1 Apache::PerlSections - Default Handler for Perl sections

1.1 Synopsis

```
<Perl >
@PerlModule = qw(Mail::Send Devel::Peek);

#run the server as whoever starts it
$User  = getpwuid(>) || >;
$Group = getgrgid()) || );

$ServerAdmin = $User;

</Perl>
```

1.2 Description

With `<Perl >`...`</Perl>` sections, it is possible to configure your server entirely in Perl.

`<Perl >` sections can contain *any* and as much Perl code as you wish. These sections are compiled into a special package whose symbol table mod_perl can then walk and grind the names and values of Perl variables/structures through the Apache core configuration gears.

Block sections such as `<Location>`..`</Location>` are represented in a `%Location` hash, e.g.:

```
<Perl>

$Location{"/~dougm/"} = {
  AuthUserFile   => '/tmp/htpasswd',
  AuthType       => 'Basic',
  AuthName       => 'test',
  DirectoryIndex => [qw(index.html index.htm)],
  Limit          => {
      METHODS => 'GET POST',
      require => 'user dougm',
  },
};

</Perl>
```

If an Apache directive can take two or three arguments you may push strings (the lowest number of arguments will be shifted off the `@list`) or use an array reference to handle any number greater than the minimum for that directive:

```
push @Redirect, "/foo", "http://www.foo.com/";

push @Redirect, "/imdb", "http://www.imdb.com/";

push @Redirect, [qw(temp "/here" "http://www.there.com")];
```

Other section counterparts include `%VirtualHost`, `%Directory` and `%Files`.

To pass all environment variables to the children with a single configuration directive, rather than listing each one via `PassEnv` or `PerlPassEnv`, a `<Perl >` section could read in a file and:

```
push @PerlPassEnv, [$key => $val];
```

or

```
Apache->httpd_conf("PerlPassEnv $key $val");
```

These are somewhat simple examples, but they should give you the basic idea. You can mix in any Perl code you desire. See *eg/httpd.conf.pl* and *eg/perl_sections.txt* in the mod_perl distribution for more examples.

Assume that you have a cluster of machines with similar configurations and only small distinctions between them: ideally you would want to maintain a single configuration file, but because the configurations aren't *exactly* the same (e.g. the `ServerName` directive) it's not quite that simple.

`<Perl >` sections come to rescue. Now you have a single configuration file and the full power of Perl to tweak the local configuration. For example to solve the problem of the `ServerName` directive you might have this `<Perl >` section:

```
<Perl >
$ServerName = ‘hostname‘;
</Perl>
```

For example if you want to allow personal directories on all machines except the ones whose names start with *secure*:

```
<Perl >
$ServerName = ‘hostname‘;
if ($ServerName !~ /^secure/) {
    $UserDir = "public.html";
}
else {
    $UserDir = "DISABLED";
}
</Perl>
```

1.3 Configuration Variables

There are a few variables that can be set to change the default behaviour of `<Perl >` sections.

1.3.1 *`$Apache::Server::SaveConfig`*

By default, the namespace in which `<Perl >` sections are evaluated is cleared after each block closes. By setting it to a true value, the content of those namespaces will be preserved and will be available for inspection by modules like `Apache::Status`.

1.3.2 *`$Apache::Server::StrictPerlSections`*

By default, compilation and run-time errors within `<Perl >` sections will cause a warning to be printed in the error_log. By setting this variable to a true value, code in the sections will be evaluated as if "use strict" was in usage, and all warning and errors will cause the server to abort startup and report the first error.

1.4 Advanced API

mod_perl 2.0 now introduces the same general concept of handlers to `<Perl >` sections. Apache::PerlSections simply being the default handler for them.

To specify a different handler for a given perl section, an extra handler argument must be given to the section:

```
<Perl handler="My::PerlSection::Handler" somearg="test1">
  $foo = 1;
  $bar = 2;
</Perl>
```

And in My/PerlSection/Handler.pm:

```
sub My::Handler::handler : handler {
    my($self, $parms, $args) = @_;
    #do your thing!
}
```

So, when that given `<Perl >` block in encountered, the code within will first be evaluated, then the handler routine will be invoked with 3 arguments

`$self` is self-explanatory

`$parms` is the *`Apache::CmdParms`* for this Container, for example, you might want to call `$parms`->server() to get the current server.

`$args` is an `APR::Table` object of the section arguments, the 2 guaranteed ones will be:

```
$args->{'handler'} = 'My::PerlSection::Handler';

$args->{'package'} = 'Apache::ReadConfig';
```

Other `name="value"` pairs given on the `<Perl >` line will also be included.

At this point, it's up to the handler routing to inspect the namespace of the `$args`->{'package'} and chooses what to do.

The most likely thing to do is to feed configuration data back into apache. To do that, use Apache::Server->add_config("directive"), for example:

```
  $parms->server->add_config("Alias /foo /bar");
```

Would create a new alias. The source code of `Apache::PerlSections` is a good place to look for a practical example.

1.5 Bugs

1.5.1 <Perl> directive missing closing '>'

httpd-2.0.47 and earlier had a bug in the configuration parser which caused the startup failure with the following error:

```
  Starting httpd:
  Syntax error on line ... of /etc/httpd/conf/httpd.conf:
  <Perl> directive missing closing '>'     [FAILED]
```

This has been fixed in httpd-2.0.48. If you can't upgrade to this or a higher version, please add a space before the closing '>' of the opening tag as a workaround. So if you had:

```
  <Perl>
  # some code
  </Perl>
```

change it to be:

```
  <Perl >
  # some code
  </Perl>
```

Table of Contents: