

A Practical Guide to

Microsoft
Active Server Pages

3.0

By

Manas Tungare

www.manastungare.com



About this guide ...

This practical guide aims to be a complete programming guide as well as a reference for the serious ASP programmer.

It does not assume any prior knowledge of ASP, and starts from the ground up. The chapters are organized according to the increasing complexity of ASP scripts that you will be writing. Simple scripts come first, and then the techniques needed for more complex scripts are covered. This is especially useful for the beginner who is usually inundated by long, boring technical dissertations in the first chapter of any book.

Experienced programmers will also find this guide useful, for it contains the following references in one easy-to-locate booklet.

VBScript Reference

A complete documentation of all VBScript functions with the necessary information for using them. Ideal for those situations where you know what you want to do, but can't remember the function that will do it for you.

SQL Reference

This includes complete syntactic specifications of the Structured Query Language, along with examples to demonstrate the use.

ADO Reference

A guide on ASP cannot miss out this quintessential section. It gives complete details on most of the objects in the ADO hierarchy.

The examples and samples used in this guide can be found on the web at:

<http://www.manastungare.com/asp>

Thanks to Pankaj Kamat for proofreading this document.

Copyright © 2000 - 2001, Manas Tungare.

<http://www.manastungare.com/>

Every effort has been made to ensure correctness & reliability of the information provided, however the author may not be held responsible for any errors that may have crept in.

INTRODUCTION	7
THE NEED FOR ASP	7
SO WHAT IS ASP ?	7
WHAT CAN YOU DO WITH ACTIVE SERVER PAGES?	8
WHAT DO SERVER-SIDE SCRIPTS LOOK LIKE?	8
WHAT YOU NEED TO RUN ASP	8
INTERNET INFORMATION SERVICES	8
PERSONAL WEB SERVER	9
BEFORE YOU BEGIN ...	9
STEPS FOR INSTALLATION	9
CREATING VIRTUAL DIRECTORIES	9
ACCESSING YOUR WEBPAGE	9
WHAT IS LOCALHOST?	10
HELLO, WORLD (AND MORE) !	11
ANOTHER WAY ...	11
DISPLAYING THE DATE ...	12
... AND MORE	12
VARIABLES AND CONSTRUCTS	13
DIM 'EM FIRST	13
THE BIG IF	14
FOR-NEXT LOOPS	16
FOR EACH OBJECT IN COLLECTION ...	17
WHILE ... WEND	18
SELECT CASE:	18
COMPLEX CONDITIONS & CONNECTIVES:	19
AND, OR AND NOT	19
SUBROUTINES, FUNCTIONS AND INCLUDES	20
SUBROUTINES	20
FUNCTIONS	20
INCLUDES	21
THE OBJECT MODEL	23
THE REQUEST OBJECT	23
SYNTAX	24
COLLECTIONS	24
METHODS	24
NOTE	24
REQUEST.SERVERVARIABLES	25
THE RESPONSE OBJECT	26
SYNTAX	26
COLLECTIONS	26
PROPERTIES	26
METHODS	27
THE SERVER OBJECT	27
SYNTAX	28
PROPERTIES	28
METHODS	28
THE SESSION OBJECT	28
SYNTAX	28
COLLECTIONS	29
PROPERTIES	29
METHODS	29
EVENTS	29
THE APPLICATION OBJECT	29
SYNTAX	30
COLLECTIONS	30
EVENTS	30
HANDLING USER INPUT : FORMS & QUERYSTRINGS	31

THE REQUEST.FORM COLLECTION	31
THE REQUEST.QUERYSTRING COLLECTION	32
GET AND POST	33
WHEN TO USE GET?	33
WHEN TO USE POST?	33
DATA MANIPULATION USING ASP	34
DISPLAYING DATA FROM A TABLE	34
RETRIEVING DATA	34
MOVING ON TO COMPLEX QUERIES	37
INSERTING DATA INTO A TABLE	37
UPDATING RECORDS	38
DELETING RECORDS	39
MORE ...	40
VBSRIPT REFERENCE	41
STATEMENTS AND KEYWORDS	41
OPERATORS	41
VBSRIPT FUNCTIONS	42
TYPE CHECKING FUNCTIONS	42
VALUE	42
CONSTANT	42
DATA TYPE	42
TYPECASTING FUNCTIONS	43
FORMATTING FUNCTIONS	44
MATH FUNCTIONS	44
DATE FUNCTIONS	45
DATE CONSTANTS	45
DAY OF THE WEEK CONSTANTS	46
STRING FUNCTIONS	46
OTHER FUNCTIONS	48
CONTROL STRUCTURES	48
SQL REFERENCE	52
THE SELECT STATEMENT	52
<i>INNER</i> AND <i>OUTER JOIN</i> STATEMENTS	53
CALCULATED VALUES AND THE <i>GROUP BY</i> CLAUSE	54
THE INSERT STATEMENT	54
THE <i>UPDATE</i> STATEMENT	55
THE <i>DELETE</i> STATEMENT	56
ACTIVEX DATA OBJECTS (ADO) REFERENCE	57
THE CONNECTION OBJECT	57
OPENING A DATABASE CONNECTION	57
THE CONNECTMODEENUM CONSTANTS	58
THE CONNECTIONSTRING	58
THE CONNECTION.EXECUTE METHOD	60
MANAGING TRANSACTIONS WITH A CONNECTION OBJECT	60
THE RECORDSET OBJECT	61
USING THE RECORDSET.OPEN METHOD	61
POSITIONING A RECORDSET OBJECT — THE <i>MOVE</i> METHODS	63
RECORDSET SORTING AND SEARCHING METHODS	63
THE FIELD OBJECT	65
EXTENDING ASP : COM COMPONENTS	66
THE BASICS	66
PROPERTIES	66
METHODS	67
ARGUMENTS	67
COLLECTIONS	68
THE DEFAULT METHOD OR PROPERTY	68

INSTANTIATING AN OBJECT	69
BUILT-IN COM OBJECTS	70
PROGID	70
FURTHER ON ...	70
COPY & PASTE ASP SCRIPTS	71
FEEDBACK PAGE	71
TELL A FRIEND ABOUT THIS SITE	71
FURTHER EXAMPLES	73
THE ASP RESOURCE GUIDE	74
EDITORS	74
TEXTPAD	74
MACROMEDIA DREAMWEAVER ULTRADEV 1.0	74
ASP HOSTING	74
DOMAINLX	74
BRINKSTER	75
SOFTCOM TECHNOLOGIES	75
COM COMPONENTS FOR USE WITH ASP	77
ASPEMAIL	77
JMAIL	77
ASPUPLOAD	77

INTRODUCTION

The need for ASP

Why bother with ASP at all, when HTML can serve your needs? If you want to display information, all you have to do is fire up your favorite text editor, type in a few HTML tags, and save it as an HTML file. Bingo, you're done!

But wait – what if you want to display information *that changes*? Supposing you're writing a page that provides constantly changing information to your visitors, for example, weather reports, stock quotes, a list of your girlfriends, etc, HTML can no longer keep up with the pace. What you need is a system that can present dynamic information. And ASP fits the bill perfectly.

So what is ASP ?

In the language of Microsoft, Active Server Pages is an open, compile-free application environment in which you can combine HTML, scripts, and reusable ActiveX server components to create dynamic and powerful Web-based business solutions. Active Server Pages enables server side scripting for IIS with native support for both VBScript and JScript.

Translated into plain English, that reads -

Active Server Pages (ASPs) are Web pages that contain *server-side scripts* in addition to the usual mixture of text and HTML tags. Server-side scripts are special commands you put in Web pages that are processed before the pages are sent from the server to the web-browser of someone who's visiting your website. When you type a URL in the Address box or click a link on a webpage, you're asking a web-server on a computer somewhere to send a file to the web-browser (also called a "client") on your computer. If that file is a normal HTML file, it looks the same when your web-browser receives it as it did before the server sent it. After receiving the file, your web-browser displays its contents as a combination of text, images, and sounds.

In the case of an Active Server Page, the process is similar, except there's an extra processing step that takes place just before the server sends the file.

Before the server sends the Active Server Page to the browser, it *runs* all server-side scripts contained in the page. Some of these scripts display the current date, time, and other information. Others process information the user has just typed into a form, such as a page in the website's guestbook. And you can write your own code to put in whatever dynamic information you want.

To distinguish Active Server Pages from normal HTML pages, Active Server Pages are given the ".asp" extension.

What Can You Do with Active Server Pages?

There are many things you can do with Active Server Pages.

- You can display date, time, and other information in different ways.
- You can make a survey form and ask people who visit your site to fill it out, send emails, save the information to a file, etc
- You can have a database which people can access via the web. People can get information from the database as well as update or insert information into it.
- You can password-protect certain sections of your site, and make sure that only authorized users can see that information.
- The possibilities are virtually endless. Most widgetry that you see on webpages nowadays can be easily done using ASP.

What Do Server-Side Scripts Look Like?

Server-side scripts typically start with `<%` and end with `%>`. The `<%` is called an *opening tag*, and the `%>` is called a *closing tag*. In between these tags are the server-side scripts. You can insert server-side scripts anywhere in your webpage - even inside HTML tags.

What you need to run ASP

Since the server must do additional processing on the ASP scripts, it must have the ability to do so. The only servers which support this facility are Microsoft Internet Information Services & Microsoft Personal Web Server. Let us look at both in detail, so that you can decide which one is most suitable for you.

Internet Information Services

This is Microsoft's web server designed for the Windows NT platform. It can only run on Microsoft Windows NT 4.0, Windows 2000 Professional, & Windows 2000 Server. The current version is 5.0, and it ships as a part of the Windows 2000 operating system.

Personal Web Server

This is a stripped-down version of IIS and supports most of the features of ASP. It can run on all Windows platforms, including Windows 95, Windows 98 & Windows Me. Typically, ASP developers use PWS to develop their sites on their own machines and later upload their files to a server running IIS. If you are running Windows 9x or Me, your only option is to use Personal Web Server 4.0.

Before you begin ...

Here a few quick tips before you begin your ASP session!

Unlike normal HTML pages, you cannot view Active Server Pages without running a web-server. To test your own pages, you should save your pages in a directory mapped as a virtual directory, and then use your web-browser to view the page.

Steps for Installation

- From the CD, run the SETUP.EXE program for starting the web-server installation.
- After the installation is complete, go to

Start > Programs > Microsoft PWS > Personal Web Manager.

and click the “Start” button under Publishing.

- Now your web-server is up & running.

Creating Virtual Directories

After you have installed the web-server, you can create virtual directories as follows:

- Right-Click on the folder that you wish to add as a virtual directory.
- Select “Properties” from the context-menu.
- In the second tab titled “Web Sharing,” click “Share this folder,” then “Add Alias”.

(If you do not see these options enabled, your web-server is not properly running. Please see the steps above under “Installation.”)

Accessing your webpage

Now that your server is completely configured and ready to use, why not give it a try?

Start your web-browser, and enter the following address into the address-bar.

http://localhost/

You should see a page come up that tells you more about Microsoft IIS (or PWS, as the case may be)

What is localhost?

Let us first see, what we mean by a hostname. Whenever you connect to a remote computer using its URL, you are in effect calling it by its hostname. For example, when you type in

http://www.google.com/

you are really asking the network to connect to a computer named `www.google.com`. It is called the “hostname” of that computer.

`localhost` is a special hostname. It always references your own machine. So what you just did, was to try to access a webpage on your own machine (which is what you wanted to do anyway.) For testing all your pages, you will need to use `localhost` as the hostname. By the way, there is also a special IP address associated with `localhost`, that is

127.0.0.1

So you could as well have typed:

http://127.0.0.1/

and would have received the same page.

To access pages in a virtual directory called `myscripts` for example, you should type in:

http://localhost/myscripts/

in the address bar. I hope the concept is now clear ...

HELLO, WORLD (AND MORE) !

Now let's write the ubiquitous first program, Hello World.

```
<HTML>
<HEAD>
<TITLE>Hello, World !</TITLE>
</HEAD>
<BODY>
<%
    Response.Write "Hello, World!"
%>
</BODY>
</HTML>
```

As you can see above, we have enclosed a single line of VBScript within the opening and closing tags. It says,

Response. Write "Hello, World!"

This statement displays the string "Hello, World!" on the webpage.

Another way ...

Let us try that in a different way that is shorter than this one.

```
<HTML>
<HEAD>
<TITLE>Hello, World !</TITLE>
</HEAD>
<BODY>
    <%= "Hello, World!" %>
</BODY>
</HTML>
```

Notice the presence of the = sign just after the <%. It has a similar effect to that of the `Response.Write` statement.

Displaying the Date ...

Now let us go one step further, and make a page that tells you the date today!

```
<HTML>
<HEAD>
<TITLE>Hello, World !</TITLE>
</HEAD>
<BODY>
    <%= Date %>
</BODY>
</HTML>
```

Using the function “Date” gives you the current date. And the function, “Time” returns the time. To get both, use the function, “Now”.

The following code shows how the “Now” function is used.

```
<HTML>
<HEAD>
<TITLE>Hello, World !</TITLE>
</HEAD>
<BODY>
<%
    Response.Write Now
%>
</BODY>
</HTML>
```

... and more

You can also get the individual elements, Year, Date, Month, Hour, Minute & Second of the time by using the above functions.

```
<HTML>
<HEAD>
<TITLE>Hello, World !</TITLE>
</HEAD>
<BODY>
<%
Response.Write "Year: " & Year (Now)
Response.Write "Month: " & Month (Now)
Response.Write "MonthName: " & MonthName (Month(Now))

Response.Write "Hour: " & Hour (Now)
Response.Write "Minute: " & Minute (Now)
Response.Write "Second: " & Second (Now)
%>
</BODY>
</HTML>
```

Notice the mixing of plain text and VBScript code. With this beginning, let us now move on to handling variables, constants, and various constructs.

VARIABLES AND CONSTRUCTS

Dim ‘em first

To “Dim” it means to Dimension it. That’s VB lingo. A variable is declared in VBScript using the Dim keyword.

```
<%  
    Dim myVar  
>%
```

VB programmers will notice here, that we have not included any indication of the type of the said variable. E.g. **Dim myString as String**, or **Dim myString\$**.

In VBScript, all variables are variants. Their type is determined automatically by the runtime interpreter, and the programmer need not (and should not) bother with them.

By default, VBScript does not force requiring variable declaration. That is, it is allowed to use a variable directly without declaring it first. However, experienced programmers know the importance of making it compulsory to declare all your variables first – without that, the bugs that may result are verrry difficult to detect. Considering this, we have here a simple directive to make variable declaration compulsory.

```
<%  
    Option Explicit  
    Dim myVar  
>%
```

Remember that **Option Explicit** must necessarily be the first statement of your ASP page, otherwise a server error is generated.

To illustrate what I mean, if you had a page that read:

```
<%  
    Pi = 3.141592654  
    Response.Write Pi  
>%
```

this is a perfectly valid page. You will get the value of Pi written back to the page, as you really expected.

Now, using the Option Explicit directive as above, let's rewrite the same page as follows:

```
<%  
    Option Explicit  
    Pi = 3.141592654  
>%
```

Now you have an error that says:

```
Microsoft VBScript runtime error (0x800A01F4)  
Variable is undefined: 'Pi'  
/asp/test.asp, line 3
```

The reason is that, now, with the Option Explicit directive, IIS expects to see a declaration of every variable that is used.

So, in this case, the correct script must read:

```
<%  
    Option Explicit  
    Dim Pi  
    Pi = 3.141592654  
>%
```

The Big If

The simplest of constructs, found in every language, is the If-Then-Else statement. I think it's familiar to everyone, so let's start with an example.

```
<%  
    If OK = True Then  
        Response.Write "OK"  
    Else  
        Response.Write "Error"  
    End If  
>%
```

Some important points to note:

- The condition after the `IF` must be followed by the `Then` keyword. This is unlike C, C++ or Java which do not require the `Then` keyword to follow.
- If only a single statement is to be executed in the `Then` block, it may directly follow the `Then` on the same line. If there are multiple statements to execute in the `Then`-block, the first statement should begin on the line after `Then`.
- The `Else`-block, as in most languages, is optional.
- The complete set of statements in the `Then`-block as well as the `Else`-block need to be “closed” by the `End IF` keyword. This is very important, and the source of many hard-to-locate errors! Take care not to forget it!

The following are further examples of valid as well as invalid If-constructs:

```
<%
    If OK = True Then Response.Write "OK" Else Response.Write
    "Error"
%>
```

This is valid. Since only one statement is to be executed in each of the `Then`- & `Else`-blocks, we do not require an `End If`. In addition, the `Else` statement must be on the same line.

```
<%
    If OK = True Then Response.Write "OK"
    Else Response.Write "Error"
%>
```

This is invalid. Since only one statement belongs to the `Then`-block, the first part of the construct above is fine. However, the `Else` cannot continue on the next line in such a case.

A simple rule of thumb to follow, is to use only one form of the `IF`-statement for all your needs:

```
<%
    If OK = True Then
        Response.Write "OK"
        ' ... Any more statements
    Else
        Response.Write "Error"
        ' ... Any more statements
    End If
%>
```

Incidentally, any line that begins with an apostrophe, `'`, is a comment; it is ignored by the interpreter. The following line is partly executable and partly a comment.

```
<%
    OK = True      ' Sets OK to True
```

```
%>
```

The comment begins after the apostrophe.

For-Next Loops

The syntax is as follows

```
<%  
    For I = 1 to 10  
        Response. Write "Number = " & I & vbCrLf  
    Next  
%>
```

And the output ...

```
Number = 1  
Number = 2  
Number = 3  
Number = 4  
Number = 5  
Number = 6  
Number = 7  
Number = 8  
Number = 9  
Number = 10
```

The `vbCrLf` used in the statement above is a predefined constant that equals the combination of the Carriage-Return character (CR for short), and the Line Feed character (LF for short.) Using it causes the output to continue on the next line.

Without the `vbCrLf`, our output would have appeared on one long line:

```
Number = 1Number = 2Number = 3Number = 4Number = 5Number =  
6Number = 7Number = 8Number = 9Number = 10
```

Let us take a case of nested loops to clarify things:

```
<%  
    For I = 1 to 8  
        For j =1 to 8  
            Response. Write "X"  
        Next  
        Response. Write vbCrLf  
    Next  
%>
```

This will draw a nice chessboard pattern on the screen. (You will need to view the source of the page in your browser however. If you look at the page in the browser itself, you will not see the true result. More about that later.)

A very important point to note is that the `Next` statement that completes the `For` does not take an argument. You cannot say:

```
Next I
```

Or

```
Next J
```

This is invalid. Each `Next` statement encountered is automatically assumed to complete the immediately preceding `For` statement.

Finally, VBScript also allows the `Step` keyword to modify the interval or step-size of the `For`-loop variable.

```
<%  
    For I = 1 to 10 Step 2  
        Response.Write "Number = " & I & vbCrLf  
    Next  
%>
```

gives you:

```
Number = 1  
Number = 3  
Number = 5  
Number = 7  
Number = 9
```

The loop counted `I` in steps of 2, thus taking on only odd values from the entire set of 10.

For Each Object In Collection ...

The `For-Each` construct is unique to VBScript (and its parent, Visual Basic, of course!) It allows you to iterate through the items in a collection one by one.

```
<%  
    For Each Member in Team  
        Response.Write Member  
    Next  
%>
```

Here, Team is assumed to be a collection of items. This statement is very useful in scenarios, where the size of the collection is not known in advance. Using the `For Next` statement assures that all items in that collection will be processed, and no “Array Index Out Of Bounds” errors will be generated.

While ... Wend

Again, here is one of the popular looping constructs of all time.

```
<%  
    While Not RS.EOF  
        Response.Write RS.Fields("Name")  
        RS.MoveNext  
    Wend  
%>
```

or,

```
<%  
    Do While Not RS.EOF  
        Response.Write RS.Fields("Name")  
        RS.MoveNext  
    Loop  
%>
```

The `While` statement executes the statements in its loop until the given condition remains true. The moment it becomes false, the loop halts.

Remember to end the `while` Statement with the `wend` Keyword.

A variation of the `While` loop that tests the condition after the loop is the `Do-loop`.

```
<%  
    Do  
        TimePass()  
    Until ExamDate - Now = 30  
%>
```

I hope the meaning is amply clear from the example above.

Select Case:

To make a choice between a set of items that can be assigned to a variable, use the `Select Case` statement.

```
<%
```

```
Select Case Choice
  Case "1":
    Response. Write "You chose 1"
  Case "2":
    Response. Write "You chose 2"
  Case "3":
    Response. Write "You chose 3"
  Case "4":
    Response. Write "You chose 4"
End Select
%>
```

Complex conditions & connectives:

You may use more than one condition within each of the constructs above.

And, Or and Not

```
Condition1 And Condition2 Or Condition3
```

The conditional connectives are the words, `And`, `Or` and `Not` themselves. The C symbols `||`, `&&` and `!` are no longer valid.

The precedence is `Not` having the highest priority, followed by `And` and finally `Or`. So, use brackets when necessary to separate your connectives.

SUBROUTINES, FUNCTIONS AND INCLUDES

Like any programming language worth its salt, VBScript allows you to define and use your own Subroutines, Functions & Includes.

Subroutines

Subroutines are defined via the `sub` keyword.

```
<%  
    Sub SayHello  
        Response. Write "Hello !"  
    End Sub  
>%
```

A subroutine may accept parameters too, which can be of any type.

```
<%  
    Sub SayHelloTo (Person)  
        Response. Write "Hello, " & Person & "!"  
    End Sub  
>%
```

Parameters do not have defined types; their usage determines the type. All parameter types are variants by default.

Subroutines cannot return a value; that is, they can only be called, their value cannot be used. To return values to calling programs, we use `Functions`.

Functions

Functions are defined similar to Subroutines:

```
<%  
    Function Add (A, B)  
        Add = A + B  
    End Function  
>%
```

```
%>
```

As seen above, the function `Add` adds two numbers and returns their result. You can call this function as:

```
<%
Response.Write Add (2, 3)
%>
```

which will produce the sum as the output. More complex processing may be done inside function bodies.

```
<%
Function Calculate (A, B, Op)
  Select Case Op
    Case "+"
      Calculate = A + B
    Case "-"
      Calculate = A - B
    Case "*"
      Calculate = A * B
    Case "/"
      Calculate = A / B
  End Select
End Function
%>
```

```
<%
Response.Write Calculate(2, 3, "+")
Response.Write Calculate(2, 3, "-")
%>
```

Includes

Server Side Includes or SSI is a very simple programming language but it also has a very limited number of instructions. We will consider only one option SSI allows us to use within our asp scripts: `include/virtual`.

```
<!-- #INCLUDE FILE="filename.inc" -->
```

As we have the `#include <stdio.h>` statement in C, we have the `#include` directive here. The purpose is exactly similar: `#INCLUDE` actually includes the said file at the given location. Any script that is present within that file is automatically executed.

Include files are typically used to store application-wide functions and procedures, as well as various utility functions, e.g. `IsValidEmail (...)` a function that checks if a given string is a valid email address. You can put such functions in just 1 file, and include that on each one of your pages.

Or, you may use this functionality to insert headers & footers on every page. Putting all the standard content in one file, you simply include that file in each of your pages, so you do not need to copy & paste it everywhere. Updates are easier too, since you can modify just one file and not worry about forgetting to update another.

Another form of the `INCLUDE` directive uses the keyword `virtual`:

```
<!-- #INCLUDE VIRTUAL="/directory/file.inc" -->
```

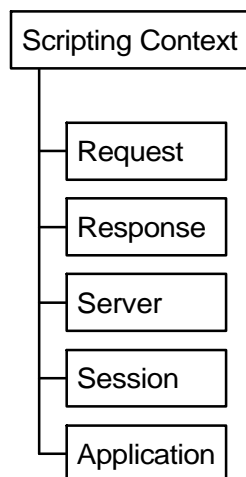
which will locate the file considering it as the virtual path. According to the line above, the file is expected to be found at

```
www.domain.com/directory/file.inc
```

Note that the virtual path of the file above, is the one that is entered in the `#INCLUDE` statement.

THE OBJECT MODEL

ASP is a scripting environment revolving around its Object Model. An Object Model is simply a hierarchy of objects that you may use to get services from. In the case of ASP, all commands are issued to certain inbuilt objects, that correspond to the Client Request, Client Response, the Server, the Session & the Application respectively. All of these are for global use



Request: To get information from the user

Response: To send information to the user

Server: To control the Internet Information Server

Session: To store information about and change settings for the user's current Web-server session

Application: To share application-level information and control settings for the lifetime of the application

The Request and Response objects contain collections (bits of information that are accessed in the same way). Objects use methods to do some type of procedure (if you know any object-oriented programming language, you know already what a method is) and properties to store any of the object's attributes (such as color, font, or size).

The Request object

The Request object retrieves the values that the client browser passed to the server during an HTTP request.

Syntax

`Request [. collection / property / method] (variable)`

Collections

ClientCertificate

To get the certification fields from the request issued by the Web browser. The fields that you can request are specified in the X.509 standard

Cookies

The values of cookies sent in the HTTP request.

Form

The values of form elements in the HTTP request body.

QueryString

The values of variables in the HTTP query string.

ServerVariables

The values of predetermined environment variables.

Properties

TotalBytes

Read-only. Specifies the total number of bytes the client is sending in the body of the request.

Methods

BinaryRead

Retrieves data sent to the server from the client as part of a POST request.

Variable parameters are strings that specify the item to be retrieved from a collection or to be used as input for a method or property.

Note

All variables can be accessed directly by calling Request(variable) without the collection name. In this case, the Web server searches the collections in the following order.

- QueryString
- Form
- Cookies
- ClientCertificate
- ServerVariables

If a variable with the same name exists in more than one collection, the Request object returns the first instance that the object encounters.

It is strongly recommended that when referring to members of the ServerVariables collection the full name be used. For example, rather than Request.(AUTH_USER) use Request.ServerVariables(AUTH_USER).

Request.ServerVariables

A complete list of all server variables is as below:

<i>Variable</i>	<i>Meaning</i>
ALL_HTTP	HTTP Headers from client
ALL_RAW	Raw HTTP Headers from client
APPL_MD_PATH	Metabase path for the ISAPI DLL
APPL_PHYSICAL_PATH	Physical path to the metabase
AUTH_PASSWORD	What the user entered in the client's authentication dialog
AUTH_TYPE	Authentication the server used
AUTH_USER	Authenticated user name
CERT_COOKIE	Unique ID of the client certificate
CERT_FLAGS	Is client certification valid?
CERT_ISSUER	Issuer of the client certificate
CERT_KEYSIZE	Number of bits in the SSL key
CERT_SECRETKEYSIZE	Number of bits in the secret key
CERT_SERIALNUMBER	Serial Number for the client certificate
CERT_SERVER_ISSUER	Issuer of the the server certificate
CERT_SERVER_SUBJECT	Subject of the server certificate
CERT_SUBJECT	Subject of the client certificate
CONTENT_LENGTH	Length of the content
CONTENT_TYPE	MIME type of the current page
GATEWAY_INTERFACE	CGI version from server
HTTPS	Is this secure through SSL?
HTTPS_KEYSIZE	Number of bits in the SSL key
HTTPS_SECRETKEYSIZE	Number of bits in the secret key
HTTPS_SERVER_ISSUER	Issuer of the server certificate
HTTPS_SERVER_SUBJECT	Subject of the server certificate
INSTANCE_ID	ID for this instance in IIS
INSTANCE_META_PATH	Metabase path for this instance
LOCAL_ADDR	IP of server
LOGON_USER	NT login for current user
PATH_INFO	Server virtual path
PATH_TRANSLATED	Server absolute path
QUERY_STRING	Variable name value pairs from the url string
REMOTE_ADDR	Client IP address for requesting machine
REMOTE_HOST	Client IP address for requesting host
REMOTE_USER	Remote User
REQUEST_METHOD	Method of request
SCRIPT_NAME	virtual path and file name of current script
SERVER_NAME	Server name
SERVER_PORT	Port being accessed
SERVER_PORT_SECURE	0=not secure, 1=secure

SERVER_PROTOCOL	Name/Version of protocol used
SERVER_SOFTWARE	HTTP software used on the server
URL	URL without the domain name
HTTP_ACCEPT	MIME types the browser knows
HTTP_ACCEPT_LANGUAGE	Browser's language setting
HTTP_CONNECTION	HTTP Connection
HTTP_HOST	Domain hosting this request
HTTP_USER_AGENT	Browser being used
HTTP_PRAGMA	Cache page or not?
HTTP_COOKIE	Cookie related to this page
HTTP_ACCEPT_CHARSET	ISO character set being accepted

The Response object

The Response object is used to send information to the user. The Response object supports only Cookies as a collection (to set cookie values). The Response object also supports a number of properties and methods.

Syntax

`Response.collection|property|method`

Collections

Cookies

Specifies cookie values. Using this collection, you can set cookie values.

Properties

Buffer

Indicates whether page output is buffered.

CacheControl

Determines whether proxy servers are able to cache the output generated by ASP.

Charset

Appends the name of the character set to the content-type header.

ContentType

Specifies the HTTP content type for the response.

Expires

Specifies the length of time before a page cached on a browser expires.

ExpiresAbsolute

Specifies the date and time on which a page cached on a browser expires.

IsClientConnected

Indicates whether the client has disconnected from the server.

Pics

Adds the value of a PICS label to the pics-label field of the response header.

Status

The value of the status line returned by the server.

Methods*AddHeader*

Sets the HTML header name to value.

AppendToLog

Adds a string to the end of the Web server log entry for this request.

BinaryWrite

Writes the given information to the current HTTP output without any character-set conversion.

Clear

Erases any buffered HTML output.

End

Stops processing the .asp file and returns the current result.

Flush

Sends buffered output immediately.

Redirect

Sends a redirect message to the browser, causing it to attempt to connect to a different URL.

Write

Writes a variable to the current HTTP output as a string. This can be done by using the construct

```
Response. Write("Hello")
```

or the shortcut command

```
<%= "Hello" %>
```

The Server object

The Server object provides access to methods and properties on the server. Most of these methods and properties serve as utility functions.

Syntax

Server. property/method

Properties

ScriptTimeout

The amount of time that a script can run before it times out.

Methods

CreateObject

Creates an instance of a server component. This component can be any component that you have installed on your server (such as an ActiveX).

HTMLEncode

Applies HTML encoding to the specified string.

MapPath

Maps the specified virtual path, either the absolute path on the current server or the path relative to the current page, into a physical path.

URLEncode

Applies URL encoding rules, including escape characters, to the string.

The Session object

You can use the Session object to store information needed for a particular user-session. Variables stored in the Session object are not discarded when the user jumps between pages in the application; instead, these variables persist for the entire user-session.

The Web server automatically creates a Session object when a Web page from the application is requested by a user who does not already have a session. The server destroys the Session object when the session expires or is abandoned.

One common use for the Session object is to store user preferences. For example, if a user indicates that they prefer not to view graphics, you could store that information in the Session object.

Note Session state is only maintained for browsers that support cookies.

Syntax

Session. collection/property/method

Collections

Contents

Contains the items that you have added to the session with script commands.

StaticObjects

Contains the objects created with the <OBJECT> tag and given session scope.

Properties

CodePage

The codepage that will be used for symbol mapping.

LCID

The locale identifier.

SessionID

Returns the session identification for this user.

Timeout

The timeout period for the session state for this application, in minutes.

Methods

Abandon

This method destroys a Session object and releases its resources.

Events

Scripts for the following events are declared in the global.asa file.

Session_OnEnd

Session_OnStart

The Application object

The Application object can store information that persists for the entire lifetime of an application (a group of pages with a common root). Generally, this is the whole time that the IIS server is running. This makes it a great place to store information that has to exist for more than one user (such as a page counter). The downside of this is that since this object isn't created anew for each user, errors that may not show up when the code is called once may show up when it is called 10,000 times in a row. In addition, because the Application object is shared by all the users, threading can be a nightmare to implement.

You can use the Application object to share information among all users of a given application. An ASP-based application is defined as all the .asp files in a virtual directory and its subdirectories. Because the Application object can be shared by more than one user, there are Lock and Unlock methods to ensure that multiple users do not try to alter a property simultaneously.

Syntax

<i>Application.method</i>

Collections

Contents

Contains all of the items that have been added to the Application through script commands.

StaticObjects

Contains all of the objects added to the session with the <OBJECT> tag.

Lock

The Lock method prevents other clients from modifying Application object properties.

Unlock

The Unlock method allows other clients to modify Application object properties.

Events

Application_OnEnd

Application_OnStart

Scripts for the preceding events are declared in the global.asa file. For more information about these events and the global.asa file, see the Global.asa Reference.

Remarks

You can store values in the Application Collections. Information stored in the Application collections is available throughout the application and has application scope.

HANDLING USER INPUT : FORMS & QUERYSTRINGS

What good is a language that won't allow you to read user input effectively! HTML, the good old markup language provides the user with forms to enter his data in, and you, as an ASP programmer, can write scripts to process the input.

The Request.Form Collection

When you have an HTML form, say,

```
<FORM METHOD="post" ACTION="process.asp">
  <INPUT TYPE="text" NAME="FirstName">
  <INPUT TYPE="text" NAME="LastName">
  <INPUT TYPE="radio" NAME="Sex" VALUE="M">
  <INPUT TYPE="radio" NAME="Sex" VALUE="F">
  <TEXTAREA NAME="Address">
  </TEXTAREA>
  <INPUT TYPE="submit" VALUE="Send">
</FORM>
```

you have within it a number of elements, each with a unique name. The fields in the form above are FirstName (Text), LastName (Text), Sex (Option: M or F), and Address (Multiline Text). The last input type is "submit" that is a button required to submit the user input to your script. On clicking the Submit button, the contents of each of these fields are posted to the script that you specified in the FORM Action attribute. In the above example, it is "process.asp".

The form processing script can access these input values as below:

```
Request.Form ("FirstName")
```

```
Request.Form ("LastName")
```

Once you have this value, you can process it as you need – enter it into a database, mail it to yourself, - anything you want.

Please note that the METHOD specified in the FORM tag must be POST if you want to use the Request.Form collection to process it.

To know how to enter these into a database, skip to the next chapter. To know about another technique of passing input to an ASP page, read on ...

The Request.QueryString Collection

Quite often, you might have seen page URL's like the one below:

```
http://www.greetings.com/show.asp?CardID=128762173676
```

This is a direct link to a card that your friend sent you. You just need to click on the link, and the card shows up. You do not need to identify yourself or enter any code number anywhere. All the information that the site needs, is encoded in the string,

```
CardID=128762173676
```

This is known as the Query String and forms part of a URL.

You can pass multiple values too, using something like:

```
Page.asp?FirstName=Manas&LastName=Tungare&Sex=M
```

The Request.QueryString Collection helps you sort this stuff out and extract only what you need – the values of the variables themselves.

So to access the data contained in the variable FirstName above, you would use:

```
Request.QueryString("FirstName")
```

This again, is a regular variable that you can assign to another, or do arithmetic on.

The Request.QueryString collection gives you access to yet another class of variables – those passed via a FORM with it's METHOD = "get." However, there is a limit to the amount of data that can be passed on via the QueryString and you are expected to use a form for more data.

GET and POST

One thing we ignored in our discussion about forms was that the METHOD by which the form is submitted may be one of the two: GET or POST.

When to use GET?

Any data that you pass via a GET can be retrieved in your script by using the Request.QueryString collection. GET may be used for small amounts of data – the reason being that, the data items are appended to the URL by your browser, and obviously, you cannot have an infinitely long URL (with the QueryString).

When to use POST?

Almost always. Stick to POST for your forms, and be sure to use the Request.Form collection to access them (and *not* the Request.QueryString collection.)

DATA MANIPULATION USING ASP

Here we shall look at a practical example of using ASP & ADO to create a database-driven website. We'll tackle it piecemeal, looking at the simpler queries first and then moving on to more complex ones.

Displaying Data from a Table

Let us have a running database example for this section. Consider a database for a class of students.

The database schema is as follows:

Table: Student	
ID	Student ID Numbers; also the primary key of the table.
FirstName	First name of the Student.
LastName	Last name of the Student.
DateofBirth	Birthdate of the Student.
Email	Email address of the student.

Retrieving Data

You'll need to use the SQL SELECT statement for reading in data from a table. (More about SQL can be found in a later chapter, an SQL Reference).

Let's say, we want to display a complete list of all the students in the class. Here is a complete page that lists out all the student records in a Table.

```
<HTML>
<HEAD>
  <TITLE>Student Records</TITLE>
</HEAD>

<BODY>
<%
Dim DB
```

```

Set DB = Server.CreateObject ("ADODB.Connection")
DB.Open ("PROVIDER=Microsoft.Jet.OLEDB.4.0; DATA SOURCE=" +
"C:\Databases\Students.mdb")

Dim RS
Set RS = Server.CreateObject ("ADODB.Recordset")
RS.Open "SELECT * FROM Students", DB

If RS.EOF And RS.BOF Then
    Response.Write "There are 0 records."
Else
    RS.MoveFirst
    While Not RS.EOF
        Response.Write RS.Fields ("FirstName")
        Response.Write RS.Fields ("LastName")
        Response.Write "<HR>"
        RS.MoveNext
    Wend
End If
%>
</BODY>
</HTML>

```

Let's look at the example line by line.

The first few lines are the opening HTML tags for any page. There's no ASP code within them. The ASP block begins with the statement,

```
Dim DB
```

which is a declaration of the variable that we're gonna use later on. The second line,

```
Set DB = Server.CreateObject ("ADODB.Connection")
```

does the following two things:

Firstly, the right-hand-side statement, `Server.CreateObject()` is used to create an instance of a COM object which has the ProgID `ADODB.Connection`. The `Set` Statement then assigns this reference to our variable, `DB`. Now, we use the object just created to connect to the database using a `Connection String`.

The string,

```
"PROVIDER=Microsoft.Jet.OLEDB.4.0; DATA SOURCE=" +
"C:\Databases\Students.mdb"
```

is a string expression that tells our object where to locate the database, and more importantly, what type the database is – whether it is an Access database, or a Sybase database, or else, is it Oracle. (Please note that this is a Connection String specific to Access 2000 databases. This example does not use ODBC.)

If the **DB.Open** statement succeeds without an error, we have a valid connection to our database under consideration. Only after this can we begin to use the database.

The immediate next lines,

```
Dim RS
Set RS = Server.CreateObject ("ADODB.Recordset")
```

serve the same purpose as the lines for creating the **ADODB.Connection** object. Only now we're creating an **ADODB.Recordset**!

Now,

```
RS.Open "SELECT * FROM Students", DB
```

is perhaps the most important line of this example. Given an SQL statement, this line executes the query, and assigns the records returned to our Recordset object. The bare-minimum syntax, as you can see, is pretty straight-forward. Of course, the **Recordset.Open (...)** method takes a couple of more arguments, but they are optional, and would just complicate things at this juncture.

Now, assuming that all the records we want are in our Recordset object, we proceed to display it.

```
If RS.EOF And RS.BOF Then
    Response.Write "There are 0 records."
```

In any scenario where it is expected that no records might exist, this is an important error check to be performed. In case your query returned no results, the **Recordset.BOF** (beginning of file) & **Recordset.EOF** (end of file) are both **True** at the same time. So you can easily write an If-statement to perform a very basic error check. (If you don't do this now, you'll encounter errors in the later part of the script. It's always wise to prevent rather than cure errors.)

We shall look at the next few lines as a complete block and not as separate lines of code.

```
Else
    RS.MoveFirst
    While Not RS.EOF
        Response.Write RS.Fields ("FirstName")
        Response.Write RS.Fields ("LastName")
        Response.Write "<HR>"
```

```

        RS. MoveNext
    Wend
End If

```

RS. MoveFirst is a method that moves the record pointer (for now, consider this to be an imaginary structure that always points to the current record in the Recordset) to the First record. By default, it may or may not be positioned correctly, so it is imperative to position it before you begin any operations.

Then we have a While-loop that iterates through all the records contained in the Recordset. The condition that we check is that **RS. EOF** should be **False**. The moment it is **True**, it can be inferred that there are no more records to be found.

RS. Fields("FirstName") retrieves the value of the "FirstName" field of the current record. We use a **Response. Write** statement to write it out to the page. Similarly, we write the **RS. Fields ("LastName")** after the first name.

You may also use a shortcut syntax for this, which takes the form:

```
RS ("FirstName")
```

After you're done displaying, you must advance the record pointer to the next record, so you execute a **RS. MoveNext**. And that's all you wanted to do within the loop, so you end the loop now. Just write **Wend** and the loop ends! And so does our little example!

Moving on to complex queries

Queries are what get complex, not the method of accessing data! So no matter what data you want, you use exactly the same syntax in your code. The only thing that changes is the SQL statement, and perhaps, the fields that you actually display on the page.

Inserting Data into a Table

Although SQL provides us the INSERT INTO statement for inserting records into a database, I would suggest using the ADODB.Recordset object for doing this to make things simpler.

So here's how you insert a new record:

```

<HTML>
<HEAD>
  <TITLE>Student Records</TITLE>
</HEAD>
<BODY>
  <%
Dim DB

```

```
Set DB = Server.CreateObject ("ADODB.Connection")
DB.Mode = adModeReadWrite
DB.Open ("PROVIDER=Microsoft.Jet.OLEDB.4.0; DATA SOURCE=" +
"C:\Databases\Students.mdb")

Dim RS
Set RS = Server.CreateObject ("ADODB.Recordset")
RS.Open "Students", DB, adOpenStatic, adLockPessimistic

RS.AddNew
    RS ("FirstName") = "Manas"
    RS ("LastName") = "Tungare"
    RS ("Email") = "manas@manastungare.com"
    RS ("DateOfBirth") = CDate("4 Feb, 1980")
RS.Update
%>
</BODY>
</HTML>
```

The first few lines are exactly the same as in the previous example. Note that we set the `Connection.Mode` to `adModeReadWrite` since we are going to insert data, which is a Write-operation. We also use the ADO constants, `adOpenStatic` & `adLockOptimistic` while opening the Recordset for it to be updateable.

The lines,

```
RS.AddNew
    RS ("FirstName") = "Manas"
    RS ("LastName") = "Tungare"
    RS ("Email") = "manas@manastungare.com"
    RS ("DateOfBirth") = CDate("4 Feb, 1980")
RS.Update
```

are what do the main processing. `RS.AddNew` adds a new, blank record to the database. Then you set the fields by assigning your data to the respective fields of the Recordset. Note the short-cut syntax used in this example.

Finally, when you're done assigning all the values, execute the `Recordset.Update` method to commit all changes to the record.

Updating Records

If you know how to insert records, then updating them is a breeze. Because you're already come more than halfway while inserting records!

```
<HTML>
<HEAD>
    <TITLE>Student Records</TITLE>
</HEAD>
<BODY>
```

```

<%
Dim DB

Set DB = Server.CreateObject ("ADODB.Connection")
DB.Mode = adModeReadWrite
DB.Open ("PROVIDER=Microsoft.Jet.OLEDB.4.0; DATA SOURCE=" +
"C:\Databases\Students.mdb")

Dim RS
Set RS = Server.CreateObject ("ADODB.Recordset")
RS.Open "SELECT * FROM Students WHERE FirstName = 'Manas'",
DB, adOpenStatic, adLockPessimistic

RS ("Email") = "mynewemail@manastungare.com"
RS ("DateOfBirth") = CDate("4 Feb, 1980")
RS.Update
%>
</BODY>
</HTML>

```

As you can see, everything else remains the same. Firstly, you need just position the current pointer to the record that you wish to update. Use a proper SQL statement to achieve this. (It is advisable to check if that record exists, prior to modifying it.)

Then, as earlier, modify the records by assigning new values to them. You need not assign values to all fields; just modify the fields you need. Then execute the RS.Update statement to write the changes back to the database. Lo!

Deleting Records

Use the SQL DELETE statement to delete one or more records satisfying a particular criterion.

```

<HTML>
<HEAD>
<TITLE>Student Records</TITLE>
</HEAD>
<BODY>
<%
Dim DB
Set DB = Server.CreateObject ("ADODB.Connection")
DB.Mode = adModeReadWrite
DB.Open ("PROVIDER=Microsoft.Jet.OLEDB.4.0; DATA SOURCE=" +
"C:\Databases\Students.mdb")

DB.Execute ("DELETE * FROM Students WHERE FirstName =
'Manas' ")

%>
</BODY>
</HTML>

```

Exercise the utmost caution while using the DELETE statement for two reasons:

- Firstly, because there's no Undo available to restore your changes! Once it's gone, it's gone.
- And secondly, because if you forget the WHERE clause, it proceeds to delete *all* of the records in the table.

There – we're done!

More ...

To gain an in-depth understanding of accessing data and complex queries, please refer to the SQL reference and ADO reference in this booklet. These are pretty exhaustive guides that you can refer to anytime for looking up a particular item easily.

VBSRIPT REFERENCE

This section covers the VBScript keywords, operators, functions, and control structures.

Statements and Keywords

Dim is used to declare variables. VBScript variables are variants, which means that they do not have to have a fixed data type.

Const is used to declare constants, which are like variables except that they cannot be changed in the script.

Option Explicit is put at the top of a page to force explicit declaration of all variables.

Operators

In order of precedence, this is a list of operators supported in VBScript.

- Anything in parentheses.
- Exponentiation (^)
- Negation (-)
- Multiplication, Division (*, /)
- Integer Division (\)
- Modulus (Mod)
- Addition, Subtraction (+,-)
- String Concatenation (&)
- Comparison Operators (=, <>, <, >, <=, >=)
- Not
- And
- Or
- Xor
- Eqv
- Imp

VBScript Functions

This will provide you with a quick look at the more important VBScript functions. They include functions for type checking, typecasting, formatting, math, date manipulation, string manipulation, and more.

Type Checking Functions

These functions allow you to determine the data subtype of a variable or expression.

- **VarType** (*expression*) returns an integer code that corresponds to the data type.
- **TypeName** (*expression*) returns a string with the name of the data type rather than a code.
- **IsNumeric** (*expression*) returns a Boolean value of **True** if the expression is numeric data, and **False** otherwise.
- **IsArray** (*expression*) returns a Boolean value of **True** if the expression is an array, and **False** otherwise.
- **IsDate**(*expression*) returns a Boolean value of **True** if the expression is date/time data, and **False** otherwise.
- **IsEmpty** (*expression*) returns a Boolean value of **True** if the expression is an empty value (uninitialized variable), and **False** otherwise.
- **IsNull** (*expression*) returns a Boolean value of **True** if the expression contains **no** valid data, and **False** otherwise.
- **IsObject** (*expression*) returns a Boolean value of **True** if the expression is an object, and **False** otherwise.

Value	Constant	Data Type
0	vbEmpty	Empty (This is the type for a variable that has not been used yet. In other words. Empty is the default datatype.)
1	vbNull	Null (No valid data)
2	vbInteger	Integer
3	vbLong	Long
4	vbSingle	Single
5	vbDouble	Double
6	vbCurrency	Currency
7	vbDate	Date
8	vbString	String
9	vbObject	Object

10	vbError	Error
11	vbBoolean	Boolean
12	vbVariant	Variant (used with vbArray)
13	vbDataObject	Data Access Object
14	vbDecimal	Decimal
17	vbByte	Byte
8192	vbArray	Array (VBScript uses 8192 as a base for arrays and adds the code for the data type to indicate an array. 8204 indicates a variant array, the only real kind of array in VBScript.)

Typecasting Functions

Typecasting allows you to convert between data subtypes.

- **CInt(expression)** casts expression to an integer. If expression is a floating-point value or a currency value, it is rounded. If it is a string that looks like a number, it is turned into that number and then rounded if necessary. If it is a Boolean value of True, it becomes -1. False becomes 0. It also must be within the range that an integer can store.
- **CByte(expression)** casts expression to a byte value provided that expression falls between 0 and 255. expression should be numeric or something that can be cast to a number.
- **CDBl(expression)** casts expression to a double, expression should be numeric or something that can be cast to a number.
- **CSng(expression)** casts expression to a single. It works like CDBl(), but must fall within the range represented by a single.
- **CBool(expression)** casts expression to a Boolean value. If expression is zero, the result is False. Otherwise, the result is True. Expression should be numeric or something that can be cast to a number.
- **CCur(expression)** casts expression to a currency value, expression should be numeric or something that can be cast to a number.
- **CDate(expression)** casts expression to a date value, expression should be numeric or something that can be cast to a number, or a string of a commonly used date format. DateValue(expression) or TimeValue(expression) can also be used for this.
- **CStr(expression)** casts expression to a string, expression can be any kind of data.

Formatting Functions

FormatDateTime(expression, format) is used to format the date/time data in expression. **format** is an optional argument that should be one of the following:

- **vbGeneralDate** Display date, if present, as short date. Display time, if present, as long time. Value is 0. This is the default setting if no format is specified.
- **vbLongDate** Display date using the server's long date format. Value is 1.
- **vbShortDate** Display date using the server's short date format. Value is 2.
- **vbLongTime** Display time using the server's long time format. Value is 3.
- **vbShortTime** Display time using the server's short time format. Value is 4.

FormatCurrency(value, numdigits, leadingzero, negparen, delimiter) is used to format the monetary value specified by value.

- **numdigits** specifies the number of digits after the decimal place to display. -1 indicates to use the system default.
- **Tristate** options have three possible values. If the value is -2, it means use the system default. If it is -1, it means turn on the option. If it is 0, turn off the option.
- **leadingzero** is a Tristate option indicating whether to include leading zeroes on values less than 1.
- **negparen** is a Tristate option indicating whether to enclose negative values in parentheses.
- **delimiter** is a Tristate option indicating whether to use the delimiter specified in the computer's settings to group digits.

FormatNumber is used to format numerical values. It is almost exactly like **FormatCurrency**, only it does not display a dollar sign.

FormatPercent works like the previous two. The options are the same, but it turns the value it is given into a percentage.

Math Functions

- **Abs(number)** returns the absolute value of number.
- **Atn(number)** returns the arctangent, in radians, of number.
- **Cos(number)** returns the cosine of number, number should be in radians.
- **Exp(number)** returns e (approx. 2.71828) raised to the power number.
- **Fix(number)** returns the integer portion of number. If number is negative, Fix returns the first integer greater than or equal to number.
- **Hex(number)** converts number from base 10 to a hexadecimal string.
- **Int(number)** returns the integer portion of number. If number is negative, Int returns the first integer less than or equal to number.
- **Log(number)** returns the natural logarithm of number.
- **Oct(number)** converts number from base 10 to an octal string.
- **Rnd(number)** returns a random number less than one and greater than or equal to zero.

If the argument `number` is less than 0, the same random number is always returned, using `number` as a seed. If `number` is greater than zero, or not provided, `Rnd` generates the next random number in the sequence. If `number` is 0, `Rnd` returns the most recently generated number.

- **Randomize** initializes the random number generator.
- **Round(number)** returns `number` rounded to an integer.
- **Round(number, dec)** returns `number` rounded to `dec` decimal places.
- **Sgn(number)** returns 1 if `number` is greater than zero, 0 if `number` equals zero, and -1 if `number` is less than zero.
- **Sin(number)** returns the sine of `number`, `number` should be in radians.
- **Sqr(number)** returns the square root of `number`, `number` must be positive.
- **Tan(number)** returns the tangent of `number`, `number` should be in radians.

Date Functions

- **Date** returns the current date on the server.
- **Time** returns the current time on the server.
- **Now** returns the current date and time on the server.
- **DateAdd(interval, number, date)** is used to add to the date specified by `date`. **Interval** is a string that represents whether you want to add days, months, years, and so on. **Number** indicates the number of intervals you want to add; that is, the number of days, months, years, and so on.
- **DateDiff(interval, date1, date2, firstDOW, firstWOY)** is used to find the time between two dates. **DateDiff** returns the number of intervals elapsed between `date1` and `date2`. The optional integer **firstDOW** specifies what day of the week to treat as the first. The optional **firstWOY** specifies which week of the year to treat as the first.
- **DateSerial(year, month, day)** takes the integers `year`, `month`, and `day` and puts them together into a date value. They may be negative.
- **TimeSerial(hour, minute, second)** is similar to **DateSerial**. **TimeSerial** returns the number of seconds elapsed since midnight.
- **DatePart(interval, datetime, firstDOW, firstWOY)** allows you to retrieve the part of `datetime` specified by **interval**. The optional integer **firstDOW** specifies what day of the week to treat as the first. The optional **firstWOY** specifies which week of the year to treat as the first.

Date Constants

<i>Value</i>	<i>Meaning</i>
yyyy"	Year
"q"	Quarter
"m"	Month
"y"	Day of year
"D"	Day
"w"	Weekday
"ww"	Week of year
"h"	Hour
"n"	Minute

"s"	Second
-----	--------

Day of the Week Constants

0	vbUseSystem	National Language Support API Setting
1	vbSunday	Sunday (default
2	vbMonday	Monday
3	vbTuesday	Tuesday
4	vbWednesday	Wednesday
5	vbThursday	Thursday
6	vbFriday	Friday
7	vbSaturday	Saturday
	vbUseSystem	National Language Support API Setting
	vbFirstJan1	Week of January 1
	vbFirstFourDays	First week with four days of new yea
	vbFirstFullWeek	First full week

- **Year(date)** returns the year portion from date as a number.
- **Month(date)** returns the month portion from date as a number.
- **MonthName(date)** returns the month portion from date.
- **Day(date)** returns the day portion from date as a number.
- **Weekday(date)** returns the day of the week of date as a number.
- **Hour(time)** returns the hour portion from time.
- **Minute(time)** returns the minute portion from time.
- **Second(time)** returns the second portion from time.

String Functions

- **UCase(string)** returns string with all its lowercase letters converted to uppercase letters.
- **LCase(string)** returns string with all its uppercase letters converted to lowercase letters.
- **LTrim(string)** removes all the spaces from the left side of string.
- **RTrim(string)** removes all the spaces from the right side of string.
- **Trim(string)** removes spaces from both the left and the right sides.
- **Space(number)** returns a string consisting of number spaces.
- **String(number, character)** returns a string consisting of character repeated number times.
- **Len(string)** returns the number of characters in string.
- **Len(variable)** returns the number of bytes required by variable.
- **LenB(string)** returns the number of bytes required to store string.
- **StrReverse(string)** returns string with the characters in reverse order.
- **StrComp(string1, string2, comparetype)** is used to perform string comparisons. If **comparetype** is zero or omitted, the two strings are compared as if uppercase letters come before lowercase letters. If **comparetype** is one, the two strings are compared as if upper and lowercase letters are the same. **StrComp** returns -1 if **string1** is less than

`string2`. It returns 0 if they are the same, and 1 if `string1` is greater than `string2`.

- **Right(string, number)** returns the number rightmost characters of string.
- **RightB(string, number)** works like **Right**, but number is taken to be a number of bytes rather than characters.
- **Left(string, number)**, as you may guess, returns the number leftmost characters of string.
- **LeftB(string, number)** works like **Left**, but number is taken to be a number of bytes rather than characters.
- **Mid(string, start, length)** returns length characters from string, starting at position start. When length is greater than the number of characters left in the string, the rest of the string is returned. If length is not specified, the rest of the string starting at the specified starting position is returned.
- **MidB(string, start, length)** works like **Mid**, but start and length are both taken to be byte numbers rather than character numbers.
- **InStr(start, string1, string2, comparetype)** is used to check if and where `string2` occurs within `string1`. **start** is an optional argument that specifies where in `string1` to start looking for `string2`. **comparetype** is an optional argument that specifies which type of comparison to perform. If **comparetype** is 0, a binary comparison is performed, and uppercase letters are distinct from lowercase letters. If **comparetype** is 1, a textual comparison is performed, and uppercase and lowercase letters are the same. **InStr** returns zero if `string1` is empty (""), if `string2` is not found in `string1`, or if **start** is greater than the length of `string2`. It returns `Null` if either string is `Null`. It returns **start** if `string2` is empty. If `string2` is successfully found in `string1`, it returns the starting position where it is first found.
- **InStrB** works like **InStr** except that the **start** position and return value are byte positions, not character positions.
- **InStrRev(string1, string2, start, comparetype)** starts looking for a match at the right side of the string rather than the left side. **start** is by default -1, which means to start at the end of the string.
- **Replace(string, find, replace, start, count, comparetype)** is used to replace occurrences of **find** with **replace** in `string`. **start**, **count**, and **comparetype** are optional, but if you want to use one, you must use the ones that come before it. **start** indicates where the resulting string will start and where to start searching for **find**. It defaults to 1. **count** indicates how many times to perform the replacement. By default, **count** is -1, which means to replace every occurrence. If **comparetype** is 0, a binary comparison is performed, and uppercase letters are distinct from lowercase letters. If **comparetype** is 1, a textual comparison is performed, and uppercase and lowercase letters are the same.
- **Filter(arrStrings, SearchFor, include, comparetype)** searches an array of strings, `arrStrings`, and returns a subset of the array, **include** is a Boolean value. If **include** is `True`, **Filter** searches through all the strings in `arrStrings` and returns an array containing the strings that contain `SearchFor`. If **include** is `False`, **Filter** returns an array of the strings that do not contain `SearchFor`. **include** is optional and defaults to

True. `comparetype` works the same as in the other string functions we have discussed. If you want to use `comparetype`, you must use `include`.

- `Split(expression, delimiter, count, comparetype)` takes a string and splits it into an array of strings. `expression` is the string to be split up. If `expression` is zero length, `Split` returns an array of no elements, `delimiter` is a string that indicates what is used to separate the sub-strings in `expression`. This is optional; by default the delimiter is the space. If `delimiter` is zero length (""), an array of one element consisting of the whole string is returned, `count` is used to specify a maximum number of sub-strings to be created. The **default** for `count` is -1, which means no limit. If `comparetype` is 0, a binary comparison is performed, and uppercase letters are distinct from lowercase letters. If `comparetype` is 1, a textual comparison is performed, and uppercase and lowercase letters are the same. `comparetype` is only useful when the delimiter you have chosen is a letter.
- `Join(stringarray, delimiter)` does just the opposite of `Split`. It takes an array of strings and joins them into one string, using `delimiter` to separate them. `delimiter` is optional; the space is the default.

Other functions

- `LBound(array)` returns the smallest valid index for array.
- `UBound(array)` returns the largest valid index for array.
- `Asc(string)` returns the ANSI character code for the first character of string.
- `Chr(integer)` returns a string consisting of the character that matches the ANSI character code specified by integer.
- `Array(value1, value2, ..., valueN)` returns an array containing the specified values. This is an alternative to assigning the values to array elements one at a time.

Control Structures

Control structures allow you to control the flow of execution of your scripts. You can specify that some code should be executed only under certain circumstances, using conditional structures. You can specify that some code should be executed repeatedly, using looping structures. Lastly, you can specify that code from somewhere else in the script should be executed using branching controls.

Conditional Structures

The If...Then...Else construct allows you to choose which block of code to execute based on a condition or series of conditions.

```
<%  
If condition1 Then  
    codeblock1  
ElseIf condition2 Then  
    codeblock2  
Else
```

```

        codeblock3
End If
%>

```

If *condition1* is true, *codeblock1* is executed. If it is false, and *condition2* is true, *codeblock 2* is executed. If *condition1* and *condition2* are both false, *codeblock3* executes. An If-Then construct may have zero or more ElseIf statements, and zero or one Else statements.

In place of some really complex If ...Then constructs, you can use a Select Case statement. It takes the following form:

```

Select Case variable
  Case choice1
    codeblock1
  Case choice2
    codeblock2
  Case choice n
    codeblock n
  Case default
    default code block
End Select

```

This compares the value of *variable* with *choice1*, *choice2*, and so on. If it finds a match, it executes the code associated with that choice. If it does not, it executes the default code.

Looping Structures

Looping structures allow you to execute the same block of code repeatedly. The number of times it executes may be fixed or may be based on one or more conditions.

The For...Next looping structure takes the following form:

```

For counter = start to stop
  codeblock
Next

```

codeblock is executed with *counter* having the value *start*, then with *counter* having the value *start+1*, then *start+2*, and so forth through the value *stop*.

Optionally, you may specify a different value to increment *counter* by. In this case the form looks like this:

```

For counter = start to stop Step stepvalue
  codeblock
Next

```

Now *counter* will take the values *start+stepvalue*, *start+stepvalue+stepvalue*, and so forth. Notice that if *stepvalue* is negative, *stop* should be less than *start*.

The For Each...Next looping structure takes the following form:

```
For Each item In Set
    codeblock
Next
```

codeblock is executed with *item* taking the value of each member of *Set*. *Set* should be an array or a collection.

The Do While-Loop looping structure has the following form:

```
Do While booleanValue
    code block
Loop
```

codeblock is executed as long as *booleanValue* is True. If it is False to begin with, the loop is not executed at all.

The While...Wend looping structure has the following form:

```
While booleanValue
    codeblock
Wend
```

codeblock is executed as long as *booleanValue* is True. If it is False to begin with, the loop is not executed at all.

The Do-Loop While looping structure has the following form:

```
Do
    code block
Loop While booleanValue
```

codeblock is executed as long as *booleanValue* is True. The loop is executed at least once no matter what.

The Do Until-Loop looping structure has the following form:

```
Do Until booleanValue
    codeblock
Loop
```

code block is executed as long as *booleanValue* is false. If it is true to begin with, the loop is not executed at all.

The Do...Loop Until looping structure has the following form:

```
Do
    code block
Loop Until booleanValue
```

code block is executed as long as *booleanValue* is false. The loop is executed at least once no matter what.

Branching Structures

Branching structures allow you to jump from one position in the code to another. A subroutine does not return a value. It simply executes. Subroutines look like this:

```
Sub name (argumentlist)
    code block
End Sub
```

Functions do return values and have the following form:

```
Function name (argumentlist)
    code block
    name = expression
End Function
```

SQL REFERENCE

Structured Query Language (SQL) is a straightforward subject, partly because it doesn't do much, and partly because the language is standardized. Most modern databases use a variant of SQL that, for the most part, conforms to the American National Standards Institute (ANSI) 92 standard. That standard means you can use similar, although not quite identical, SQL code to access many different databases. Fortunately, for basic operations, there's no difference between most common databases.

SQL lets you perform four basic operations:

- SELECT - Retrieve data
- INSERT - Add data
- UPDATE - Change data
- DELETE - Remove data

The SELECT Statement

The SELECT statement retrieves data from the database. To retrieve the data, you specify a field list, a table list, a list of fields to sort by, and the sort order. The parts of a SQL statement are called clauses. A basic SELECT statement has up to four clauses. For example:

```
SELECT (field1 [, field2] ...)  
FROM (table1 [, table2] ...)  
WHERE (condition1 [, condition2] ...)  
ORDER BY (field1 [ASC|DESC] [, field2 [ASC|DESC]] ...)
```

The WHERE and ORDER BY clauses are optional. If you omit the WHERE clause, the query returns all rows from the specified tables. If you omit the ORDER BY clause, SQL retrieves rows in the sequence in which they're stored in a table. By default, when you retrieve data from multiple tables, SQL uses the row order from the first specified field.

At the most basic level, you can obtain all the information from a table using an asterisk (*) as a shorthand way of specifying all fields.

Of course, you don't have to select all fields, you can specify the exact fields and field order that you wish.

Programmers moving from file-based databases to relational databases often make the mistake of thinking that the simple SELECT statement is all they need. They are accustomed to scrolling (moving sequentially from field to field) through a set of records to find the info they need. That's absolutely the wrong way to approach relational databases. Don't search for records yourself — let the database do the work. That's what the WHERE clause does — it limits the returned records to exactly the ones you need

The ORDER BY clause of the SELECT statement controls the order of the records returned by the query.

The fields in the ORDER BY clause do not have to appear in the selected field list. The default sort order is ascending (ASC), but you can retrieve fields in reverse order by specifying the DESC keyword after the appropriate field name. You don't have to select all the fields, and you may select them in any order you desire.

The following SELECT statement includes all the basic SELECT clauses:

```
SELECT StudentID, LastName, FirstName
FROM Students
ORDER BY Grade DESC
```

INNER and OUTER JOIN Statements

You can use the SELECT statement to retrieve data from more than one table at a time. SQL statements referencing more than one table typically (but not necessarily) use a JOIN statement to connect the tables on a common field or value.

For example:

```
SELECT StudentID
FROM TeacherStudent INNER JOIN Teachers ON
TeacherStudent.TeacherID=Teachers.TeacherID
WHERE Teachers.LastName='Franklin' AND
Teachers.FirstName='Marsha'
```

When you use two tables, you can't use the asterisk shorthand to retrieve all the fields from only one of the tables (although you can use it to retrieve all the fields in both tables). In such cases, the tablename.* syntax selects all the fields from the named table.

The INNER JOIN statement requires that you specify which tables and fields the database should join to produce the query. Also, when you work with more than one table you must specify the table name as well as the column name for each field where the field name appears in more than one table. In

other words, if the column name is not unique among all fields in all tables in the FROM clause, the server will raise an error, because it can't distinguish the table from which to extract the data.

When you know that a foreign key may not exist, or may not match a key value in the joined table, you can perform a LEFT (OUTER) JOIN or a RIGHT (OUTER) JOIN. The OUTER keyword is optional. Outer joins return all the values from one of the tables even if there's no matching key.

Calculated Values and the *GROUP BY* Clause

Transact-SQL (T-SQL) contains a number of functions to calculate values. A calculated value is a result of an operation on one or more columns in multiple rows. For example, a sum, average, or total. In T-SQL, calculated values are called aggregates, and the functions are aggregate functions because they aggregate, or collect a number of values into a single value using a calculation. For example, you can retrieve the total number of rows in any table with the following SELECT statement, substituting an appropriate table name in the FROM clause:

```
SELECT count(*) FROM <tablename>
```

The syntax of the GROUP BY clause is:

```
SELECT (field1 [, field2] ...)  
FROM (table1 [, table2] ...)  
WHERE (condition)  
GROUP BY (field1 [, field2] ...)  
HAVING (condition)  
ORDER BY (field1 [ASC|DESC] [, field2 [ASC|DESC]] ...)
```

You've seen the rudiments of how to select data. Selecting data doesn't change it, so selecting is a safe operation. All the other statements change data in some way. You'll be happy to know that the other statements are considerably less complex than the SELECT statement. I suggest you make a backup copy of your database before you continue.

The INSERT Statement

SQL INSERT statements add one or more new rows to a table. The INSERT statement has two variations. The first variation adds one row by assigning values to a specified list of columns in a specified table. The values you want to insert follow a VALUES statement. You put parentheses around both the field list and the values list.

For example:

```
INSERT INTO tablename (field1 [, field2] ...)
VALUES (value1 [, value2] ...)
```

You must provide a value for all fields that cannot accept a null value and do not have a default value. You do not have to provide values for identity columns.

The second variation lets you add multiple rows using a SELECT query in place of the VALUES list, as follows:

```
INSERT INTO tablename (field1 [, field2] ...) SELECT query
```

If you're inserting data into all the columns in the target table, you can omit the field list. The SELECT statement you use to obtain the data you want to insert can include any clause or condition discussed in the previous section, including calculated fields and a GROUP BY clause.

The **UPDATE** Statement

UPDATE statements change data in one or more columns and in one or more rows. The UPDATE statement is dangerous, because if you forget to specify conditions, your database will happily update all the rows in the table. You should always specify a WHERE condition when updating data. The UPDATE statement has the following syntax:

```
UPDATE (tablename)
SET field1=(value|expression) [, field2=(value|expression)]...
FROM (table|query source)
WHERE (condition)
```

The UPDATE statement has four clauses. In the UPDATE clause, you must specify a table name containing the fields to update. You may not update multiple tables simultaneously.

The SET clause contains the list of fields you wish to update. You separate the list with commas. Each item in the list consists of a field name, an equals sign, and a new value. You can use a constant, a variable, a field from another table, or an expression for the value on the right-hand side of the equals sign.

The FROM clause is optional. If you're updating a single row with constant values, you can omit the FROM clause. You need the FROM clause when you're updating data in one table from values stored in a different table (or in another place in the same table). Fortunately, the FROM clause is identical to the FROM clause you saw earlier in *The SELECT Statement* section. You may update from multiple tables using JOIN statements as appropriate.

The WHERE clause (**Important: don't forget the WHERE clause!**), again, is a condition that identifies the rows in the target table you wish to update.

The *DELETE* Statement

The *DELETE* statement is the simplest of all, but quite powerful. You can use the *DELETE* statement to delete one or more rows in one or more tables. The *DELETE* statement is just as dangerous as the *UPDATE* statement, as you can see, because it cheerfully deletes data without prompting. If you accidentally run a *DELETE* statement, it's difficult to recover your data. You should rarely use a *DELETE* statement without a *WHERE* clause. If you want to delete all the data from a table it's much more efficient to use a different type of statement, one of a group of statements that alters the database itself—the *TRUNCATE TABLE* statement.

Truncating a table removes all the data and resets the identity column value to its default.

You should rarely use *DELETE* without a *WHERE* clause. There is one reason to do so. The *TRUNCATE* statement is not logged — that means you can't recover if you use it automatically, whereas the *DELETE* statement is a logged operation. That's the reason *TRUNCATE* is so much more efficient—it avoids the log operations, but it also means the data is unrecoverable from the transaction log.

ACTIVE X DATA OBJECTS (ADO) REFERENCE

ASP's primary interface to relational databases is through Microsoft's ActiveX Data Objects (ADO). This ability to access multiple types of data stores, along with a relatively simple and flat object model, make ADO the simplest method yet devised for retrieving data.

The three main objects in the ADO object model and their most useful and common methods are reviewed here. In ADO, there are often several ways to accomplish a task. However, there are reasons why you should prefer one object or method instead of another.

The Connection Object

Before you can retrieve any data from a database, you have to create and initialize a connection to that database. In ADO, you use a Connection object to make and break database connections. A Connection object is a high-level object that works through a provider (think driver) that actually makes the data requests.

Opening a Database Connection

A single project called ActiveX Data Objects Database (ADODB) contains all the ADO objects. You create a Connection object in the same way as any other ASP object - with the `Server.CreateObject` method.

```
Dim Conn
Set Conn = Server.CreateObject("ADODB.Connection")
```

By default, connections are read-only, but you can create a read-write or write-only connection by setting the Connection object's Mode property. There are several Mode constants - in fact, ADO is rife with constants. You have to include the `adovbs.inc` file (provided in the appendix to this guide). To use the ADO constants, include the following line in each file where you use ADO, substituting the appropriate drive and path for your server:

```
<!-- #INCLUDE FILE="adovbs.inc" -->
```

If you open the `adovbs.inc` file with Notepad or another text editor, you'll see groups of constants.

The `ConnectModeEnum` constants

Typically, you only need to select one of the first three values. If you only need to read information from the database (the most common action), use the `adModeRead` constant. If you only need to write data, use the `adModeWrite` constant. If you need to read and write data within a single page, use the `adModeReadWrite` constant. Some people always use the `adModeReadWrite` constant, but that slows down data access when you only need to read or write, but not both.

The last five constants are of less use in a Web application where you may not know how many people are simultaneously connected. The `adModeShareDenyRead` constant prevents other connections from reading the database. Similarly, the `adModeShareDenyWrite` constant lets other connections read from, but not write to the database. The misnamed `adModeShareExclusive` constant prevents all other connections from opening the database. To thoroughly confuse the issue, the `adModeShareDenyNone` constant allows all other connections to attach to the database with any type of permission. The `adModeRecursive` constant works in concert with all of the Share-type constants except `adModeShareDenyNone` to propagate the setting to sub-records of the current record. For example, you can use `adShareDenyRead + adModeRecursive` to deny read permissions to sub-records.

The `ConnectionString`

After setting the mode, you must set the Connection object's `ConnectionString` property. Although you must set this property each time you open a new Connection object, you should define the connection string (or strings) in your application's `global.asa` file as an application-level or session-level variable. There are at least three reasons to define the connection string in the `global.asa` file; it means you only have one place to check for connection string problems, you can change the connection from one database to another with minimal code changes during development, and you can copy or move your application from one server to another very quickly.

The `ConnectionString` property is both simple and complicated. It has several parts, all of which are optional, depending on which type of connection string you're using, but typically, you specify the following:

- Provider name
- Name of the database server
- Name of the database you want to use
- User ID (UID) with which to connect
- Password (PWD) for that user ID.

You separate the parts of the connection string with semicolons. For example, at the simplest level, you can use an Open Database Connectivity (ODBC) Data Source Name (DSN), a user ID, and password to connect to your database. A DSN already contains the provider, the database server, and the database name, so you don't have to specify those again.

For example:

```
Dim Conn
Set Conn = Server.CreateObject("ADODB.Connection")
Conn.Mode = adModeReadWrite
Conn.ConnectionString = "DSN=myDSN; UID=manas; PWD=manas; "
```

Unfortunately, that's not the best method. By default, ODBC DSNs use the MSDASQL provider, but the JET OLEDB provider is faster and provides more functionality. Use this type of connection string instead.

```
Dim Conn, ConnStr
ConnStr= "PROVIDER=Microsoft.Jet.OLEDB.4.0; DATA SOURCE=" +
Server.MapPath(Path2DB)
Set Conn = Server.CreateObject("ADODB.Connection")
Conn.Mode = adModeReadWrite
Conn.ConnectionString = ConnStr
```

The connection string contains the provider name, the name of the server (in this case, and the path to the MDB file. We use the Server.MapPath function to translate the virtual path to the actual path on the server's disk. For example, a database at the location

```
http://www.manastungare.com/users.mdb
```

can actually be the file

```
E:\Web\Databases\users.mdb
```

Server.MapPath translates the first address to the second (which is what is needed by the ADODB.Connection object.)

You must set most of the Connection object's properties before opening the connection. If you later want to change a property, close the connection, change the property value, and then reopen the connection.

To open a connection, use the Open method of the Connection object.

```
Conn.Open
```

If the Open method executes without errors, you have a working connection to the database.

All the procedures so far can be simplified with an alternate syntax. The Open method accepts up to four optional arguments: a `ConnectionString`, a user ID, a password, and the `Options` argument consisting of a `ConnectOptionEnum` constant.

```
Conn. Open ConnStr, "manas", "manas", adAsyncConnect
```

You can specify more than one value for the options by adding the constants together.

The Connection.Execute Method

There are three ways to obtain data from a database using ADO. All of them require a connection. The simplest way is to use the `Execute` method of the `Connection` object.

The `Execute` method accepts three arguments

- An SQL statement or query, table, view, or stored procedure name called the `CommandText` argument
- A variable called `RecordsAffected` that will contain the number of records affected by the statement or query after the `Execute` method completes
- And a `CommandTypeEnum` constant called `Options` that tells the database what type of statement or query you want to run, and whether to return a `Recordset` object.

`Connection` objects can open tables directly, and can execute SQL in either pre-compiled form (called stored procedures) or dynamically by interpreting and executing the SQL statement at runtime. All these types of requests return records. The returned records are called result sets, and ADO wraps the resulting rows in a `Recordset` object.

The return value of the `Execute` method, by default, is a `Recordset` object that contains the result of the statement or query. You can control whether the `Execute` method returns a `Recordset` object by adding the `adExecuteNoRecords` constant to the `Options` constant. If you're running a `SELECT` statement, you generally need the resulting `Recordset` object; but when you're running an `INSERT` or `UPDATE` query, you usually don't need any records returned.

Managing Transactions with a Connection Object

When you submit a command that changes data to SQL Server, such as an `UPDATE`, `INSERT`, or `DELETE` query, it always treats the statement as a transaction. If the statement fails, all operations in the statement fail. However, you often need to execute one or more statements as a transaction. To do that, you must wrap the statements in a high-level transaction yourself.

You manage transactions with the Connection object's `BeginTrans`, `CommitTrans`, and `RollbackTrans` methods.

The Recordset Object

Recordset objects provide more functionality than simply a method for holding and scrolling through data. A Recordset object is a table of values. It has rows and columns like a database table; but a Recordset object is not a table. It's more like a virtual table or view.

First, the values of the Recordset object's columns may come from several different tables via a JOIN operation. Second, the column values may be calculated values — they may not match any value in the database. Finally, you can search and sort Recordset objects, run them into strings or arrays, and even persist them to and retrieve them from disk storage as objects or as XML data.

Using the Recordset.Open Method

If you need a Recordset object with any type of cursor other than a forward-only, read-only cursor, you need to open it directly rather than calling the `Execute` method of a Connection object. Recordset objects also have an `Open` method, which takes several arguments

Recordset.Open `CommandText`, `Connection|ConnectionString`, `CursorType`, `LockType`, `Options`

The `CommandText` argument contains the SQL query. The `Connection | ConnectionString` argument contains either a reference to an open Connection object or a valid `ConnectionString` argument. If you use a `ConnectionString` argument, the `Open` method creates a Connection object for you.

If you're going to make only one call to the database in a page, letting ADO create a Connection object is a viable option. However, if you're going to make more than one call, you should create and open your own Connection object. The reason is that you have more control over the type and duration of the Connection object if you open and close it yourself.

The `CursorType` argument is a value derived from one or more `adCursorTypeEnum` values. The following list shows the valid values and a description of each:

- **adOpenForwardOnly**
Returns a forward-only cursor. This is the default cursor type. If you don't specify a cursor type, ADO always returns a forward-only cursor. As the name implies, you can only move forward, not backward, through the Recordset object. You should use this whenever you need to make only one pass through a Recordset object because it's the fastest type of cursor.
- **adOpenKeyset**
Returns a keyset cursor. You can move in any direction with this cursor

type first, last, forward, backward, skip, or move to bookmark (if the provider supports bookmarks). You can see changes that others make to the records in the Recordset object, but you can't see records added since you opened the Recordset object. You cannot access or change records that other users delete. Use a keyset cursor for large record sets where you need to be able to scroll backward or you need to change. The server creates a unique bookmark for each row when you first run the query. Those bookmarks don't change during the life of the Recordset object, which is why you can't see new records.

- **adOpenDynamic**
Returns a dynamic cursor. This type is exactly like a keyset cursor except that you can see new records that others add. A dynamic cursor checks constantly for updates and additions to the result set. It does not build a set of bookmarks for the result set, so a dynamic cursor often opens more quickly than a keyset cursor. Dynamic cursors require the most resources of all cursor types, so you should not use them unless you need to see additions to the result set while the Recordset object is open.
- **adOpenStatic**
Returns a static cursor, which is a fixed copy of a set of records. You cannot see any changes or inserts by others without querying the database again. Recordset objects with static cursors can be updated.

Depending on the provider, you may be able to see changes your application makes to data with a static cursor.

The **LockType** argument tells ADO how to treat database locks. In general, you want to avoid locking data for updates or inserts because locks created by one user can cause problems for other users in your application. Read-only locks do not cause such problems. The valid **LockType** arguments are:

- **adLockReadOnly**
Read-only — you cannot alter the data.
- **adLockPessimistic**
Pessimistic locking is the strongest type of lock. Records with pessimistic locking are unavailable to other users of your application. Pessimistic locks occur when the server delivers the record. The record remains locked until you close the Recordset object. You should avoid pessimistic locking in Web applications whenever possible.
- **adLockOptimistic**
Optimistic locking locks records just before an update occurs, and unlocks them immediately afterward. Other users can access data during the time you're updating the record, which means they may potentially be viewing outdated data. Similarly, with optimistic locking, multiple users may

simultaneously try to update the same data, leading to problems. You should avoid optimistic locking in Web applications whenever possible.

- **adLockBatchOptimistic**

Optimistic batch locks act like optimistic locks, except they work for batch updates — deferring immediate updates in favor of updating many records at one time rather than updating each record immediately as with **adLockOptimistic** locking. It's your call whether batch updates or immediate updates are better for your application, in part, it depends on the level of interactivity your application demands and how people expect to use the application.

The final **Recordset.Open** argument is the **Options** flag. The **Options** flag takes exactly the same values as the **Connection.Execute** options argument. Again, the options argument is not required, but you should always include it. It tells ADO whether the query is a table, view, stored procedure, or dynamic SQL statement.

The **Update** method fails because even though the **Recordset** object may have an updateable cursor type, the underlying connection is read-only. To open an updateable **Recordset** object you must set the **ConnectionMode** property to **adModeReadWrite**.

Positioning a Recordset Object — the *Move* Methods

After opening a **Recordset** object, you can use the **Move** methods to move forward and (depending on the cursor) backward through the data rows. **Recordset** objects provide a **RecordCount** property, which tells you the number of records in the **Recordset** object.

Think of a **Recordset** object as a table with an empty row at the top and bottom, and a current-record pointer. The record pointer points to only one record at a time. When you use one of the **Move** methods, you don't scroll the record set - you move the record pointer. **Recordset** objects have **EOF** (end-of-file) and **BOF** (beginning-of-file) methods to let you know when the record pointer has moved past the last record or prior to the first record. **EOF** and **BOF** are Boolean properties.

```
While Not R.EOF
    ' Do something
    R.MoveNext
Wend
```

It's important to check whether a **Recordset** object is at the **BOF** or **EOF** position before requesting data, because **Recordset** objects raise an error if you request data when the **Recordset** object is at either of these two positions.

Recordset Sorting and Searching Methods

You can search and sort data with **Recordset** methods, although it's much more efficient to obtain only the data you need from the server and retrieve it

already sorted. To sort a record set, assign its `Sort` property the names of the field(s) you want to sort by. For example, to sort the `Recordset`:

```
SELECT * FROM Students
```

by `LastName`, you would write:

```
R.Sort = "LastName"
```

To sort by more than one field, separate the field names with commas, as follows:

```
R.Sort = "LastName, FirstName"
```

The default sort order is always ascending, so you don't need to write an explicit direction (although you can). To sort in a specific order, append either `ASC` or `DESC` to the end of the sort string.

```
R.Sort = "LastName, FirstName DESC"
```

You can also search for specific records in a record set. To perform the search, use the `Recordset.Find` method. You specify the search conditions with the equivalent of a SQL `WHERE` clause, without the word `WHERE`. After performing a find, the `Recordset` object is positioned at the first record found, or if no records are found, at the end of the `Recordset` (EOF).

You may include multiple conditions, just as in a `WHERE` clause. In addition to the search criteria, the `Find` method accepts three other optional arguments:

- **SkipRecords**
The number of records to skip before beginning the search. This argument is particularly useful when you're searching in a loop. By setting `SkipRecords` to 1, you can begin searching in the record following the current record. When searching backward, set this value to a negative number.
- **SearchDirection:**
The direction to search, either `adSearchForward` or `adSearchBackward`.
- **Start:**
The number or bookmark of the record where the search should begin. You should specify either `SkipRecords` or `Start`, but not both.

The Field Object

Although we've been talking of Recordset objects as tables, that's just a convenient mental model. Recordset objects actually consist of a two-dimensional array of Field objects. In the ADO object model, Field objects contain the data. Therefore, each Field object has a type, a size, and a value. A Field object also has properties for the numeric scale of its value, the original value, the underlying value (in the database), the defined size, the actual size, and the precision, as well as a list of other attributes. Most of the time you will not need to deal with the Field object properties and methods, but it's useful to study them in case you do need them.

EXTENDING ASP : COM COMPONENTS

You have already used Component Object Model (COM) objects to create your ASP pages. However, unless you have developed COM objects or read a detailed book on COM, you might not know about the multitude of COM objects that you can use in ASP. Also, without sufficient COM knowledge, you might not be able to infer the methods and properties that exist for those objects when reading the documentation. One of the wonderful things about COM is that once you learn the standards and restrictions, you can quickly learn to implement other COM objects.

This section will demonstrate and describe the basics of COM to those familiar with VBScript and COM objects, and is especially useful to those who have used COM objects like ActiveX™ Data Objects (ADO) without knowing that they are COM.

The Basics

COM (<http://www.microsoft.com/com/>) is the standard for the interface to objects. By definition, COM objects have only methods and properties; there are no other interfaces. There isn't much difference between properties and methods from a programmer's standpoint: Methods can take arguments, properties can't. Properties can be read/write; methods - if they return a value - are read-only.

Component designers use methods and properties for different functionality. Properties usually represent some aspect of the object's state, whereas a method can be a function that performs regardless of whether the object's state is involved.

Properties

Properties do not take any arguments and are usually used to describe or set the state of an object. All properties return a value, however some properties are read-only, and some are read/write. Here is an example of the VBScript syntax for reading a property:

```
value = object.property
```

Note there are no parentheses, not even a blank set; that is, (). Here is the Visual Basic syntax for setting a property:

```
object.property = value
```

Methods

Methods can return values and take arguments. They are most often used to initiate an event within the object. Methods can be used to set values, but only when passing the value through the argument list. If a method returns a value but doesn't take an argument, the syntax will be:

```
value = object.method()
```

Note that the method has a set of blank parentheses. Methods that have a return value must have arguments encapsulated in parentheses. For example, the **Connection** object has an **Execute** method that returns a **RecordSet** object. Here is an example:

```
Set RS = Conn.Execute("SELECT * FROM TABLE")
```

Methods that do not return values do not have parentheses around the arguments. For example, the **Close** method of the **Connection** object is not encapsulated in parentheses:

```
Conn.Close
```

Arguments

Methods can take one or more arguments, or take none at all. However, arguments might be optional. If they are, you do not have to enter anything for an argument. Once one argument is optional, all arguments following it are also optional. For example, if arguments one and two are required, and three is optional, argument four has to be optional. A good example of an optional argument method is the **Open** method of the **Connection** object. The **Open** method has eight optional arguments. The first three are for establishing the database and the logon information.

```
Conn.Open "DSN", "sa", ""
```

This indicates a DSN of "DSN", a logon of "sa", and a password of "". You can also call the **Open** method as:

```
Conn. Open "driver=SQL
Server; server=yourServerName; uid=someUID; pwd=somePWD; database=
someDatabase; "
```

Calling the arguments by delimiting with the argument and leaving it blank causes the method to execute with nulls instead of the optional argument's default values.

```
Conn. Open "DSN", "sa", "", , , ,
```

This calls the optional methods with null values, which is different than earlier.

Collections

Collections are objects that represent a set of objects. All collections have predefined methods and properties. A collection object has an **Item** method, a **Count** property, and a **_NewEnum** method. A collection can also create objects of the collection type. In other words, if a particular object can be grouped in a set, that object will have a collection object that can create an instance of an object within the set. For example, a **Drives** collection object will contain a set of drives that can represent all the drives on a particular computer.

The **Count** property returns a LONG value that specifies how many objects are in the collection. By passing a LONG value - that is between one and the value returned by the **Count** property -- to the **Item** method, the collection method will return the object in the set that is associated with that position. Accessing an item in an array works similarly.

The **_NewEnum** method enables a programmer to iterate through the collection in a For...Next statement.

```
For Each Object in Collection
    ...
Next Object
```

Note that the **_NewEnum** method is not referenced within the syntax of the statement in Example 6. This is because the **_NewEnum** method has a special index that is used for the For...Next statement. In fact, all methods and properties in a COM object are indexed and certain indexes are used for particular tasks. For example, the zero index is used for the default method or property.

The Default Method or Property

The method or property that has the COM index of zero is called the default property. Visual Basic enables a programmer to not use the regular

method/property syntax when calling the default value; you can leave the syntactical call to the method/property off altogether. For example, the default method in all collections is the **Item** method.

```
Set Object = Collection.Item(2)
```

This would get the second item in the collection and assign the object variable to that object. Because the **Item** method is the default method, you can also call the **Item** method as below:

```
Set Object = Collection(2)
```

Note that both the period and the actual name of the method are missing; only the argument to the method remains.

Instantiating an Object

To create an instance of a COM object in ASP, you can use a statement like the following:

```
Set Object = Server.CreateObject("ADODB.Connection")
```

There is only one argument to the **CreateObject** method of **Server** that is the **ProgId** (the program ID). The **ProgId** is assigned by every component vendor to uniquely identify the COM object. To create an instance of the COM object, you must know the **ProgId** of the COM object.

There is another way to get an instance of a COM object. You can have another COM object create the object and return the newly created object to you. This is how a collection works. You call the **Item** method of a collection, and a COM object is returned that represents the subset of the collection, which you index. Whenever a COM object is returned by another object, you must preface the statement with **Set**.

```
Set Object = Collection.Item(2)
```

Because **Server** is a COM object, both the examples above are much alike. They both return COM objects with a call to another COM object. The difference is that the **CreateObject** method of the **Server** object can return any COM object, and the **Item** method can only return COM objects that are stored in the collection. If you need to have a COM object to create another COM object, where did the **Server** object come from? ASP has a set of built-in COM objects that solve this chicken-or-the-egg problem.

Built-in COM Objects

There are six built-in COM objects in the ASP environment:

- Server
- Request
- Response
- ObjectContext
- Application
- Session

The difference between these COM objects and the others is that you do not need to create an instance of these objects to call them. Built-in objects are present in the page without instantiation. They adhere to all the other rules of COM and have their own methods and properties. You do not need to know their **ProgIds** because you don't have to call the **CreateObject** method to instantiate them.

ProgId

If one of the major ways to create a COM object is by using the **CreateObject** method, knowing the **ProgIds** of the objects you are creating is very important. So where are the **ProgIds** located? The component vendor should supply the component **ProgIds** as part of the documentation.

However, not all **ProgIds** are supplied, because a vendor doesn't always want you to create an instance of the object using the **CreateObject** method. Some objects inherit properties from the object that creates them, so if they are not created from calling a method in that object they are not initialized correctly. For example, creating an instance of an ADO **Field** object would not do you much good without going through the **RecordSet** object, because the ADO **Field** object would not contain any data unless you went through the **RecordSet** object.

Further On ...

In a booklet as brief as this, there is not much scope to include how to write your own COM objects. The idea is to use languages such as Visual Basic (not the scripting language VBScript) or Visual C++ (or any language that allows you to generate Win32 COM objects).

COM objects, as you must have noted, are typically compiled code. Hence they execute faster than ASP scripts. In a complex project with stringent efficiency requirements, you might want to code the Business Rules layer in a COM object rather than ASP scripts.

COPY & PASTE ASP SCRIPTS

Feedback Page

You can use this simple page to get feedback from your visitors. This will generate a form to ask your visitors the information that you want, and then it will mail the same to you. It assumes that you have the **Persits Software AspEmail Component** installed on the Server.

```
<HTML>
<HEAD>
<TITLE>Feedback</TITLE>
</HEAD>
<BODY>
<% If Request ("Action") <> "Send" Then %>
  <FORM METHOD="post" ACTION="<%= Request.ServerVariables
("PATH_INFO")%>">
    <P>Name: <BR>
      <INPUT TYPE="text" NAME="Name">
    <P>Email: <BR>
      <INPUT TYPE="text" NAME="Email">
    <P>Comments: <BR>
      <TEXTAREA NAME="Comments" ROWS=3 COLS=30>
      </TEXTAREA>
    <P><INPUT TYPE="submit" NAME="Action" VALUE="Send">
  </FORM>
<% Else
  Set Mail = Server.CreateObject("Persits.MailSender")
  Mail.Host = "smtp.your-isp.com"
  Mail.From = Request ("Email")
  Mail.FromName = Request ("Name")
  Mail.AddAddress "you@domain.com", "Your Name"
  Mail.Subject = "Web Feedback"
  Mail.Body = Request ("Comments")
  Mail.Send
  Response.Write "Your message was delivered."
End If %>
</BODY>
</HTML>
```

Tell a Friend about this site

```
<HTML>
```

```

<HEAD>
<TITLE>Tell a Friend</TITLE>
</HEAD>
<BODY>
<%
    URL = Request.QueryString("URL")
    If Len(URL) = 0 Then URL = "http://www.manastungare.com/"
    ' We need a Default URL

    If Len(Request("SenderEmail")) > 0 Then _
        Dim Mail, FriendEmail, I
        sBody = Request("SenderName") + " wants to tell you that
he found the webpage at " & URL & " very interesting. He would
like you to visit it. " & vbCrLf & "A personal message
follows: " & vbCrLf & Request("Message")

        I = 1
        Do While True
            FriendEmail = Request("FriendEmail" & I)
            If Len(FriendEmail) = 0 Then
                Exit Do
            Else
                Set Mail =
Server.CreateObject("Persits.MailSender")
                Mail.Host = "smtp.domainlx.com"
                Mail.From = "webmaster@yourdomain.com"
                Mail.FromName = "Webmaster at yourdomain.com"
                Mail.AddAddress FriendEmail
                Mail.Subject = "Recommended Page"
                Mail.Body = sBody
                Mail.Send

                End If
                I=I+1
            Loop

            Response.Write "<H1>Thank you for spreading the word</H1>"
            Response.Write "<A HREF="" & URL & "">Click here to return
to " & URL & "</A>"
            Else
                %>
<FORM METHOD="post" ACTION=""<%=
Request.ServerVariables("PATH_INFO") %>?URL=<%= URL %>">
<P>
    Recommended URL: <%= URL %><BR>
    Your Name: <INPUT TYPE="text" NAME="SenderName"
SIZE="25"><BR>

    Your Email: <INPUT TYPE="text" NAME="SenderEmail"
SIZE="25"><BR>

<P>Your friends' emails: <BR>

<OL>
    <LI><INPUT TYPE="text" NAME="FriendEmail1"><BR>
    <LI><INPUT TYPE="text" NAME="FriendEmail2"><BR>
    <LI><INPUT TYPE="text" NAME="FriendEmail3"><BR>
    <LI><INPUT TYPE="text" NAME="FriendEmail4"><BR>
    <LI><INPUT TYPE="text" NAME="FriendEmail5"><BR>
</OL>

<P>Message: <BR>
<TEXTAREA NAME="Message" ROWS="6" COLS="41"></TEXTAREA>

<P><INPUT TYPE="submit" VALUE="Recommend Page">
</FORM>

```

```
<% End If %>
```

Further examples

Further examples can be found on the web at my website,
<http://www.manastungare.com/asp>
or by request, at manas@manastungare.com

Since these are long and complex, they cannot be included in a booklet of this size.

THE ASP RESOURCE GUIDE

Editors

Once you start writing ASP code, you'll soon realize that Notepad makes it a tedious job. You could do with a more feature-packed editor for ASP scripts.

TextPad

www.textpad.com

TextPad is my personal favorite. It's a plain-jane ASCII text editor, with lots of features to simplify your job. For one, it provides Syntax Highlighting for a wide variety of source files, so whether you are writing ASP, or C, or C++, Java, HTML – you name it – TextPad supports it. Definitely worth more than a dekho.

Macromedia Dreamweaver UltraDev 1.0

<http://www.macromedia.com/>

UltraDev is a WYSIWYG HTML editor with enhancements for ASP. Although I do not use the ASP-wizards that come with UD, they're there for those want to. The best part is integrated site-management with superior ASP support (it won't mess up your code like FrontPage does.)

ASP Hosting

Once you're ready with your pages, you'll need a server to host your site. Since Active Server Pages may be hosted only on Windows NT servers (though the scene is fast changing with the advent of [Chili!ASP](#)), ASP hosting is usually costlier than regular hosting. (Maintaining NT costs more than Linux-Apache)

But, of course, there are freebies – here is a pick of some of the free ASP hosts around. They insert their ads on your pages, and that's how they earn.

DomainDLX

<http://www.domaindlx.com/>

- DomainDeluxe offer the following features to their members:
- Microsoft Windows 2000
- Internet Information Server 5.0
- 15 MB of Space
- Additional 25 MB Free Server Disk Space Upgrade!
- Unlimited Bandwidth
- Unlimited 24/7 Account Access via FTP
- 6MB Free Web Based E-mail
- Free Statistical Counter
- Free Intranet
- SSI (DHTML)
- Active Server Pages Support (ASP 3.0)
- AspEmail Component
- Free Database Connectivity: MS Access databases

Brinkster

<http://www.brinkster.com/>

They offer two hosting plans, General & Premium.

General

- Cost: FREE!
- 30 MBs of Web Space
- ASP Support
- MS Access DB Support
- Web Based File Manager
- ADO & FileSystemObject
- No Ads On Your Site

Premium Plan

- \$10.95/Setup \$10.95/Month
- All FREE Features PLUS
- FTP To Upload Files
- Your Own Domain Name
- 1 POP3 Email Address
- 2000 MB/Month Data Transfer
- 15+ 3rd Party Components

SoftCom Technologies

<http://www.softcomca.com/>

SoftCom offers four plans: Standard, Premium, Gold, and Platinum. The Standard plan is an excellent start for those hosting a site for the first time. It gives you all of the features you need to get your site up and running. With one mailing list, 50 MB of disk storage, and 20 e-mail accounts you have all the features to get your business on-line.

The Standard plan

For the monthly fee of \$9.95 you will receive:

- Free domain name registration / transfer
-

- Microsoft® FrontPage® 2000 Extensions
- 50 MB disk space
- Microsoft® Active Server Pages
- 20 e-mail accounts
- Microsoft® Visual InterDev Support
- Unlimited web traffic
- Your own cgi-bin
- Unlimited FTP updates and traffic
- Audio/Video streaming
- Unlimited e-mail forwarding
- Account management console
- 1 mailing list
- Monthly billing cycle with no minimum contract
- Web site statistics
- Access to raw web site log files
- E-mail autoresponders
- Redundant Internet connections
- Dedicated IP address
- Daily tape backups
- Free scripts
- UPS protection

After you begin to build your web presence, you can consider purchasing advanced services to enhance the functionality of your site. Compare the standard hosting plan with other plans.

The Premium plan - improve your Web presence.

For \$14.95 per month you will receive all the same features of the standard plan plus an additional mailing list, an ODBC connection for a Microsoft Access database, a secure web site connection using SoftCom's key, (SSL), 25 MB more storage space and 5 more e-mail accounts.

The Gold plan - as you expand so can your service.

2 mailing lists, 2 ODBC connections to Microsoft Access database, a secure website connection (SSL), directory password protection (for membership based services), an additional FTP user with upload capability, twice the amount of storage space and 10 more e-mail accounts.

The Platinum plan - increase your storage space as your business grows.

For \$44.95 per month you will receive all the features of the standard plan plus 3 mailing lists, 3 ODBC connections to a Microsoft Access database, a secure web site (SSL) connection, directory password protection (for membership based services), two additional FTP users with upload capability, 100 MB more of storage space and 20 more e-mail accounts.

Feature plus services

As you begin to build your online presence you can purchase advanced services to enhance the functionality of your site, as well as gaining more control of how you conduct business online.

COM Components for use with ASP

AspEmail

<http://www.aspemail.com/>

AspEmail 4.4 is a free active server component that enables your ASP application to send email messages via any external SMTP server. The component supports multiple file attachments, multiple recipients, Cc's, Bcc's, and Reply-To.

In addition to basic functionality available for free, AspEmail 4.4 offers a number of premium features that require a registration key after a 30-day evaluation period. These features are support for message queuing, embedded images, Quoted-Printable format and authentication. Regular file attachments are still free, of course.

JMail

<http://tech.dimac.net/>

JMail is another email component, much more powerful than AspEmail and offers a whole lot of functions including POP3 support, MailMerge with a simple template, PGP signing, etc.

Nonetheless, JMail is fairly complex to use, and exceeds the needs of most ASP sites. Highly recommended for large sites, not so useful for small ones. AspEmail will suffice just the same!

AspUpload

<http://www.aspupload.com/>

AspUpload is an Active Server component which enables an ASP application to accept, save and manipulate files uploaded with a browser. The files are uploaded via an HTML POST form with one or more `<INPUT TYPE=FILE>` tags. The `<FORM>` tag must contain the attribute `ENCTYPE="multipart/form-data"`.

Its rich set of features include

- Uploads to memory.
- Directory uploads.
- Image size extraction functionality.
- Compatibility with IIS 3, IIS 4, IIS 5 (Window 2000), and PWS.
- Ability to upload multiple files at once.
- Ability to change file attributes.

- Ability to save files in the database as blobs.
- Ability to export files from the database.
- Automatic generation of unique file names to prevent collisions with existing files.
- Ability to put a limit on the size of files being uploaded.
- Encryption support.
- Directory Listing with sorting.
- File copying, moving and deletion.
- Directory creation and deletion.

Again, this is not something everybody will need. Evaluate your budget & requirements before going in for any component.