VBScript Database Tutorial Part 1

Probably the most popular use for ASP scripting is connections to databases. It's incredibly useful and surprisingly easy to do.

The first thing you need is the database, of course. A variety of programs can be used to create it, but probably the most popular is Microsoft Access. You can also use FoxPro or create it directly in an SQL Server using whichever utilities are supplied with the server (Enterprise Manager in the case of Microsoft SQL Server), or native SQL commands.

The choice of final format for the database is important. Almost any Windows NT web server can host an Access or FoxPro database, but this method is problematic - because of the inability to make changes to the structure of the database while it is live, and because of performance considerations. Don't be too scared off though - if you're looking to learn ASP Database interfacing, or you're developing a new site or low traffic site, there's absolutely nothing to stop you starting with an Access or FoxPro database.

The best method from several points of view is to create your database using Access or FoxPro, and then upscale it to SQL Server - you can then use Enterprise Manager to make changes while the database is live.

Because of the heavy server load involved in Internet Information Server 4 and SQL Server, it is almost essential that you select a web host that offers dedicated SQL Servers - especially where you are sharing servers (most often the case) or you have a high-traffic Site.

We'll attempt here to show you the basics of database usage with Active Server Pages and VBScript. We don't claim that these are the best methods available, but they will get you started down the path to developing real Web Applications.

We will be developing a range of articles over coming weeks that will show you how to optimise your code to cause the minimum possible server load and provide the maximum possible performance for your application.

VBScript Database Tutorial Part 2

Connecting to the Database

The Datasource is essentially a connection from the web server to a database, which can either be on a dedicated machine running SQL server or a database file sitting somewhere on the web server.

There are essentially two types of Datasources (DSN's):

1. System DSN
A datasource created on the web server by the administrator of the server. The most popular type of DSN and generally a lot more reliable.

2. File DSN
Essentially a connection that your script makes itself each time access to the database is required, specifying the path to and name of the database. The database must reside on the server in a directory that your script can access for this to work.

The code below is designed around a System DSN - we will incorporate File DSN code over coming weeks.

Setup Datasource - use at the beginning of the page

```
<%
Session.timeout = 15
Set conn = Server.CreateObject("ADODB.Connection")
conn.open "ODBCDSNName","username","password"
Set Session("MyDB_conn") = conn
%>
```

'ODBCDSNName' should be the name assigned to the DSN by the server administrator when they created the Datasource.

Close the Connection to the Datasource

Should be done at the end of each page for each Datasource opened.

```
<%
conn.close
set conn = Nothing
%>
```

VBScript Database Tutorial Part 3

Working With Recordsets

In order to read information from a Datasource, you need to open a 'Recordset' - a set of database records based on some type of criteria, either all of the records in a table or those matching some condition or set of conditions.

To create a recordset containing all of the records in one table of the database, without sorting them:

```
<%
strSQLQuery = "SELECT * FROM tablename"
Set rs = Server.CreateObject("ADODB.Recordset")
rs.Open strSQLQuery, conn, 3, 3
%>
```

To create a recordset containing only those records in one table where the field 'Name' consists of only the word 'Fred':

```
<%
strSQLQuery = "SELECT * FROM tablename WHERE Name = 'Fred'"
Set rs = Server.CreateObject("ADODB.Recordset")
rs.Open strSQLQuery, conn, 3, 3
%>
```

To create a recordset containing all of the records in one table and sort them alphabetically based on the field Name starting at 'a':

```
<%
strSQLQuery = "SELECT * FROM tablename ORDER BY Name ASC"
Set rs = Server.CreateObject("ADODB.Recordset")
rs.Open strSQLQuery, conn, 3, 3
%>
```

To create a recordset containing all of the records in one table and sort them alphabetically based on the field 'Name' starting at 'z':

```
<%
strSQLQuery = "SELECT * FROM tablename ORDER BY Name DESC"
Set rs = Server.CreateObject("ADODB.Recordset")
rs.Open strSQLQuery, conn, 3, 3
%>
```

If you want to be able to be sure of selecting only one particular record from a database, each table should have a 'Unique Key' field - usually a simple auto-incrementing number field. You can then select, update or delete records in the database using this field as the criteria.

```
<%
strSQLQuery = "SELECT * FROM tablename WHERE RecordID = 15"
Set rs = Server.CreateObject("ADODB.Recordset")
rs.Open strSQLQuery, conn, 3, 3
%>
```

To retrieve the values of various fields in a Recordset use:

```
<%
strValue = rs("FieldName")
%>
```

strValue can of course be any variable name, *FieldName* is the field name in the table that makes up the recordset. Multiple recordsets can be opened for the same database and/or table - just change the 'rs' to whatever name you want to give to each recordset. Standard SQL queries can be used to update, add or delete records, or create recordsets based on conditions. For example:

```
<%
strSQLQuery = "DELETE * FROM tablename WHERE Name = 'Fred'"
Set rs = Server.CreateObject("ADODB.Recordset")
rs.Open strSQLQuery, conn, 3, 3
%>
```

```
<%
strSQLQuery = "DELETE * FROM tablename WHERE Name = 'Fred' AND Address
= 'Smith St'"
Set rs = Server.CreateObject("ADODB.Recordset")
rs.Open strSQLQuery, conn, 3, 3
%>
```

```
<%
strSQLQuery = "DELETE * FROM tablename WHERE Name = 'Fred' OR Name =
'John'"
Set rs = Server.CreateObject("ADODB.Recordset")
rs.Open strSQLQuery, conn, 3, 3
%>
```

```
<%
strSQLQuery = "UPDATE tablename SET FieldName1 = " & strValue1 & " WHERE
FieldName2 = " & strValue2 & ";"
Set rs = Server.CreateObject("ADODB.Recordset")
rs.Open strSQLQuery, conn, 3, 3
%>
```

```
<%
strSQLQuery = "INSERT INTO tablename (FieldName1, FieldName2) VALUES (" &
strValue1 & ", " & strValue2 & ")"
Set rs = Server.CreateObject("ADODB.Recordset")
rs.Open strSQLQuery, conn, 3, 3
%>
```

There are some minor differences in the SQL queries used for Access versus SQL Server - the above works on SQL Server.

Note:
tablename would normally be just the name of the table in the database when using an Access database for instance, but when using SQL Server you also need to specify a UserName with access permissions to that database - in the format 'UserName.TableName'. Talk to your server administrator for more information.

SQL Query Reference

SQL Queries have 3 basic parts:

1. The Statement
Essentially determines the purpose of the SQL query - to retrieve records, delete them, change them, etc.

2. The Clause(s)
Determines which records are affected by the query and how they are sorted - and can also perform a number of other functions, including grouping of indentical records.

3. The Operation(s)
Actually perform operations on the records, such as combining or joining result sets.

SQL SubQueries

SQL Date Formats

Statements	
SELECT Statement	Retrieve records from a database.
INSERT INTO Statement	Insert records into a database.
SELECT...INTO Statement	Select records from one table/database and insert them into another table/database.
UPDATE Statement	Update/change records.
DELETE Statement	Delete Records.
TRANSFORM Statement	Creates a crosstab query.

Clauses	
FROM Clause	Determines the tables in the database that the query is to affect/retrieve.
WHERE Clause	Determines the selection criteria to be used.
ORDER BY Clause	Determines the field(s) to be used to sort the results of a SELECT or SELECT...INTO statement, together with the order - either 'ASC' (ascending) or 'DESC' (descending).
GROUP BY Clause	Determines the field(s) to be used for the grouping of the results of a SELECT or SELECT...INTO statement.
HAVING Clause	Specifies which grouped records are displayed in a SELECT statement with a GROUP BY clause.

IN Clause	Identifies tables in any external database to which the database engine can connect.

Operations	
UNION Operation	Used in combination with both Statements and Clauses for combining the results of two or more independent queries or tables.
INNER JOIN Operation	
LEFT JOIN Operation	
RIGHT JOIN Operation	

SQL SubQueries

Top

A subquery is a SELECT statement nested inside a SELECT, SELECT...INTO, INSERT...INTO, DELETE, or UPDATE statement or inside another subquery.

You can use three forms of syntax to create a subquery:

comparison [ANY | ALL | SOME] (*sqlstatement*)

expression [NOT] IN (*sqlstatement*)

[NOT] EXISTS (*sqlstatement*)

Part	Description
comparison	An expression and a comparison operator that compares the expression with the results of the subquery.
expression	An expression for which the result set of the subquery is searched.
sqlstatement	A SELECT statement, following the same format and rules as any other SELECT statement. It must be enclosed in parentheses.

You can use a subquery instead of an expression in the field list of a SELECT statement or in a WHERE or HAVING clause. In a subquery, you use a SELECT statement to provide a set of one or more specific values to evaluate in the WHERE or HAVING clause expression.

Use the ANY or SOME predicate, which are synonymous, to retrieve records in the main query that satisfy the comparison with any records retrieved in the subquery. The following example returns all products whose unit price is greater than that of any product sold at a discount of 25 percent or more:

```
SELECT * FROM Products
WHERE UnitPrice > ANY
(SELECT UnitPrice FROM OrderDetails
WHERE [Discount] >= .25);
```

Use the ALL predicate to retrieve only those records in the main query that satisfy the comparison with all records retrieved in the subquery. If you changed ANY to ALL in the above example, the query would return only those products whose unit price is greater than that of all products sold at a discount of 25 percent or more. This is much more restrictive.

Use the IN predicate to retrieve only those records in the main query for which some record in the subquery contains an equal value. The following example returns all products sold at a discount of 25 percent or more:

```
SELECT * FROM Products
WHERE ProductID IN
(SELECT ProductID FROM OrderDetails
WHERE [Discount] >= .25);
```

Conversely, you can use NOT IN to retrieve only those records in the main query for which no record in the subquery contains an equal value.

Use the EXISTS predicate (with the optional NOT reserved word) in true/false comparisons to determine whether the subquery returns any records.

You can also use table name aliases in a subquery to refer to tables listed in a FROM clause outside the subquery. The following example returns the names of employees whose salaries are equal to or greater than the average salary of all employees with the same job title. The Employees table is given the alias “T1":

```
SELECT LastName,
FirstName, Title, Salary
FROM Employees AS T1
WHERE Salary >=
(SELECT Avg(Salary)
FROM Employees
WHERE T1.Title = Employees.Title) Order by Title;
```

The AS reserved word is optional.

Examples

```
SELECT LastName, FirstName, Title, Salary
FROM Employees
WHERE Title LIKE "*Sales Rep*" AND Salary > ALL
(SELECT Salary FROM Employees
WHERE (Title LIKE "*Manager*") OR (Title LIKE "*Director*"));
```

Lists the name, title, and salary of every sales representative whose salary is higher than that of all managers and directors.

```
SELECT DISTINCTROW ProductName, UnitPrice
FROM Products WHERE UnitPrice =
(SELECT UnitPrice FROM [Products]
WHERE ProductName = "Aniseed Syrup");
```

Lists the name and unit price of every product whose unit price is the same as that of Aniseed Syrup.

```
SELECT DISTINCTROW [Contact Name], CompanyName, [Contact Title], Phone
FROM Customers WHERE CustomerID IN
(SELECT DISTINCTROW CustomerID
FROM Orders WHERE OrderDate BETWEEN #04/1/94# AND #07/1/94#);
```

Lists the company and contact of every customer who placed an order in the second quarter of 1994.

```
SELECT LastName, FirstName, Title, Salary
FROM Employees T1
WHERE Salary >=
(SELECT AVG(Salary)
FROM Employees
WHERE Employees.Title = T1.Title)
ORDER BY Title;
```

Lists employees whose salary is higher than the average salary for all employees.

```
SELECT FirstName, LastName
FROM Employees AS E
WHERE EXISTS
(SELECT *
FROM Orders AS O
WHERE O.EmployeeID = E.EmployeeID);
```

Selects the name of every employee who has booked at least one order. This could also be done with an INNER JOIN.

VBScript Database Tutorial Part 4

The code used to move around in a recordset is:

```
<%
' Move to the first record
rs.movefirst

' Move to the last record
rs.movelast

' Move to the next record
rs.movenext

' Move to the previous record
rs.moveprevious
%>
```

To delete the current record in a recordset:

```
<%
rs.delete
%>
```

To loop through a recordset, writing a field from each record to the web page:

```
<%
' Open the recordset first (see above)
do until rs.eof
response.write(rs("FieldName") & "<br>")
rs.movenext
loop
%>
```

The same thing, but starting with the last record and working in reverse:

```
<%
' Open the recordset first (see above)
rs.movelast
do until rs.bof
response.write(rs("FieldName") & "<br>")
rs.moveprevious
loop
%>
```

The most important thing to remember is that there are essentially four steps involved:

1. Create the connection to the database.

2. Specify a set of records (Recordset) that you want to retrieve, delete, update etc.

3. Do whatever you need to do with those records - ie. write them to the web page.

4. Close the connection to the database.

Once you get a handle on the basics here, the next step is to work out how SQL Queries work - check our Database Section for the SQL Query reference.

Code Snippets - VBScript

Counting Records Without Creating a Full Recordset

```
<%
Set rs = Server.CreateObject("ADODB.Recordset")
strSQLQuery = "SELECT Count(*) AS MyRecordCount FROM tablename"
rs.Open strSQLQuery, conn, 3, 3
response.write(rs("MyRecordCount") & "<br>" & vbCrLf)
%>
```

Counting records in this way is desirable where you don't need to use the contents of the set of records you need to count - it imposes much less load on the server. A full range of SQL Queries can be used to count records matching specific criteria.

Recordset Filtering Using VBScript

To Set The Filter

```
<% rs.Filter = "Name = '" & strName & "'" %>
```

To Clear The Filter

```
<% rs.Filter = 0 %>
```

'rs' is the Recordset created earlier in the script.
Any valid SQL query can be used to create the filter.

Remove Single Quotes from SQL Queries

Single quotes - ' - in SQL Query strings will break the query and generally cause an 'Incorrect Syntax Near' error, but one simple line of code can be used to strip out any single quotes:

```
<%
strSQLQuery = Replace(strSQLQuery, "'", "", 1, -1, 1)
%>
```

In situations where you want to add the single quotes to the database record, you can replace them with double quotes:

```
<%
strSQLQuery = Replace(strSQLQuery, "'", "''", 1, -1, 1)
%>
```

This should be a standard part of any database add/update routine - especially where the data to be added is user input - an order form for instance.