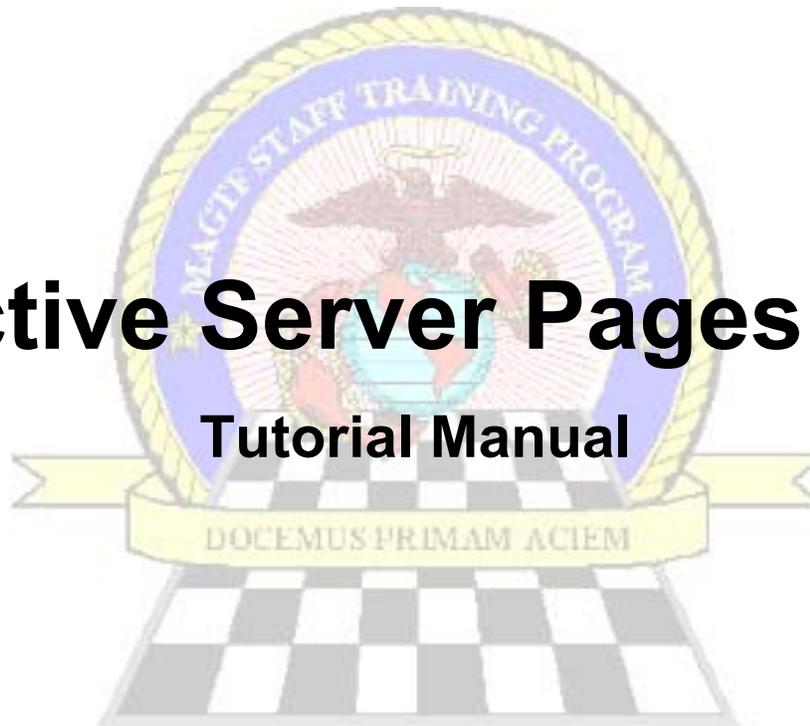




**United States Marine Corps  
MAGTF Staff Training Program  
2084 South Street  
Quantico, VA 22134-5001**

# **Active Server Pages 3.0 Tutorial Manual**



*"Training The First To Fight"*



MAGTF Staff Training Program  
April 2004



## Table of Content

<b>I. HTML BASICS.....</b>	<b>3</b>
A. BASIC STRUCTURE OF AN HTML PAGE .....	3
B. TAGS .....	4
C. TEXT FORMATTING .....	4
1. <i>Heading styles</i> <H> ... </H> .....	4
2. <i>Paragraph</i> <P> ... </P> .....	5
3. <i>Font</i> <FONT> ... </FONT> .....	6
a. <i>Bold</i> <b> ... </b> .....	6
b. <i>Italic</i> <i> ... </i> .....	6
c. <i>Underline</i> <u> ... </u> .....	6
d. <i>Font size</i> <SIZE> ... </SIZE> .....	6
4. <i>Break</i>   .....	7
D. TABLES .....	8
1. <i>Table Tag</i> <Table> ... </Table> .....	8
2. <i>Row</i> <TR> ... </TR> .....	9
a. <i>Row Alignment</i> .....	9
b. <i>Vertical Alignment</i> .....	10
c. <i>Background Color</i> .....	10
3. <i>Column</i> <TD> ... </TD> .....	11
a. <i>Column and Vertical Alignment</i> .....	12
b. <i>Background Color</i> .....	12
4. <i>Heading</i> <TH> ... </TH> .....	13
E. LISTS .....	13
1. <i>Unordered List</i> .....	13
a. <i>The &lt;UL&gt; Tag</i> .....	13
i. <i>Type Attribute</i> .....	14
ii. <i>Nesting Unordered Lists</i> .....	14
2. <i>Ordered List</i> .....	15
b. <i>The &lt;OL&gt; Tag</i> .....	15
i. <i>Type Attribute</i> .....	15
ii. <i>Nesting Ordered Lists</i> .....	16
iii. <i>Start Attribute</i> .....	16
F. HTML FORMS .....	17
1. <i>The &lt;FORM&gt; Tag</i> .....	18
a. <i>NAME attribute</i> .....	18
b. <i>ACTION attribute</i> .....	18
c. <i>METHOD attribute</i> .....	18
2. <i>&lt;INPUT TYPE = "TEXT"&gt; Tag</i> .....	19
a. <i>Type attribute</i> .....	19
b. <i>NAME attribute</i> .....	20
c. <i>SIZE attribute</i> .....	20
3. <i>&lt;INPUT TYPE = "PASSWORD"&gt; Tag</i> .....	21
4. <i>Drop Down Lists</i> .....	21
a. <i>The &lt;SELECT&gt; Tag</i> .....	21

i.	NAME attribute .....	22
ii.	SIZE attribute.....	22
iii.	MULTIPLE attribute .....	22
b.	The <OPTION> Tag.....	22
i.	VALUE attribute.....	22
5.	The <INPUT TYPE = "SUBMIT"> Tag.....	23
a.	TYPE attribute .....	23
b.	NAME attribute .....	23
c.	VALUE attribute.....	23
6.	The <INPUT TYPE="IMAGE"> Tag .....	24
7.	The <INPUT TYPE="RESET"> Tag .....	24
<b>II.</b>	<b>GETTING STARTED .....</b>	<b>26</b>
A.	WHAT DO WE NEED TO CREATE A WEB SITE? .....	26
B.	SETTING UP THE ENVIRONMENT .....	26
1.	IIS 5.0 overview .....	26
a.	Installation.....	26
C.	CREATING VIRTUAL DIRECTORIES.....	27
1.	Creating Virtual Directories With the Wizard.....	28
2.	Creating Virtual Directories Without the Wizard.....	31
<b>III.</b>	<b>ASP OVERVIEW .....</b>	<b>33</b>
A.	WHAT IS ACTIVE SERVER PAGES? .....	33
1.	Composition of Active Server Pages?.....	33
a.	ASP Delimiters .....	33
b.	Setting the ASP Scripting Language.....	33
c.	Variables, Operators, and Statements .....	33
d.	Active Server Components and Objects .....	33
2.	Running ASP Scripts.....	34
<b>IV.</b>	<b>LAB 1: YOUR FIRST CODE.....</b>	<b>37</b>
A.	HELLO WORLD! .....	37
1.	Creating an HTML Page .....	37
2.	Creating your First ASP page.....	38
<b>V.</b>	<b>PROGRAMMING AND SCRIPTING IN VBSCRIPT .....</b>	<b>41</b>
A.	DIFFERENCES BETWEEN VISUAL BASIC AND VISUAL BASIC SCRIPT .....	41
B.	VARIABLES AND DATA TYPES .....	41
1.	Data types .....	41
2.	Variables.....	42
a.	Declaring Variables .....	42
b.	Naming Restrictions.....	44
c.	Constants.....	44
C.	CONTROL STRUCTURES.....	44
1.	If...Then...Else .....	45
2.	Select Case.....	46
D.	LOOPING STRUCTURE .....	46

1.	<i>For...Next</i> .....	47
2.	<i>Do...Loop</i> .....	48
3.	<i>While...Wend</i> .....	48
<b>VI.</b>	<b>PROCESSING USER INPUT REQUEST (REQUEST OBJECT).....</b>	<b>50</b>
A.	THE REQUEST OBJECT .....	50
B.	HOW TO GET DATA FROM THE USER TO THE SERVER?.....	50
1.	<i>HTML Forms</i> .....	50
C.	PROCESSING RESULTS .....	52
<b>VII.</b>	<b>LAB 2: USING A FORM TO GATHER USER INPUT AND DISPLAYING RESULTS .....</b>	<b>53</b>
<b>VIII.</b>	<b>SESSION OBJECT.....</b>	<b>57</b>
A.	SESSION VARIABLES .....	57
1.	<i>Assigning Session Variables</i> .....	57
2.	<i>Clearing Session Variables</i> .....	57
<b>IX.</b>	<b>LAB 3: USE OF SESSION VARIABLES .....</b>	<b>59</b>
<b>X.</b>	<b>RESPONDING TO THE USER (RESPONSE OBJECT) .....</b>	<b>61</b>
A.	HOW TO SEND OUTPUT TO THE USER'S BROWSER.....	61
B.	THE RESPONSE OBJECT.....	61
C.	RESPONSE.WRITE .....	61
D.	RESPONSE.REDIRECT .....	63
<b>XI.</b>	<b>LAB 4: RESPONDING TO THE USER .....</b>	<b>65</b>
<b>XII.</b>	<b>DATABASE OVERVIEW .....</b>	<b>67</b>
A.	PLANNING A DATABASE .....	67
1.	<i>Database Structure</i> .....	67
2.	<i>Building the database in Microsoft Access</i> .....	68
a.	Creating a Database .....	68
b.	Creating a Table.....	70
<b>XIII.</b>	<b>INTRODUCTION TO STRUCTURED QUERY LANGUAGE (SQL)....</b>	<b>73</b>
A.	SQL STATEMENTS.....	73
1.	<i>The SELECT Statement</i> .....	73
a.	The WHERE Clause .....	74
b.	The ORDER BY Clause .....	74
<b>XIV.</b>	<b>ACCESSING A DATABASE.....</b>	<b>77</b>
A.	THE CONNECTION OBJECT .....	77
B.	INCLUDING ACTIVE X DATA OBJECTS (ADO) CONSTANTS .....	77
C.	CREATING AN ODBC CONNECTION .....	78
D.	ODBC DATA SOURCE NAME (DSN)-LESS CONNECTION .....	78
1.	<i>DSN vs DSN-less</i> .....	78
E.	THE RECORDSET OBJECT .....	79
1.	<i>The Beginning of File (BOF) Object</i> .....	80

2. <i>The End of File (EOF) Object</i> .....	80
F. CONNECTING TO A DATABASE .....	81
1. <i>Opening a Table</i> .....	81
2. <i>Selecting Records</i> .....	82
3. <i>Iterating thorough a Recordset</i> .....	83
4. <i>Checking for Matching Records</i> .....	84
5. <i>Closing Connections and Recordsets</i> .....	87
G. UPDATING A DATABASE .....	88
1. <i>Adding records</i> .....	88
2. <i>Updating records</i> .....	88
3. <i>Deleting Records</i> .....	89
<b>XV. LAB 5: DATABASE CONNECTION.....</b>	<b>91</b>
<b>XVI. PUTTING IT ALL TOGETHER.....</b>	<b>97</b>
A. LINKING PAGES .....	97
<b>XVII. LAB 6: ALPHA ROSTER (FINAL PRODUCT).....</b>	<b>99</b>
<b>APPENDIX A. FINAL PRODUCT DESCRIPTION.....</b>	<b>101</b>
<b>APPENDIX B. VBSCRIPT REFERENCE .....</b>	<b>107</b>
<b>APPENDIX C. HTML TAGS.....</b>	<b>119</b>
<b>APPENDIX D. ASP OBJECTS .....</b>	<b>125</b>
<b>APPENDIX E. HELPFUL WEBSITES.....</b>	<b>135</b>
<b>POINT OF CONTACTS .....</b>	<b>137</b>

## Table of Figures

Figure 1 Basic HTML Output.....	4
Figure 2 HTML headings.....	5
Figure 3 Text Formatting, Font.....	7
Figure 4 Colors in an HTML Table.....	11
Figure 5 HTML Table (2 Rows, 2 Columns).....	12
Figure 6 HTML Table with Headers and Caption.....	13
Figure 7 HTML Form Controls.....	17
Figure 8 Windows Components Add/Remove dialog window.....	26
Figure 9 Internet Information Manager window.....	27
Figure 10 Creating Virtual Directories with the Wizard.....	28
Figure 11 Naming your virtual directory using the wizard.....	29
Figure 12 Specifying the directory for the alias.....	30
Figure 13 Setting Access Permission.....	30
Figure 14 Creating a virtual directory without the wizard.....	31
Figure 15 Naming the Alias and assigning Access permissions without the wizard.....	32
Figure 16 ASP Composition.....	34
Figure 17 Running ASP Scripts.....	35
Figure 18 Hello World! Web page.....	37
Figure 19 Hello World! ASP page.....	39
Figure 20 HTML Form.....	51
Figure 21 Lab 2 Using a Form Example.....	54
Figure 22 Lab 2 Response to the user.....	55
Figure 23 Lab 3 Session Variables.....	60
Figure 24 Creating a new Microsoft Access Database.....	69
Figure 25 Naming and saving the new database in Access.....	69
Figure 26 Access Main Window.....	70
Figure 27 Table Design View.....	70
Figure 28 Adding Information into the Database.....	71
Figure 29 Selecting Records from a Database.....	82
Figure 30 Lab 5 Connecting to a Database and displaying results.....	95
Figure A- 1 Login Page.....	101
Figure A- 2 Alpha Roster.....	102
Figure A- 3 Detail Information.....	102
Figure A- 4 Change Information.....	103
Figure A- 5 Updated information.....	103
Figure A- 6 Search.....	104
Figure A- 7 Delete User.....	104
Figure A- 8 Add Kid.....	104
Figure A- 9 Search Results.....	105

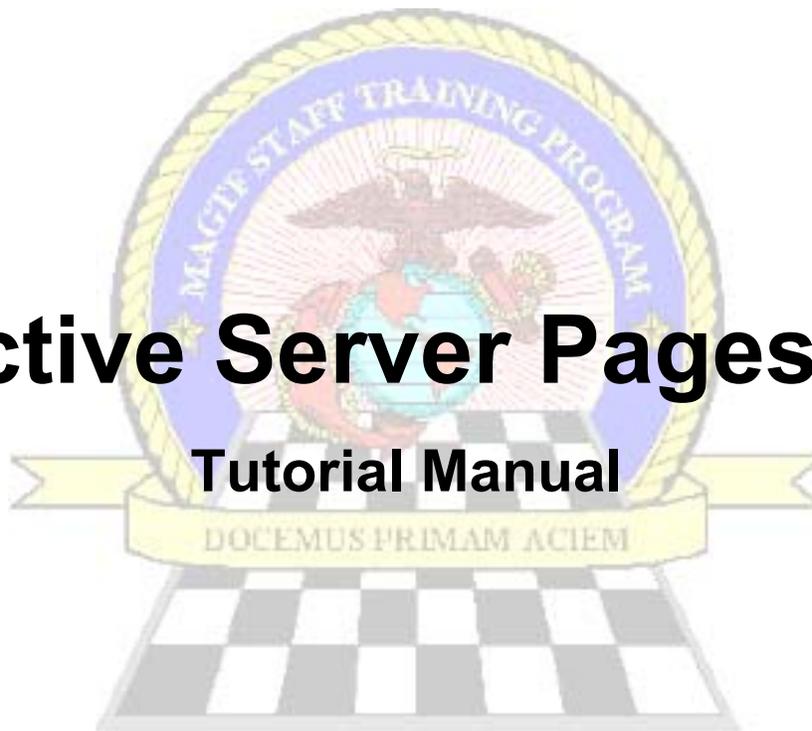


## List of Tables

Table 1 Names Table Structure.....	68
Table 2 Names Table Data.....	68



# Active Server Pages 3.0 Tutorial Manual



*"Training The First To Fight"*



## I. HTML Basics

### A. Basic Structure of an HTML page

HTML (Hypertext Markup Language) is the language used to create web documents. It is the primary language for formatting web pages. With HTML you describe what a page should look like, what types of fonts to use, what color text should be, where paragraph marks come and many more aspects of the document.

HTML defines the syntax and placement of special instructions that aren't displayed, but tell the browser how to display the document's contents. It is the job of the browser that requests the HTML file to format the page according to the various tags included in the HTML. It is also used to create links to other documents, either locally or over a network such as the Internet.

All HTML documents are created by using a set of tags. Tags have beginning and ending identifiers to communicate to the browser the beginning and ending text that is to be formatted by the tag in question. The tags have an opening and a closing tag. Each tag is enclosed with the "less than" (i.e. <) and "greater than" (i.e. >) sign.

Opening tag <>

Closing tag </>

HTML lets you create structured documents. The heading commands separate and categorize sections of your documents. It also has commands to format and display text, display images, accept input from users, and send information to a server for a back-end processing.

Each document has a *head* and a *body*, delimited by the <head> and <body> tags. The head is where you give your HTML document a title and where you indicate other parameters the browser may use when displaying the document. The body is where you put the actual contents of the HTML document. This includes the text for display and document control markers (tags) that advise the browser how to display the text.

An HTML document consists of text, which defines the content of the document, and tags, which define the structure and appearance of the document. The structure of an HTML document is simple, consisting of an outer <html> tag enclosing the document head and body:

```

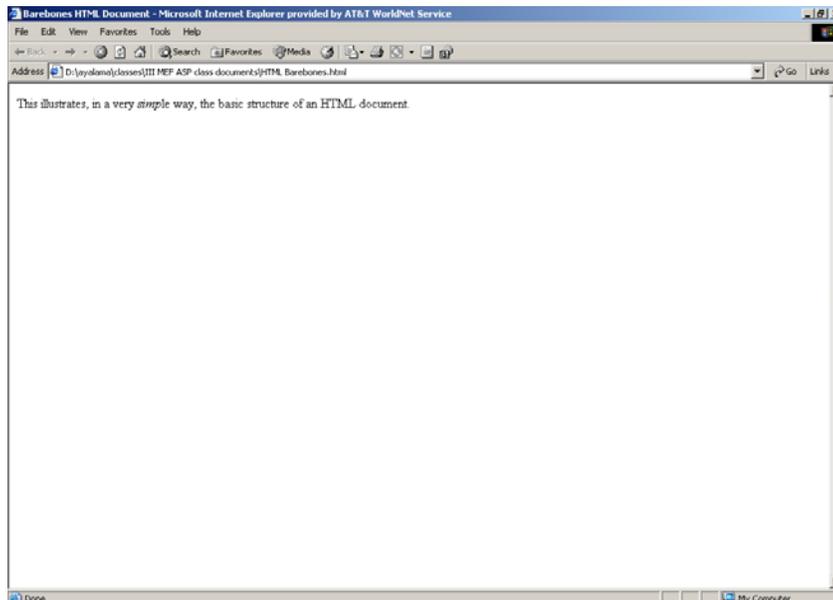
<HTML>
  <HEAD>
    <TITLE>Barebones HTML Document</TITLE>
  </HEAD>
  <BODY>
    This illustrates, in a very <i>simple</i> way,
    the basic structure of an HTML document.
  </BODY>
</HTML>

```

← Opening tag

← Closing tag

Notice the beginning of the `<html>` tag is closed at the end of the document by its counterpart closing tag `</html>`.



**Figure 1 Basic HTML Output**

## **B. TAGS**

The tag marks a portion of text for special treatment by the browser. That treatment may be anything from make the next character bold, to treat the following line as code.

For the most part, HTML document tags are simple to understand and use, since they are made up of common words, abbreviations, and notations. The HTML standard and its various extensions define how and where you place tags within a document.

Every HTML tag consists of a tag *name*, sometimes followed by an optional list of tag *attributes*, all placed between opening and closing brackets (`<` and `>`). The simplest tag is nothing more than a name appropriately enclosed in brackets, such as `<head>`. More complicated tags contain one or more attributes, which specify or modify the behavior of the tag. Tags are not case-sensitive. There's no difference in effect between `<head>`, `<Head>`, `<HEAD>`, or even `<HeaD>`; they are all equivalent.

## **C. Text Formatting**

### **1. Heading styles `<H>...</H>`**

HTML recognizes six levels of heading, written as `<h1>` through `<h6>`. The number signifies the position of the heading content in a hierarchy, where the smaller number mean that the content is higher in the hierarchy.

`<HTML>`

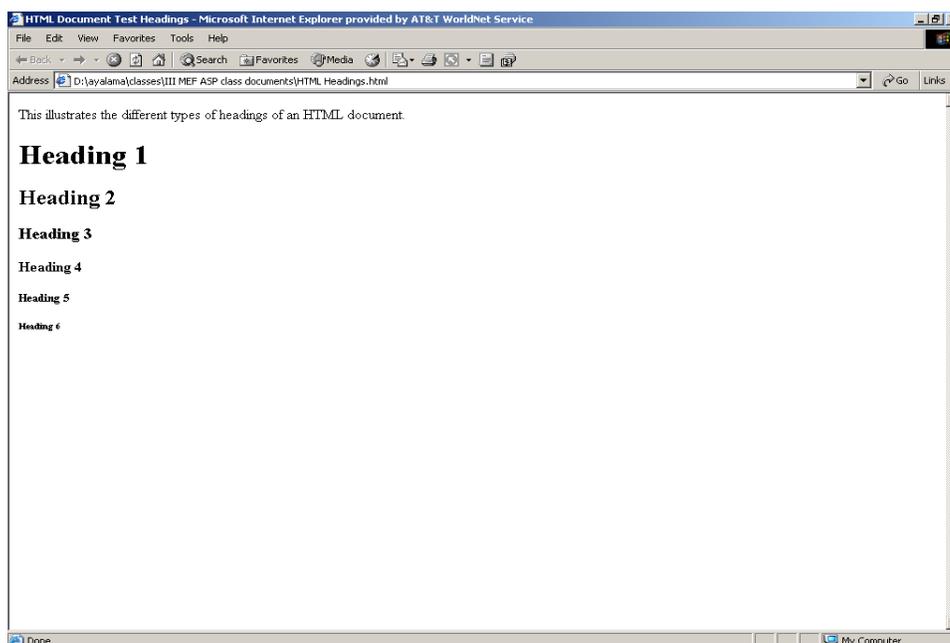
```

<HEAD>
  <TITLE>HTML Document Test Headings</TITLE>
</HEAD>
<BODY>
  This illustrates the different types of headings of an
  HTML document.

  <H1> Heading 1 </H1>
  <H2> Heading 2 </H2>
  <H3> Heading 3 </H3>
  <H4> Heading 4 </H4>
  <H5> Heading 5 </H5>
  <H6> Heading 6 </H6>

</BODY>
</HTML>

```



**Figure 2 HTML headings**

## 2. Paragraph <P>...</P>

Paragraphs allow you to add text to a document in such a way that it will automatically adjust the end of a line to suit the window size of the browser in which it is being displayed. Each line of text will stretch the entire length of the windows. Paragraph tags can contain child tags, such as text formatting commands and a table.

```

<HTML>
<HEAD>
  <TITLE>HTML Document Paragraph Headings</TITLE>

```

```
</HEAD>
  <BODY>
    This illustrates the paragraph tag of an
    HTML document.

    <p> paragraph 1 </p>
    <p> paragraph 2 </p>
    <p> paragraph 3 </p>

  </BODY>
</HTML>
```

### 3. Font <FONT>...</FONT>

Used to modify the characteristics of the characters.

a. Bold <b>...</b>

The <b> tag explicitly boldfaces a character or segment of text that is enclosed between it and its corresponding </b> end tag.

b. Italic <i>...</i>

The <i> tag renders the enclosed text between it and </i> end tag into italic or oblique typeface.

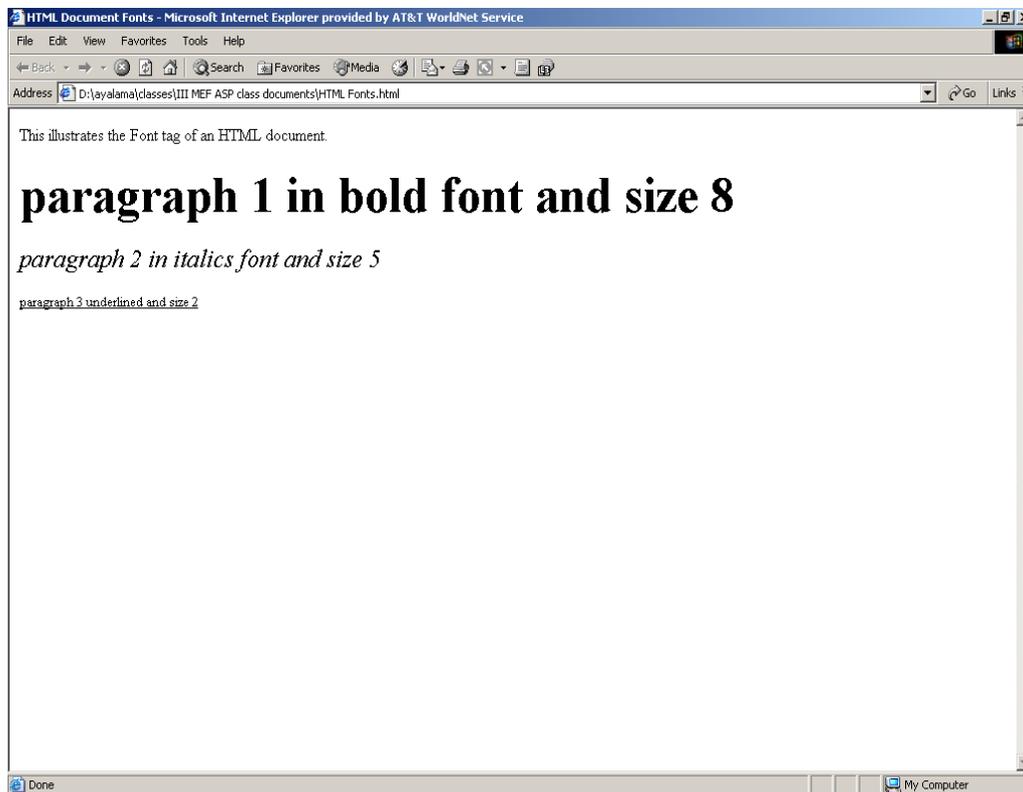
c. Underline <u>...</u>

This tag tells the browser to underline the text contained between the <u> and the corresponding </u> tag. The underlining technique is simplistic, drawing the line under spaces and punctuation as well as the text.

d. Font size <SIZE>...</SIZE>

Changes the tag text font size to the specified size. It is a number.

*size=value*



**Figure 3 Text Formatting, Font**

#### **4. Break <BR>**

The <br> tag interrupts the normal line filling and word wrapping of paragraphs within an HTML document. It has no ending tag, but simply marks the point in the flow where a new line should begin.

An example that brings all the previously discussed tags together:

```
<HTML>
<HEAD>
  <TITLE>HTML Document Fonts</TITLE>
</HEAD>
<BODY>
  This illustrates the Font tag of an HTML
  document.

  <FONT SIZE = 8>
```

```
<p> <b> paragraph 1 in bold font and size 8 </b>
</p>
</FONT>
```

```
<FONT SIZE = 5 >
<p> <i> paragraph 2 in italics font and size 5
</i> </p>
</FONT>
```

```
<FONT SIZE = 2 >
<p> <u> paragraph 3 underlined and size 2 </u>
</p>
</FONT>
```

After a long sentence you can simply add a break to specify `<br>` where the next line should begin.

```
</BODY>
</HTML>
```

## D. Tables

Tables are useful devices for presentation of information in a meaningful form. At the same time, tables can be used to structure the layout of a Web page regardless of its content. You can produce a variety of appearances simply by designing different table structures into which your page information will fit. In short, tables are important presentation devices, and you need to know as much as you can about how to use them. There are three basic steps to defining a simple table. First define the table itself (appearance of table, cell spacing, backgrounds, etc.) then define a row of the table and finally defining the columns for a table.

The rows, columns, and cell intersections of a table are defined with three basic tags. `<TABLE>...</TABLE>` tags surround the entire table description; `<TR>...</TR>` (table row) tags defined the rows of the table; and `<TD>...</TD>` (table data) tags define the cells, or columns, that appear in each row.

### 1. Table Tag `<Table>...</Table>`

To define a Table, we will use two tags. The opening table tag `<TABLE>` and the closing table tag `</TABLE>`.

```
<TABLE>...</TABLE>
ALIGN = "LEFT | CENTER | RIGHT"
HSPACE = "n"
VSPACE = "n" ← A number
BORDER = "n"
BORDERCOLOR = "color name | #rrggbb"
BORDERCOLORDARK = "color name | #rrggbb"
BORDERCOLORLIGHT = "color name | #rrggbb"
BGCOLOR = "color name | #rrggbb"
```

```

BACKGROUND = "URL"
CELLPADDING = "n"
CELLSPACING = "n"
WIDTH = "n | n%" ← Percent or number
<table>

<TR>...</TR>
ALIGN = "LEFT | CENTER | RIGHT"
VALIGN = "TOP | MIDDLE | BOTTOM"
BGCOLOR = "color name | #rrggbb"

<TD>...</TD>
<TH>...</TH>
ALIGN = "LEFT | CENTER | RIGHT"
VALIGN = "TOP | MIDDLE | BOTTOM"
BGCOLOR = "color name | #rrggbb"
BACKGROUND = "URL"
COLSPAN = "n"
ROWSPAN = "n"
NOWRAP

<CAPTION>...</CAPTION>
ALIGN = "TOP | BOTTOM"

```

## 2. Row <TR>...</TR>

A table is made up of rows and columns. Before a column can be defined a row must be defined. Rows will take their default attributes from what is defined in the <table> tag, however these attributes may be overridden when defining a table row (i.e. <tr> tag) - The following is an example of defining table row with a single row defined:

```

<table width="43%" height="70%" border="1" cellspacing="15"
cellpadding="8" >
  <tr>
  </tr>
</table>

```

← Row definition

The following attributes may be defined for a Table Row:

- Row Alignment
- Vertical Alignment
- Background Color

### a. Row Alignment

The values for the Row Alignment are:

- Left
- Center
- Right

The following is an example of defining the Table Row for each of these values:

```
<tr align=left>
<tr align=center>
<tr align=right>
```

#### b. Vertical Alignment

The vertical alignment defines where the row is to line up vertically. Typically you would probably want the row to line up at the top, however there may be times which you want it to align vertically to create different effects. The following are examples of vertical alignment:

```
<tr align=center valign=top>
<tr align=center valign=center>
<tr align=center valign=bottom>
<tr align=center valign=baseline>
```

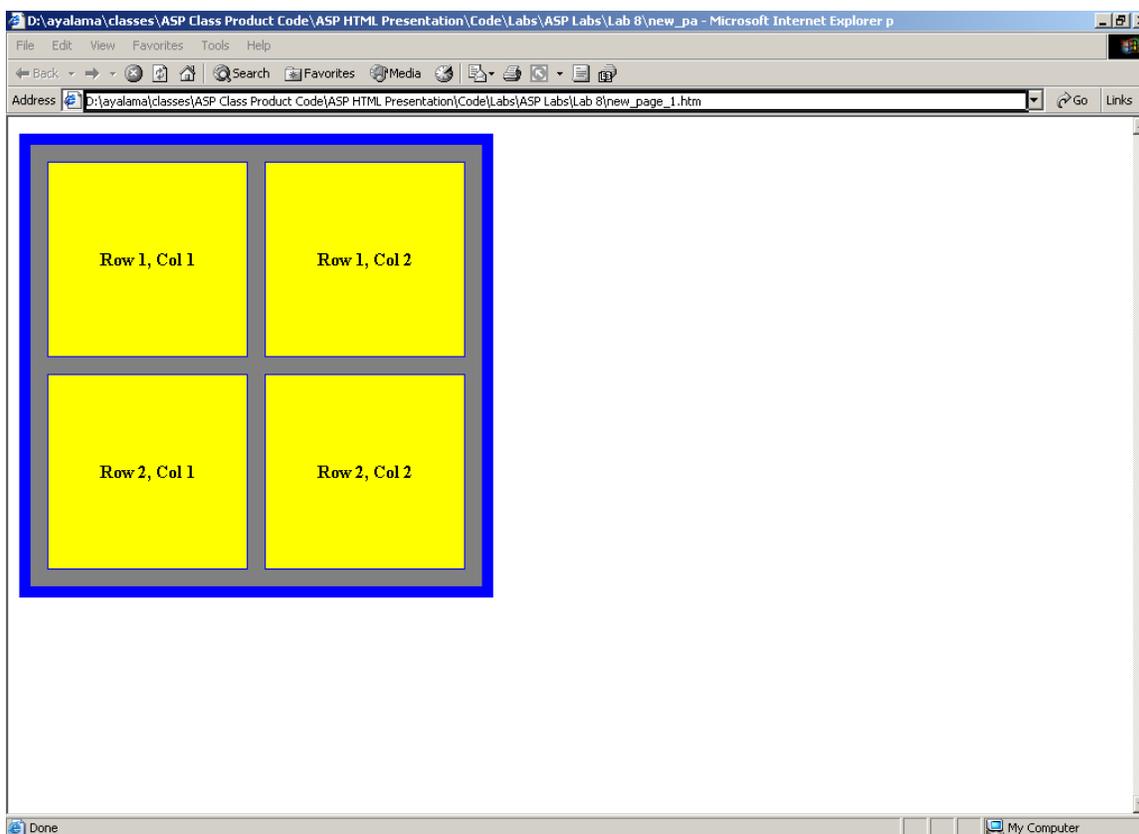
#### c. Background Color

Finally, you can define the background color of the Row. This following is an example of adding in the background color for the row to be displayed in Burgundy Color and the Column in Green:

```
<HTML>
<BODY>

<table bgcolor="gray" width="43%" height="70%" border="10"
  cellspacing="15" cellpadding="8" bordercolor="blue" ><tr
align=center valign=top bgcolor="yellow"><td
  valign="middle" align="center" > <b> Row 1, Col 1 <b>
</td><td valign="middle" align="center"><b> Row 1, Col 2
<b> </td></tr>
<tr align=center valign=top bgcolor="yellow"><td
  valign="middle" align="center" > <b> Row 2, Col 1 <b>
</td><td valign="middle" align="center" > <b> Row 2, Col
2 <b> </td></tr></table>

<BODY>
</HTML>
```



**Figure 4 Colors in an HTML Table**

### 3. Column <TD>...</TD>

A Row is divided into Columns (or cells) - Each Cell has many attributes that define what the appearance of that cell is. Each column (or cell) is defined by the <td> and </td> tags. Below we have placed these two tags between the table row tags (i.e. <tr> and </tr> tags).

```
<HTML>
```

```
<BODY>
```

```
  <table width="43%" height="70%" border="2"
    cellspacing="3" cellpadding="2" >
```

```
  <tr align=center valign=center >
```

```
    <td> Row 1, Col 1 </td>
```

```
    <td> Row 1, Col 2 </td>
```

```
  </tr>
```

```
  <tr align=center valign=center >
```

```
    <td> Row 2, Col 1 </td>
```

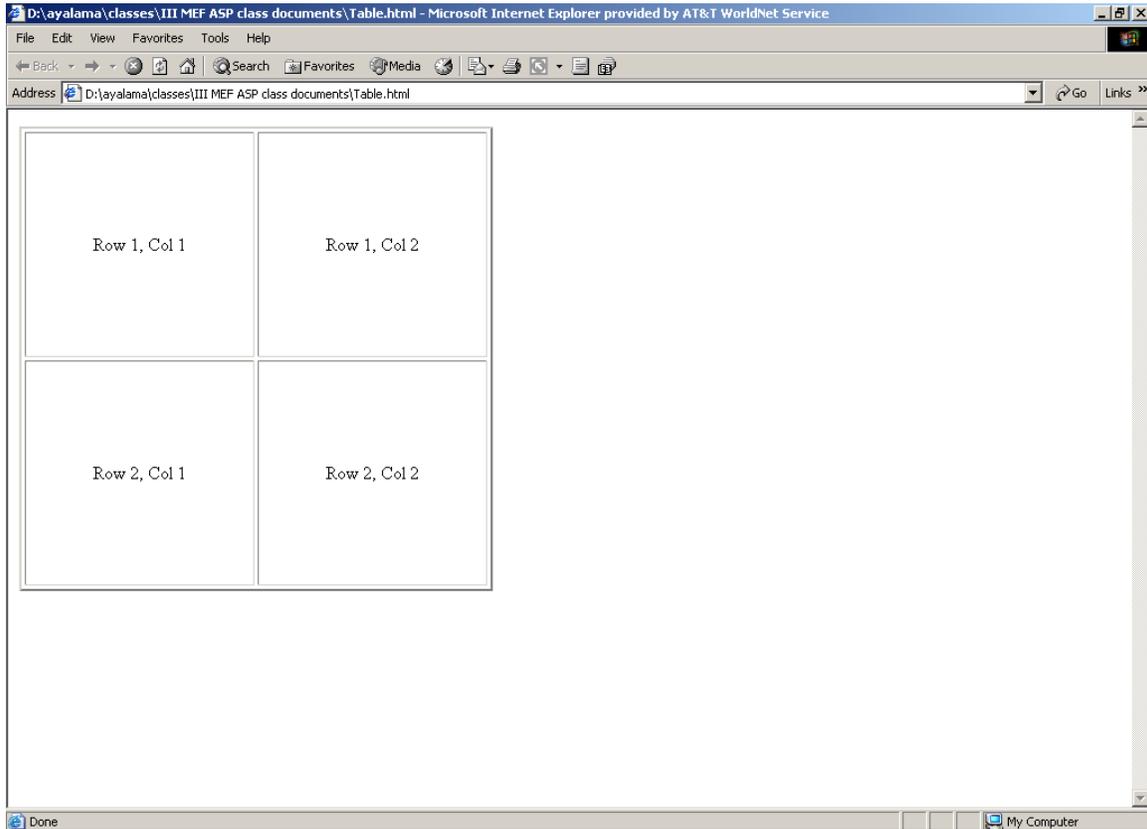
```
    <td> Row 2, Col 2 </td>
```

```
  </tr>
```

```
</table>
```

Column definition

```
<BODY>
</HTML>
```



**Figure 5 HTML Table (2 Rows, 2 Columns)**

a. Column and Vertical Alignment

The same principle as the row applies to the column in terms of alignment.

```
<td valign="middle" align="center"> </td>
<td valign="middle" align="center"> </td>
<td valign="middle" align="center"> </td>
```

b. Background Color

Background color for a column is done in the same way as for the row.

```
<tr align=center valign=top bgcolor="blue">
<td valign="middle" align="center" bgcolor="#FFFFCC"></td>
<td valign="middle" align="center" bgcolor="#FFFFCC"></td>
<td valign="middle" align="center" bgcolor="#FFFFCC"></td>
</tr>
```

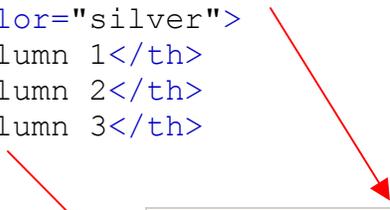
Row color, either simple name or hex number.

Column color

#### 4. Heading <TH>...</TH>

Column headings can be supplied with <TH>...</TH> tags appearing within the first row of the table. These tags center and bold the enclosed text over the associated column. In the following example a background color is specified for the row of column headings.

```
<table border="1">
<caption><b>This is My Table</b></caption>
<tr bgcolor="silver">
  <th>Column 1</th>
  <th>Column 2</th>
  <th>Column 3</th>
</tr>
```



This is My Table		
Column 1	Column 2	Column 3

Figure 6 HTML Table with Headers and Caption

### E. Lists

HTML also contains tags to format bulleted and numbered lists.

Lists and the items within them are block-level elements, meaning that line spaces will automatically be added before and after them. Extra space may be added above and below the entire list element but, in general, if you want to add space between individual list items, you need to insert a <p> tag between them (although, technically, that is not good HTML form).

#### 1. Unordered List

An unordered list is a series of items set off from surrounding text by a single blank line and are preceded by bullet characters. The list is single spaced and indented from the left margin. An example unordered list is shown below.

- List Item 1
- List Item 2
- List Item 3

##### a. The <UL> Tag

An unordered list is created with <UL> tags wherein each item in the list is identified by an <LI> (list item) tag. The general format for an unordered list is shown below.

```
<UL>
  TYPE = "DISC | CIRCLE | SQUARE"
```

```
<LI>List item 1</LI>
<LI>List item 2</LI>
...
</UL>
```

Items in the listing are single spaced. You can code `<BR>` tags between the items to increase the line spacing. List items containing text paragraphs are word wrapped and indented inside the bullet character.

*i. Type Attribute*

The *TYPE* attribute can be coded within the opening `<UL>` tag in order to specify a different type of bullet character. There are three types of attributes:

- circle (default)
- square
- disc

```
<ul type="circle">
  • List Item 1
  • List Item 2
```

```
<ul type="square">
  ▪ List Item 1
  ▪ List Item 2
```

```
<ul type="disk">
  ○ List Item 1
  ○ List Item 2
```

*ii. Nesting Unordered Lists*

Unordered lists can be nested within each other. For example, a bulleted list appearing inside a bulleted list is produced by the following code,

```
<ul type="square">
  <li>List Item 1</li>
  <li>List Item 2</li>
  <ul type="circle">
    <li>List Item 2a</li>
    <li>List Item 2b</li>
  </ul>
</ul>
```

which is rendered in the browser as

- List Item 1
- List Item 2
  - List Item 2a
  - List Item 2b

## 2. Ordered List

An ordered list is a series of numbered items set off from surrounding text by a single blank line. The list is single spaced and indented from the left margin. An example ordered list is shown below.

1. List Item 1
2. List Item 2
3. List Item 3

### b. The <OL> Tag

An ordered list is created with <OL> tags wherein each item in the list is identified by an <LI> (list item) tag. The general format for an ordered list is shown below.

```
<OL TYPE = "1 | A | a | I | i" >

    <LI>List item 1</LI>
    <LI>List item 2</LI>
    ...
</OL>
```

Items in the listing are single spaced. If you want additional spacing between items, you can code <BR> tags between them. List items containing text paragraphs are word wrapped and indented inside the numeral character.

### i. Type Attribute

You can code the TYPE attribute within the opening <OL> tag in order to specify one of five different numbering characters. The attribute value can be

- "1" for Arabic numerals (the default)
- "A" for upper-case letters
- "a" for lower-case letters
- "I" for upper-case Roman numeral
- "i" for lower-case Roman numerals

For example, the tag <ol type="A"> produces the following list of alphabetically ordered items:

- A. List Item 1
- B. List Item 2
- C. List Item 3

```
<ol type="1">
    1. List Item 1
    2. List Item 2
```

```
<ol type="A">
```

- A. List Item 1
- B. List Item 2

```
<ol type="a">  
  a. List Item 1  
  b. List Item 2
```

```
<ol type="I">  
  I. List Item 1  
  II. List Item 2
```

```
<ol type="i">  
  i. List Item 1  
  ii. List Item 2
```

### *ii. Nesting Ordered Lists*

Ordered lists can be nested within each other, each list having its own numbering scheme. In the following example the outer list is numbered with upper-case Roman numerals and the inner list is numbered with lower-case Roman numerals.

```
<ol type="I">  
  <li>List Item 1</li>  
  <li>List Item 2</li>  
    <ol type="i">  
      <li>List Item 2a</li>  
      <li>List Item 2b</li>  
    </ol>  
  <li>List Item 3</li>  
</ol>
```

This code is rendered in the browser as

- I. List Item 1
- II. List Item 2
  - i. List Item 2a
  - ii. List Item 2b
- III. List Item 3

Note that when lists are contained within other lists that no blank lines surround the interior list as they do when the list appears within the normal flow of text.

### *iii. Start Attribute*

When using numerals in an ordered list, you have a choice of the beginning number for the list. For example, to start an ordered listing with the number 5, code the opening tag with the `START="5"` attribute and the items will be numbered in sequence accordingly:

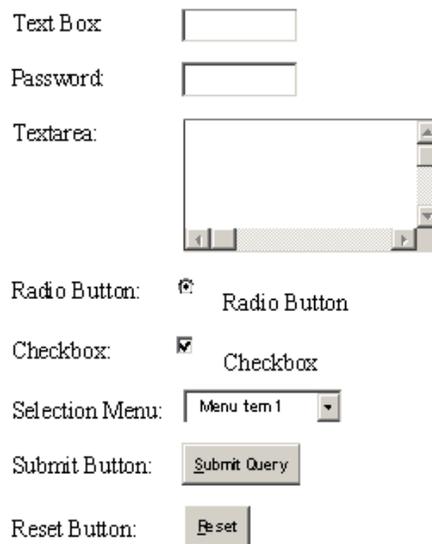
5. List Item E
6. List Item F
7. List Item G
8. List Item H
9. List Item I

## F. HTML Forms

HTML forms provide a way for users to interact with Web pages. A form is, basically, a data capture device. It presents the user with one or more data input or selection controls, or fields, through which the user submits information to a Web page. On the basis of this submitted information, the page can react to the user. This response can vary depending on the purpose of the form. The submitted data may be used:

- to direct visitors to a different page, much like what happens when clicking a link
- to present visitors with personalized pages containing information and links pertinent to their interests or preferences
- to trigger a complex search process to locate information or services about which the user is interested
- to generate automated email responses

Forms gather information from users by displaying special form fields that permit the user to enter data or make selections. The varieties of standard controls that can be coded on a Web form are shown below.



**Figure 7 HTML Form Controls**

An HTML form can appear anywhere within the body of a Web page. In fact, the entire page can be a form or you can define more than one form per page. The manner in which you do this is dependent upon the purposes of the forms.

## 1. The <FORM> Tag

Irrespective of the number of forms or their locations on the page, each must be surrounded by a <FORM> tag, the general format for which is shown below.

<FORM>

```
NAME = "form name"  
ACTION = "URL"  
METHOD = "GET | POST"
```

</FORM>

All fields that are part of a form must be enclosed within <FORM> tags. These tags can appear anywhere on the page as long as they enclose all the form fields. If a page contains a single form, you can code the opening tag immediately following the <BODY> tag and the closing tag immediately preceding the </BODY> tag to encompass the entire page as a form. Then, form fields can appear anywhere on the page and be part of the form. If a page contains more than one form, then <FORM> tags need to enclose only those fields comprising each form.

### a. NAME attribute

The *NAME* attribute assigns a name to the form. Naming a form is necessary if you need to refer to the form in browser-based scripts. This is required, for example, if your page includes scripts written in the VBScript language to process form information or to edit or verify user-entered information prior to submitting it to the server for processing. You can use any name of your choosing for a form; it is best if the name does not include embedded blank spaces.

### b. ACTION attribute

User information entered into a form is made available to a page containing a script to process that information in some fashion. Form information can be submitted to the page containing the form, or it can be sent to a different page, depending on where the processing script is coded.

The *ACTION* attribute identifies the page to which form information is submitted. If the page is in the same directory as the form page, then the URL can simply be the name of the page; if the page is at a remote location, a complete URL is coded. If the *ACTION* attribute is not coded in the <FORM> tag, the information is made available to the current page when it is reloaded following form submission.

### c. METHOD attribute

The *METHOD* attribute specifies the manner in which form information will be submitted. The two possible values are *GET* and *POST*. The *POST* is the default. When the *GET* method is used, information from the form is submitted appended to the *ACTION* URL; when the *POST* method is used, the information is transmitted as a separate data stream. We'll take a look at these methods in more detail later.

All that is required at present is to code the basic `<FORM>` tag to enclose its fields. Within these tags you can also include any other types of tags to structure and format your page.

```
<html>

<head>
  <title>A Form Page</title>
</head>

<body>

  <form name="MyForm" action="AspPage.asp"
        method="POST">
    ...
  </form>

</body>

</html>
```

## 2. `<INPUT TYPE = "TEXT">` Tag

The most commonly encountered type of form field is the text box. This control presents a standard text entry box into which information can be typed. A text field is created using the `<INPUT>` tag in the following format.

```
<INPUT>

TYPE = "TEXT | PASSWORD"
NAME = "field name"
SIZE = "n". Determines the textbox in characters.
        Default is 20 characters.
MAXLENGTH = "n". Determines the maximum number of
        characters that the field will
        accept.
VALUE = "text string". Will display its contents
        as the default value.
```

### a. Type attribute

The `<INPUT>` tag includes the `TYPE="TEXT"` attribute to define this as a text entry field. If no type attribute is specified, the default field type is `TEXT`, although you probably will want to include this attribute to help document your code. Also, you will likely need to include a label accompanying the field to prompt users about the type of information being solicited.

```
<form>
  Last Name: <input type="text">
```

```
</form>
```

Last Name:   A text box

#### b. NAME attribute

You will nearly always need to name your text fields, as you will any other form field. As mentioned above, field naming is important because it provides identification for the field, which, in turn, identifies the text value entered into the field by the user. Any browser or server scripts written to process this information does so through these field names.

The name of the field can be of your choosing. It should, however, be representative of the field contents. In any case, you should not use names with embedded blank spaces. Although this practice is syntactically correct so long as the name is enclosed in quotation marks, it can cause problems in referencing the names in processing scripts.

```
<form>
  Last Name: <input type="text" name="LastName">
</form>
```

The name of the text box is LastName. The Request object will be able to access the information through this name.

#### c. SIZE attribute

Unless you specify a **SIZE** attribute for a text field, it is displayed at its default size, which is large enough for approximately 20 typed characters. In most cases you will want to specify a size that is suggestive of the number of characters expected to be entered. For example, the three text fields below are sized at 15 (City), 2 (State), and 5 (Zip code) characters, respectively. Note also that the labels and fields appears within a table structure to help align them.

```
...
<tr>
  <td>City: </td>
  <td><input type="text" name="City" size="15"></td>
<tr>
<tr>
  <td>State: </td>
  <td><input type="text" name="State" size="2"></td>
<tr>
  <td>Zip: </td>
  <td><input type="text" name="Zip" size="5"></td>
</tr>
...
```

City:

State:   
 Zip:

### 3. <INPUT TYPE = "PASSWORD"> Tag

An input field that is similar in function to a text field is the password field. All the attributes associated with the text field are applicable to the password type. When coding `TYPE="PASSWORD"` a text box is displayed; however, instead of the typed characters being echoed in the box, a line of asterisks (\*), or bullets, are echoed. This practice is valuable to keep passwords private to the persons entering them. Password field specifications should include the NAME, SIZE, and MAXLENGTH attributes, just like a text field.

```
<form>
  Password: <input type="password" name="Password">
</form>
```

Password:

### 4. Drop Down Lists

#### a. The <SELECT> Tag

A *drop-down list*, or *selection menu* presents items that are chosen from a drop-down list. By clicking on the down-arrow to the right of the menu the list is exposed. Then one or more items can be chosen by clicking the entry. The menu of choices is created with the `<SELECT>` tag. Inside this tag are `<OPTION>` tags representing the items in the menu. The general formats for the `<SELECT>` and `<OPTION>` tags are shown below.

```
<SELECT>
  NAME = "field name"
  SIZE = "n"  This sets the number of visible
             choices
  MULTIPLE:  The presence of this attribute
             signifies that the user can make
             multiple selections.  By default only
             one selection is allowed

  <OPTION> Label
             VALUE = "text string"
             SELECTED
</OPTION>
  ...
</SELECT>
```

i. *NAME attribute*

As with all form elements, the selection menu must be assigned a name with the *NAME* attribute. The menu name becomes associated with the value of the item or items selected.

ii. *SIZE attribute*

The *SIZE* attribute indicates the number of items that are visible at a time in the menu. By default, the menu displays the first item in the list when the form is loaded. If a size is specified, the menu displays that number of items and, if necessary, a scroll bar for accessing them.

iii. *MULTIPLE attribute*

One or more items can be chosen from the menu. The default is one; however, with the *MULTIPLE* attribute, more than one item can be chosen. Multiple items are chosen using the Shift-Click or Ctrl-Click method.

The `<SELECT>` tag for the following menu has a *SIZE* attribute of 4 to display four items at a time and the *MULTIPLE* attribute to permit multiple choices. If multiple items are chosen, the name of this field is associated with that collection of values, and any scripts that process the field need to take this into account.

```
<form>
  Choose your favorite color:<br>
  <select name="Color" size="4" multiple>
    <OPTION> Red </OPTION>
    <OPTION> Green </OPTION>
    <OPTION> Blue </OPTION>
    <OPTION> Yellow </OPTION>
    <OPTION> White </OPTION>
  </select>
</form>
```

Choose your favorite color:



b. The `<OPTION>` Tag

An item is defined for a selection menu with an `<OPTION>` tag. There are as many tags as there are items in the menu. The label for the option is entered following the tag. This is the text string that is visible in the menu and becomes the default value for the selection.

i. *VALUE attribute*

The value associated with a menu option normally is given by the text label following the `<OPTION>` tag. However, the `VALUE` attribute can be coded to supply a different value from the label. You might choose to do this, for example, when the labels are extended text strings but you need only to collect abbreviated codes for the values.

```
<form>
  How do you like to travel?<br>
  <select name="Mode">
    <option value="1">Airplane</option>
    <option value="2">Car</option>
    <option value="3">Bus</option>
    <option value="4">Ship</option>
  </select>
</form>
```

How do you like to travel?

Notice the size attribute here is not stated. Default is size one, providing you a classical drop-down list.

## 5. The `<INPUT TYPE = "SUBMIT">` Tag

All forms must include at least one "submit" button to submit the form information for processing. This button is defined with an `<INPUT TYPE = "SUBMIT">` tag and can appear anywhere on the form. The default appearance of the button with its "Submit Query" label is shown below.

The general format for the `<INPUT>` tag to define a submit button is given below.

```
<INPUT TYPE="SUBMIT">
  NAME = "field name"
  VALUE = "text string"
```

### a. TYPE attribute

The attribute type must be `SUBMIT`. This value specifies that the control is a form submission button and differentiates it from other `<INPUT>` controls.

### b. NAME attribute

Submit buttons need to be named. This `NAME` attribute is associated with the value of the button, and together these become part of the names and values that are submitted with the form.

### c. VALUE attribute

The `VALUE` attribute provides two types of identification for the button. On the one hand, the value is associated with the name; on the other, this value is used as

the label for the button. If a *VALUE* attribute is not assigned, the button is labeled "Submit Query," but you can assign any text string you wish to provide helpful identification for the user.

```
<form>
...
  <input type="submit" name="SubmitButton"
    value="Submit">
...
</form>
```

## 6. The <INPUT TYPE="IMAGE"> Tag

An alternative to use of the standard submit button is to use a graphic image to trigger form submission. For example, the following "Go" button functions identically to a submit button.



```
<form>
...
  <input type="image" src="gobutton.gif" border="0"
    name="SubmitButton"
    alt="Click to Submit">
</form>
```

The <INPUT> tag uses *TYPE="IMAGE"* to identify this as a graphic form submission control. The particular image that is used is given by the URL in the SRC attribute. Other attributes of the <IMG> tag can be coded, including *BORDER="0"* to remove the blue border from around the image and the ALT attribute for an alternative text description of the image. As with standard submit buttons you will need to assign the image button a name with the *NAME* attribute.

## 7. The <INPUT TYPE="RESET"> Tag

A reset button can be defined to permit users to clear all information from a form. Its default appearance is shown below.



This button is created by coding an <INPUT TYPE="RESET"> tag. You can name the button and can replace the default "Reset" label by coding the *VALUE* attribute. The

action of the button is to automatically reset the form, clearing all text input areas and resetting radio buttons, checkboxes, and selection menus back to their defaults.

```
<form>
  ...
  <input type="reset" value="Clear the Form">
  ...
</form>
```

## II. Getting Started

### A. What do we need to create a web site?

Before you can write ASP code, you have to have a place to run it. ASP files do not run in your browser like HTML files do. ASP code is evaluated on the server before you ever see it in your Web browser. You need to have Internet Information Server (IIS) installed in your computer. IIS version 5.0 comes with Windows 2000.

To create an ASP file you could use Microsoft FrontPage, Microsoft InterDev, or even Notepad (if you are really brave!).

### B. Setting up the environment

#### 1. IIS 5.0 overview

IIS is not installed with Windows 2000 Professional if you accepted the default settings during the installation process. If you change the installation settings, you can add the Web services as you install Windows 2000 professional. You can also add IIS once you have got Windows installed.

##### a. Installation

If you are installing IIS after completing the Windows installation, go to Control Panel ⇒ Add/Remove Programs, and pick the Add/Remove Windows Components button. You should have a dialog window like figure 7.

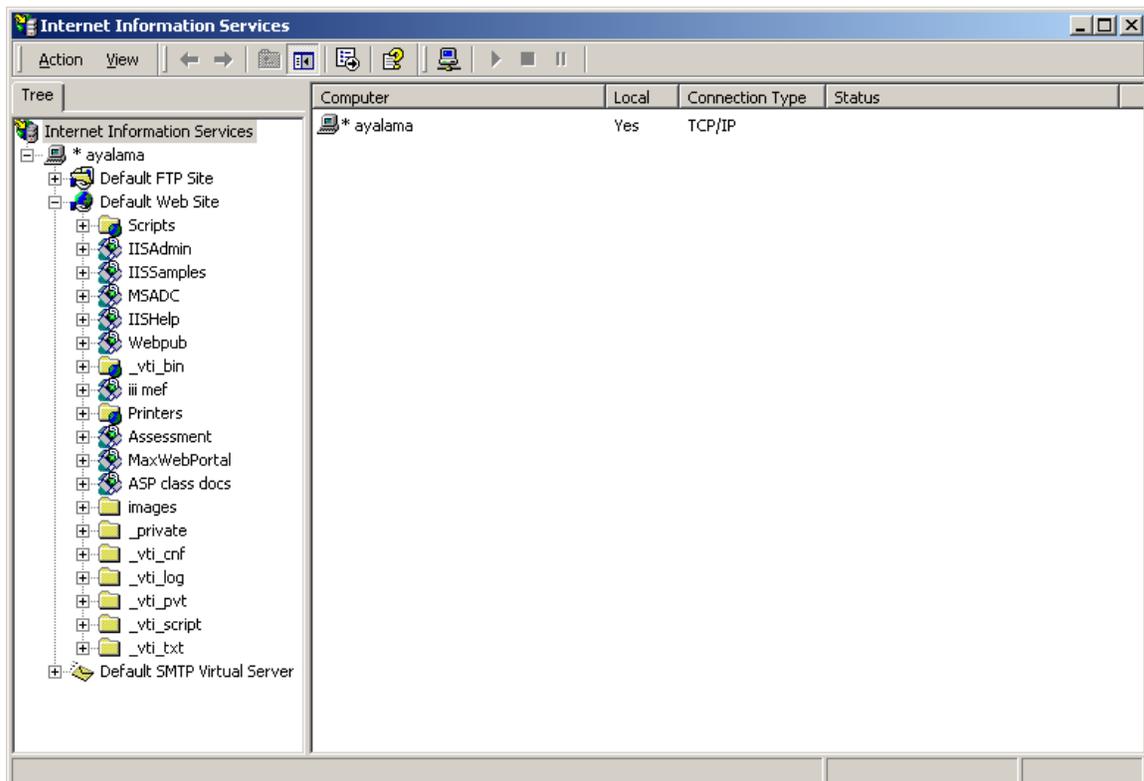


Figure 8 Windows Components Add/Remove dialog window

There are a number of components in the IIS that you can choose to add or leave out. Go ahead and leave the default options. If you decide to install certain components only, stick with this minimum set of features:

- Common Files
- Personal Web Manager
- World Wide Web Server

Now that we have installed IIS, there is a new icon under the Control Panel ⇒ Administrative Tools folder ⇒ Internet Services Manager. When you start Internet Services Manager, you will see the window shown in Figure 8.



**Figure 9 Internet Information Manager window**

You will see the name of your computer on the left at the top of the screen. Under the name of your computer (it behaves like a directory) you will see the sites configured on your server.

The Default Web Site is:

`http://machinename/`

For example from Figure 8: `http://ayalama/`

### **C. Creating virtual directories**

A virtual directory is a pointer to a real or physical directory. It is nothing more than a shorthand name for a real directory. When a user requests a URL, the server looks up the

virtual directory name and translates it to a physical directory before accessing the file. One positive thing about creating virtual directories (besides avoiding a lot of writing in the URL) is that virtual directories hides the physical implementation of your site from the users, giving (in some way) an extra layer of security.

When you installed your Web server, a default Web site and a set of directories were created. By default they are located in `C:\InetPub`. The default Web directory is `wwwroot` in that directory. Any directory that you create in the root Web directory are available through the Web browser. However, you may want to create another directory that is not in the same directory as everything else. That is when a virtual directory comes into place.

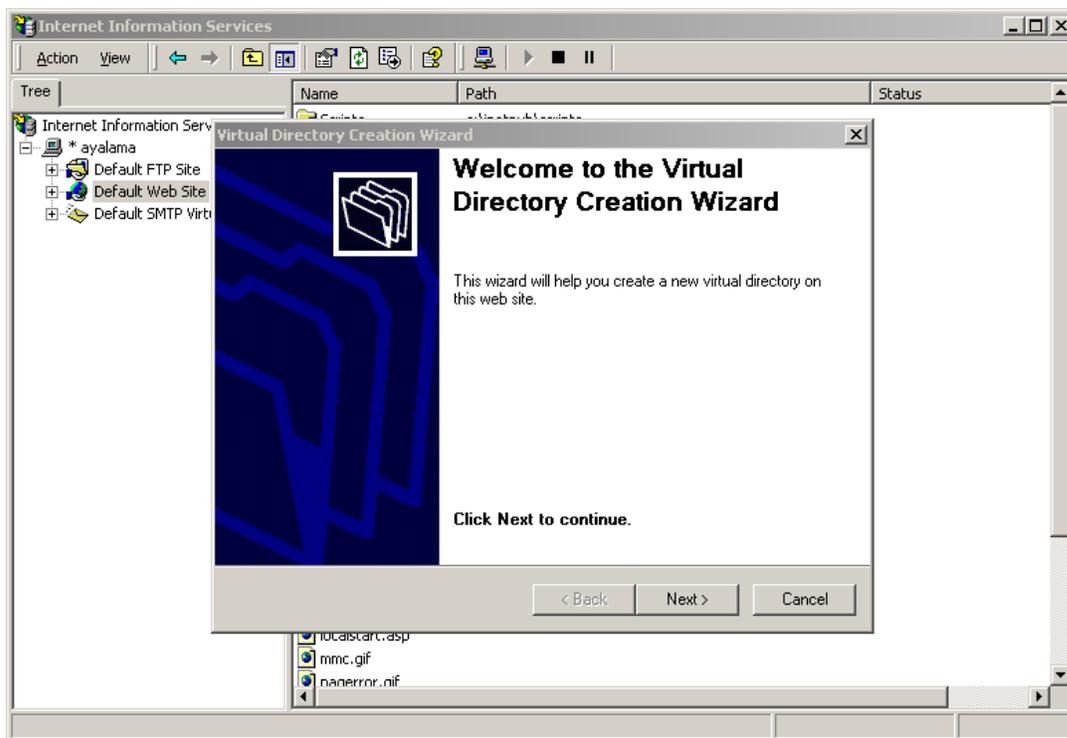
There are two ways to create a virtual directory, using the IIS Manager (virtual directory creation wizard) or without the wizard.

To create a virtual directory, basically you follow three steps:

- Creation of an alias
- Assign a physical directory to the alias
- Assign access permissions

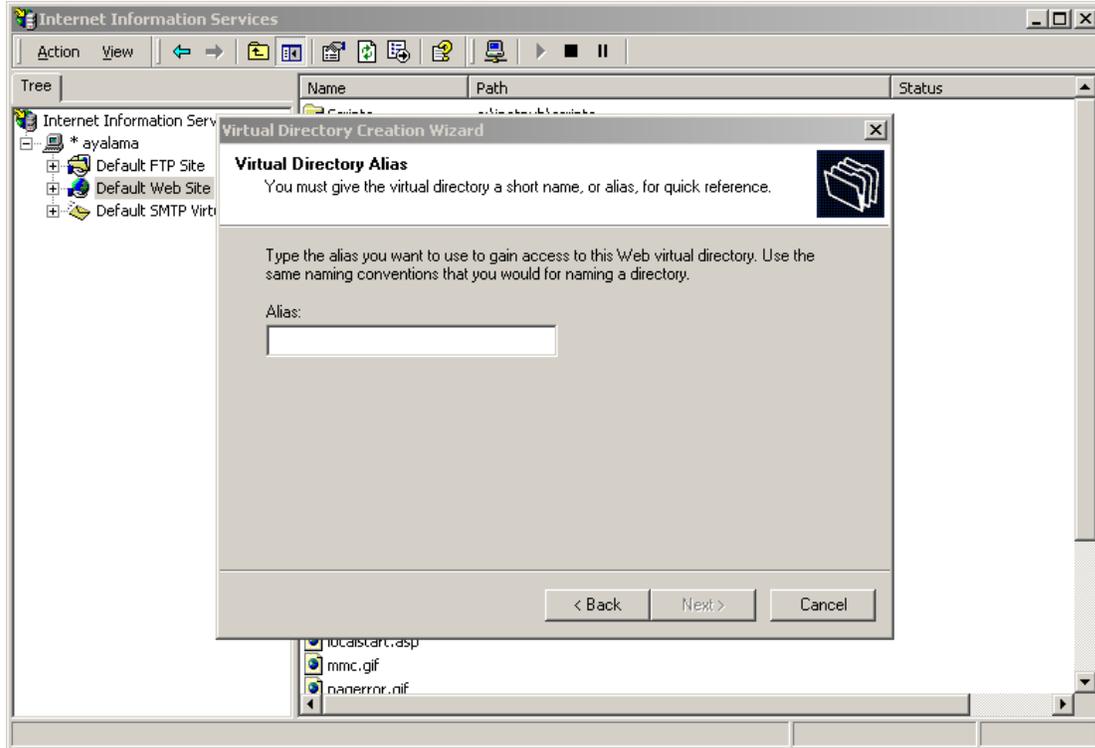
## 1. Creating Virtual Directories With the Wizard

To create a virtual directory from the IIS Manager, right-click the Default Web Site and select `New ⇒ Virtual Directory` from the popup menu. You should have a window as shown in Figure 10.



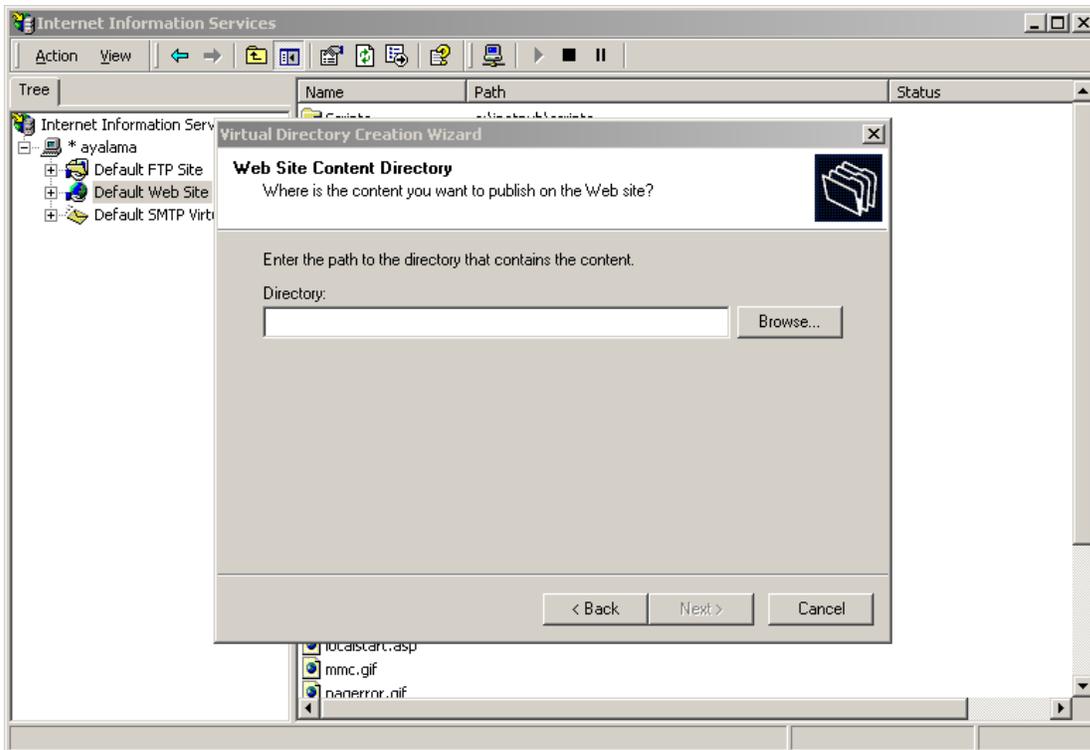
**Figure 10** Creating Virtual Directories with the Wizard

Figure 11 shows the next step in the creation of a virtual directory using the wizard. In this window you specify the name or alias that you want to use in your Web browser. This is the name that you will type in the URL section of the browser in order to access the desired page.



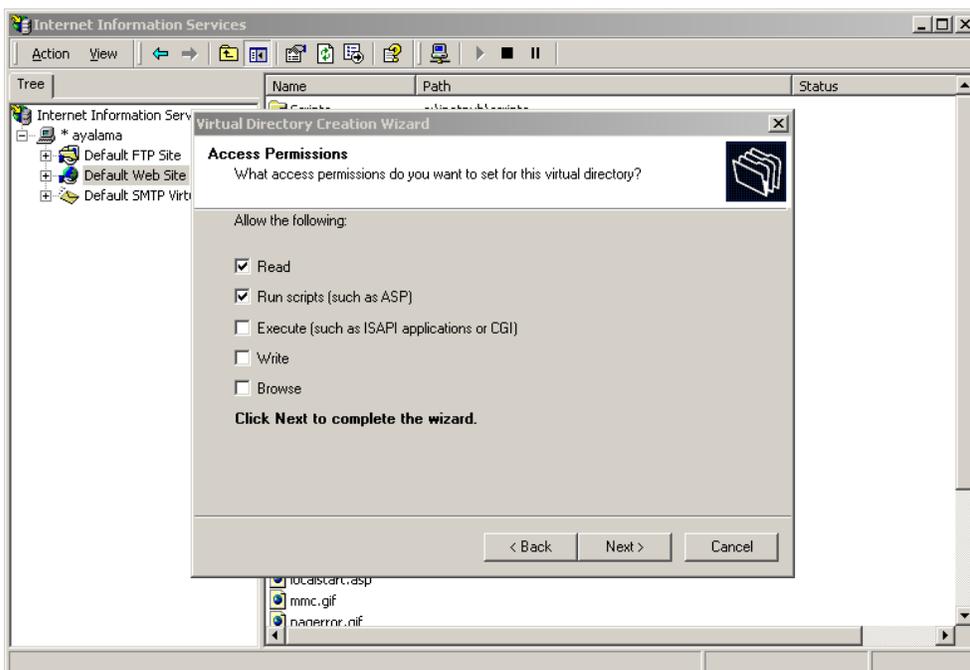
**Figure 11 Naming your virtual directory using the wizard**

The next step is to specify the physical directory that the alias or virtual directory will be pointed at. Figure 12 shows the appropriate window for that.



**Figure 12 Specifying the directory for the alias**

Finally, you specify the security settings or properties for the virtual directory as shown in Figure 13.



**Figure 13 Setting Access Permission**

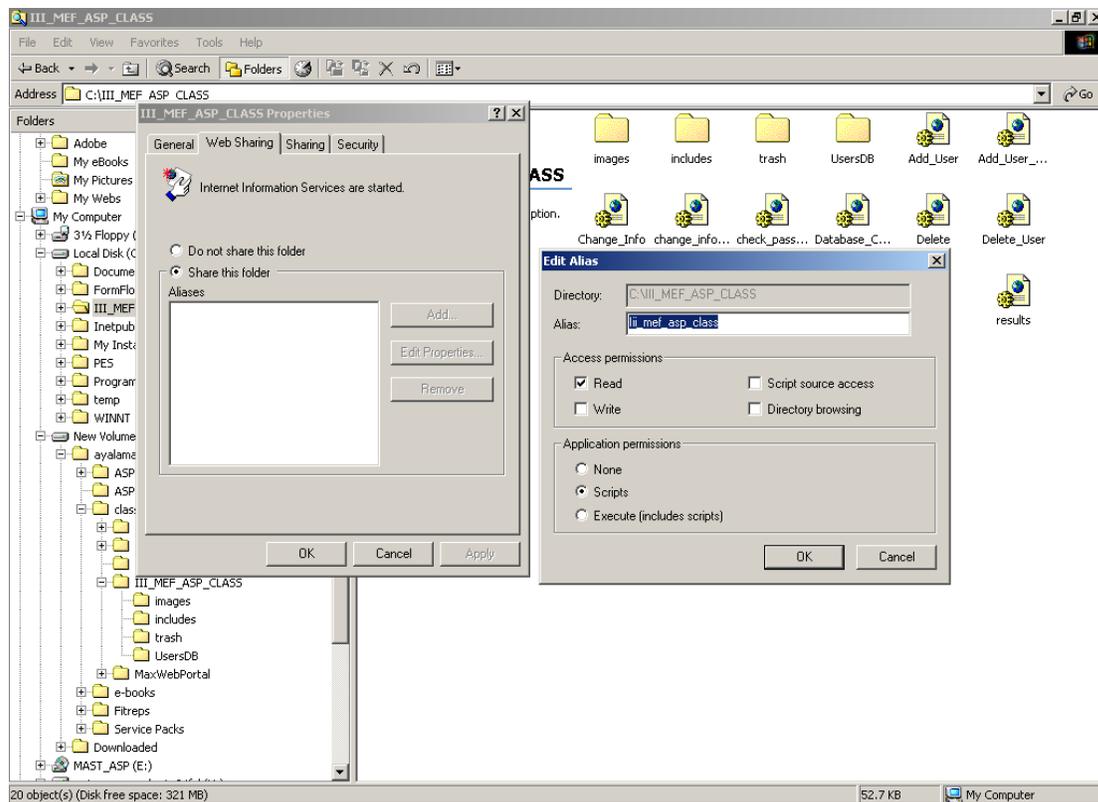
## 2. Creating Virtual Directories Without the Wizard

You can create virtual directories without using the wizard. The three steps still apply. Using Windows Explorer or My Computer in the Desktop, navigate where the desire directory is located. Right-click the desire directory and select Properties. You will have a window like Figure 14.



**Figure 14 Creating a virtual directory without the wizard**

Select Web Sharing tab. The window will look like Figure 14 but with the Do not share this folder radio button selected. Go ahead and select the Share this folder option. The Edit Alias window will pop-up as shown in Figure 15. In this window you will create the alias and assign Permission Access parameters.



**Figure 15 Naming the Alias and assigning Access permissions without the wizard**

By default, you have read and script permissions, which enable you to see HTML, image, and ASP files. You have to have both permissions checked. If you do not mark the script permission box, ASP files will not be able to run. If you do not mark the read permission box, you will not be able to view the pages.

---

## III. ASP Overview

### A. What are Active Server Pages?

ASP is design to let you create pages that can change each time a user loads them. You write code in the page that is run on the Web server before the user sees the page. It 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 applications. ASP enables server side scripting for IIS with native support for both VBScript and JScript.

#### 1. Composition of Active Server Pages?

Active Server Pages are text based files comprised of a combination of HTML tags and Active Server scripts. The Active Server scripts, whether written in VBScript or JScript, are interpreted by the Active Server engine residing in the server. The Active Server scripts usually contain variables, operators, and statements to control the application logic processed by the server.

##### a. ASP Delimiters

Active Server scripts are distinguished on the page from HTML tags by using `<%` and `%>` delimiters. The delimiter can be embedded within HTML tags.

##### b. Setting the ASP Scripting Language

There are two different ways to set the scripting language. The first one is to rely on the default scripting language of the IIS, VBScript (not a good practice). The LANGUAGE attribute allows you to specify the scripting language for the server to interpret, and therefore the interpreter.

```
<%@ LANGUAGE = "VBScript" %>
```

If JScript was the desired scripting language, use JScript with the LANGUAGE tag,

```
<%@ LANGUAGE = "JScript" %>
```

##### c. Variables, Operators, and Statements

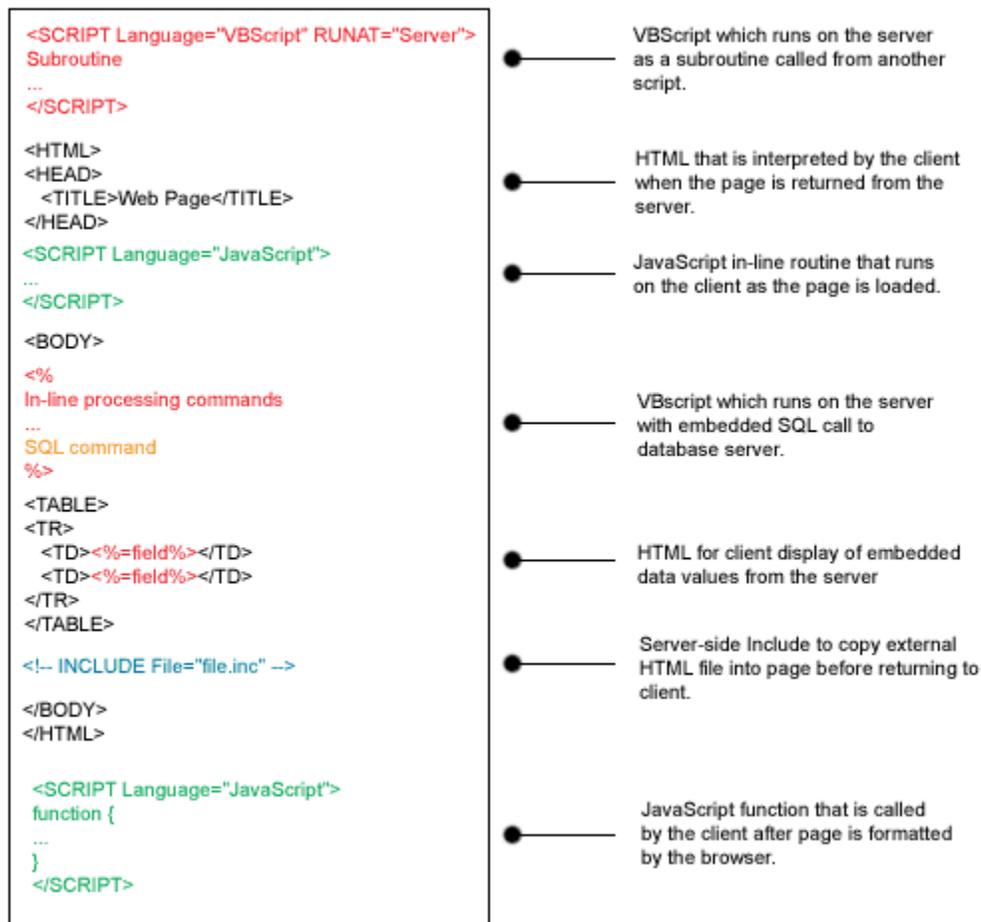
Each scripting language has its own specific syntax that is used to define and set variables, use operators for comparing items, and use statements to help define and organize the code.

##### d. Active Server Components and Objects

The scripting variables, operators, and statements can be used to tap into special Active Server tools that add programming functionality to the ActiveX Server. These tools consist of Active Server Objects components. There are six individual objects with different roles and responsibilities. The most commonly used are:

- Request: responsible for retrieving information from the browser
- Response: responsible for sending information to the browser

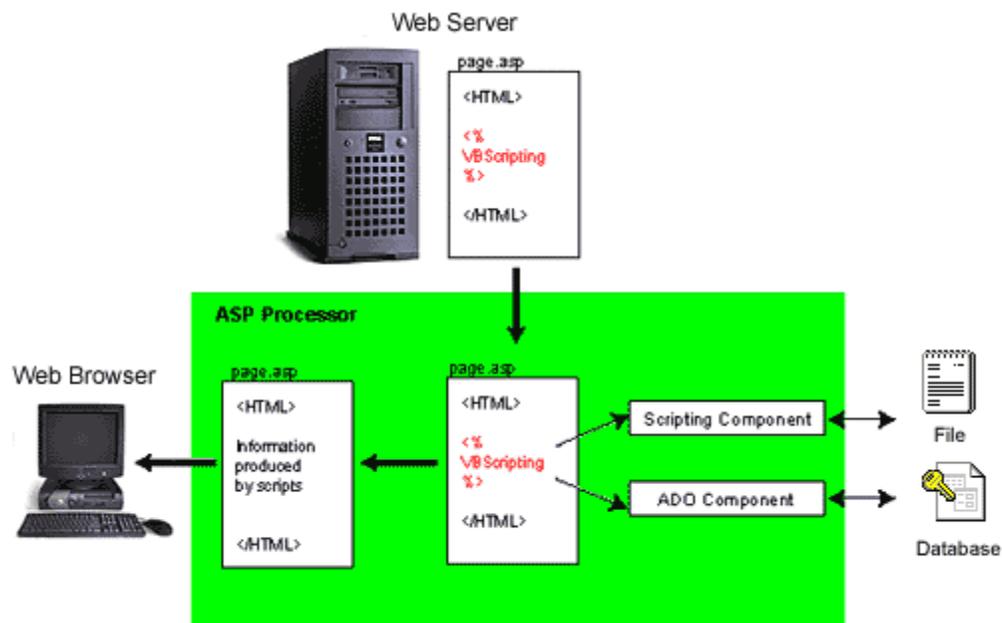
- Session: responsible for managing information for a specific user session
- Server: responsible for administrative functionality of the server



**Figure 16 ASP Composition**

## 2. Running ASP Scripts

Web pages containing scripts must be saved with the special file extension *.asp* to inform the server that this is a scripted page. When the ASP page is retrieved by the server, it is not sent directly to the browser as is the case with normal *.htm* pages. Instead, the page is routed to the ASP processing routine (*asp.dll*) where the scripts on the page are run and the information produced by the scripts is generated.



**Figure 17 Running ASP Scripts**

The server "composes" an HTML page for return to the user, inserting the information generated by the scripts within the HTML formatting. The end result is a page composed entirely of HTML and text information where much, if not all, of the text information has been generated by the scripts. If you view the source listing of the returned page in the browser, you see only the final results of processing. You do not see any of the scripts contained on the original page.

Scripted pages (.asp pages) must be run under the http protocol. This means that you cannot open these pages as files in your browser and view the results as you can with standard .htm pages. Pages *must be run from a server* and accessed with a URL beginning with `http://servername/... .asp`



## IV. Lab 1: Your First Code

### A. Hello World!

#### 1. Creating an HTML Page

In your text editor (Notepad, Wordpad, MS Word) enter the following code:

```
<html>

<head>

    <title> My First HTML Page Hello World! </title>
</head>

<body>

    <h1> Hello, World! </h1>

</body>

</html>
```

Save the file to the root directory of your Web. Once saved, go to your Web browser, type the following URL, substituting the appropriate names with the one you have selected:

```
http://servername/filename.html
```

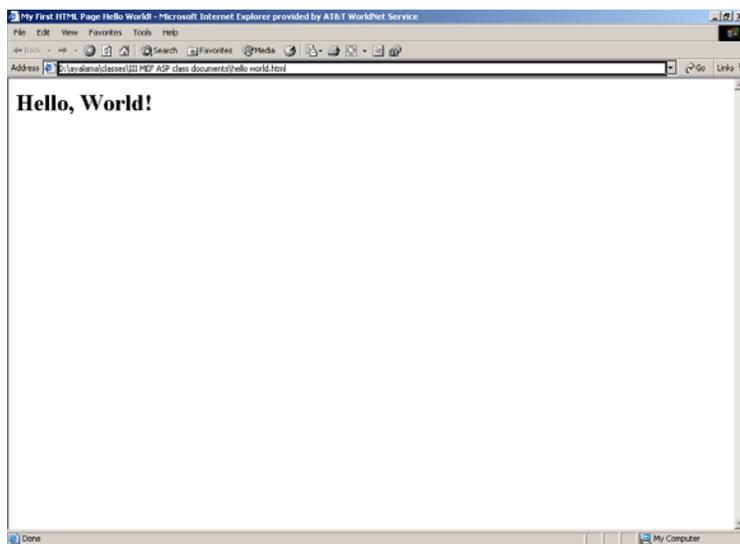


Figure 18 Hello World! Web page

And here you have it, your first HTML page.

## 2. Creating your First ASP page

Lets take the page created above and add some script to it. We want to display the current time in the server and based on the time to say Good morning, afternoon or Good evening, in other words to make a decision. So using the same steps as above, we will proceed to make the additions.

```
<html>

<head>

    <title> My First ASP Page Hello World! </title>

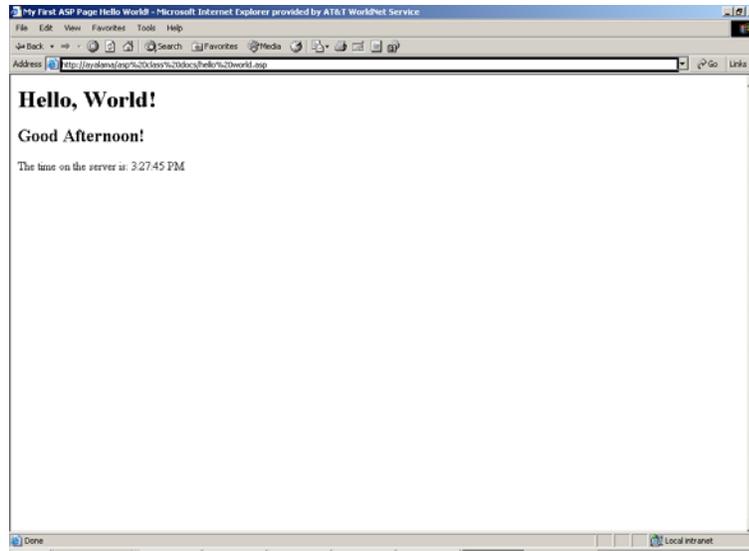
</head>

<body>
    <h1> Hello, World! </h1>
    <% If Time >= #12:00:00 AM# AND Time < #12:00:00
    PM# Then %>
        <h2> Good Morning! </h2><p>
            The time on the server is: <% =Time()%>
    <% Elseif Time >= #12:00:00 PM# AND Time <
    #6:00:00 PM# Then %>
        <h2> Good Afternoon! </h2><p>
            The time on the server is: <% =Time()%>
    <% Else %>
        <h2> Good Evening! </h2><p>
            The time on the server is: <% =Time()%>
    <% End if %>
</body>

</html>
```

Save the file to the root directory of your Web server. Once saved, go to your Web browser, type the following URL, substituting the appropriate names with the one you have selected:

```
http://servername/filename.asp
```



**Figure 19 Hello World! ASP page**

Your very first ASP page.



## V. Programming and Scripting in VBScript

### A. Differences between Visual Basic and Visual Basic Script

VBScript was designed as a subset of the Visual Basic language. VBScript is much easier to learn than programming languages such as Java, C/C++, and other scripting languages such as JavaScript. Derived from the BASIC language, VBScript should not be difficult for anyone who has any computer programming experience.

One of the first striking differences between VBScript and Visual Basic is that Visual Basic has a design-time environment. When you run Visual Basic, you get an attractive editing environment where you can craft forms and write code using an interactive shell. When you work with VBScript, on the other hand, you have no such environment. VBScript code "lives" within an HTML document, which is a plain text file. Visual Basic code creates Windows applications that operate in and of themselves. On the other hand, VBScript code works inside of HTML documents and runs along with HTML.

The other primary difference between VBScript and Visual Basic, aside from development environments, is the language itself. Visual Basic supports many commands, keywords, and data types that VBScript does not support.

### B. Variables and Data Types

#### 1. Data types

VBScript has only one data type called a *Variant*. A Variant is a special kind of data type that can contain different kinds of information, depending on how it is used. Because Variant is the only data type in VBScript, it is also the data type returned by all functions in VBScript.

A Variant can contain either numeric or string information. A Variant behaves as a number when you use it in a numeric context and as a string when you use it in a string context. That is, if you are working with data that looks like numbers, VBScript assumes that it is numbers and does what is most appropriate for numbers. Similarly, if you're working with data that can only be string data, VBScript treats it as string data. Below is a list with all the subtypes of data that a *Variant* can contain:

Empty	<i>Variant</i> is uninitialized. Value is 0 for numeric variables or a zero-length string ("" ) for string variables.
Null	<i>Variant</i> intentionally contains no valid data.
Boolean	Contains either <a href="#">True</a> or <a href="#">False</a> .
Byte	Contains integer in the range 0 to 255.
Integer	Contains integer in the range -32,768 to 32,767.
Currency	-922,337,203,685,477.5808 to 922,337,203,685,477.5807.

Long	Contains integer in the range -2,147,483,648 to 2,147,483,647.
Single	Contains a single-precision, floating-point number in the range -3.402823E38 to -1.401298E-45 for negative values; 1.401298E-45 to 3.402823E38 for positive values.
Double	Contains a double-precision, floating-point number in the range -1.79769313486232E308 to -4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486232E308 for positive values.
Date (Time)	Contains a number that represents a date between January 1, 100 to December 31, 9999.
String	Contains a variable-length string that can be up to approximately 2 billion characters in length.
Object	Contains an object.
Error	

## 2. Variables

A variable is a placeholder that refers to a computer memory location where you can store program information that may change during the time your script is running. In VBScript, variables are always of one fundamental data type, *Variant*.

### a. Declaring Variables

After you've decided on a name for your variable, you have two ways to create it. The first way, called the *explicit method*, is where you use the *Dim* keyword to tell VBScript you are about to create a variable. You then follow this keyword with the name of the variable. For example, if you want to create a variable called *Quantity*, you would enter :

```
Dim Quantity
```

and the variable is created. If you want to create more than one variable, you can put several on the same line and separate them by commas,

```
Dim X, Y, Z
```

The second way to create a variable is called the *implicit method*. In this case, you don't need to use the *Dim* statement. You can just start using the variable in your code, and VBScript creates it automatically. If, for example, you want to store the quantity of an item,

```
Quantity = 10
```

Using the implicit method you don't have to create the variable explicitly with a *Dim* statement.

If you take no special steps, you can freely intermix the implicit and explicit methods of declaring variables. When you want to, you can choose to set aside a named storage space before you use it by giving it a name in advance through the *Dim* statement. On the other hand, you can also rely on the fact that when you refer to something by name in a statement and space hasn't yet been reserved for storing that variable, it will be created for you on-the-fly.

**A word of caution!!** This method of intermixing implicit and explicit declarations can produce programs that are confusing to follow. Fortunately, VBScript gives you a way to force a consistent explicit declaration approach. To make the explicit method a requirement and prevent the implicit allocation of names, you must place the command

```
Option Explicit
```

in the first line of the first script in your HTML document.

```
<%
Option Explicit
Dim Quantity
...
%>
```

Why use the explicit method when I can just start using variables? The implicit method is certainly more convenient while you are writing the code—that is, until you spell a variable wrong. What will happen? VBScript will go ahead and create another variable based on your misspelling, and you will get the wrong result. Consider, for example:

```
<%
...
Quantity = 2
Quantity = Quantity + 3
...
%>
```

You would expect the result to be 5, right? And it would be. Suppose you misspelled *Quantity* in the second line of code:

```
<%
...
Quantity = 2
Quantity = Quantite + 3
...
%>
```

The variable *Quantite* would be created on the spot, and because it was never defined or previously assigned a value, it would be assigned a value of zero. The result variable *Quantity*, then, would wind up being 3, not 5. If, on the other hand, you were using the explicit method of creating variables, you would enter the code as

```
<%  
...  
Option Explicit  
Quantity = 2  
Quantity = Quantity + 3  
...  
%>
```

which would give you a runtime error if you spelled *Quantity* wrong because you had not declared it first.

#### b. Naming Restrictions

Variable names follow the standard rules for naming anything in VBScript. A variable name:

- Must begin with an alphabetic character.
- Cannot contain an embedded period.
- Must not exceed 255 characters.
- Must be unique in the scope in which it is declared.

#### c. Constants

A constant is a meaningful name that takes the place of a number or string and never changes. A number of useful constants you can use in your code are built into VBScript. You create user-defined constants in VBScript using the *Const* statement. You can create string or numeric constants with meaningful names and assign them literal values. For example:

```
Const MY_STRING = "VBScript is fun"  
Const MY_AGE = 31
```

You may want to adopt a naming scheme to differentiate constants from variables. This will prevent you from trying to reassign constant values while your script is running. For example, you might name your constants in all capital letters. It is a good practice to do that. Differentiating constants from variables eliminates confusion as you develop more complex scripts.

## **C. Control Structures**

You can control the flow of your script with conditional statements. Using conditional statements, you can write VBScript code that makes decisions.

## 1. If...Then...Else

Conditionally executes a group of statements, depending on the value of an expression.

```
If condition Then statements
```

Or, you can use the block form syntax:

```
If condition Then
    statements
ElseIf condition-n Then
    else if statements . . .
Else
    else statements
End If
```

The *If...Then...Else* statement is used to evaluate whether a condition is *True* or *False* and, depending on the result, to specify one or more statements to run. Usually the condition is an expression that uses a comparison operator to compare one value or variable with another. *If...Then...Else* statements can be nested to as many levels as you need.

To run only one statement when a condition is *True*, use the single-line syntax for the *If...Then...Else* statement. The following example shows the single-line syntax. Notice that this example omits the *Else* keyword.

```
Dim myDate
    MyDate = #2/13/95#
    If myDate < Date() Then myDate = Date()
```

MyDate is assigned 13 Feb 1995. Today's date is 21 May 2003. Since MyDate is less than today's date, MyDate will be assigned 21 May 2003.

To run more than one line of code, you must use the multiple-line (or block) syntax. This syntax includes the *End If* statement.

```
Sub Status(value)
    If value = "L" Then
        MarineStatus = "On Leave"
    End If
End Sub
```

A variation on the *If...Then...Else* statement allows you to choose from several alternatives. Adding *ElseIf* clauses expands the functionality of the *If...Then...Else* statement so you can control program flow based on different possibilities.

```
Dim Status
Status = "T"
If Status = "T" Then
    MarineStatus = "TAD"
ElseIf Status = "L" Then
```

```
MarineStatus = "On Leave"  
ElseIf Status = "A" then  
    MarineStatus = "Accounted for"  
Else  
    MarineStatus = "UA"  
End If
```

You can add as many *ElseIf* clauses as you need to provide alternative choices. Extensive use of the *ElseIf* clauses often becomes cumbersome. A better way to choose between several alternatives is the *Select Case* statement.

## 2. Select Case

The *Select Case* structure provides an alternative to *If...Then...ElseIf* for selectively executing one block of statements from among multiple blocks of statements. A *Select Case* statement provides capability similar to the *If...Then...Else statement*, but it makes code more efficient and readable.

A *Select Case* structure works with a single test expression that is evaluated once, at the top of the structure. The result of the expression is then compared with the values for each *Case* in the structure. If there is a match, the block of statements associated with that *Case* is executed, as in the following example.

```
Dim Status, MarineStatus  
Select Case Status  
    Case "A"  
        MarineStatus = "Accounted for"  
    Case "L"  
        MarineStatus = "On Leave"  
    Case "T"  
        MarineStatus = "TAD"  
    Case Else  
        MarineStatus = "UA"  
End Select
```

## D. Looping Structure

Looping allows you to run a group of statements repeatedly. Some loops repeat statements until a condition is False; others repeat statements until a condition is True. There are also loops that repeat statements a specific number of times.

The following looping statements are available in VBScript:

- *Do...Loop*: Loops while or until a condition is True.
- *While...Wend*: Loops while a condition is True.
- *For...Next*: Uses a counter to run statements a specified number of times.
- *For Each...Next*: Repeats a group of statements for each item in a collection or each element of an array.

We will discuss the first three.

## 1. For...Next

When you need to loop a pre-determined number of times, use a *For...Next* loop structure. The structure uses a loop variable to control the number of loops to be executed through the code. The programmer provides the control variable and the parameters of starting value and ending value as well as an increment step. The syntax is:

```
For variable = start to end Step increment
    `code that gets repeated
Next
```

The default increment step is one so the Step specification can be omitted if the default increment is desired.

```
Dim x, result
x = 0
For x = 1 to 50
    result = result + 1
Next
Response.Write "The value of result = " &result
```

The code inside the For loop will be executed 50 times. What is the value of result at the end of the For loop execution?

For increments of 2:

```
Dim x, result
x = 0
For x = 1 to 50 Step 2
    result = result + 1
Next
Response.Write "The value of result = " &result
```

The code inside the For loop will be executed 25 times since the increment is 2.

Negative increments can be used like Step -2. The only difference is that the start value must be higher than the end value.

```
For x = 50 to 1 Step -2
```

Nested Loop can be performed as well,

```
Dim x, y, result
x = 0
For x = 1 to 100
    For y = 1 to 100
        result = result + 1
    Next
Next
Response.Write "The value of result = " &result
```



The code inside the For loop will be executed 10000 times.

## 2. Do...Loop

You can use *Do...Loop* statements to run a block of statements an indefinite number of times. The statements are repeated either while a condition is *True* or until a condition becomes *True*. You can place a conditional test at either the start or end of the loop structure. This provides you with the ability to force a loop to execute at least one. The syntax is:

```
Do While condition
    ... code within the loop goes here
Loop
or
Do
    ... code within the loop goes here
Loop While condition
```

Examples:

```
Dim result
Result = 0;
Do While result < 50
    result = result + 1
Loop
Response.Write "The value of result = " &result
```

It is the same behavior from the For...Next structure. The condition is tested **before** executing the code.

```
Dim result
Result = 0;
Do
    result = result + 1
Loop While result < 50
Response.Write "The value of result = " &result
```

The condition is tested **after** execution of the code.

The *Do...While* version may not execute once because the conditional test at the start of the loop may not evaluate True. On the other hand, the *Do..Loop While* version places the conditional test at the end of the loop, so the loop must execute at least once. As with all the other looping structures, the *Do...Loop* structure can be nested like the *For...Next* loop.

## 3. While...Wend

The *While...Wend* structure is identical to the *Do...While* loop structure. *Wend* takes place of the *Loop* keyword in the previous examples. The *While...Wend* executes until the conditional statement following the *While* becomes True. The syntax is:

```
While condition  
    'do something  
Wend
```

Example:

```
While result < 50  
    result = result + 1  
Wend
```

The looping will continue while *result* is less than 50.

## VI. Processing user input request (Request Object)

Anytime you visit a Web page, there is a communication going on between your computer, the client, and the computer providing you the Web page, the server. There are times when the client specifically sends data to the server like entering an e-mail into a form. To interact with the client or visitor, your script should be able to request information from the visitor. This is done by the use of the *Request* object.

### A. The Request Object

You can think of the *Request* object as the *input* object. This object holds information that was sent from the browser to the web server. It is responsible to make that information accessible to the ASP application. So, you use the *Request* object to read that information from the client's browser. It uses five collections (a group of common objects) to provide for communication between the client browser and Web server.

### B. How to get data from the user to the server?

HTML forms allow the user to enter data into control on a Web page. Then they can send the information to the Web server by clicking a Submit button. On the server, you can use an ASP page along with the *Form* collection to handle the information sent.

#### 1. HTML Forms

When you create a form in HTML, you have to give it a *method* describing how the form is to be processed. The two options are *GET* and *POST*, being *POST* the most commonly used. If the form's method is *POST*, the information entered in the form by the user on the client side is accessed through this *FORM* collection. If the method is *GET*, the information can be found in the *QueryString* collection.

If you create a form in HTML, you can also specify an action. If you intend to handle the form using ASP page, put the page's name that will handle the form in the action attribute.

This is the method that most people think of when they want to send user data to a Web server. A form can be as simple as a single text box for searching, or as complex as a on-line mortgage application. A simple form looks like:

```

<form method="post" name="LoginForm" action="check_password.asp">
  <table border="2" width="400">
    <tr>
      <td width="91"><b>Login: &nbsp;</b></td>
      <td width="295"><input name="login"></td>
    </tr>
    <tr>
      <td width="91"><b>Password:</b></td>
      <td width="295"><input type="password"
        name="password"></td>
    </tr>
  </table>

```

Form's name

Page that will process the data received

Method *post* so *Request* object will be capable of accessing the information

Name of the text box. Request object can ask for the information

```
<p><input type="submit" value="Login" name="Login"> <input
type="reset" value="Reset" name="Reset"></p>
</form>
```

Type of submit. This will be a submit button with label Login

---

Login:	<input type="text"/>
Password:	<input type="password"/>

Login    Reset

---

**Figure 20 HTML Form**

We have two “variables” or “text boxes” that we need to get information from, *login* and *password*. In the page that is handling the form, you may want to process the values received from the form. For example you want to validate and authenticate the login and password provided by the user in order to allow the access to your web site. You can do that using the *Request* object:

```
<%
...
password = Request.Form("password")
login = Request.Form("login")
...
%>
```

We are getting the information from the form and assigning it to two local variables.

or you can use the shortcut:

```
<%
...
password = Request ("password")
login = Request ("login")
...
%>
```

*Request.QueryString* is used when you are using the *Get* method on the form to pass information.

You do not have to specify the *Form* collection. When you use this syntax, the *Request* object looks in the *Form* collection for the name you send it.

If the Form was using the *Get* method like:

```
<form method="get" name="LoginForm" action="check_password.asp">
  <table border="2" width="400">
```

then you would be using *Request.QueryString* instead of *Request.Form*. This method is noticeable in the URL part of the Web browser. All your items passed will show there. Remember, there is a size limit when using the *GET* method of about 1KB.

### **C. Processing results**

No that you know how to get the information out of the form, there are basically two ways to manage the information, assigning it to local variable or requesting them directly from the form. For example:

```
<%  
...  
Users_Password = Request ("password")  
Users_Login = Request ("login")  
  
strSQL = "SELECT * FROM Names WHERE Password = '" &  
Users_Password & "' and Login = '" & Users_Login & "'"
```

In the previous piece of code we are assigning to local variable password and login the respective information inputted by the user in the form. Then, we are using those variable in the query statement. Another way to do that is bypassing the assignment of those variables and substitute them for the Request object itself.

```
<%  
...  
strSQL = "SELECT * FROM Names WHERE Password = '" &  
(Request ("password")) & "' and Login = '" & (Request  
("login")) & "'"
```

As you can notice, the second method looks crowded and could be confusing at the beginning. The first method is more readable.

## VII. Lab 2: Using a form to gather user input and displaying results

In this Lab you will create a form that will take information from the user, First Name, MI, and Last Name and submit the information to the server. You will have an Active Server Page processing that information and displaying it in a separate page.

Type the following code to create the form that will capture the user's information. Save the file with an .html extension under the virtual directory you created or under the default virtual root directory C:\Inetpub\wwwroot.

```
<HTML>

<HEAD>
  <TITLE>
    Personal Information
  </TITLE>
</HEAD>

<BODY>

  <form method = "POST" action = "answering.asp">

  <FONT COLOR="#FF0000" SIZE=5>
    Personal Information
  </FONT>

  <table border="1" width="667" height="30">
  <tr>
    <td width="104" height="1">First Name</td>
    <td width="160" height="1"><input type="text"
      name="FName" size="20"></td>
    <td width="65" height="1">MI</td>
    <td width="121" height="1"><input type="text"
      name="MI" size="5"></td>
    <td width="175" height="1">Last Name</td>
    <td width="186" height="1"><input type="text"
      name="LName" size="20"></td>
  </tr>
</table>

  <p>

  <input type="submit" value="Add New User" name="Add">
  <input type="reset" value="Reset" name="reset">
```

```
</p>
</form>
</BODY>
</HTML>
```

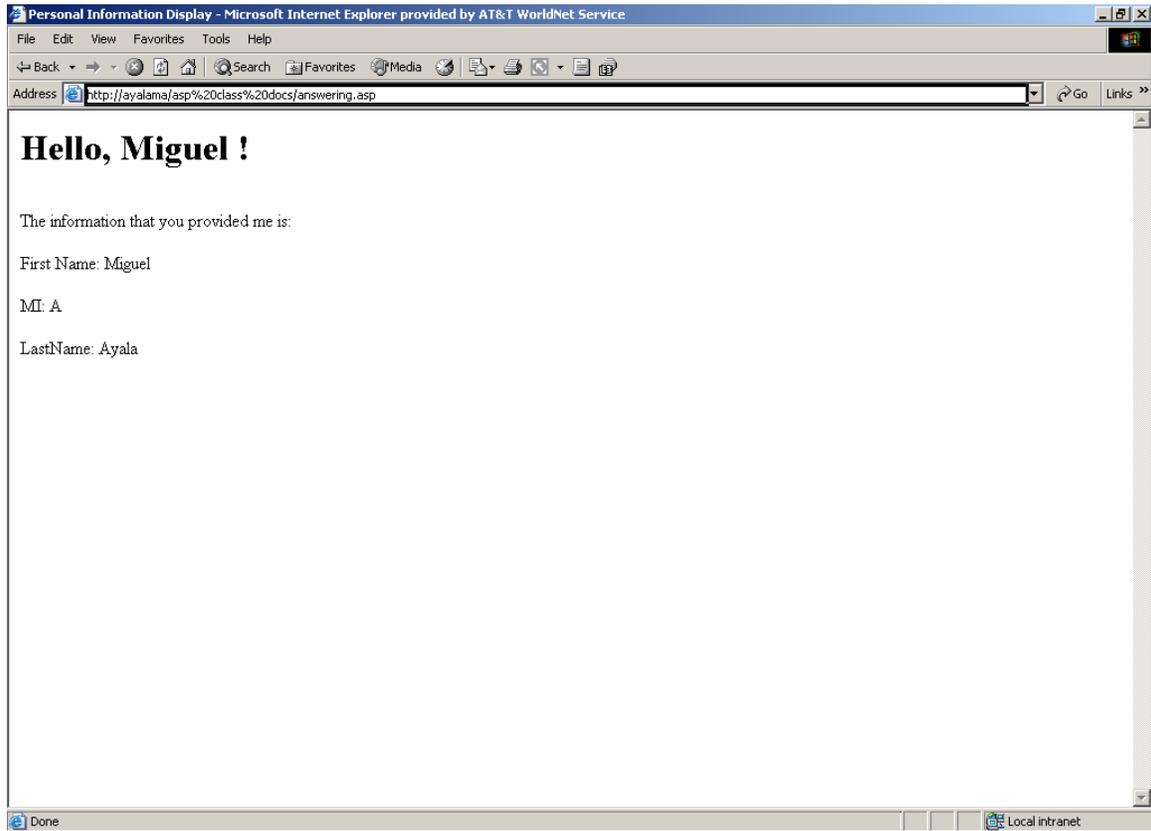
### Personal Information

First Name	<input type="text" value="Miguel"/>	MI	<input type="text" value="A"/>	Last Name	<input type="text" value="Ayala"/>
<input type="button" value="Add New User"/>		<input type="button" value="Reset"/>			

**Figure 21 Lab 2 Using a Form Example**

Type the following Active Server code and save it as `answering.asp` in the same virtual directory you saved the previous file.

```
<HTML>
<HEAD>
  <TITLE>
    Personal Information Display
  </TITLE>
</HEAD>
<BODY>
<h1> Hello, <%=Request ("FName")%> !</h1>
<br>
The information that you provided me is:
<p>
First Name: <%=Request ("FName") %>
</p>
<p>
MI: <%=Request ("MI") %>
</p>
<p>
LastName: <%=Request ("LName") %>
</p>
</BODY>
</HTML>
```



**Figure 22 Lab 2 Response to the user**



---

## VIII. Session Object

### A. Session Variables

The Session object represents the current user's session on the web server. It is user specific, and its properties and methods allows you to manipulate the information on the server that is specific to that user for the duration of that user's connection. So, a session is the interval a single user spends interacting with your application during a contiguous length of time.

#### 1. Assigning Session Variables

Session variables are declared in a manner similar to declaring application variables. To create or reference a session variable, use the *Session* object and name of the variable with the following syntax:

```
Session (variableName)
```

Where *variableName* is the name of the session-level variable. Session-level variables are only available to the user session that created the object and cannot be shared among separate user sessions. This variable can be accessed from any page as long as the user is using the application.

To add a Session variable, you simply assign a value to session variable name.

```
Session ("Permissions") = "user"
```

In the above line of code a *Session* variable named *Permissions* is assigned the value of *user*. This variable will be accessible from any page as long as the user is in the same application. In this way we can provide the user specific access to certain pages based on the permission given.

```
If Session("Permissions") = "user" Then
    response.redirect "Show_Records.asp"
Elseif Session ("Permissions") = "admin" Then
    response.redirect "Admin.asp"
Else
    response.redirect "login.asp"
End If
```

#### 2. Clearing Session Variables

You can clear a *Session* variable by setting it to a null string ("") or by a setting it to the VBScript *Empty* value. For example:

```
Session ("Permissions") = ""
Session ("Permissions") = Empty
```



## IX. Lab 3: Use of Session Variables

In this Lab you will use the form from Lab 2. We will add some code in order to have Session variables.

Type the following code and save the file as *answering.asp* under the virtual directory you created or under the default virtual root directory C:\Inetpub\wwwroot.

```

<HTML>

<HEAD>
  <TITLE>
    Personal Information Display
  </TITLE>
</HEAD>

<BODY>

  <%
    Session ("FName") = Request ("FName")
  %>

  <br>

  <% If Session("FName") = "Miguel" Then %>

    <h1> Hello and welcome back <%=Request
  ("FName")%> !</h1>

  <% Else %>

    <h1> Hello, <%=Request ("FName")%> !</h1>

  <% End If %>

  <br>
  The information that you provided me is:
  <p>
  First Name: <%=Request("FName") %>
  </p>
  <p>
  MI: <%=Request ("MI") %>
  </p>
  <p>
  LastName: <%=Request ("LName") %>

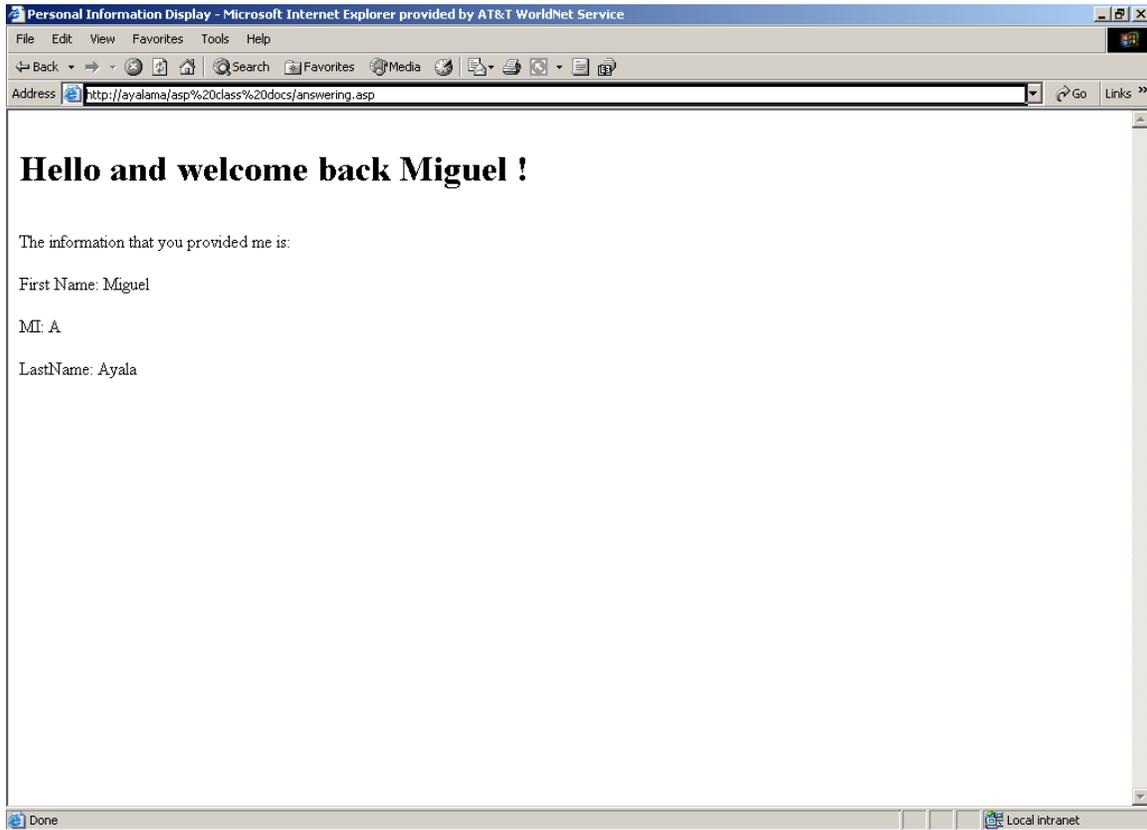
```

We assign a session variable (FName) and we use an If statement to check the value of it. If the session variable meets our condition then the first block is executed, the second block otherwise.

</p>

</BODY>

</HTML>



**Figure 23 Lab 3 Session Variables**

## X. Responding to the user (Response Object)

### A. How to send output to the user's browser

In the chapter *Processing User Input (The Request Object)*, we talked about how to get information from the user and in Lab 2 we used the *Request* object to display user information using the syntax:

```
<p>
First Name: <%Request ("FName") %>
</p>
```

A better method to print output uses the *Write* method of the *Response* object.

### B. The Response Object

If the *Request* object is the *input* object, then the *Response* object is just the opposite, the *output* object. The *Response* object allows you to send information from the Web server to the browser.

Probably the most important methods of the *Response* object are the *Redirect* method, and the *Write* method.

### C. Response.Write

The *Write* method is used to send output to the browser. It has the following syntax:

```
Response.Write variant
```

where *variant* is any variant data type supported. For example, to display the time a page was requested by using the *Write* method, we could use the following syntax:

```
<% Response.Write Now %>
```

or the shortcut

```
<%=Now%>
```

The above piece of script will produce:

```
5/22/03 8:39:20 PM
```

So, the *Response.Write()* method can be coded inside a script at the location on the page where the system date and time should appear, as well as other dynamic information/variables. In the previous example, it can write HTML text combined with the *Date()* and *Time()* values retrieved from the server.

```
<%
...
```

```
Response.Write("<b>The current date is " & Date())
Response.Write(" and the current time is " & Time() &
".</b>")
...
%>
```

Note that the `Response.Write()` method can contain literal text strings (enclosed in quotes), values retrieved from VBScript functions or other VBScript commands, along with HTML tags (enclosed in quotes as part of a text string) to control output formatting. Text strings and server values are connected together, with the VBScript concatenation operator `&` (ampersand) to produce a resulting string of text characters that can be written to the Web page. The general format for the `Response.Write()` method is shown below:

```
Response.Write("text string" | & | server value | "<HTML
tag>")
```

In the first line of the above script the literal text string `"<b>The current date is "` is concatenated with the server value produced by the `Date()` function to create the output string *The current date is 5/24/2003*. In the second line of the script the literal text string `" and the current time is "` is concatenated with the value of the `Time()` function and with the string `".</b>"` to create the output string *and the current time is 11:18:19 PM*. Alternately, the script can be written as a single line of code,

```
<%
...
Response.Write("<b>The current date is " & Date() & " and
the current time is " & Time() & ".</b>")
...
%>
```

where various text strings and server values are concatenated to produce the output line. The `Response.Write()` method has a second, abbreviated format that makes it easy to embed server values within the HTML coded on the page. These scripts are in the following format,

```
<%= server value %>
```

where the symbols `<%=` and `%>` enclose a server-generated data value. This script is placed anywhere within the HTML coding on a page to display the value at that location on the page. The following output, for example,

```
The current date is 5/24/2003 and the current time is
11:18:19 PM.
```

is displayed with the following HTML code and embedded server values positioned at the above location:

```
<b>The current date is <%=Date() %> and the current  
time is <%=Time() %>.</b>
```

As the ASP processor renders the HTML code for return to the browser, it inserts the data values given by the Date() and Time() functions within the HTML code. This is a shorthand way of coding scripts that produce single data values and is an alternative to coding

```
<b>The current date is <% Response.Write(Date()) %> and the  
current time is <% Response.Write(Time()) %>.</b>
```

where a full script (even if a single line of code) is embedded on the page.

### **D. Response.Redirect**

The *Redirect* method simply tells the browser where to go. It is useful for redirection when you know in advance that you need to redirect and you have the exact location of the redirection URL. The syntax:

```
Response.Redirect (http://righthpage.htm)
```

For example, redirection is particularly useful when you are making decisions based on a user's actions. Suppose you have a login page and you are authenticating and validating users in order to provide access to certain pages based on privileges. If the privilege is *user* then you can direct the user to the respective page. On the other hand, a user with an administrative privilege will be redirected to a page where all the administration tools would be available. Example:

```
<%  
...  
Permissions = Session("Permissions")  
  
If Permissions = "user" Then  
    response.redirect "Show_Records.asp"  
Elseif Permissions = "admin" Then  
    response.redirect "Show_RecordsAdmin.asp"  
Else  
    response.redirect "login.asp"  
End If  
...  
>
```



## XI. Lab 4: Responding to the user

In this Lab you will use the form from Lab 3. We will add some code in order to use the Response object.

Type the following code and save the file with an *answering.asp* under the virtual directory you created.

```
<HTML>
<HEAD>
  <TITLE>
    Personal Information Display
  </TITLE>
</HEAD>
<BODY>
  <%
    Session ("FName") = Request ("FName")
  %>

  <br>

  <% If Session("FName") = "Miguel" Then %>

    <h1> Hello and welcome back <%Response.Write
  ("FName")%> !</h1>

  <% Else %>

    <h1> Hello, <%Response.Write ("FName")%> !</h1>

  <% End If %>

  <br>
  The information that you provided me is:
  <p>
  First Name: <%= ("FName") %>
  </p>
  <p>
  MI: <%= ("MI") %>
  </p>
  <p>
  LastName: <%= ("LName") %>
  </p>
</BODY>
</HTML>
```

The code is exactly the same as Lab 3 with the only difference that we substituted the `=Request("FName")` with `Response.Write ("FName")`.

In fact, as discussed earlier in this chapter the `= ("FName")` is equivalent to `Response.Write ("FName")`. So we were using it in Lab 3.

The result from will look exactly the same as for Lab 3.



---

## XII. Database Overview

A database is a repository for related collections of data. An address book is an example of a database in which you store names, addresses, and telephone numbers of friends, relatives, and business contacts. The information in a database is stored as a *table*.

### A. Planning a database

The goal of the database planning process is to identify the following:

- the information you currently track
- the information you want or need to track in the future
- the reports you need to produce

#### 1. Database Structure

Before storing the information, you need to design a *database structure*. Each database file has the following elements:

**Field:** contains one portion of the data (name or telephone number), also known as a column.

**Record:** contains related information (name, address, and telephone number), also known as a row. A single record is made up by one or more fields.

**Database Table:** made up by one or more records (table of products on the display, table of customers' names and addresses).

*Database File* is the physical file stored on a disk and contains Table, Query, Form, and Report.

The application we are trying to create builds onto the password checking pages that we will be using. It is quite simple, yet it introduces you to most of the considerations you will face when designing data-driven Web pages. In this example, we will set up a table of accounts against which we can check user accounts and passwords during log on. If the user does not enter a valid account and password, they will not be permitted to view the `show_records.asp` page. We will need to create a database and tables to contain all the necessary user information pertaining our application.

For purposes of this tutorial you will only need to create one database. Any and all tables necessary for future applications can be created in this database. To simplify access to the database, you can place it in the same directory as your Web pages or create a folder to hold the database. The important thing is that at time of coding you need to know the location of the database. In this and subsequent lessons we will assume the existence of a database named *Users.mdb*. You can name your database whatever you choose.

Within this database you need to create a new table named *Names*. It's always a good idea to name your databases and tables without embedded blank spaces in the names. The structure of this table is shown below.

Field Name	Field Type	Field Size
ID	AutoNumber	Long Integer
SSN	Number	Long Integer
FName	Text	15
LName	Text	15
MI	Text	2
Rank	Text	10
Unit	Text	50
Login	Text	15
Password	Text	15
Permissions	Text	10
Phone	Text	15
Spouse	Text	15

In order to have the passwords to show as asterisk ( \* ) in the table, you need to specify the input format as *Password*.

**Table 1 Names Table Structure**

These specifications are entered when creating a database table in Access. They control the "structure" of the table, i.e., the *fields* of data that will be maintained in the table. At the same time, as you define each field, you will need to make sure that three other specifications are made.

The point to be kept in mind is that you are using Access only to create a database and table structure. You will not be using any of the processing features of Access. All considerations related to key fields, indexing, retrieval, updating, and the like are controlled through scripts, not through the Access software. Once you have created a database structure, you can effectively forget about Access.

Once you have created the table structure, open the table in "datasheet" view and insert a few example *records*. This is done for testing purposes. Later, we will consider how to populate database tables from Web forms. When you are done, you should have created a *Names* table that looks like the following (Enter your own information):

ID	SSN	FName	LName	MI	Rank	Unit	Login	Password	Permissions	Phone	Spouse
1	123456789	Miguel	Ayala	A	Capt	MSTP	ayalama	*****	admin	7037846001	Daisy

**Table 2 Names Table Data**

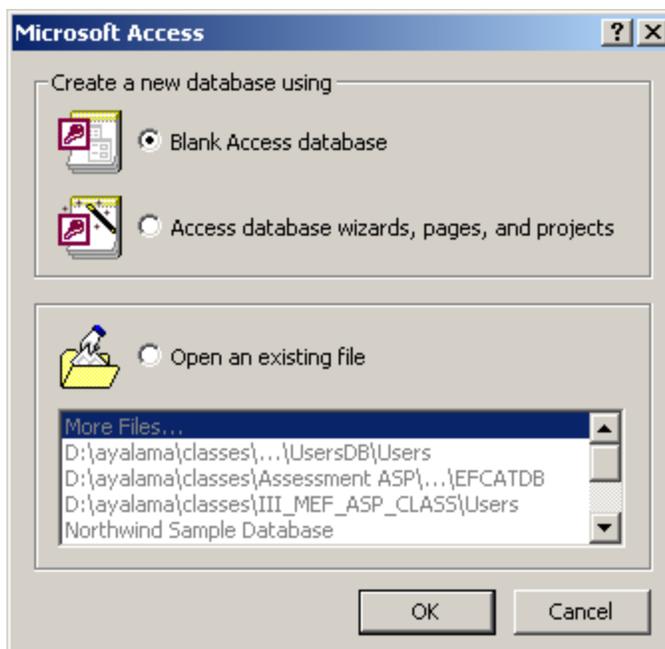
## 2. Building the database in Microsoft Access

A Microsoft Access *database* is a repository for *tables* of information. A database table is a collection of rows and columns for storing data. The columns represent the different *fields* of data; the rows are the different *records*.

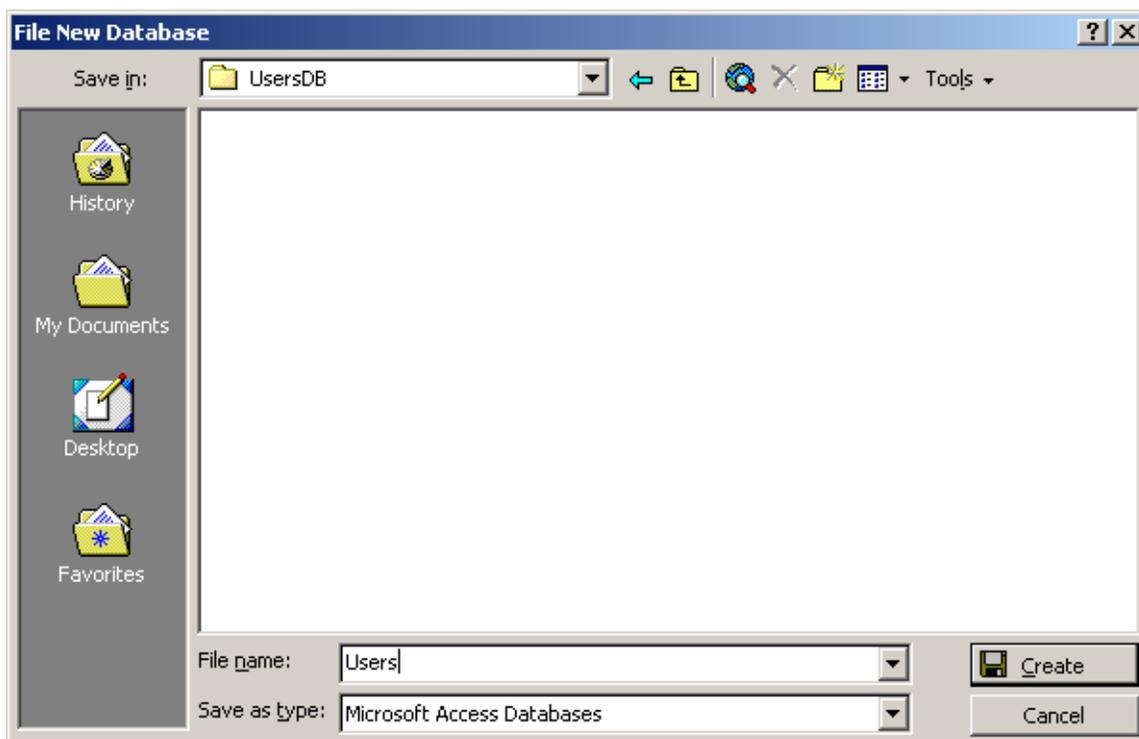
### a. Creating a Database

Tables are the main units of data storage in Access. Recall that a table is made up of one or more *columns* (or *fields*) and that a given column may appear in more than one table in order to indicate a relationship between the tables.

When you open Microsoft Access you will see the dialog box at the right. Click the "Blank Database" button to indicate that you are creating a new database. Then click "OK".



**Figure 24 Creating a new Microsoft Access Database**

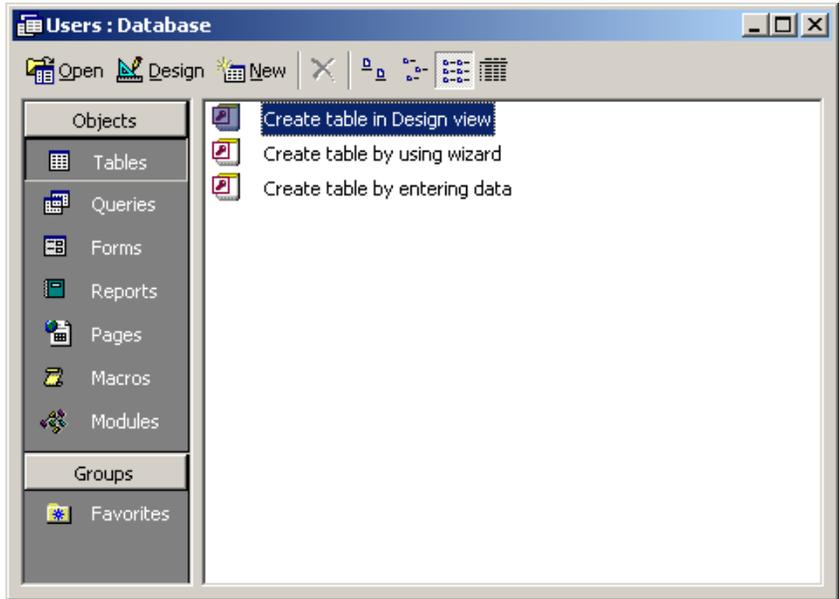


**Figure 25 Naming and saving the new database in Access**

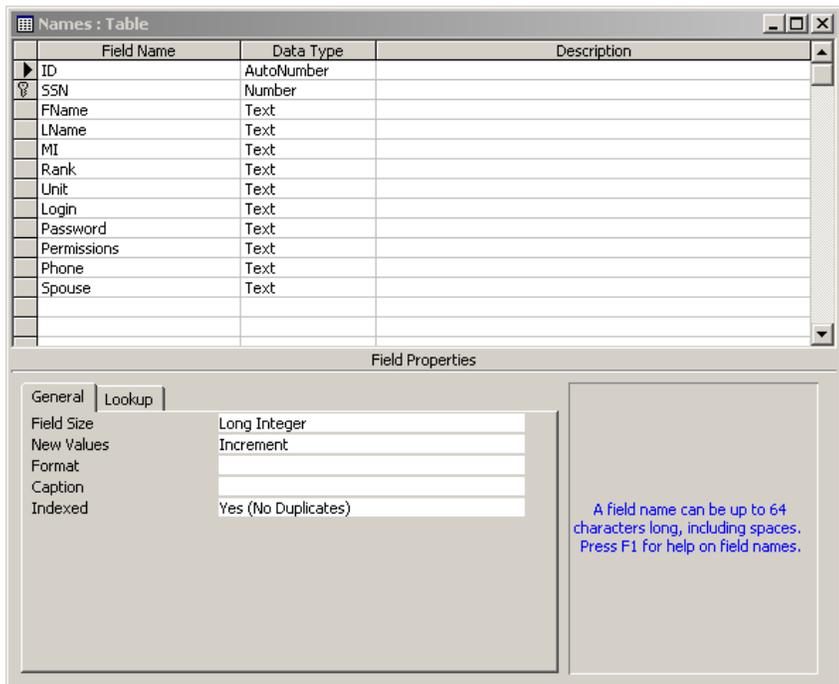
When the "File New Database" dialog box appears, name it with your `Users.mdb` and save it in your directory on the server by clicking the "Create" button.

b. Creating a Table

Next, you are presented with a set of options for creating or selecting tables and other components of your database. You can create your table using the Wizard, in Design View or by entering data into generic fields. We will do it in Design View. Click the *Design* button.



**Figure 26 Access Main Window**



**Figure 27 Table Design View**

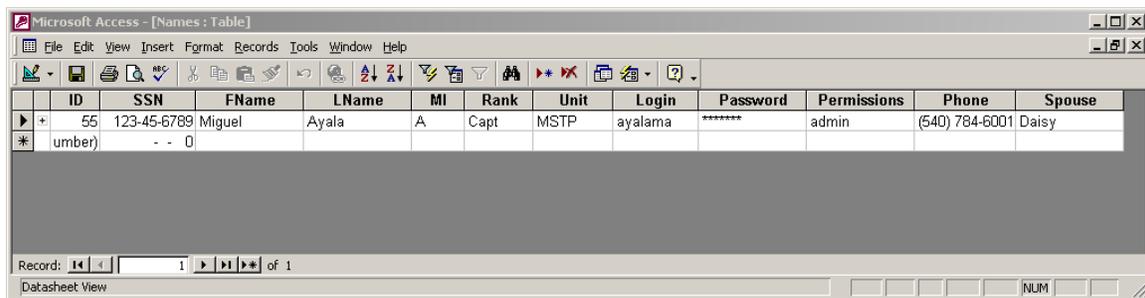
You are now ready to enter your field specifications. For each field you create, assign a Field Name (cannot include blank spaces), select the Data Type of the field (Text, Memo, etc.), and specify the Field Size to hold the largest number of expected characters. Assign the primary key to the SSN field.

After you are done designing your table, choose "Save" from the File menu. The Save dialog box appears and you can assign a name to your table and click "OK."

The new table is listed under the Table tab of the database dialog.

By double-clicking on the table in the database dialog you can open the table in "datasheet" view. This view presents the columns and rows of the table where you can enter your data. Just type your information in the columns and tab to the next column.

When you have finished with one record, a new blank line is appended to the table for entering the next record. After entering all your information, close the table.



**Figure 28 Adding Information into the Database**

When you are creating tables to be accessed through scripting it is usually not necessary to enter original data into the tables. You only need to create the table structure. You will normally fill the table with information collected from Web forms.



## XIII. Introduction to Structured Query Language (SQL)

Most database management systems provide easy and faster ways of extracting information from their tables. The common denominator of these methods is *SQL*, the *Structured Query Language* built into these systems. *SQL* is a full-service language for maintaining information in a database.

When using *SQL* methods of working with databases, you are relying on the database management system to perform the work. Rather than coding a server script to access tables or to maintain the data in the databases, this work has been passed to the DBMS. The script simply issues a request to the DBMS, which independently carries out the task. This method promotes the notion of a three-tier, client/server processing system where data access and database processing is localized to the database server.

### A. SQL Statements

#### 1. The SELECT Statement

The most useful of the SQL statements is the *SELECT* statement. This statement is used to select records from a database table. The selection can encompass the entire table with all of its fields, or it can be restricted to certain fields in certain records matching a given criteria. The group of selected records itself becomes a recordset that can be processed in the same fashion as used for an entire table. The general format for the *SELECT* statement is shown below:

```
SELECT * | field1[,field2]...
FROM TableName
WHERE condition
ORDER BY field1 [ASC/DESC] [, field2 [ASC/DESC] ]
```

The keyword *SELECT* is followed by one of two specifications identifying the fields of data to be selected from a table. An asterisk (\*) denotes that *all* fields are to be selected for each record. Otherwise, you can provide a list of field names, separated by commas, and only those data fields will be selected. The *FROM* clause identifies the table from which these records and fields are to be selected.

For example, the statement:

```
SELECT * FROM MyTable
```

selects all records from *MyTable* and includes all (\*) of the fields that make up a record. The resulting recordset is identical to the one returned when opening a full table. In contrast, the statement:

```
SELECT LastName,FirstName FROM MyTable
```

selects all records from the table, but only provides the fields named *LastName* and *FirstName* from among all the fields in the records. In this case the resulting recordset contains as many rows as there are records in the table, but only two columns.

### a. The WHERE Clause

In both of the above instances, all records are retrieved from the table. Only the fields that comprise a record differ. There may be cases where you do not want or need to retrieve each and every record in a table. You might wish to select only those records that meet certain condition. The *WHERE* clause is used for this purposes. The keyword *WHERE* is followed by one or more selection criteria. A common way of using this feature is to check for equality, that is, to look for a matching value in one of the record's fields. For example, if you are processing a set of Marines records based on the base in which they are located, you might wish to select only those records where the *State* field contains the value "Quantic". So, you would issue the SQL statement:

```
SELECT * FROM Marines WHERE Base='Quantic'
```

and the database management system would deliver only those records that had a matching criteria. You can, in fact, use any of the common conditional operators,

- = (equal to)
- <> (not equal to)
- < (less than)
- > (greater than)
- <= (less than or equal to)
- => (equal to or greater than)

to formulate your selection criterion. Plus, you can combine tests using the logical operators AND, OR, and NOT to expand or contract your selection:

```
SELECT * FROM Marines WHERE base='Quantic' OR base='Camp  
Lejeune'
```

Note in these examples that the selection criterion values are enclosed in single quotes (apostrophes). Any time you are matching against a database *text* field, the criterion value must be enclosed in single quotes (*WHERE base = 'Quantic'*). If you are testing against a *numeric* field, the data value is not enclosed in quotes (*WHERE Number > 10*). If you are testing against a *date/time* field, the criterion value is surrounded by # symbols (*WHERE TheDate > #1/1/01#*).

### b. The ORDER BY Clause

A *SELECT* statement can also include the *ORDER BY* clause in order to arrange, or sort, the set of records retrieved from a table. The *ORDER BY* clause identifies the names of fields on which to sort the records. If more than one field name is supplied, then sorting takes place in the order in which the names appear, separated by commas. The first field becomes the *major* sort field, the second field becomes the *intermediate* sort field, and the third field becomes the *minor* sort field.

Thus, you could arrange a set of names in order by last name, first name, and middle initial by using a *SELECT* statement that looks like the following:

```
SELECT * FROM Marines
ORDER By LastName,FirstName,MiddleInitial
```

You can also specify whether ordering is to take place in ascending or descending sequence by coding ASC or DESC following the field name. The default order is ascending (ASC), which doesn't need to be coded.

```
SELECT * FROM Marines
ORDER By LastName (DESC) ,FirstName (ASC) ,MiddleInitial
```

The *WHERE* and *ORDER BY* clauses are optional in a *SELECT* statement and either can appear. If both are included, however, the *WHERE* clause must precede the *ORDER BY* clause.



## XIV. Accessing a Database

The most basic purpose of ASP is to allow a website to connect to a database and show "Live data". It is called live data because ideally the database administrator will be updating the database routinely which will therefore automatically update the website. So how do you do it? Well, it's actually pretty simple. First, you need to understand that there are two ways to connect to a database. You can use a DSN or DSN-less connection, both accomplish the same thing. A DSN is a Data Source Name that is setup on the server. You can think of it as a shortcut to your database because it contains the driver and database path information to your database. If you have your website hosted by an outside company like most people do, you will need to contact them directly and ask them to setup the DSN for you. You will have to tell them where your database is located within your website and you will have to give the DSN a name.

### A. The Connection Object

Before you can retrieve any data from a database, you have to create a connection to that database. The ASP *Connection Object* contains the properties and methods necessary to make a link between a Web page and a database so that the database can be accessed through scripts appearing on the page. In order to make available these properties and methods to our own page we need to create a Connection Object for our own use.

### B. Including ActiveX Data Objects (ADO) Constants

The ADO are a set of objects that you can use to access databases. Each constant represents a numeric value. For example, in the *adovbs.inc* you will find the four constants for the recordset type defined as follows:

```
Const adOpenForwardOnly = 0
Const adOpenKeySet = 1
Const adOpenDynamic = 2
Const adOpenStatic = 3
```

So, why do we use constants? Consider the following two lines:

```
objRec.Open "Contact", strConnect, 0, 1, 2
```

or

```
objRec.Open "Contact", strConnect, adOpenForwardOnly, _
    adLockReadOnly, adCmdTable
```

Which one of these lines is easier to read? At the time of finding a mistake, the one with a meaningful constant name will actually tell you something.

You can use the *#include* directive to read another page in and make it part of the current page. 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. In order to have those constants available to your page, you have to include the file *adovbs.inc*, if

you are using VBScript or *adojavas.inc* if using JScript, if the file is in the same folder as your application.

```
<!--#include file="adovbs.inc"-->
```

A better solution may be to create a folder under your main folder and called it *include*. Then you can have all the include files there. If you decide to do that, your *#include* line looks more like:

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

### **C. Creating an ODBC connection**

ODBC stands for Open Database Connectivity. It allows programs to access lots of different kinds of databases in almost the exact way.

The method for creating a Connection Object is to use the VBScript *Set* statement, calling upon the Server Object to create a Connection Object for our script. For example

```
<%  
...  
Dim ConnectionObjectName  
Set ConnectionObjectName =  
    Server.CreateObject("ADODB.Connection")  
...  
%>
```

where *ConnectionObjectName* the name you want to assign to the connection object.

The Server Object uses its *CreateObject* method to create an ADODB Connection object. This object is assigned to a *ConnectionObject* name that we provide and through which we use the properties and methods of the object. The *ConnectionObject* can be any name of your choosing. In creating Connection objects, your only decision is the name to which you want it assigned. All of the other parameters in the Set statement must be coded as shown.

The primary need for a Connection object is to open a connection to a database. The Connection object has an *Open* method for just this purpose. This method is supplied with a "connection string" identifying the two pieces of information necessary for a script to link to a database:

1. the location of the database and
2. the type of database driver used to access it. Database connections can be made through either of two methods.

### **D. ODBC Data Source Name (DSN)-less Connection**

#### **1. DSN vs DSN-less**

A Data Source Name, or DSN, is a system name that is given to a database. Here is an example of a DSN connection:

```

<%
DIM Connect
Set Connect = Server.CreateObject("ADODB.Connection")
Connect.Open = "MyDSNDatabase"

%>

```

where *MyDSNDatabase* is the DSN name given to the database.

Personally, I prefer to use DSN-less connections to databases. The reason is that for maintenance and updating purposes, it is easier to make changes to database connections on your own rather than having to call or email your hosting company and wait for them to update your DSN.

There is a little more code involved with DSN-less connections, but it is worth it. Here is an example of a DSN-less connection:

```

<%
...
Dim dsn      'data source (database) name & path i.e.
Dim Conn     'Connection object

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

dsn="DBQ=" & Server.MapPath("UsersDb/Users.mdb") &
";Driver={Microsoft Access Driver (*.mdb)};"

Conn.Open dsn
...
%>

```

To avoid writing absolute path use the `Server.MapPath` method. This will accept a relative or virtual path and returns a physical path. Good when you do not have control of the server.

The database file is located under that folder in the same directory as the application.

DSN-less which requires no server setup, just a carefully constructed connection string as demonstrated below. DSN-less connections demand that that you know the name of the file (i.e. file based databases like Access). This is faster than a system DSN since it saves a trip to read the registry each attempt.

### **E. The Recordset Object**

The ASP *Recordset Object* contains the properties and methods necessary to extract data from a database table and to make that set of records available to a script. Normally, you need to create as many Recordset Objects as there are tables being accessed. In the

present example, we are accessing the single Accounts table from Database.mdb. Therefore, we need a single Recordset Object for our script. The general format for creating a Recordset Object is similar to the method used to create a Connection Object,

```
<%  
...  
Dim RecordsetObjectName  
Set RecordsetObjectName =  
    Server.CreateObject ("ADODB.Recordset")  
...  
%>
```

where, *RecordsetObjectName* is a name assigned to the object and by which it can be referenced in our script. The Server Object uses its CreateObject method to create an ADODB (Active Data Object DataBase) Recordset Object and assigns it to the name provided.

## 1. The Beginning of File (BOF) Object

If the value of the BOF property of a Recordset object is True, the current record pointer is positioned one record before the first record in the recordset. This is a read-only property. You can use the BOF property in conjunction with the EOF property to ensure that your recordset contains records and that you have not navigated beyond the boundaries of the recordset.

```
<%  
...  
If Not rs.BOF Then  
    ' There are records. Use the EOF property to loop  
    ' through all the records in the recordset and  
    ' display them to the screen.  
...  
%>
```

## 2. The End of File (EOF) Object

If the value of a Recordset object's EOF property is True, the current record pointer is positioned one record after the last record in the recordset. This is a read-only property. You can use the EOF property in conjunction with the BOF property to ensure that your recordset contains records and that you have not navigated beyond the boundaries of the recordset. Note that the value of EOF is also True if there are no records in the recordset.

```
<%  
    Do While Not rs.EOF  
%>
```

```

<tr>

  <td width="63"><%=rs("Rank")%></a></td>
  <td width="158"><%=rs("FName")%></td>
  <td width="41"><%=rs("MI")%></td>
  <td width="113">
  <a href=detail.asp?IDNum=<%=rs("ID")%>><%=rs("LName")%></a></td>
  <td width="54"><%=rs("Unit")%></td>
  <td width="152"><%=rs("Phone")%></td>

</tr>

<%
  rs.MoveNext
  Loop
%>

```

## F. Connecting to a Database

Once you have a *Connection* object already created, you are ready to connect to the database.

```

<%
...
  ConnectionObjectName.Open dsn
...
%>

```

### 1. Opening a Table

Once a Recordset Object is created, its built-in *Open* method is available to extract information from a database table and make it available to our script for processing. The general format for extracting ALL the information from a table is to open the entire table.

```

<%
...
RecordsetObject.Open "TableName", ConnectionObject
...
%>

```

Here, *TableName* is the name of a database table and *Connection object* is the name of an existing connection to the database containing the table.

```

<%
...
Set Conn = Server.CreateObject("ADODB.Connection")
Set rs = Server.CreateObject("ADODB.Recordset")
dsn="DBQ=" & Server.MapPath("UsersDb/Users.mdb") &
";Driver={Microsoft Access Driver (*.mdb)};"

```

```
Conn.Open dsN
rs.Open "Names", Conn
...
%>
```

With this statement in place, our script has extracted the entire Names table from the Users.mdb database and made it available for processing by the script.

## 2. Selecting Records

When the Names table is opened, the entire data can be loaded using the *Recordset* object. As in the Request.Form or Request.QueryString Collection, the data values contained in the recordset can be referenced through the notation rs ("FieldName"), where the *FieldName* is taken from the field names assigned when the table was created in Access. There is more than one data value that can be referenced through a recordset field name (that is the purpose of a database). For instance, under FName and the LName fields in the Names' table you could be referenced them as rs ("FName") and rs ("LName").

```
<tr>
<td width="63"><%=rs("Rank")%></a> </td>
<td width="158"><%=rs("FName")%> </td>
<td width="41"><%=rs("MI")%> </td>
<td width="113"><a href=detail.asp?IDNum=<%=rs("ID")%>>
<%=rs("LName")%> </a> </td>
<td width="54"><%=rs("Unit")%> </td>
<td width="152"><%=rs("Phone")%> </td>
</tr>
```

The HTML code of `<href>` will create a link to the `detail.asp` page passing the information `ID` and `LName` contained in the recordset named `rs`.

[Home](#)  
[Add User](#)  
[Delete User](#)  
[Search User](#)

1st Battalion Web Page



Alpha Roster					
Rank	First Name	MI	Last Name	Unit	Phone
LtCol	Michael	J	<a href="#">Burke</a>	MSTP	7037844972
Capt	Miguel	A	<a href="#">Ayala</a>	MSTP	7037846001
Capt	Sean	M	<a href="#">Sadtler</a>	MSTP	7037844315

**Figure 29 Selecting Records from a Database**

The above code builds this part of the page.

### 3. Iterating thorough a Recordset

It is obvious that a database will have more than one entry in the tables. You will want to iterate through all the records in order to find a specific record or to display them all. An appropriate VBScript iteration structure is the *Do While...Loop*.

When a recordset is first opened, the recordset cursor is positioned at the first record. As we continue our login and password checking (login page application), we need to advance the cursor through the recordset, examining each record, in turn, until we find one with a matching login and password. If we don't happen to find a matching record, the cursor will eventually be advanced *past* the end of the recordset .

Recordset objects have a property setting that indicates whether a recordset cursor has advanced beyond the last record in the recordset. This is the *EOF* (end-of-file) property. If there is an EOF property associated with the recordset, then the cursor has advanced beyond its last record; If there is NOT an EOF property associated with the recordset, then the cursor has not yet reached beyond the last record.

We can set up a program loop, then, that advances the recordset cursor from one record to the next, looking for a matching login and password. This loop will continue until we run out of records to check; until the Recordset object's EOF property is true. By using the VBScript Do While...Loop construct we can create this loop in the syntax:

Do While Not rs.EOF...Loop; that is, continue the loop so long as there is not an EOF property associated with our rs recordset.

We still need a method of advancing the recordset cursor from one record to the next.

Recordset objects provide this ability with the *MoveNext* method. Therefore, we need to use our rs.MoveNext method to advance the cursor each time through the loop.

Let's add these statements to previous script to create the structure of the loop that iterates through our recordset:

```
<%
Dim dsN
Dim Conn
Dim rs

Set Conn = Server.CreateObject("ADODB.Connection")
Set rs = Server.CreateObject("ADODB.Recordset")

dsN="DBQ=" & Server.MapPath("UsersDb/Users.mdb") &
";Driver={Microsoft Access Driver (*.mdb)};"

Conn.Open dsN

rs.Open "Names", Conn

Do While Not rs.EOF
    <tr>
        <td width="63"><%=rs("Rank")%></a> </td>
        <td width="158"><%=rs("FName")%> </td>
        <td width="41"><%=rs("MI")%> </td>
```

```
<td width="113"><a
href=detail.asp?IDNum=<%=rs ("ID") %>><%=rs ("LName") %>
</a> </td>
<td width="54"><%=rs ("Unit") %> </td>
<td width="152"><%=rs ("Phone") %> </td>
</tr>

<%
    rs.MoveNext
    Loop
%>
```

With this code all of the records under the *Names* table will be presented in the page (only the fields of Rank, Fname, MI, Lname, Unit, and Phone).

#### 4. Checking for Matching Records

For our login/password page, inside the Do While loop is where we need to check for a record with a matching login and password.

As the script iterates through its loop, we need to compare rs.Fields("login") with Request("login") and rs.Fields("Password") with Request("Password") for each record in the recordset, looking for matching values. If *both* matches are made, then we have found a valid login and password.

If we find matches to the values entered on the logon form, then we'll redirect the visitor to the show\_records.asp page, effectively ending the script.

If, on the other hand, no matches are found, then the recordset cursor gets advanced beyond the last record in the recordset, and the EOF property ends the loop. If the loop ends, this is a signal that no matching record was found. In this circumstance we redirect to the login.asp page. Another method is to create a message and let the user know that no matching set was found.

```
<%

Dim password, login
Password = Request("password")
Login = Request("login")

Dim dsn      'data source name & path    i.e. the database
            name and a path to it
Dim Conn    'Connection object
Dim rs      'recordset object
Dim strSQL  'structured query language statement

Set Conn = Server.CreateObject("ADODB.Connection")
Set rs = Server.CreateObject("ADODB.Recordset")

dsn="DBQ=" & Server.MapPath("UsersDb/Users.mdb") &
";Driver={Microsoft Access Driver (*.mdb)};"
```

```

Conn.Open dsN
rs.Open "Names", Conn

Do While Not rs.EOF
    If rs("login") = Login AND rs("password") = Password Then
        Response.Redirect "Show_Records.asp"
    Else
        Response.Redirect "Login.asp"
    End If
    Rs.MoveNext
Loop
...
%>

```

Lets see how SQL can be used to simplify our checks for matching login and passwords. We'll need to replace a portion of the previous code that we used previously to iterate through the recordset. We'll still need a Connection object to link to our database and we'll need a Recordset object for use in extracting records from it.

Rather than retrieving the entire table of accounts and passwords and looking through them one at a time, we'll use an SQL *SELECT* statement to attempt to retrieve a matching record. That is, we'll look for a record in which the Password value from the form matches the Password value in the table, and where the Login value from the form matches the Login value in the table. We'll compose an SQL statement to select this record.

If there is a matching record in the table, then this single record will be retrieved; if there is not a matching record in the table, then NO records will be retrieved. Thus, after issuing the *SELECT* statement, the mere existence of a retrieved record indicates that the user entered a matching account and password; on the other hand, an EOF property condition for the recordset indicates that the user did NOT enter a correct account and password.

In general, our statement must end up being in the following format in order to satisfy the syntactical rules of SQL:

```

SELECT * FROM Names WHERE Login='the form Account value'
AND Password='the form Password value'

```

That is, we want to SELECT all (\*) fields (the Login field and the Password field ) from the record in the Names table WHERE the value in the Login field of the table matches the Login value entered on the form, and the value in the Password field of the table matches the Password value entered on the form. Both the Login and Password criterion values must be enclosed in apostrophes since these are defined as text fields in the table. From a coding standpoint, then, we need to insert references inside the single quotes to the corresponding form values. In order to accomplish this, we can piece together an SQL statement from both the "literal" text strings that represent the fixed, unchanging code of

the *SELECT* statement along with the **variable** values taken from the *Request* object at the login form. The various pieces of the *SELECT* statement are

```
"SELECT * FROM Names WHERE Login='"  
Request ("Login")  
' AND Password='"  
Request ("Password")  
"'"
```

When these pieces are strung together, we will have a valid SQL *SELECT* statement that attempts to retrieve a matching record from the Accounts table. Let's concatenate these five elements and assign them to a variable named *strSQL*. We can either compose one long assignment statement,

```
strSQL = "SELECT * FROM Names WHERE Login='" &  
Request.Form("Account") & "' AND Password='" &  
Request.Form("Password") & "'"
```

or we can piece the statement together through separate concatenations:

```
SQL = "SELECT * FROM Names WHERE Login='"  
SQL = SQL & Request.Form("Account")  
SQL = SQL & "' AND Password='"  
SQL = SQL & Request.Form("Password")  
SQL = SQL & "'"
```

Assuming, for instance, that the visitor submitted the *login* aaaaa and the *Password* 11111, then the concatenations would produce the following *SELECT* statement assigned to variable *SQL*:

```
SELECT * FROM Names WHERE Login='aaaaa' AND Password='11111'
```

This is exactly the statement we need to perform the search. If, on the other hand, the visitor submits the *Login* bbbbb and the *Password* 2222, then the *SQL* variable would end up containing

```
SELECT * FROM Names WHERE Login='bbbbb' AND Password='22222'
```

As you can see, any *Login* and *Password* value submitted by the visitor gets plugged into the *SELECT* statement. The statement is assigned to variable *SQL*, which, in turn, is issued through the recordset *Open* statement.

```
<%
```

```
Dim dsn      'data source name & path   i.e. the database  
name and a path to it  
Dim Conn    'Connection object  
Dim rs      'recordset object  
Dim strSQL  'structured query language statement
```

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

```

Set rs = Server.CreateObject("ADODB.Recordset")

dsn="DBQ=" & Server.MapPath("UsersDb/Users.mdb") &
";Driver={Microsoft Access Driver (*.mdb)};"

Conn.Open dsn

strSQL = "SELECT * FROM Names WHERE Password = '" &
Request("password") & "' and Login = '" & Request("login")
& "'"

rs.Open strSQL, Conn

If RS.EOF = true Then
    Conn.Close 'closes the database connection
    set rs = Nothing 'sets the recordset equal to nothing
    set Conn= Nothing 'sets the connection object equal to
nothing
    response.redirect "Login.asp"
End If

%>

```

## 5. Closing Connections and Recordsets

Both Connection objects and Recordset objects have formal *Close* methods that can be applied when you are finished with either. Standard programming practice normally dictates that you close open items when you are done with them. Under ASP, however, this is not necessary. In fact, ASP automatically closes any open connections and recordsets when it finishes processing a page.

About the only time you need to close a recordset is when you intend to open it again on the same page. Or, you might close a recordset if you wish to use the same recordset name to open a different recordset. However, you should probably use different names for different recordsets just to keep them straight in your scripts. Whether you use the *Close* method is up to you.

```

<%
...
Conn.Close 'closes the database connection
set rs = Nothing 'sets the recordset equal to nothing
set Conn= Nothing 'sets the connection object equal to
nothing
...
%>

```

Both the objects are closed prior to redirecting the user to the welcome.asp page. They are both closed when the script ends following the loop.

## G. Updating a Database

Before you can make changes to a *Recordset*, you must make sure you do two things first:

1. Open the *Recordset* object with the *adOpenStatic* or *adOpenDynamic* cursor used in the second argument of the *Recordset* object's *Open* method.
2. Use *adLockOptimistic* for the lock type in the third argument of the *Recordset* object's *Open* method.

After the above steps, you will be ready to edit the *Recordset*.

### 1. Adding records

Adding a new record is relatively easy. It is just a matter of assigning values to the fields and that the bulk of it. You accomplish this with the *AddNew* method. This method adds a new, blank record to the database. Then you set the fields by assigning your data to the respective fields of the *Recordset*. When you are done assigning all the values, execute the *Recordset.Update* method to commit all changes to the record.

```
rs.AddNew

rs("SSN")=Request("SSN")
rs("FName") = Request("FName")
rs("MI") = Request("MI")
rs("LName") = Request("LName")
rs("Rank") = Request("Rank")
rs("Unit") = Request("Unit")
rs("login") = Request("login")
rs("Password") = Request("Password")
rs("Permissions") = Request("Permissions")
rs("Phone") = Request("Phone")

rs.Update
```

### 2. Updating records

If you know how to insert records, then updating then is piece of cake. First, you need to position the current pointer to the record that you wish to update. Use a proper SQL statement to achieve this (just like selecting a record for viewing). Modify the record by assigning the new values to the fields that need changes only. Finally, execute the *Update* method statement to write the changes back to the database.

```
IDNum = Request("IDNum")

 strSQL = "SELECT * FROM Names WHERE ID=" & IDNum

rs.Open strSQL, Conn, adOpenDynamic, adLockOptimistic

rs("SSN") = Request("SSN")
rs("FName") = Request("FName")
```



The SQL statement is selecting a record in which the ID number from the database matches the ID passed from the form. After selecting the record, the changes are made to the selected record.

```
rs("MI") = Request("MI")
rs("LName") = Request("LName")
rs("Rank") = Request("Rank")
rs("Unit") = Request("Unit")
rs("Phone") = Request("Phone")
```

```
rs.Update
```

### 3. Deleting Records

There are two easy ways to delete a record. Using the *Recordset.Delete* method or using the *DELETE* statement in SQL. I will be discussing the *Recordset.Delete* method to be consistent using the *Recordset* object. Like in the update part, you need to position the current pointer to the record that you wish to update. Use a proper SQL statement as well. Delete the record by using the *Recordset.Delete* method.

```
IDNum = Request("IDNum")

strSQL = "SELECT * FROM Names WHERE ID=" & IDNum

rs.Open strSQL, Conn, adOpenDynamic, adLockOptimistic

rs.Delete
```



## XV. Lab 5: Database Connection

In this Lab you will create a script to check a password and logon against the database you created in Chapter XIII and based on the permission you assigned to yourself we will extract information from the database and display it in the page.

Type the following code and save the file with an *check\_password.asp* extension under the virtual directory you created or under the default virtual root directory C:\Inetpub\wwwroot. Make sure that you point to the location of the database you created previously.

```
check_password.asp
```

```
<!--#include file="includes/adovbs.inc"-->
```

```
<%
```

```
Dim password, login
```

```
password = Request("password")
login = Request("login")
```

```
Dim dsn      'data source name & path    i.e. the database
            name and a path to it
Dim Conn     'Connection object
Dim rs       'recordset object
Dim strSQL  'structured query language statement
```

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

```
dsn="DBQ=" & Server.MapPath("UsersDb/Users.mdb") &
";Driver={Microsoft Access Driver (*.mdb)};"
```

Make sure you point to the location of your database

```
Conn.Open dsn
```

```
strSQL = "SELECT * FROM Names WHERE Password = '" &
password & "' and Login = '" & login & "'"
```

```
rs.Open strSQL, Conn
```

```
If RS.EOF = true Then
    Conn.Close 'closes the database connection
    set rs = Nothing 'sets the recordset equal to nothing
```

```
        set Conn= Nothing 'sets the connection object equal to
nothing
        response.redirect "Login.asp"
End If
```

```
Session ("SSN") = RS("SSN")
Session ("LName") = RS("LName")
Session ("FName") = RS("FName")
Session ("MI") = RS("MI")
Session ("Rank") = RS("Rank")
Session ("LoginPass") = True
Session ("Login") = RS("Login")
Session ("Permissions") = RS("Permissions")
```

```
Conn.Close 'closes the database connection
set rs = Nothing 'sets the recordset equal to nothing
set Conn= Nothing 'sets the connection object equal to
nothing
```

```
Permissions = Session("Permissions")
    If Permissions = "user" Then
        response.redirect "Show_Records.asp"
    ElseIf Permissions = "admin" Then
        response.redirect "Show_Records.asp"
    Else
        response.redirect "login.asp"
    End If
%>
```

*Show\_Records.asp*

```
<!-- #include file="includes/adovbs.inc" -->
```

```
<HTML>
```

```
<%
Dim dsn      'data source name & path    i.e. the database
name and a path to it
Dim Conn    'Connection object
Dim rs      'recordset object
Dim strSQL  'structured query language statement
Dim IDNum
dim selection
```

```
set Selection = Request.QueryString("Choice")
Set Conn = Server.CreateObject("ADODB.Connection")
Set rs = Server.CreateObject("ADODB.Recordset")
```

```

dsn="DBQ=" & Server.MapPath("UsersDb/Users.mdb") &
";Driver={Microsoft Access Driver (*.mdb)};"

Conn.Open dsn

 strSQL = "SELECT * FROM Names"

rs.Open strSQL, Conn, adOpenDynamic, adLockOptimistic,
adCmdText
%>
<table border=1 bgcolor=white width=655 bordercolor=Black
cellspacing=0 cellpadding=3>
<tr>
<td width="63"> Rank </td>
<td width="158"> First Name </td>
<th width="41"> MI </th>
<td width="113"> Last Name </td>
<td width="54"> Unit</td>
<th width="152"> Phone </th>

<%
  Do While Not rs.EOF
%>

<tr>

<td width="63"><%=rs("Rank")%></a> </td>
<td width="158"><%=rs("FName")%> </td>
<td width="41"><%=rs("MI")%> </td>
<td width="113"><%=rs("LName")%> </a> </td>
<td width="54"><%=rs("Unit")%> </td>
<td width="152"><%=rs("Phone")%> </td>

</tr>

<%
  rs.MoveNext
  Loop

Conn.Close 'closes the database connection
set rs= Nothing 'sets the recordset equal to nothing
set Conn = Nothing 'sets the connection object equal to
nothing

%>
</table>
</BODY>
</HTML>

```

Make the following changes to the Login.asp file

```
<html>

<head>
<title>Login</title>
</head>

<body>

<hr size="4" color="#000080">

<form method="post" name="LoginForm"
action="check_password.asp">
  <table border="2" width="400">
    <tr>
      <td width="91"><b>Login:&nbsp;</b></td>
      <td width="295"><input name="login"></td>
    </tr>
    <tr>
      <td width="91"><b>Password:</b></td>
      <td width="295"><input type="password"
name="password"></td>
    </tr>
  </table>
  <p><input type="submit" value="Login" name="submit">
<input type="reset" value="Reset" name="Reset"></p>
</form>

<hr size="4" color="#000080">

</body>

</html>
```

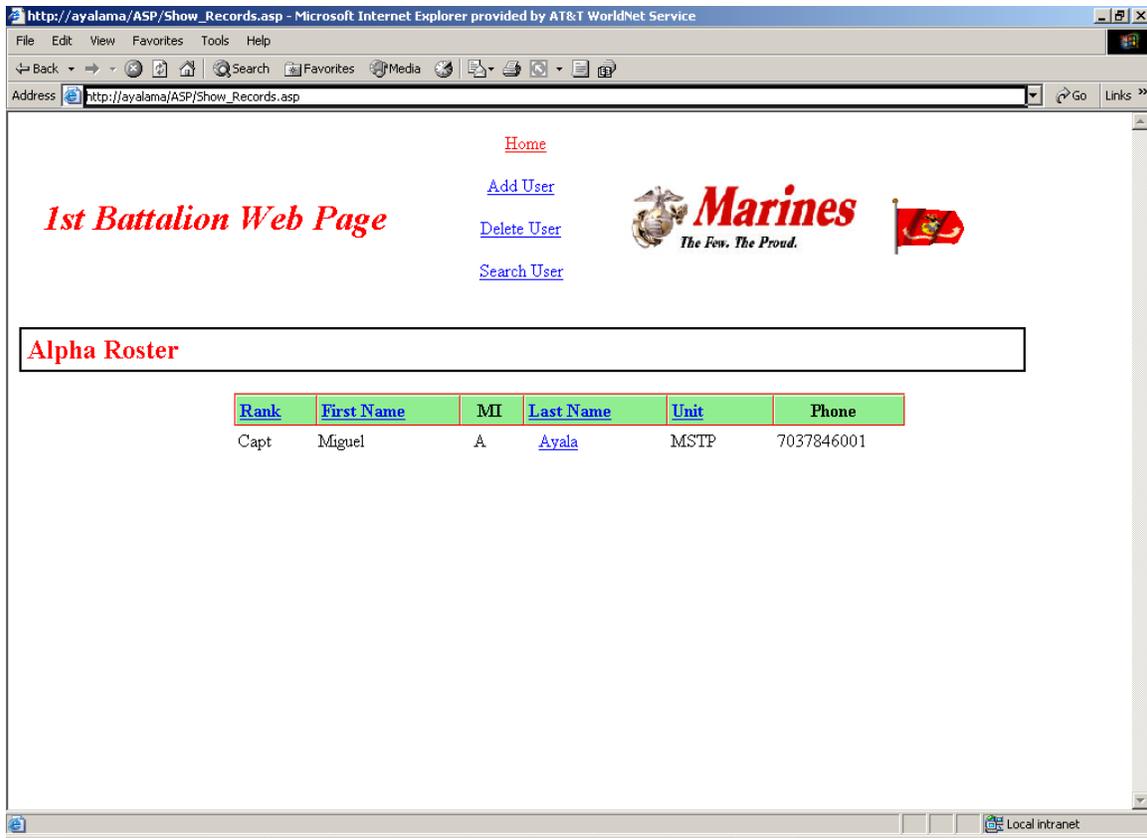
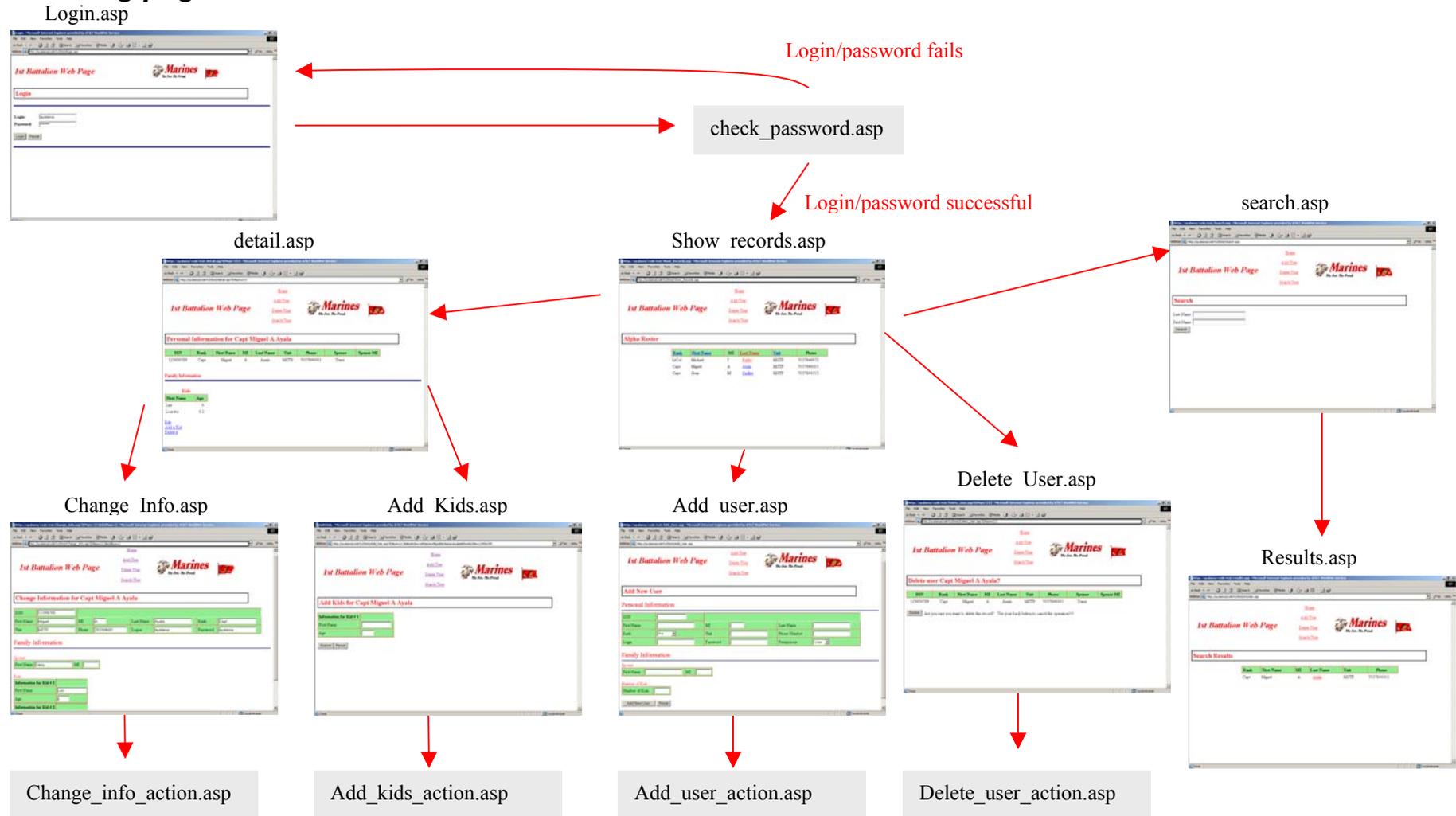


Figure 30 Lab 5 Connecting to a Database and displaying results



## XVI. Putting it all together

### A. Linking pages





## XVII. Lab 6: Alpha Roster (Final Product)

Create an alpha roster web application with the following functionalities:

1. **Login Page:** Ask the user for a logon id and a password. The page must be able to process the information either on its own or pass it to another .asp file to process the information. That pair needs to be check against information in a database. The end product is to be able to check the logon id and password provided by the user against a database.
2. **Database Connectivity:** The application must be able to have connectivity with a database where all the information about the users reside.
3. **Show Records:** Display the appropriate information to the user, retrieving it from the database.
4. **Update Records:** The application must be able to update existing information from the database.
5. **Add Records:** The application must provide the functionality of adding new users to the database, collecting at least the following information:
  - SSN
  - Rank
  - First Name
  - MI
  - Last Name
  - Unit
  - Phone Number
  - Spouse Name
  - Kids Name
  - Kids Age
  - Permission for the database (User or Admin)
  - Password
  - Logon Id

This functionality must be granted to an administrator

6. **Delete Records:** The application must provide the functionality of deleting users. This functionality must be granted to an administrator.
7. **Sort Records:** When displaying the records, the functionality of sorting them should be available. The application should sort the records by:
  - Rank

- First Name
- Last Name
- Unit

**8. Search Records:** The application should have the functionality of searching a record under certain parameters. At least should provide the functionality of searching records by:

- Last Name
- First Name

The application should return all the records that matches the criteria given by the user.

## Appendix A. Final Product Description

Final product:

Upon completion of the class the student should be able to build the following:

You have an alpha roster database with all the members of the Battalion. The information in the database is composed of:

- Rank
- First Name
- MI
- Last Name
- SSN
- Unit
- Phone
- Login
- Password
- Permission
- Spouse Name
- Children Name

You should be able to create a web application that reads all the information from the database and presents it to you. The application must have a login page that when the user types the userid and password it validates and authenticate the user by looking into the database for that user. Access to the rest of the pages is allowed if the user is found, otherwise redirect back to the login page. Based on the user's permission (user or admin) allow the access to different functionalities of the home page. In the home page the user should have the functionalities of searching for an individual in the alpha roster and get the details of that user (for a user permission). For an admin permission the user can add, delete, modify as well as search for a user.

### *1st Battalion Web Page*



Login

Login:   
Password:

**Figure A- 1 Login Page**

[Home](#)  
[Add User](#)  
[Delete User](#)  
[Search User](#)



**Alpha Roster**

Rank	First Name	MI	Last Name	Unit	Phone
Capt	Miguel	A	Ayala	MSTP	7037846001

**Figure A- 2 Alpha Roster**

[Home](#)  
[Add User](#)  
[Delete User](#)  
[Search User](#)



**Personal Information for Capt Miguel A Ayala**

SSN	Rank	First Name	MI	Last Name	Unit	Phone	Spouse	Spouse MI
123456789	Capt	Miguel	A	Ayala	MSTP	7037846001	Daisy	

**Family Information**

**Kids**

First Name	Age
Luis	6
Lourdes	0.2

[Edit](#)  
[Add a Kid](#)  
[Delete it](#)

**Figure A- 3 Detail Information**

[Search User](#)

**Change Information for Capt Miguel A Ayala**

SSN:	123456789						
First Name:	Miguel	MI:	A	Last Name:	Ayala	Rank:	Capt
Unit:	MSTP	Phone:	7037846001	Logon:	ayalama	Password:	ayalama

**Family Information**

**Spouse**

First Name	Daisy	MI	
------------	-------	----	--

**Kids**

<b>Information for Kid # 1</b>	
First Name	Luis
Age	6
<b>Information for Kid # 2</b>	
First Name	Lourdes
Age	0.2

Make Changes

**Figure A- 4 Change Information**

*1st Battalion Web Page*

[Home](#)  
[Add User](#)  
[Delete User](#)  
[Search User](#)



**Information updated on Capt Miguel A Ayala**

SSN	Rank	First Name	MI	Last Name	Unit	Phone	Spouse	Spouse MI
123456789	Capt	Miguel	A	Ayala	MSTP	7037846001	Daisy	

**Family Information**

**Kids**

First Name	Age
Luis	6
Lourdes	0.2

OK

**Figure A- 5 Updated information**

[Home](#)  
[Add User](#)  
[Delete User](#)  
[Search User](#)

*1st Battalion Web Page*



**Search**

Last Name:

First Name:

Figure A- 6 Search

[Home](#)  
[Add User](#)  
[Delete User](#)  
[Search User](#)

*1st Battalion Web Page*



**Delete user Capt Miguel A Ayala?**

SSN	Rank	First Name	MI	Last Name	Unit	Phone	Spouse	Spouse MI
123456789	Capt	Miguel	A	Ayala	MSTP	7037846001	Daisy	

Are you sure you want to delete this record? Use your back button to cancel the operation!!!!

Figure A- 7 Delete User

[Home](#)  
[Add User](#)  
[Delete User](#)  
[Search User](#)

*1st Battalion Web Page*



**Add Kids for Capt Miguel A Ayala**

Information for Kid # 1	
First Name	<input type="text"/>
Age	<input type="text"/>

Figure A- 8 Add Kid

[Home](#)  
[Add User](#)  
[Delete User](#)  
[Search User](#)

*1st Battalion Web Page*



**Search Results**

Rank	First Name	MI	Last Name	Unit	Phone
Capt	Miguel	A	<a href="#">Ayala</a>	MSTP	7037846001

**Figure A- 9 Search Results**



---

## Appendix B. VBScript Reference

---

---

### 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	<code>vbEmpty</code>	Empty (This is the type for a variable that has not been used yet. In other words, Empty is the default datatype.)
1	<code>vbNull</code>	Null (No valid data)
2	<code>vbInteger</code>	Integer
3	<code>vbLong</code>	Long
4	<code>vbSingle</code>	Single
5	<code>vbDouble</code>	Double
6	<code>vbCurrency</code>	Currency
7	<code>vbDate</code>	Date
8	<code>vbString</code>	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.)

### Typcasting Functions

Typcasting 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 year
	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,compare type)` 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 choicen  
        codeblockn  
    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
```



## Appendix C. HTML Tags

### Reference **HTML Cheat sheet**

---

#### Basic Tags

**<html></html>**

Creates an HTML document

**<head></head>**

Sets off the title and other information that isn't displayed on the Web page itself

**<body></body>**

Sets off the visible portion of the document

#### Header Tags

**<title></title>**

Puts the name of the document in the title bar

#### Body Attributes

**<body bgcolor=?>**

Sets the background color, using name or hex value

**<body text=?>**

Sets the text color, using name or hex value

**<body link=?>**

Sets the color of links, using name or hex value

**<body vlink=?>**

Sets the color of followed links, using name or hex value

**<body alink=?>**

Sets the color of links on click

#### Text Tags

**<pre></pre>**

Creates preformatted text

**<h1></h1>**

Creates the largest headline

**<h6></h6>**

Creates the smallest headline

**<b></b>**

Creates bold text

`<i></i>`

Creates italic text

`<tt></tt>`

Creates teletype, or typewriter-style text

`<cite></cite>`

Creates a citation, usually italic

`<em></em>`

Emphasizes a word (with italic or bold)

`<strong></strong>`

Emphasizes a word (with italic or bold)

`<font size=?></font>`

Sets size of font, from 1 to 7)

`<font color=?></font>`

Sets font color, using name or hex value

## Links

`<a href="URL"></a>`

Creates a hyperlink

`<a href="mailto:EMAIL"></a>`

Creates a mailto link

`<a name="NAME"></a>`

Creates a target location within a document

`<a href="#NAME"></a>`

Links to that target location from elsewhere in the document

## Formatting

`<p></p>`

Creates a new paragraph

`<p align=?>`

Aligns a paragraph to the left, right, or center

`<br>`

Inserts a line break

`<blockquote>`

`</blockquote>`

Indents text from both sides

`<dl></dl>`

Creates a definition list

`<dt>`

Precedes each definition term

**<dd>**

Precedes each definition

**<ol></ol>**

Creates a numbered list

**<li></li>**

Precedes each list item, and adds a number

**<ul></ul>**

Creates a bulleted list

**<div align=?>**

A generic tag used to format large blocks of HTML, also used for stylesheets

## Graphical Elements

****

Adds an image

****

Aligns an image: left, right, center; bottom, top, middle

****

Sets size of border around an image

**<hr>**

Inserts a horizontal rule

**<hr size=?>**

Sets size (height) of rule

**<hr width=?>**

Sets width of rule, in percentage or absolute value

**<hr noshade>**

Creates a rule without a shadow

## Tables

**<table></table>**

Creates a table

**<tr></tr>**

Sets off each row in a table

**<td></td>**

Sets off each cell in a row

**<th></th>**

Sets off the table header (a normal cell with bold, centered text)

### Table Attributes

**<table border=#>**

Sets width of border around table cells

**<table cellspacing=#>**

Sets amount of space between table cells

**<table cellpadding=#>**

Sets amount of space between a cell's border and its contents

**<table width=# or %>**

Sets width of table — in pixels or as a percentage of document width

**<tr align=?> or <td align=?>**

Sets alignment for cell(s) (left, center, or right)

**<tr valign=?> or <td valign=?>**

Sets vertical alignment for cell(s) (top, middle, or bottom)

**<td colspan=#>**

Sets number of columns a cell should span

**<td rowspan=#>**

Sets number of rows a cell should span (default=1)

**<td nowrap>**

Prevents the lines within a cell from being broken to fit

### Frames

**<frameset></frameset>**

Replaces the <body> tag in a frames document; can also be nested in other framesets

**<frameset rows="value,value">**

Defines the rows within a frameset, using number in pixels, or percentage of width

**<frameset cols="value,value">**

Defines the columns within a frameset, using number in pixels, or percentage of width

**<frame>**

Defines a single frame — or region — within a frameset

**<noframes></noframes>**

Defines what will appear on browsers that don't support frames

### Frames Attributes

**<frame src="URL">**

Specifies which HTML document should be displayed

**<frame name="name">**

Names the frame, or region, so it may be targeted by other frames

**<frame marginwidth=#>**

Defines the left and right margins for the frame; must be equal to or greater than 1

**<frame marginheight=#>**

Defines the top and bottom margins for the frame; must be equal to or greater than 1

**<frame scrolling=VALUE>**

Sets whether the frame has a scrollbar; value may equal "yes," "no," or "auto." The default, as in ordinary documents, is auto.

**<frame noresize>**

Prevents the user from resizing a frame

## Forms

For functional forms, you'll have to run a [CGI script](#). The HTML just creates the appearance of a form.

**<form></form>**

Creates all forms

**<select multiple name="NAME" size=?></select>**

Creates a scrolling menu. Size sets the number of menu items visible before you need to scroll.

**<option>**

Sets off each menu item

**<select name="NAME"></select>**

Creates a pulldown menu

**<option>**

Sets off each menu item

**<textarea name="NAME" cols=40 rows=8></textarea>**

Creates a text box area. Columns set the width; rows set the height.

**<input type="checkbox" name="NAME">**

Creates a checkbox. Text follows tag.

**<input type="radio" name="NAME" value="x">**

Creates a radio button. Text follows tag

**<input type="text" name="foo" size=20>**

Creates a one-line text area. Size sets length, in characters.

**<input type="submit" value="NAME">**

Creates a Submit button

**<input type="image" border=0 name="NAME" src="name.gif">**

Creates a Submit button using an image

**<input type="reset">**

Creates a Reset button



## Appendix D. ASP Objects

### Response Object

The ASP Response object is used to send output to the user from the server. Its collections, properties, and methods are described below:

#### Collections

Collection	Description
<a href="#">Cookies</a>	Sets a cookie value. If the cookie does not exist, it will be created, and take the value that is specified

#### Properties

Property	Description
<a href="#">Buffer</a>	Specifies whether to buffer the page output or not
<a href="#">CacheControl</a>	Sets whether a proxy server can cache the output generated by ASP or not
<a href="#">Charset</a>	Appends the name of a character-set to the content-type header in the Response object
<a href="#">ContentType</a>	Sets the HTTP content type for the Response object
<a href="#">Expires</a>	Sets how long (in minutes) a page will be cached on a browser before it expires
<a href="#">ExpiresAbsolute</a>	Sets a date and time when a page cached on a browser will expire
<a href="#">IsClientConnected</a>	Indicates if the client has disconnected from the server
<a href="#">Pics</a>	Appends a value to the PICS label response header
<a href="#">Status</a>	Specifies the value of the status line returned by the server

#### Methods

Method	Description
<a href="#">AddHeader</a>	Adds a new HTTP header and a value to the HTTP response
<a href="#">AppendToLog</a>	Adds a string to the end of the server log entry
<a href="#">BinaryWrite</a>	Writes data directly to the output without any character conversion

<a href="#">Clear</a>	Clears any buffered HTML output
<a href="#">End</a>	Stops processing a script, and returns the current result
<a href="#">Flush</a>	Sends buffered HTML output immediately
<a href="#">Redirect</a>	Redirects the user to a different URL
<a href="#">Write</a>	Writes a specified string to the output

## Request Object

When a browser asks for a page from a server, it is called a request. The ASP Request object is used to get information from the user. Its collections, properties, and methods are described below:

### Collections

Collection	Description
ClientCertificate	Contains all the field values stored in the client certificate
<a href="#">Cookies</a>	Contains all the cookie values sent in a HTTP request
<a href="#">Form</a>	Contains all the form (input) values from a form that uses the post method
<a href="#">QueryString</a>	Contains all the variable values in a HTTP query string
<a href="#">ServerVariables</a>	Contains all the server variable values

### Properties

Property	Description
<a href="#">TotalBytes</a>	Returns the total number of bytes the client sent in the body of the request

### Methods

Method	Description
<a href="#">BinaryRead</a>	Retrieves the data sent to the server from the client as part of a post request and stores it in a safe array

## Application Object

An application on the Web may be a group of ASP files. The ASP files work together to perform some purpose. The Application object in ASP is used to tie these files together. The Application object is used to store and access variables from any page, just like the Session object. The difference is that ALL users share one Application object, while with Sessions there is one Session object for EACH user.

The Application object should hold information that will be used by many pages in the application (like database connection information). This means that you can access the information from any page. It also means that you can change the information in one place and the changes will automatically be reflected on all pages.

The Application object's collections, methods, and events are described below:

### Collections

Collection	Description
<a href="#">Contents</a>	Contains all the items appended to the application through a script command
<a href="#">StaticObjects</a>	Contains all the objects appended to the application with the HTML <object> tag

### Methods

Method	Description
<a href="#">Contents.Remove</a>	Deletes an item from the Contents collection
<a href="#">Contents.RemoveAll()</a>	Deletes all items from the Contents collection
<a href="#">Lock</a>	Prevents other users from modifying the variables in the Application object
<a href="#">Unlock</a>	Enables other users to modify the variables in the Application object (after it has been locked using the Lock method)

### Events

Event	Description
<a href="#">Application_OnEnd</a>	Occurs when all user sessions are over, and the application ends
<a href="#">Application_OnStart</a>	Occurs before the first new session is created (when the Application object is first referenced)

## Session Object

When you are working with an application, you open it, do some changes and then you close it. This is much like a Session. The computer knows who you are. It knows when you start the application and when you end. But on the internet there is one problem: the web server does not know who you are and what you do because the HTTP address doesn't maintain state.

ASP solves this problem by creating a unique cookie for each user. The cookie is sent to the client and it contains information that identifies the user. This interface is called the Session object.

The Session object is used to store information about, or change settings for a user session. Variables stored in the Session object hold information about one single user, and are available to all pages in one application. Common information stored in session variables are name, id, and preferences. The server creates a new Session object for each new user, and destroys the Session object when the session expires.

The Session object's collections, properties, methods, and events are described below:

### Collections

Collection	Description
<a href="#">Contents</a>	Contains all the items appended to the session through a script command
<a href="#">StaticObjects</a>	Contains all the objects appended to the session with the HTML <object> tag

### Properties

Property	Description
<a href="#">CodePage</a>	Specifies the character set that will be used when displaying dynamic content
<a href="#">LCID</a>	Sets or returns an integer that specifies a location or region. Contents like date, time, and currency will be displayed according to that location or region
<a href="#">SessionID</a>	Returns a unique id for each user. The unique id is generated by the server
<a href="#">Timeout</a>	Sets or returns the timeout period (in minutes) for the Session object in this application

### Methods

Method	Description
<a href="#">Abandon</a>	Destroys a user session
<a href="#">Contents.Remove</a>	Deletes an item from the Contents collection

<a href="#">Contents.RemoveAll()</a>	Deletes all items from the Contents collection
--------------------------------------	--

## Events

<b>Event</b>	<b>Description</b>
<a href="#">Session_OnEnd</a>	Occurs when a session ends
<a href="#">Session_OnStart</a>	Occurs when a session starts

## Server Object

The ASP Server object is used to access properties and methods on the server. Its properties and methods are described below:

### Properties

Property	Description
<a href="#">ScriptTimeout</a>	Sets or returns the maximum number of seconds a script can run before it is terminated

### Methods

Method	Description
<a href="#">CreateObject</a>	Creates an instance of an object
<a href="#">Execute</a>	Executes an ASP file from inside another ASP file
<a href="#">GetLastError()</a>	Returns an ASPError object that describes the error condition that occurred
<a href="#">HTMLEncode</a>	Applies HTML encoding to a specified string
<a href="#">MapPath</a>	Maps a specified path to a physical path
<a href="#">Transfer</a>	Sends (transfers) all the information created in one ASP file to a second ASP file
<a href="#">URLEncode</a>	Applies URL encoding rules to a specified string

## The ASPError Object

The ASPError object is implemented in ASP 3.0 and it is only available in IIS5. The ASP Error object is used to display detailed information of any error that occurs in scripts in an ASP page. The ASPError object is created when Server.GetLastError is called, so the error information can only be accessed by using the Server.GetLastError method.

### Example

```
<html>
<body>

<%
'The following line creates an error
dim i for i=1 to 1 next

'Call the GetLastError() method to trap the error
dim objerr
set objerr=Server.GetLastError()

'The variable objerr now contains the ASPError object
response.write("ASP Code=" & objerr.ASPCode)
response.write("<br />")
response.write("Number=" & objerr.Number)
response.write("<br />")
response.write("Source=" & objerr.Source)
response.write("<br />")
response.write("Filename=" & objerr.File)
response.write("<br />")
response.write("LineNumber=" & objerr.Line)
%>
</body>
</html>
```

The ASPError object's properties are described below (all properties are read-only):  
**Note:** The properties below can only be accessed through the Server.GetLastError() method.

## Properties

<b>Property</b>	<b>Description</b>
<a href="#">ASPCode</a>	Returns an error code generated by IIS
<a href="#">ASPDescription</a>	Returns a detailed description of the error (if the error is ASP-related)
<a href="#">Category</a>	Returns the source of the error (was the error generated by ASP? By a scripting language? By an object?)
<a href="#">Column</a>	Returns the column position within the file that generated the error
<a href="#">Description</a>	Returns a short description of the error
<a href="#">File</a>	Returns the name of the ASP file that generated the error
<a href="#">Line</a>	Returns the line number where the error was detected
<a href="#">Number</a>	Returns the standard COM error code for the error
<a href="#">Source</a>	Returns the actual source code of the line where the error occurred



## Appendix E. Helpful Websites

Some helpful web sites for tutorials and information:

<http://asp-dev.aspin.com/home/tutorial>  
<http://www.free-webmaster-tools.com/programming/asp/code.html>  
<http://www.asptutorial.info/>  
<http://www.aspin.com/home/tutorial>  
<http://devguru.aspin.com/>  
[http://www.techiwarehouse.com/ASP/ASP\\_Tutorial.html](http://www.techiwarehouse.com/ASP/ASP_Tutorial.html)  
<http://www.maconstate.edu/msconline/tutorials/>





**United States Marine Corps  
MAGTF Staff Training Program  
2084 South Street  
Quantico, VA 22134-5001**

**Point of Contacts**



**LtCol Michael J. Burke**  
[burkemi@mstp.quantico.usmc.mil](mailto:burkemi@mstp.quantico.usmc.mil)  
(703) 784-4972 DSN 278

**Capt Miguel A. Ayala**  
[ayalama@mstp.quantico.usmc.mil](mailto:ayalama@mstp.quantico.usmc.mil)  
(703) 784-6001 DSN 278

**DOC MSTP web site:**  
[www.mstp.usmc.mil](http://www.mstp.usmc.mil)

*"Training The First To Fight"*