Perl Server Pages language specification

Holger Zimmermann (*zimpel@t-online.de*) V0.6, 8 October 2001

Perl Server Pages (PSP) language specification, version 0.6

Contents

1 Introduction

Perl Server Pages (PSP) is an embedded script language, which allows to mix Perl code with any tag language (HTML) pages. You can either use scriptlets to execute embedded perl code, expressions to write dynamic output or directives to include subsequent PSP files.

The PSP code could use Pi3Perl for callbacks using the perl package Pi3:: into the Pi3Web API's. Pi3Perl is similar to Java Servlet API and PSP is similar to JSP.

2 PSP Architecture

Implementations of PSP should run on perl 5.6.x or greater. For web server or other multithreaded implementations the persistence/context switching mechanisms of the perl API should be used.

2.1 Standalone vs. embedded interpreter

PSP is designed to run as a standalone interpreter as well as an embedded script interpreter engine for WWW or other servers. The standalone interpreter could be started from the command line or as a CGI program just like the perl interpreter itself is used. The embedded interpreter is used as a web server module and doesn't communicate through the STDIN/STDOUT mechanism but only by using the PSP API and the with the web server.

A standalone interpreter could be used as a backend of several web servers or a web server could distribute client requests to several PSP engines for load balancing. Particular in those scenarios the following has to be considered.

2.2 Implications of multiprocessed and multithreaded servers

It depends on the lifetime of processes, threads and the sharing of the used perl interpreters between them, what happens regarding lifetime and scope of perl variables. Currently it isn't defined by PSP, how to separate name spaces of global perl variables in different scripts. So it is assumed when a PSP script is executed, that all global values are undefined except the 4 PSP variables. If your PSP script does initialize some global values within pspLoad(), it is the responsibility of the script author to use this values with a global scope (don't write to them during request handling).

Further it is assumed, that all global values except of the 4 PSP variables have script scope. They may be visible to all clients in a multithreaded context. Thus global perl variables represent a real global, client-independant context of a script. A client- or session-scope context mechanism is not part of the current PSP language specification.

Because the PSP variables are both global *and* have script scope, the server may synchronize multiple calls to the same script interpreter at the same time.

The implementation of an embedded PSP script engine must avoid to share global values between scripts by starting each script in an own interpreter instance or by similar mechanisms. If variables must be shared between PSP scripts, other session mechanisms (cookies, hidden form fields, query string) must be used.

2.3 PSP interpreter persistence

The implementation of an embedded PSP script engine should avoid the overhead of often creation/distortion of perl interpreters by maintenance of a pool of such interpreters. Due to the script scope of global values the interpreter instance will live as long as the PSP script itself lives. Thus the memory allocated by global variables will remain allocated and memory usage could increase during lifetime of a script dependant on the embedded perl code. If the script engines are stored in a pool on a per-script basis, their number and resource usage will increase until all scripts have been called and initialized at least once.

3 PSP overview

Perl Server Pages (PSP) defines an embedded script language, to embed arbitrary perl code into a tag language body. A PSP script consists of the following parts:

- "shebang" line
- tag language body
- embedded script language elements
  - PSP API subroutines
  - read/write PSP variables
  - any perl code

The tag language body contains plain text in the respective tag language (e.g. HTML tags).

PSP language elements are a subset of the JSP 1.0 standard. This subset is really small to keep things simple and fast:

- Scriptlets
- Expressions
- Directives

The PSP API contract consists of only a few simple subroutines to be called by the server either once or per request:

- pspLoad()
- pspSetResponse()
- pspUnload()

There are only a few PSP variables, which belong also to the PSP API contract:

- %pspRequest
- %pspResponse
- %pspQueryString
- %pspForm
- $pspPlain
- $pspStatusCode

4 PSP scripts

PSP scripts use a one byte, ASCII character set. The syntax of the plain tag language body is determined by the respective language (e.g. HTML). The script interpreter doesn't depend on it.

4.1 MIME type

If a web server does map client requests to the PSP interpreter through a MIME type, the type "application/x-perl-serverpages" should be used.

4.2 Content-type

An embedded interpreter will use the Content-Type "text/html" in responses by default. This could be overwritten by the PSP script by setting an appropriate value in *%pspResponse*{*Content-Type*}.

4.3 File extensions

PSP scripts should be stored by using the file extensions *.psp* or *.ppsp*.

4.4 Shebang line

The standalone PSP interpreter could be located by a POSIX compliant shell e.g. if your PSP script starts with:

```
#!/usr/local/bin/psp
```

An embedded PSP interpreter should ignore it and doesn't send the shebang line back to the client.

5 Language elements

The PSP language elements are ordered in an arbitrary sequence. It isn't allowed to start elements recursive within another elements. If the tag delimiters of PSP should be used within plain text, they must be replaced by the respective character entities or will probably throwe a parse error.

5.1 Plain text

Plain text is simply written back to the client. If one of the following language elements resides on a separate line, the final NL is evaluated as plain text an will be sent to the client as well.

5.2 Scriptlets

An embedded PSP scriptlet is defined by:

```
<% scriptlet %>
```

The scriptlet is rather evaluated than written back to the client. More concrete the behaviour depends on the type of interpreter. If a standalone interpreter is used and the scriptlet uses e.g. *print()* to write to STDOUT, this is part of the script output. An embedded interpreter doesn't care about STDOUT, the output created by *print()* will probably end on the server console.

5.3 Expressions

An embedded PSP expression is defined by:

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

Expressions are always evaluated and written back to the client immediately. An expression must return in a scalar context.

5.4 Directives

An embedded PSP directive is defined by:

```
<%@ directive %>
directive:      include file="filename"
                  | page contentType="<bf>text/html" parsedLength="<bf>true | false"
```

A directive is never written back to the client but will change something in the configuration of the script interpreter.

5.4.1 Include

The *include* directive is used to define an included file, it's content will occur in the output at the position of the *include* directive. The filename could contain a relative or absolute filesystem path.

For standalone interpreters it depends on the current directory of the script engine and the usage of *cwd()* in scriptlets or expressions, how the filename is resolved.

An embedded interpreter running in a server context may restrict to resolve a path only relative to a predefined root path for security reasons. This doesn't affect usage of *cwd()* in scriptlets or expressions.

There could be subsequent *include* directives in included files, but the number of recursions is restricted by the PSP interpreter. As written above, it isn't possible to include other PSP files from within an element.

5.4.2 Page

The *page* directive is used to change attributes of the page sent back to the client. The content-type is adjusted by populating the parameter *contentType* with an appropriate value. If omitted, the default Content-Type is "text/html".

The content-length is calculated by the parser as default. Since this could lead to an error, it could be switched off by setting parameter *parsedLength* to *false*.

6 The PSP API contract

The following describes the perl subs and some global values, which are part of the API contract between the PSP script author and the script engine. All subroutines are optional. It wouldn't lead to an error, if one of the subs is undefined in a PSP script. If a subroutine is defined it must return either *0* on success or *-1* on error.

6.1 pspLoad

This optional subroutine is invoked once the first time that an instance of a PSP script is parsed and loaded. Once off initialization can be accomplished in this function. A typical use of this is to create a database connection, TCP/IP socket or other connection to external datasource that will be used across multiple PSP script requests.

6.2 pspUnload

This optional subroutine is invoked after the last request to a PSP script has been completed and the script and interpreter is being unloaded. A typical use of this function is to destroy database connections or TCP/IP sockets that were opened in the *pspLoad* function.

6.3 pspSetResponse

This optional subroutine is invoked at the beginning of a new web request before any data or headers have been sent. The function provides the user with the opportunity to manipulate the headers that are sent to the browser. HTTP headers are sent to the browser after this function but before the script as a whole is executed. Since the script will not have been executed when this function is invoked care must be taken to note that any global variables initialized in global scope will not have been executed.

The *pspSetResponse* function is used very often to redirect or set a cookie:

```
# ---
# Redirect all requests to http://localhost with the given PathInfo
# ---
sub pspSetResponse
{
        $pspResponse{"Location"} = "http://localhost".$pspRequest{"PathInfo"};
        $pspResponseStatus = 302;
}

# ---
# Set a transient cookie
# ---
sub pspSetResponse
{
        my $value = "foo";  # value to set in cookie
        $pspResponse{"Set-Cookie"} = "MyCookie=$value; path=/";
}
```

6.4 %pspRequest

All request headers from the browser are stored in this hash list. Browser variables are currently *not* canonicized (i.e. doesn't converting letters to upper case and/or replacing the hyphen '-' character with the underscore '_'). Additional the following special hash keys are added (the values are examples):

CLF

GET /index.psp/test.txt?param1=abc+123&param2=xyz HTTP/1.0

Method

GET

URI

/index.psp

QueryString

param1=abc+123&param2=xyz

Protocol

HTTP/1.0

More variables could be added dependant on the used server and the server configuration. The *%pspRequest* will only be filled by the script interpreter if it is used by the PSP script.

```
# ---
# Put the browser type into '$browser'
# ---
my $browser = $pspRequest{"User-Agent"};
```

6.5 %pspResponse

Headers sent to the browser. Values in *%pspResponse* need to be set within the subroutine *pspSetResponse* before the headers are sent to the browser. For an example refer to 6.3 (pspSetResponse). The *%pspResponse* will be pre-filled by the script interpreter only if the variable is really used by the PSP script. The following keys are used (the values are examples):

Path

C:\Pi3Web\WebRoot\index.psp

PathInfo

/test.txt

PathTranslated

C:\Pi3Web\WebRoot\test.txt

ScriptName

/index.psp

6.6 %pspQueryString

This hash list contains parsed and urldecoded name value pairs from the query string of the URI if the request method is not POST. It will only be filled by the script interpreter if it is used by the PSP script.

```
# ---
# check if this request was http://dallas.com/index.psp?name=J.%20R.
# ---
if ( $pspQueryString{name} eq "J. R." )
        {
        # ...
        };
```

6.7 %pspForm

Parsed and urldecoded name value pairs from data POSTed to the script with content-type *application/x-www-form-urlencoded*. The *%pspForm* will only be filled by the script interpreter if the variable is used by the PSP script.

6.8 $pspPlain

This hash list contains raw data from the request if the content-type of the request body is other than *application/x-www-form-urlencoded*. The *$pspPlain* will be generated even if the variable isn't used by the PSP script.

6.9 $pspStatusCode

The HTTP response code to send to the browser. By default this is 200. As with *%pspResponse* these values need to be set in the sub *pspSetResponse* before the headers are sent to the browser.

6.10 Invoke Pi3:: callbacks

It is possible to invoke Pi3 API functions by using *Pi3::* callbacks from within PSP scriptlets or expressions if the Pi3Web PSP is used. A *PIObject* reference is given as parameter to *pspLoad* and *pspUnload*. Additional parameters of *pspSetResponse* are *PIHTTP* and *PIIOBuffer* references. This values could handed over to the respective *Pi3::* callbacks. This mechanisms are proprietary to Pi3Web and not part of the PSP language specification.

7 Example PSP scripts

7.1 Generic example

```
#!/usr/local/psp
<%
sub pspLoad {
   $counter = 0;
   return 0;
}

sub pspSetResponse {
   $counter++;
   return 0;
}
%>
<html><body>
<h1>PSP example</h1>
<p><b>The name of PSP script is: </b><%= $pspRequest{"Path"}; %>
<p><b>I've been invoked: </b><%= $counter; %>
<p><b>The local time is: </b><%= localtime(); %>
<p><b>Your browser is: </b><%= $pspRequest{"User-Agent"}; %>
</body></html>
```

7.2 Example, how to use package Pi3::

```
#!/usr/local/psp
<%
# this may extend perl library path
BEGIN { push @INC, qw(/my/perl/libs); }
use Pi3;
sub pspSetResponse
{
   $piObject = shift;
   return 0;
}
%>
<html><body>
<h1>PSP example, which invokes Pi3::</h1>
<p><b>Your Pi3 platform is: </b><%= Pi3::PIPlatform_getDescription(); %>
<p><b>Your Pi3Web server is: </b><%= Pi3::HTTPCore_getServerStamp(); %>
<p><b>The Pi3Web handler is: </b><%= Pi3::PIObject_getName($piObject); %>
</body></html>
```

7.3 Example, how to perform redirection

```
#!/usr/local/psp
<%
sub pspSetResponse
{
   $pspResponse{"Location"} = "http://localhost".$pspRequest{"PathInfo"};
   $pspResponseStatus = 302;
   return 0;
}
%>
<html><body>
If automatic redirect doesn’t work, use this <a href="<%= $pspResponse{"Location"} %>">link</a>.
</html></body>
```