedure. Notice that the above procedure doesn't return anything, so you don't need to set a recordset. The same will be true for UPDATE and DELETE calls. The only statement that returns a recordset is the SELECT statement.

Now, just because a recordset isn't returned, it doesn't mean that there won't be a return value. Stored procedures have the ability to return single values, not just recordsets. Let me show you a practical example of this. Suppose you have a login on your site, the user enters a username and password, and you need to look these up in the database, if they match, then you allow the user to logon, otherwise you redirect them to an incorrect logon page. Without a stored procedure you would do something like this:

```
<%
dim dataConn, sSql, rs

set dataConn = Server.CreateObject("ADODB.Connection")
dataConn.Open  "DSN=webData;uid=user;pwd=password" 'make connection

sSql = "Select * From User_Table Where UserName = '" & _
       Request.Form("UserName") & "' And Password = '" & _
       Request.Form("Password") & "'"

Set rs = dataConn.Execute(sSql) 'execute sql call

If rs.EOF Then
       'Redirect user, incorrect login
       Response.Redirect "Incorrect.htm"
End If

'process logon code
.............
%>
```

Now let's look at how we would accomplish this same task using a stored procedure. First let's write the procedure.

```
CREATE PROCEDURE sp_IsValidLogon
  @UserName  varchar(16),
  @Password  varchar(16)
As
if exists(Select * From User_Table
          Where UserName = @UserName
                        And
                Password = @Password)
  return(1)
else
  return(0)

Go
```

What this procedure does is take the username and password as input parameters and performs the lookup. If a record is returned the stored procedure will return a single value of 1, if not the procedure will return 0. No recordset is returned. Let's look at the asp you would use:

```
<%
```

```
<!--#INCLUDE VIRTUAL="/include/adovbs.inc"-->

dim dataConn, adocmd, IsValid

set dataConn = Server.CreateObject("ADODB.Connection")
dataConn.Open  "DSN=webData;uid=user;pwd=password" 'make connection

Set adocmd = Server.CreateObject("ADODB.Command")
adocmd.CommandText = "sp_IsValidLogon"

adocmd.ActiveConnection = dataConn
adocmd.CommandType = adCmdStoredProc
adocmd.Parameters.Append adocmd.CreateParameter("return", _
                            adInteger, adParamReturnValue, 4)
adocmd.Parameters.Append adocmd.CreateParameter("username", _
                            adVarChar, adParamInput, 16, _
                            Request.Form("UserName"))
adocmd.Parameters.Append adocmd.CreateParameter("password", _
                            adVarChar, adParamInput, 16, _
                            Request.Form("Password"))

adocmd.Execute

IsValid = adocmd.Parameters("return").Value

If IsValid = 0 Then
        'Redirect user, incorrect login
        Response.Redirect "Incorrect.htm"
End If

'process logon code
.............
%>
```

In Part 2 we'll look at the ADO Command Object, and how you can use it to execute stored procedures through your ASP pages. We'll also look at *why* you should use stored procedures as opposed to dynamic queries.

In Part 1, I introduced a lot of new things, so lets slow down for a minute and I'll go through them. First thing I did was create a command object for ADO. I did this with:

```
Set adocmd = Server.CreateObject("ADODB.Command")
```

Next I had to tell the object what command it would be executing, with this line:

```
adocmd.CommandText = "sp_IsValidLogon"
```

Notice that the command is the name of the stored procedure. You must tell the command object which connection (database) to use, to do this you use `.ActiveConnection`. `.CommandType` is a property that tells sql what type of command it is trying to execute. `adCmdStoredProc` is a constant variable declared in the include file `adovbs.inc`. (For more information on `adovbs.inc`, be sure to read ADOVBS.INC - Use It!) It represents the number telling sql that the command is to execute a stored procedure. The `.Append` method is used to add return values and parameters. I had to add the username and password parameters, as well as set up the return value. I then executed the command with `.Execute`, and `.Parameters("return").Value` held the return value from the procedure. I set that to the variable `IsValid`. If `IsValid` is 0, the login is incorrect, if it is 1, the login is correct.

Now even after the explanation this is still a lot to take in. My recommendation to you is to dive into your server and try a few simple tasks like this. Practice makes perfect. One note: sometimes I get errors when I try to `.Append` the return value after I have already set the parameters. Meaning I might get an error if the above code looked like this:

```
<%
.....
Set adocmd = Server.CreateObject("ADODB.Command")
adocmd.CommandText = "sp_IsValidLogon"

adocmd.ActiveConnection = dataConn
adocmd.CommandType = adCmdStoredProc
adocmd.Parameters.Append adocmd.CreateParameter("username", _
            adVarChar, adParamInput, 16, Request.Form("UserName"))
adocmd.Parameters.Append .CreateParameter("password", _
            adVarChar, adParamInput, 16, Request.Form("Password"))
adocmd.Parameters.Append .CreateParameter("return", _
            adInteger, adParamReturnValue, 4)
adocmd.Execute

IsValid = adocmd.Parameters("return").Value
.....
%>
```

I'm not exactly sure why this happens, but I just made it a habit to declare the return value first, then the parameters.

Now I know what some of you are saying. "The original ASP example for checking the username and password without using a stored procedure is so much easier, all you did was confuse me! Can Stored Procedures actually be used to improve efficiency?" Well I'm glad you asked, because although the example above did require a bit more code, it is important to realize that it is much more efficient. Stored procedures have other benefits besides efficiency, though. For a full explanation of the benefits of stored procedures, be sure to read the SQL Guru's Advice on the issue. And now I am going to show you an example of a task where using stored procedures minimizes your database calls.

Assume you have the same script as before for validating usernames and passwords. All it really does is say whether it is a valid username and password. Suppose you want to add functionality in to log all failed attempts at logging on into another table called `FailedLogons`. If you weren't using a stored procedure you would have to make another call to the database from your ASP code. However, in the example using the stored procedure, we don't have to touch the ASP code at all, we simply modify the procedure like so:

```
ALTER PROCEDURE sp_IsValidLogon
  @UserName  varchar(16),
  @Password  varchar(16)
As
if exists(Select * From User_Table
          Where UserName = @UserName
                        And
                Password = @Password)
  begin
    return(1)
  end
else
  begin
    INSERT INTO FailedLogons(UserName, Password)
        values(@UserName, @Password)

    return(0)
  end

Go
```

Wasn't that neat? But that's not all, while we're at it why not add a little more functionality? Let's say that we want to run a check on each incorrect login, and if there have been more than 5 incorrect logins for that username within the past day, that account will be disabled. We would have to have the `FailedLogons` table set up to have a `dtFailed`

column with a default value of (GetDate()). So when the incorrect logon is inserted into the table, the date and time is recorded automatically. Then we would modify our stored procedure like this:

```
ALTER PROCEDURE sp_IsValidLogon
  @UserName  varchar(16),
  @Password  varchar(16)
As
if exists(Select * From User_Table
          Where UserName = @UserName
                        And
                Password = @Password)
  begin
    return(1)
  end
else
  begin
    INSERT INTO FailedLogons(UserName, Password)
        values(@UserName, @Password)

    declare @totalFails  int
    Select @totalFails = Count(*) From FailedLogons
        Where UserName = @UserName
        And dtFailed > GetDate()-1

    if (@totalFails > 5)
      UPDATE User_Table Set Active = 0
      Where UserName = @UserName

    return(0)
  end

Go
```

Now, let's take a closer look at what I was doing. First thing, check to see if the username and password exist on the same row, if they do, login is fine, return 1 to the user and exit the procedure. If the login is not ok though, we want to log it. The first thing the procedure does is insert the record into the FailedLogons table. Next we declare a variable to hold the number of failed logons for that same day. Next we assign that value by using a sql statement to retrieve the number of records for that username, within the same day. If that number is greater than 5, it's likely someone is trying to hack that account so the username will be disabled by setting the active flag in the User_Table to 0. Finally, return 0 letting the calling code (ASP) know that the login was unsuccessful. To accomplish this same task using only ASP, you would have needed to make 4 database calls. The way we just did it is still only one database call, plus the fact that all that functionality we added at the end was in the stored procedure, we didn't have to touch the ASP code at all!

Note about begin/end: When using an If statement in a stored procedure, as long as you keep the conditional code to one line you won't need a begin or end statement. Example:

```
if (@myvar=1)
  return(1)
else
  return(2)
```

However, if you need more than one line, it is required that you use begin and end. Example:

```
if (@myvar=1)
  begin
    do this.....
    and this.....
    return(1)
```

```
  end
else
  begin
    do this....
    return(2)
  end
```

I hope that I have given enough information to keep you active in learning stored procedures. If you're anything like me, getting the basics is the hard part, from there you can experiment and learn on your own. That is why I decided to create these two articles. Remember, feel free to e-mail me at npond@bgnet.bgsu.edu with any questions or comments about either of my articles. And thanks to everyone who wrote to me regarding part one of this series.

Stored Procedures

by: stanleytan

http://www.csharpfriends.com/Articles/getArticle.aspx?articleID=78

Why Use Stored Procedures?

There are several advantages of using stored procedures instead of standard SQL. First, stored procedures allow a lot more flexibility offering capabilities such as conditional logic. Second, because stored procedures are stored within the DBMS, bandwidth and execution time are reduced. This is because a single stored procedure can execute a complex set of SQL statements. Third, SQL Server pre-compiles stored procedures such that they execute optimally. Fourth, client developers are abstracted from complex designs. They would simply need to know the stored procedure's name and the type of data it returns.

Creating a Stored Procedure

Enterprise Manager provides an easy way to create stored procedures. First, select the database to create the stored procedure on. Expand the database node, right-click on "Stored Procedures" and select "New Stored Procedure...". You should see the following:

```
CREATE PROCEDURE [OWNER].[PROCEDURE NAME] AS
```

Substitute OWNER with "dbo" (database owner) and PROCEDURE NAME with the name of the procedure. For example:

```
CREATE PROCEDURE [dbo].[GetProducts] AS
```

So far, we are telling SQL Server to create a new stored procedure with the name GetProducts. We specify the body of the procedure after the AS clause:

```
CREATE PROCEDURE [dbo].[GetProducts] AS
SELECT ProductID, ProductName FROM Products
```

Click on the Check Syntax button in order to confirm that the stored procedure is syntactically correct. Please note that the GetProducts example above will work on the Northwind sample database that comes with SQL Server. Modify it as necessary to suite the database you are using.

Now that we have created a stored procedure, we will examine how to call it from within a C# application.

Calling a Stored Procedure

A very nice aspect of ADO.NET is that it allows the developer to call a stored procedure in almost the exact same way as a standard SQL statement.

1. Create a new C# Windows Application project.

2. From the Toolbox, drag and drop a DataGrid onto the Form. Resize it as necessary.

3. Double-click on the Form to generate the Form_Load event handler. Before entering any code, add "using System.Data.SqlClient" at the top of the file.

Enter the following code:

```
private void Form1_Load(object sender, System.EventArgs e)
{
    SqlConnection conn = new SqlConnection("Data
Source=localhost;Database=Northwind;Integrated Security=SSPI");
    SqlCommand command = new SqlCommand("GetProducts", conn);
    SqlDataAdapter adapter = new SqlDataAdapter(command);
    DataSet ds = new DataSet();
    adapter.Fill(ds, "Products");
    this.dataGrid1.DataSource = ds;
    this.dataGrid1.DataMember = "Products";
}
```

As you can see, calling a stored procedure in this example is exactly like how you would use SQL statements, only that instead of specifying the SQL statement, you specify the name of the stored procedure. Aside from that, you can treat it exactly the same as you would an ordinary SQL statement call with all the advantages of a stored procedure.

Specifying Parameters

Most of the time, especially when using non-queries, values must be supplied to the stored procedure at runtime. For instance, a @CategoryID parameter can be added to our GetProducts procedure in order to specify to retrieve only products of a certain category. In SQL Server, parameters are specified after the procedure name and before the AS clause.

```
CREATE PROCEDURE [dbo].[GetProducts] (@CategoryID int) AS
SELECT ProductID, ProductName FROM Products WHERE CategoryID = @CategoryID
```

Parameters are enclosed within parenthesis with the parameter name first followed by the data type. If more than one parameter is accepted, they are separated by commas:

```
CREATE PROCEDURE [dbo].[SomeProcedure] (
    @Param1 int,
    @Param2 varchar(50),
    @Param3 varchar(50)
) AS
...
```

For our GetProducts example, if @CategoryID was supplied with the value 1, the query would equate to:

```
SELECT ProductID, ProductName FROM Products WHERE CategoryID = 1
```

Which would select all the products that belong to CategoryID 1 or the Beverages category. To call the stored procedure, use Query Analyzer to execute:

```
exec GetProducts X
```

where X is the @CategoryID parameter passed to the stored procedure. To call the stored procedure from within a C# application using 1 as the @CategoryID parameter value, use the following code:

```
SqlConnection conn = new SqlConnection("Data
Source=localhost;Database=Northwind;Integrated Security=SSPI");
SqlCommand command = new SqlCommand("GetProducts", conn);
```

```
command.CommandType = CommandType.StoredProcedure;
command.Parameters.Add("@CategoryID", SqlDbType.Int).Value = 1;
SqlDataAdapter adapter = new SqlDataAdapter(command);
DataSet ds = new DataSet();
adapter.Fill(ds, "Products");
this.dataGrid1.DataSource = ds;
this.dataGrid1.DataMember = "Products";
```

Note that you must now specify the CommandType property of the SqlCommand object. The reason we did not do this in the first example was that it is not required if the stored procedure does not accept parameters. Of course, specifying the CommandType property even if it is not needed may improve readability. The next line actually combines two lines in one:

```
command.Parameters.Add("@CategoryID", SqlDbType.Int);
command.Parameters["@CategoryID"].Value = 1;
```

The first line of this segment specifies that the command object (which calls the GetProducts stored procedure) accepts a parameter named @CategoryID which is of type SqlDbType.Int. The type must be the same as the data type specified by the stored procedure. The second line of this code segment gives the parameter the value 1. For simplicity, especially when using more than one parameter, I prefer to combine to two lines into a single line:

```
command.Parameters.Add("@CategoryID", SqlDbType.Int).Value = 1;
```

The rest of the code is the same as in the previous example without parameters. As illustrated in the previous examples, ADO.NET takes a lot of pain out of database programming. Calling a stored procedure uses virtually the same code as using standard SQL and specifying parameters is a painless process.

Data Retrieval

Data Retrieval with stored procedures is the same (surprise!) as if using standard SQL. You can wrap a DataAdapter around the Command object or you can use a DataReader to fetch the data one row at a time. The previous examples have already illustrated how to use a DataAdapter and fill a DataSet. The following example shows usage of the DataReader:

```
SqlConnection conn = new SqlConnection("Data
Source=localhost;Database=Northwind;Integrated Security=SSPI");
SqlCommand command = new SqlCommand("GetProducts", conn);
command.CommandType = CommandType.StoredProcedure;
command.Parameters.Add("@CategoryID", SqlDbType.Int).Value = 1;
conn.Open();
SqlDataReader reader = command.ExecuteReader();
while (reader.Read())
{
    Console.WriteLine(reader["ProductName"]);
}
conn.Close();
```

Again, using either a DataAdapter or a DataReader against a query from a stored procedure is the same as specifying the SQL from within the code.

Inserting Data Using Parameters

Using other SQL statements such as INSERT, UPDATE or DELETE follow the same procedure. First, create a stored procedure that may or may not accept parameters, and then call the stored procedure from within the code supply the necessary values if parameters are needed. The following example illustrates how to insert a new user in a users table that has a username and password field.

```
CREATE PROCEDURE [dbo].[InsertUser] (
```

```
    @Username varchar(50),
    @Password varchar(50)
) AS
INSERT INTO Users VALUES(@Username, @Password)

string username = ... // get username from user
string password = ... // get password from user
SqlConnection conn = new SqlConnection("Data
Source=localhost;Database=MyDB;Integrated Security=SSPI");
SqlCommand command = new SqlCommand("InsertUser", conn);
command.CommandType = CommandType.StoredProcedure;
command.Parameters.Add("@Username", SqlDbType.VarChar).Value = username;
command.Parameters.Add("@Password", SqlDbType.VarChar).Value = password;
conn.Open();
int rows = command.ExecuteNonQuery();
conn.Close();
```

First, we retrieve the username and password information from the user. This information may be entered onto a form, through a message dialog or through some other method. The point is, the user specifies the username and password and the application inserts the data into the database. Also notice that we called the ExecuteNonQuery() method of the Connection object. We call this method to indicate that the stored procedure does not return results for a query but rather an integer indicating how many rows were affected by the executed statement. ExecuteNonQuery() is used for DML statements such as INSERT, UPDATE and DELETE. Note that we can test the value of rows to check if the stored procedure inserted the data successfully.

```
if (rows == 1)
{
    MessageBox.Show("Create new user SUCCESS!");
}
else
{
    MessageBox.Show("Create new user FAILED!");
}
```

We check the value of rows to see if it is equal to one. Since our stored procedure only did one insert operation and if it is successful, the ExecuteNonQuery() method should return 1 to indicate the one row that was inserted. For other SQL statements, especially UPDATE and DELETE statements that affect more than one row, the stored procedure will return the number of rows affected by the statement.

```
DELETE FROM Products WHERE ProductID > 50
```

This will delete all products whose product ID is greater than 50 and will return the number of rows deleted.

Conclusion

Stored procedures offer developers a lot of flexibility with many features not available using standard SQL. ADO.NET allows us to use stored procedures in our applications seamlessly. The combination of these two allows us to create very powerful applications rapidly.