

Dimeric Server Pages (DSP) Syntax Guide

Content

1 Dimeric Server Pages Syntax Guide 1

1.1 HTML Text 1

1.2 Hidden Comment 1

1.3 Expression 2

1.4 Encoded Expression 2

1.5 Code Fragment 3

1.6 Directives 4

1.6.1 Directive: begin 4

1.6.2 Directive: define 5

1.6.3 Directive: else 6

1.6.4 Directive: end 6

1.6.5 Directive: endif 7

1.6.6 Directive: fetch 7

1.6.7 Directive: finalization 7

1.6.8 Directive: if 8

1.6.9 Directive: include 9

1.6.10 Directive: initialization 9

1.6.11 Directive: interface 9

1.6.12 Directive: options 10

1.6.13 Directive: run 11

1.6.14 Directive: uses 11

1.6.15 Directive: uses_interface 12

2 Index 13

1 Dimeric Server Pages Syntax Guide

You can contact us and find out more about DSP and our other products at <http://www.Dimeric.com/>

Copyright (c) 2002 by Dimeric, L.L.C. All rights reserved.

1.1 HTML Text

Overview

All plain HTML text is generated into the resulting page.

Syntax

any HTML text

Example 1

```
<html>
<body>
<!-- Isn't this plain? -->
This is a valid DSP, even though it contains only plain HTML.
</body>
</html>
```

Example 2

```
<html>
<body>
<!-- Generated at <%= Now %> -->

This is still a very plain HTML file.
</body>
</html>
```

Displays in the page source:

```
<html>
<body>
<!-- Generated at 3/28/2000 11:42:27 PM -->
This is still a very plain HTML file.
</body>
</html>
```

Description

All plain HTML is sent directly to the client as written. This includes all tags, plain text and HTML comments. Text outside of matched DSP tags is not interpreted by DSP in any way.

See Also

Expression

1.2 Hidden Comment

Overview

Documents the DSP file, but is not sent to the client.

Syntax

<%-- comment --%>

Example

```
<html>
<head><>Comment Test</title></head>
<body>
<h1>Comment Test</h1>
<%-- This comment will not appear in the page source or generated Delphi unit --%>
<% // This comment will not appear in the page source either, but will appear in the
generated Delphi unit %>
<!-- This comment will appear in the page source (and, of course, the generated Delphi
unit) -->
</body>
</html>
```

Description

A hidden comment is essentially ignored. It is not generated into the resulting page and is not sent to the client. The DSP engine does not process anything between the `<%--` and `--%>` characters. It is not displayed in the resulting page or in the page source. It simply documents the DSP file.

You can use any characters in the body of the comment until the comment terminator, `--%>`, is reached.

See Also

Code Fragment

1.3 Expression

Overview

Contains a valid Delphi expression that is Variant-compatible and also convertible to a string.

Syntax

```
<%= expression %>
```

Example

```
PI squared is <%= PI * PI %><br>  
<a href="<%=Table1URL.AsString%">">Some link</a>
```

Displays in the page source:

```
PI squared is 9.86960440108936<br>  
<a href="http://community.borland.com">Some link</a>
```

Description

The expression must be a valid Delphi expression that is Variant compatible. That is, it must be an expression that is directly convertible to a Variant. Once it is converted to a Variant, it is then converted to a string and so it not only must be assignable to a Variant, but must then be convertible to a string.

It is important to remember that this is an expression and not a statement. That is, it should not end in a semicolon. The same expression in a code block would require a semicolon. Also, the expression can be as simple as a string literal or number. It could also be a complex multipart expression.

You may assume that the expression is evaluated as if inside the following typecast:

```
string(Variant(expression))
```

See Also

Encoded Expression, Code Fragment

1.4 Encoded Expression

Overview

Contains a valid Delphi expression that is Variant-compatible and also convertible to a string. It is then HTML-encoded.

Syntax

```
<%| expression %>
```

Example 1

HTML has the following special characters: `<%| '& < > ' ' " ' %>`

Displays in the page source:

HTML has the following special characters: `& < > " '`

Example 2

```
<% Name := 'Jimmy "the weasel"' ; %>  
<input type=text value="<%| Name %>">
```

Displays in the page source:

```
<input type=text value="Jimmy &quot;the weasel&quot;">
```

Description

The encoded expression is identical to an expression, except that the resulting string is HTML-encoded. This means that the following transformations are made:

Character	Converted to
&	&
<	<
>	>
"	"
'	'

This is especially useful when the text being generated is not strictly under the control of the program. For example, when text is coming from a database or file.

See Also

Expression, Code Fragment

1.5 Code Fragment

Overview

Contains a Delphi code fragment.

Syntax

```
<% Delphi code %>
```

Example 1

```
<% for i := 1 to 10 do begin %> <%= i %><br> <% end; %>
```

Example 2

```
<%  
  if ( Time >= StrToTime('6am') )  
    and ( Time < StrToTime('6pm') ) then begin  
    s := 'day';  
%>  
    
<%  
  end else begin  
    s := 'night';  
%>  
    
<%  
  end;  
%>  
it is <%=s%>time
```

Description

A code fragment can contain any valid Delphi statement or statements. You can reference any variables defined in the DSP, call functions, create and destroy objects, use looping and conditional statements. As you see from the examples, you may even use loops and conditional statements across code tags.

In example 1, you will see the numbers 1 through 10 each on their own line (assuming that *i* is an integer variable). In example 2, you will either see the image in sun.gif followed by the words "it is daytime" or the image in moon.gif followed by the words "it is nighttime," but not both.

Any plain text or HTML elements must be outside the code tag.

See Also

Expression

1.6 Directives

Overview

Directives alter the generation of the DSP page from conditional compilation, to including other source files, to altering which section of the generated unit code is placed in.

General Syntax

```
<%@ directive-name directive-attributes %>
```

See Also

Expression

1.6.1 Directive: begin

Overview

Opens a scope (global, nested or main).

Syntax

```
<%@ begin { global | nested | main } %>
```

Example

```
<%@ begin global %>
<%
  // This function is at file scope
  function Factorial (i: integer): string;
  begin
    if i <= 1 then
      Result := 1
    else
      Result := i * Factorial (i - 1);
    end;
  %>
<%@ end global %>

<%@ begin nested %>
<%
  // this procedure is nested in the main service routine
  procedure HtmlBold (const v: Variant);
  begin
  %>
    <%= v %>
  %>
  end; // HtmlBold
  %>
<%@ end nested %>

<html>
<head><>Factorial example</title></head>
<body>
<%@ begin nested %>
  var
    i: integer; // declared nested in the main service routine
<%@ end nested %>
<% for i := 0 to 10 do begin %>
  Factorial of <%= i %> is <%= HtmlBold(Factorial(i)) %><br>
<% end; %>
</body>
</html>
```

Description

The begin directive must specify a scope of either global, nested or main. The begin and end scope directives must be matched but can be properly nested inside one another.

The begin and end scope directives change the context for everything that comes between them. Understanding the meaning of the scopes is easier when you understand how a DSP file is translated into a Delphi unit.

The main scope corresponds to the body of the Service procedure. That is, all HTML text, code fragments, etc., that are not between begin and end scope directives, are placed inside the Service procedure of the Delphi unit generated from the DSP. Explicitly opening the main scope is particularly useful in an include file that may be included inside a nested or global scope. The nested scope is the declaration section of the Service procedure. For example, you can declare variables in a code block inside a nested scope, as in the example above. You can also declare types, constants, nested procedures and nested functions.

The global scope is in the implementation section of the generated Delphi unit and is outside the Service procedure. Whatever you put in the global scope will be placed in the implementation section of the Delphi unit generated from the DSP.

Note that you may have HTML and plain text, code, expressions and directives inside a scope, as illustrated by the `HtmlBold` function in the example above.

Also note that you may have as many of each kind of block as you wish. They will be ordered within their scope by their relative position in the DSP file, although having only one of each section per file is recommended. More than one of each kind of scope can lead to confusion. Use the scope directives with care.

Also, scope blocks are irrelevant for many directives, such as the `uses` directive. The `uses` directive places units in the `uses` clause of the generated unit, and is not effected by its location in the file. This is also true of the `initialization`, `finalization`, `interface`, and `options` directives. They behave the same regardless of where in the file they are placed.

See Also

Directive: `end`, Expression, Code Fragment

1.6.2 Directive: define

Overview

Defines a preprocessor symbol.

Syntax

```
<%@ define name1["expression1"] name2["expression2"] ... %>
```

Example

```
<%@ define Debug %> <%@ define ExtraUnit="DSStringUtil" %>
<%@ if Debug %> <%@ uses $ExtraUnit %> <%@ endif %>
```

Description

The `define` directive can define a preprocessor symbol. Symbols are managed in a case-insensitive environment. Each symbol has a value that is a Variant data type. If no value is specified, then the value defaults to True. It is acceptable to re-define an existing symbol - the new value simply replaces any previously assigned value. There is no way to undefine a symbol, however defining it to the built-in symbol `Null` is functionally equivalent.

Symbols defined in one page are not carried over to another page. The following symbols are predefined:

Name	Value
True	True
False	False
Null	Null
Windows	True
Win32	True
DelphiVersion	5, 6 or 7 depending on the Delphi version of the translator
DSPVersion	2.0

The value of the name attribute, the part after the equals-sign that is in double-quotes, is actually an expression that is evaluated at the time the `define` directive is encountered in the source file. The following table describes the available operators (based on the Delphi language).

Operator	Precedence	Description
? :	9	Conditional operator (a la C++)
or	8	logical or
and	7	logical and
=, <>	6	equality / inequality
<, >, <=, >=	5	comparisons
+, -	4	add / subtract
*, /, div, mod	3	multiply / divide / remainder
^	2	raise to power
-, not	1	unary negation / logical negation

Literal values encountered in these expressions are handled according to the following rules:

A literal enclosed within single quotes is considered a string.

A literal that begins with a digit or period is considered a number.

A number that can be converted to an integer is considered an integer.

A number that can not be converted to an integer is considered a double.

A literal that is not quoted, and does not begin with a digit or period is considered an identifier.

The example above illustrates how to define preprocessor symbols. First, we define the Debug symbol. Because we didn't specify a value, it defaults to True. Next, we define the ExtraUnit symbol, assigning it a string value (note that single quotes are needed to make it a string, and double-quotes are required for the HTML attribute overall). This example also shows how the value of a preprocessor symbol may be referenced within another directive. Using a dollar-sign symbol, you can substitute the value of a preprocessor symbol in the following directives:

```
uses
uses_interface
include
options (for errorPage only)
```

See Also

Directive: fetch, Directive: if

1.6.3 Directive: else

Overview

Closes a conditional block opened by an "if" directive, opening an alternate block.

Syntax

```
<%@ else %>
```

Description

For details on how the else directive is used, see the if directive.

See Also

Directive: if, Directive: endif

1.6.4 Directive: end

Overview

Closes a scope (global, nested or main).

Syntax

```
<%@ end { global | nested | main } %>
```

Description

The end directive is the companion of the begin directive. The end directive must specify a scope of either global, nested or main and there must be a previous, matching begin directive with the same scope.

See the documentation on the begin directive for a more complete explanation of scopes.

See Also

Directive: begin, Code Fragment, Expression

1.6.5 Directive: endif

Overview

Closes a conditional block opened by an "if" or "else" directive.

Syntax

```
<%@ endif %>
```

Description

For details on how the endif directive is used, see the if directive.

See Also

Directive: if, Directive: else

1.6.6 Directive: fetch

Overview

Fetches the value of a preprocessor expression into a Delphi variable.

Syntax

```
<%@ fetch into="variable-name" from="expression" %>
```

Example

```
<%@ define DefaultFontSize=3 %>  
<%@ begin nested %>  
<%  
var  
    FontSize: Integer;  
%>  
<%@ end nested %>  
  
<%@ fetch into="FontSize" from="DefaultFontSize + 1" %>
```

Description

The fetch directive evaluates the expression in the "from" attribute at translation-time (i.e. while the DSP page is being translated into a Delphi unit) and creates an assignment expression in the generated unit. For example, the example above results in the following code being placed in the generated Delphi unit:

```
FontSize := 4;
```

For more information on the expression syntax see the define directive.

See Also

Directive: define, Directive: begin, Directive: end

1.6.7 Directive: finalization

Overview

Places code in the finalization section of the generated unit.

Syntax

```
<%@ finalization [Delphi code] %>
```

Example

```
<%@ begin global %>  
<% var Lock: TCriticalSection; %>  
<%@ end global %>  
  
<%@ initialization  
  Lock := TCriticalSection.Create;  
%>  
<%@ finalization  
  FreeAndNil(Lock);  
%>
```

Description

The finalization directive specifies Delphi code that placed in the generated unit's finalization section. It is typically used to free resources allocated in an initialization block.

Note that it does not matter where in the DSP file the finalization directive is placed. It will have the same effect, regardless of its placement. It is, however, considered good style to place the finalization directive at the bottom of the file in which it appears.

See Also

Directive: initialization, Code Fragment, Directive: begin, Directive: end

1.6.8 Directive: if

Overview

Conditionally translates DSP code.

Syntax

```
<%@ if expression %>  
  ...  
<%@ endif %>  
  
or  
  
<%@ if expression %>  
  ...  
<%@ else %>  
  ...  
<%@ endif %>
```

Example

```
<%@ if DelphiVersion > 5 %>  
<%@ uses DateUtils %>  
<%@ else %>  
<%@ uses MyDateUtils %>  
<%@ endif %>
```

Description

The if-endif and if-else-endif directives allow portions of a DSP to be conditionally used or not when translating the DSP page into a Delphi unit. In the example above if the DelphiVersion of the DSP translator is six or greater then the page uses the DateUtils unit, if it is five then it uses the MyDateUtils unit.

The "if" directive treats any Null expression as being False. For numeric and string types, zero and empty string are considered False. Any undefined symbol is treated as Null (and therefore evaluates to False). See the define directive for more information on the expressions syntax.

As you would expect, these directives may be nested, and the else directive is optional.

See Also

Directive: define, Directive: else, Directive: endif

1.6.9 Directive: include

Overview

Includes another file in a DSP file.

Syntax

```
<%@ include filename %>
```

Example 1

```
example.dsp
<html>
<head></head>
<body>
...
<%@ include copyright.dsi %>
</body>
</html>

copyright.dsi
<!-- include this copyright on every page -->
<small>© Copyright 2000 Myself, Inc.</small>
```

Description

When the DSP is translated, the include directive is replaced with the contents of the file indicated, as if that other file were typed in the original DSP. The filename specified may be a relative path, which is relative to the including file. That is, `<%@ include ..copyright.dsi %>` will include the `copyright.dsi` file in the parent directory to the file being included.

1.6.10 Directive: initialization

Overview

Places code in the initialization section of the generated unit.

Syntax

```
<%@ initialization [Delphi code] %>
```

Example

```
<%@ begin global %>
<% var Lock: TCriticalSection; %>
<%@ end global %>

<%@ initialization
    Lock := TCriticalSection.Create;
%>
<%@ finalization
    FreeAndNil(Lock);
%>
```

Description

The initialization directive specifies Delphi code that is placed in the generated unit's initialization section. It is typically used to initialize global variables of the page or for logging.

Note that it does not matter where in the DSP file the initialization directive is placed. It will have the same effect, regardless of its placement. It is, however, considered good style to place the initialization directive at the bottom of the file in which it appears.

See Also

Directive: finalization, Code Fragment, Directive: begin, Directive: end

1.6.11 Directive: interface

Overview

Contains declarations placed in the interface section of the generated unit.

Syntax

```
<%@ interface [Delphi code] %>
```

Example

```
<%@ interface
  type
    TSomeEnumeration = (seFirst, seSecond, seThird);
%>
```

Description

The interface directive specifies Delphi code that is placed in the interface section of the generated unit. It is used for global declarations that are used by more than one DSP page or Delphi unit.

Note that it does not matter where in the DSP file the interface directive is placed. It will have the same effect, regardless of its placement. It is, however, considered good style to place the interface directive at the top of the file in which it appears.

See Also

Directive: begin, Directive: end, Code Fragment, Expression

1.6.12 Directive: options

Overview

Sets page-wide options.

Syntax

```
<%@ options
  [ ErrorPage="dsp-page" ]
  [ BufSize=count[ { k | m } ] ]
  [ ParseContent={ all | none } ]
  [ StripSpaces={ on | off | html } ]
%>
```

Example 1

```
example.dsp
<%@ options ErrorPage="Errors.dsp" %>
<% raise Exception.Create('Only the error page is displayed--with this message'); %>

Errors.dsp
<html>
<head><>Error Page</title></head>
<body>

The exception class is: <%= Context.Exception.ClassName %>

The exception message is: <%= Context.Exception.Message %>
</body>
</html>
```

Example 2

```
<%@ options BufSize=8k %>
...
```

Description

The options directive allows you to set page-wide DSP options. Currently there are four such options: ErrorPage, BufSize, ParseContent and StripSpaces.

The ErrorPage option indicates a DSP page to run when an exception is thrown outside the DSP. That is, if an exception is thrown and not caught, and an ErrorPage option is set for the page, then that page is run. When the page specified by the errorpage option is run, the Context object's Exception property is set to the exception thrown by the original page.

The BufSize option is an advanced option that enables dynamic streaming of the response back to the client. This enables a DSP to generate a large response without keeping the entire contents in memory on the server at any one time. For DSP pages that return sizeable content, this can reduce the resource load on the server and improve perceived performance by the user. Essentially, a fixed-sized buffer is created and when it fills up, the data is sent to the client. By default the entire DSP page is buffered in memory until the page execution is complete. Example 2 above creates an 8k buffer (8192 bytes).

Dynamic streaming comes at the cost of greater complexity and should only be used if you are aware of the consequences. Once data has been sent to the client, it is not possible to retract it. This includes changing the HTTP response code, cookies, or any other data that is part of the HTTP header because the HTTP header is sent before any other data. For example, if an exception is thrown after data has been sent, then the response code cannot be set to an error code. You may print an error message at that point, but it will be appended to the end of any previously sent content. In contrast, when dynamic streaming is not used, the entire page's content is kept in memory until the DSP page has completed processing. The page can change the entire response content at any time before completing, including changing the HTTP response code or replacing the DSP page's output with that of an `ErrorPage`. Note that for this reason, the use of dynamic streaming with an `ErrorPage` is discouraged.

The `ParseContent` option allows disabling the automatic parsing of data posted to the application. When the `ParseContent` option is "all", which it is by default, the posted content is parsed as if it came from an HTML form and the result is placed in the `Query` object. If `ParseContent` is set to "none" then the content isn't parsed or even read from the `Request.Content` stream. The `Request.Content` stream is left for the page code to read explicitly.

The `StripSpaces` option allows stripping redundant white space from the dynamically generated response to the client. This can significantly reduce the size of the response to the client with no work by the page author (i.e. the page doesn't have to be written in an poorly readable format just to reduce the size of the whitespace). If this option is set to "on" then it the spaces will be stripped from the response, if set to "off" then spaces stripping won't be performed. If it is set to "html", the default, then spaces will be stripped only if the response's content type is "text/html". Note that only redundant whitespace (such as extra blank line or multiple spaces in a row) will be stripped. Non-redundant whitespace, such as any whitespace between `<>` and `<>` tags will not be stripped.

Note that it does not matter where in the DSP file the options directive is placed. It will have the same effect, regardless of its placement. It is, however, considered good style to place the options directive at the top of the file in which it appears.

See Also

Directive: run

1.6.13 Directive: run

Overview

Executes another DSP, optionally placing its output inline.

Syntax

```
<%@ run page=DSP-page [html={ on | off }] %>
```

Example 1

```
This appears before the output of header.dsp
<%@ run page="../common files/header.dsp" %>
This appears after the output of header.dsp
```

Example 2

```
<%@ run page="check_security.dsp" html="off" %>
The check_security.dsp page has executed, but its output is ignored.
```

Description

The run directive executes another DSP page within the context of an existing request. That is, at the point of the run directive, another DSP page is invoked and the `Context`, `Request` and `Response`, etc., objects are the same for page that is run as for the current page. The output of the page that is run is inserted at the point of the run directive, unless the `html=off` option is used, in which case, the output of the page that is run is discarded.

1.6.14 Directive: uses

Overview

Indicates the DSP depends on one or more Delphi units.

Syntax

```
<%@ uses unit-list %>

or

<%@ uses_implementation unit-list %>
```

Example

```
<%@ uses Math %>  
Ten cubed is <%= Power(10, 3) %>
```

Description

The uses directive (and the identical uses_implementation directive) places the units in the unit-list in the uses clause of the implementation section of the generated Delphi unit.

It is not an error to have the same unit appear in more than one uses directive. The unit will be listed only once in the order it was first encountered. If a unit is listed as both an implementation-uses and an interface-uses (using the uses_interface directive) then the unit will be placed in the interface section of the unit generated from the DSP file.

Note that it does not matter where in the DSP file the uses directive is placed. It will have the same effect, regardless of its placement. It is, however, considered good style to place the uses directive at the top of the file in which it appears.

See Also

Directive: uses_interface, Code Fragment, Expression

1.6.15 Directive: uses_interface

Overview

Indicates the DSP interface section depends on one or more Delphi units.

Syntax

```
<%@ uses_interface unit-list %>
```

Example

```
<%@ uses_interface VirtualDB, SomeOtherUnit %>  
<%@ interface  
function Login(  
    const Database: IDataBase; const UserName, Password: string): Boolean;  
%>
```

Description

The uses_interface directive places the units in the unit-list in the uses clause of the interface section of the generated Delphi unit.

It is not an error to have the same unit appear in more than one uses_interface directive. The unit will be listed only once in the order it was first encountered. If a unit is listed as both an implementation-uses (using the uses directive) and an interface-uses then the unit will be placed in the interface section of the unit generated from the DSP file.

Note that it does not matter where in the DSP file the uses_interface directive is placed. It will have the same effect, regardless of its placement. It is, however, considered good style to place the uses_interface directive at the top of the file in which it appears.

See Also

Directive: uses, Code Fragment, Expression, Directive: interface

Index

C

Code Fragment 3

D

Dimeric Server Pages Syntax Guide 1

Directive: begin 4

Directive: define 5

Directive: else 6

Directive: end 6

Directive: endif 7

Directive: fetch 7

Directive: finalization 7

Directive: if 8

Directive: include 9

Directive: initialization 9

Directive: interface 9

Directive: options 10

Directive: run 11

Directive: uses 11

Directive: uses_interface 12

Directives 4

E

Encoded Expression 2

Expression 2

H

Hidden Comment 1

HTML Text 1

