Apache Module mod_include

Description:	Server-parsed html documents (Server Side Includes)
Status:	Base
Module Identifier:	include_module
Source File:	mod_include.c
Compatibility:	Implemented as an output filter since Apache 2.0

Summary

This module provides a filter which will process files before they are sent to the client. The processing is controlled by specially formatted SGML comments, referred to as *elements*. These elements allow conditional text, the inclusion of other files or programs, as well as the setting and printing of environment variables.

Topics

Directives

See also

- `Options`
- `AcceptPathInfo`
- International Customized Server Error Messages[1]
- Filters[2]
- SSI Tutorial[3]

Enabling Server-Side Includes

Server Side Includes are implemented by the `INCLUDES` filter[2]. If documents containing server-side include directives are given the extension .shtml, the following directives will make Apache parse them and assign the resulting document the mime type of `text/html`:

```
AddType text/html .shtml
AddOutputFilter INCLUDES .shtml
```

The following directive must be given for the directories containing the shtml files (typically in a `<Directory>` section, but this directive is also valid in `.htaccess` files if `AllowOverride Options` is set):

```
Options +Includes
```

For backwards compatibility, the `server-parsed` handler[4] also activates the INCLUDES filter. As well, Apache will activate the INCLUDES filter for any document with mime type `text/x-server-parsed-html` or `text/x-server-parsed-html3` (and the resulting output will have the mime type `text/html`).

For more information, see our Tutorial on Server Side Includes[3].

PATH_INFO with Server Side Includes

Files processed for server-side includes no longer accept requests with `PATH_INFO` (trailing pathname information) by default. You can use the `AcceptPathInfo` directive to configure the server to accept requests with `PATH_INFO`.

Basic Elements

The document is parsed as an HTML document, with special commands embedded as SGML comments. A command has the syntax:

```
<!--#element attribute=value attribute=value ... -->
```

The value will often be enclosed in double quotes, but single quotes (`'`) and backticks (`` ` ``) are also possible. Many commands only allow a single attribute-value pair. Note that the comment terminator (`-->`) should be preceded by whitespace to ensure that it isn't considered part of an SSI token. The allowed elements are listed in the following table:

Element	**Description**
`config`	configure output formats
`echo`	print variables
`exec`	execute external programs
`fsize`	print size of a file
`flastmod`	print last modification time of a file
`include`	include a file
`printenv`	print all available variables
`set`	set a value of a variable

SSI elements may be defined by modules other than `mod_include`. In fact, the `exec` element is provided by `mod_cgi`, and will only be available if this module is loaded.

The config Element

This command controls various aspects of the parsing. The valid attributes are:

`errmsg`
The value is a message that is sent back to the client if an error occurs while parsing the document.

`sizefmt`
The value sets the format to be used which displaying the size of a file. Valid values are `bytes` for a count in bytes, or `abbrev` for a count in Kb or Mb as appropriate, for example a size of 1024 bytes will be printed as "1K".

`timefmt`
The value is a string to be used by the `strftime(3)` library routine when printing dates.

The echo Element

This command prints one of the include variables, defined below. If the variable is unset, the result is determined by the `SSIUndefinedEcho` directive. Any dates printed are subject to the currently configured `timefmt`.

Attributes:

`var`
The value is the name of the variable to print.

`encoding`
Specifies how Apache should encode special characters contained in the variable before outputting them. If set to `none`, no encoding will be done. If set to `url`, then URL encoding (also known as %-encoding; this is appropriate for use within URLs in links, etc.) will be performed. At the start of an `echo` element, the default is set to `entity`, resulting in entity encoding (which is appropriate in the context of a block-level HTML element, *e.g.* a paragraph of text). This can be changed by adding an `encoding` attribute, which will remain in effect until the next `encoding` attribute is encountered or the element ends, whichever comes first.

The `encoding` attribute must *precede* the corresponding `var` attribute to be effective, and only special characters as defined in the ISO-8859-1 character encoding will be encoded. This encoding process may not have the desired result if a different character encoding is in use.

In order to avoid cross-site scripting issues, you should *always* encode user supplied data.

The exec Element

The `exec` command executes a given shell command or CGI script. It requires `mod_cgi` to be present in the server. If `Options IncludesNOEXEC` is set, this command is completely disabled. The valid attributes are:

`cgi`
The value specifies a (%-encoded) URL-path to the CGI script. If the path does not begin with a slash (/), then it is taken to be relative to the current document. The document referenced by this path is invoked as a CGI script, even if the server would not normally recognize it as such. However, the directory containing the script must be enabled for CGI scripts (with `ScriptAlias` or `Options ExecCGI`).

The CGI script is given the `PATH_INFO` and query string (`QUERY_STRING`) of the original request from the client; these *cannot* be specified in the URL path. The include variables will be available to the script in addition to the standard CGI[5] environment.

Example
```
<!--#exec cgi="/cgi-bin/example.cgi" -->
```

If the script returns a `Location:` header instead of output, then this will be translated into an HTML anchor.

The `include virtual` element should be used in preference to `exec cgi`. In particular, if you need to pass additional arguments to a CGI program, using the query string, this cannot be done with `exec cgi`, but can be done with `include virtual`, as shown here:

```
<!--#include virtual="/cgi-bin/example.cgi?argument=value" -->
```

`cmd`

The server will execute the given string using `/bin/sh`. The include variables are available to the command, in addition to the usual set of CGI variables.

The use of `#include virtual` is almost always prefered to using either `#exec cgi` or `#exec cmd`. The former (`#include virtual`) uses the standard Apache sub-request mechanism to include files or scripts. It is much better tested and maintained.

In addition, on some platforms, like Win32, and on unix when using suexec[6], you cannot pass arguments to a command in an `exec` directive, or otherwise include spaces in the command. Thus, while the following will work under a non-suexec configuration on unix, it will not produce the desired result under Win32, or when running suexec:

```
<!--#exec cmd="perl /path/to/perlscript arg1 arg2" -->
```

The fsize Element

This command prints the size of the specified file, subject to the `sizefmt` format specification. Attributes:

`file`

The value is a path relative to the directory containing the current document being parsed.

`virtual`

The value is a (%-encoded) URL-path. If it does not begin with a slash (/) then it is taken to be relative to the current document. Note, that this does *not* print the size of any CGI output, but the size of the CGI script itself.

The flastmod Element

This command prints the last modification date of the specified file, subject to the `timefmt` format specification. The attributes are the same as for the `fsize` command.

The include Element

This command inserts the text of another document or file into the parsed file. Any included file is subject to the usual access control. If the directory containing the parsed file has Options[7] `IncludesNOEXEC` set, then only documents with a text MIME type (`text/plain`, `text/html` etc.) will be included. Otherwise CGI scripts are invoked as normal using the complete URL given in the command, including any query string.

An attribute defines the location of the document; the inclusion is done for each attribute given to the include command. The valid attributes are:

`file`

The value is a path relative to the directory containing the current document being parsed. It cannot contain `../`, nor can it be an absolute path. Therefore, you cannot include files that are outside of the document root, or above the current document in the directory structure. The `virtual` attribute should always be used in preference to this one.

`virtual`

The value is a (%-encoded) URL-path. The URL cannot contain a scheme or hostname, only a path and an optional query string. If it does not begin with a slash (/) then it is taken to be relative to the current document.

A URL is constructed from the attribute, and the output the server would return if the URL were accessed by the client is included in the parsed output. Thus included files can be nested.

If the specified URL is a CGI program, the program will be executed and its output inserted in place of the directive in the parsed file. You may include a query string in a CGI url:

```
<!--#include virtual="/cgi-bin/example.cgi?argument=value" -->
```

`include virtual` should be used in preference to `exec cgi` to include the output of CGI programs into an HTML document.

The printenv Element

This prints out a listing of all existing variables and their values. Special characters are entity encoded (see the `echo` element for details) before being output. There are no attributes.

Example

```
<!--#printenv -->
```

The set Element

This sets the value of a variable. Attributes:

`var`
The name of the variable to set.

`value`
The value to give a variable.

Example

```
<!--#set var="category" value="help" -->
```

Include Variables

In addition to the variables in the standard CGI environment, these are available for the `echo` command, for `if` and `elif`, and to any program invoked by the document.

`DATE_GMT`
The current date in Greenwich Mean Time.

`DATE_LOCAL`
The current date in the local time zone.

`DOCUMENT_NAME`
The filename (excluding directories) of the document requested by the user.

`DOCUMENT_URI`
The (%-decoded) URL path of the document requested by the user. Note that in the case of nested include files, this is *not* the URL for the current document.

`LAST_MODIFIED`
The last modification date of the document requested by the user.

`QUERY_STRING_UNESCAPED`
If a query string is present, this variable contains the (%-decoded) query string, which is *escaped* for shell usage (special characters like `&` etc. are preceded by backslashes).

Variable Substitution

Variable substitution is done within quoted strings in most cases where they may reasonably occur as an argument to an SSI directive. This includes the `config`, `exec`, `flastmod`, `fsize`, `include`, `echo`, and `set` directives, as well as the arguments to conditional operators. You can insert a literal dollar sign into the string using backslash quoting:

```
<!--#if expr="$a = \$test" -->
```

If a variable reference needs to be substituted in the middle of a character sequence that might otherwise be considered a valid identifier in its own right, it can be disambiguated by enclosing the reference in braces, *a la* shell substitution:

```
<!--#set var="Zed" value="${REMOTE_HOST}_${REQUEST_METHOD}" -->
```

This will result in the `Zed` variable being set to "`X_Y`" if `REMOTE_HOST` is "`X`" and `REQUEST_METHOD` is "`Y`".

The below example will print "in foo" if the `DOCUMENT_URI` is `/foo/file.html`, "in bar" if it is `/bar/file.html` and "in neither" otherwise:

```
<!--#if expr='"$DOCUMENT_URI" = "/foo/file.html"' -->
  in foo
<!--#elif expr='"$DOCUMENT_URI" = "/bar/file.html"' -->
  in bar
<!--#else -->
  in neither
<!--#endif -->
```

Flow Control Elements

The basic flow control elements are:

```
<!--#if expr="test_condition" -->
<!--#elif expr="test_condition" -->
<!--#else -->
<!--#endif -->
```

The `if` element works like an if statement in a programming language. The test condition is evaluated and if the result is true, then the text until the next `elif`, `else` or `endif` element is included in the output stream.

The `elif` or `else` statements are be used to put text into the output stream if the original *test_condition* was false. These elements are optional.

The `endif` element ends the `if` element and is required.

test_condition is one of the following:

`string`
: true if *string* is not empty

`string1 = string2`
`string1 != string2`
: Compare *string1* with *string2*. If *string2* has the form *`/string2/`* then it is treated as a regular expression. Regular expressions are implemented by the PCRE [8] engine and have the same syntax as those in perl 5[9].

  If you are matching positive (`=`), you can capture grouped parts of the regular expression. The captured parts are stored in the special variables `$1` .. `$9`.

  Example

```
<!--#if expr="$QUERY_STRING = /^sid=([a-zA-Z0-9]+)/" -->
  <!--#set var="session" value="$1" -->
<!--#endif -->
```

`string1 < string2`
`string1 <= string2`
`string1 > string2`
`string1 >= string2`
: Compare *string1* with *string2*. Note, that strings are compared *literally* (using `strcmp(3)`). Therefore the string "100" is less than "20".

`( test_condition )`
: true if *test_condition* is true

`! test_condition`
: true if *test_condition* is false

`test_condition1 && test_condition2`
: true if both *test_condition1* and *test_condition2* are true

`test_condition1 || test_condition2`
: true if either *test_condition1* or *test_condition2* is true

"`=`" and "`!=`" bind more tightly than "`&&`" and "`||`". "`!`" binds most tightly. Thus, the following are equivalent:

```
<!--#if expr="$a = test1 && $b = test2" -->
<!--#if expr="($a = test1) && ($b = test2)" -->
```

Anything that's not recognized as a variable or an operator is treated as a string. Strings can also be quoted: `'string'`. Unquoted strings can't contain whitespace (blanks and tabs) because it is used to separate tokens such as variables. If multiple strings are found in a row, they are concatenated using blanks. So,

`string1    string2` results in *`string1 string2`*

and

`'`*`string1    string2`*`'` results in *`string1    string2`*.

SSIEndTag Directive

Description:	String that ends an include element
Syntax:	`SSIEndTag` *`tag`*

Default:	`SSIEndTag "-->"`
Context:	server config, virtual host
Status:	Base
Module:	mod_include
Compatibility:	Available in version 2.0.30 and later.

This directive changes the string that `mod_include` looks for to mark the end of an include element.

Example
```
SSIEndTag "%>"
```

See also

- `SSIStartTag`

SSIErrorMsg Directive

Description:	Error message displayed when there is an SSI error
Syntax:	`SSIErrorMsg` *`message`*
Default:	`SSIErrorMsg "[an error occurred while processing this directive]"`
Context:	server config, virtual host, directory, .htaccess
Override:	All
Status:	Base
Module:	mod_include
Compatibility:	Available in version 2.0.30 and later.

The `SSIErrorMsg` directive changes the error message displayed when `mod_include` encounters an error. For production servers you may consider changing the default error message to `"<!-- Error -->"` so that the message is not presented to the user.

This directive has the same effect as the `<!--#config errmsg=`*`message`*` -->` element.

Example
```
SSIErrorMsg "<!-- Error -->"
```

SSIStartTag Directive

Description:	String that starts an include element
Syntax:	`SSIStartTag` *`tag`*
Default:	`SSIStartTag "<!--"`
Context:	server config, virtual host
Status:	Base
Module:	mod_include
Compatibility:	Available in version 2.0.30 and later.

This directive changes the string that mod_include looks for to mark an include element to process.

You may want to use this option if you have 2 servers parsing the output of a file each processing different commands (possibly at different times).

Example
```
SSIStartTag "<%"
```

The example given above, in conjunction with a matching SSIEndTag, will allow you to use SSI directives as shown in the example below:

SSI directives with alternate start and end tags
```
<%#printenv %>
```

See also

- SSIEndTag

SSITimeFormat Directive

Description:	Configures the format in which date strings are displayed
Syntax:	`SSITimeFormat` *`formatstring`*
Default:	`SSITimeFormat "%A, %d-%b-%Y %H:%M:%S %Z"`
Context:	server config, virtual host, directory, .htaccess
Override:	All
Status:	Base
Module:	mod_include
Compatibility:	Available in version 2.0.30 and later.

This directive changes the format in which date strings are displayed when echoing `DATE` environment variables. The *formatstring* is as in `strftime(3)` from the C standard library.

This directive has the same effect as the `<!--#config timefmt=`*`formatstring`*` -->` element.

Example
```
SSITimeFormat "%R, %B %d, %Y"
```

The above directive would cause times to be displayed in the format "22:26, June 14, 2002".

SSIUndefinedEcho Directive

Description:	String displayed when an unset variable is echoed
Syntax:	`SSIUndefinedEcho` *`string`*
Default:	`SSIUndefinedEcho "(none)"`
Context:	server config, virtual host
Status:	Base
Module:	mod_include

Compatibility:	Available in version 2.0.34 and later.

This directive changes the string that `mod_include` displays when a variable is not set and "echoed".

Example
```
SSIUndefinedEcho "<!-- undef -->"
```

XBitHack Directive

Description:	Parse SSI directives in files with the execute bit set
Syntax:	`XBitHack on|off|full`
Default:	`XBitHack off`
Context:	server config, virtual host, directory, .htaccess
Override:	Options
Status:	Base
Module:	mod_include

The `XBitHack` directive controls the parsing of ordinary html documents. This directive only affects files associated with the MIME type `text/html`. `XBitHack` can take on the following values:

`off`
: No special treatment of executable files.

`on`
: Any `text/html` file that has the user-execute bit set will be treated as a server-parsed html document.

`full`
: As for `on` but also test the group-execute bit. If it is set, then set the `Last-modified` date of the returned file to be the last modified time of the file. If it is not set, then no last-modified date is sent. Setting this bit allows clients and proxies to cache the result of the request.

Note
You would not want to use the full option, unless you assure the group-execute bit is unset for every SSI script which might `#include` a CGI or otherwise produces different output on each hit (or could potentially change on subsequent requests).

URI References

[1] http://httpd.apache.org/docs-2.1/misc/custom_errordocs.html
[2] http://httpd.apache.org/docs-2.1/filter.html
[3] http://httpd.apache.org/docs-2.1/howto/ssi.html
[4] http://httpd.apache.org/docs-2.1/handler.html
[5] http://httpd.apache.org/docs-2.1/mod/mod_cgi.html
[6] http://httpd.apache.org/docs-2.1/suexec.html
[7] http://httpd.apache.org/docs-2.1/mod/core.html#options
[8] http://www.pcre.org
[9] http://www.perl.com