

# **1 mod\_perl 2.0 Server Configuration**

## 1.1 Description

This chapter provides an in-depth mod\_perl 2.0 configuration details.

## 1.2 mod\_perl configuration directives

Similar to mod\_perl 1.0, in order to use mod\_perl 2.0 a few configuration settings should be added to *httpd.conf*. They are quite similar to 1.0 settings but some directives were renamed and new directives were added.

## 1.3 Enabling mod\_perl

To enable mod\_perl built as DSO add to *httpd.conf*:

```
LoadModule perl_module modules/mod_perl.so
```

This setting specifies the location of the mod\_perl module relative to the `ServerRoot` setting, therefore you should put it somewhere after `ServerRoot` is specified.

If mod\_perl has been statically linked it's automatically enabled.

For Win32 specific details, see the documentation on Win32 configuration.

Remember that you can't use mod\_perl until you have configured Apache to use it. You need to configure Registry scripts or custom handlers.

## 1.4 Server Configuration Directives

### 1.4.1 *<Perl> Sections*

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

Please refer to the `Apache2::PerlSections` manpage for more information.

META: a dedicated chapter with examples?

See also: this directive argument types and allowed location.

### 1.4.2 *=pod, =over and =cut*

It's known that anything written between tokens `=pod` and `=cut` is ignored by the Perl parser. mod\_perl allows you to use the same technique to make Apache ignore things in *httpd.conf* (similar to `#` comments). With an exception to `=over apache` and `=over httpd` sections which are visible to Apache.

For example the following configuration:

```
#file: httpd.conf
=pod

PerlSetVar A 1

=over apache

PerlSetVar B 2

=back

PerlSetVar C 3

=cut

PerlSetVar D 4
```

Apache will see:

```
PerlSetVar B 2
PerlSetVar D 4
```

but not:

```
PerlSetVar A 1
PerlSetVar C 3
```

`=over httpd` is just an alias to `=over apache`. Remember that `=over` requires a corresponding `=back`.

### 1.4.3 *PerlAddVar*

`PerlAddVar` is useful if you need to pass in multiple values into the same variable emulating arrays and hashes. For example:

```
PerlAddVar foo bar
PerlAddVar foo bar1
PerlAddVar foo bar2
```

You would retrieve these values with:

```
my @foos = $r->dir_config->get('foo');
```

This would fill the `@foos` array with 'bar', 'bar1', and 'bar2'.

To pass in hashed values you need to ensure that you use an even number of directives per key. For example:

#### 1.4.4 PerlConfigRequire

```
PerlAddVar foo key1
PerlAddVar foo value1
PerlAddVar foo key2
PerlAddVar foo value2
```

You can then retrieve these values with:

```
my %foos = $r->dir_config->get('foo');
```

Where *%foos* will have a structure like:

```
%foos = (
    key1 => 'value1',
    key2 => 'value2',
);
```

See also: this directive argument types and allowed location.

### ***1.4.4 PerlConfigRequire***

`PerlConfigRequire` does the same thing as `PerlPostConfigRequire`, but it is executed as soon as it is encountered, i.e. during the configuration phase.

You should be using this directive to load only files that introduce new configuration directives, used later in the configuration file. For any other purposes (like preloading modules) use `PerlPostConfigRequire`.

One of the reasons for avoiding using the `PerlConfigRequire` directive, is that the `STDERR` stream is not available during the restart phase, therefore the errors will be not reported. It is not a bug in `mod_perl` but an Apache limitation. Use `PerlPostConfigRequire` if you can, and there you have the `STDERR` stream sent to the `error_log` file (by default).

See also: this directive argument types and allowed location.

### ***1.4.5 PerlLoadModule***

The `PerlLoadModule` directive is similar to `PerlModule`, in a sense that it loads a module. The difference is that it's used to triggers an early Perl startup. This can be useful for modules that need to be loaded early, as is the case for modules that implement new Apache directives, which are needed during the configuration phase.

See also: this directive argument types and allowed location.

### ***1.4.6 PerlModule***

```
PerlModule Foo::Bar
```

is equivalent to Perl's:

```
require Foo::Bar;
```

PerlModule is used to load modules using their package names.

You can pass one or more module names as arguments to PerlModule:

```
PerlModule Apache::DBI CGI DBD::Mysql
```

Notice, that normally, the Perl startup is delayed until after the configuration phase.

See also: PerlRequire.

See also: this directive argument types and allowed location.

## 1.4.7 PerlOptions

The directive PerlOptions provides fine-grained configuration for what were compile-time only options in the first mod\_perl generation. It also provides control over what class of Perl interpreter pool is used for a <VirtualHost> or location configured with <Location>, <Directory>, etc.

`$r->is_perl_option_enabled($option)` and `$s->is_perl_option_enabled($option)` can be used at run-time to check whether a certain `$option` has been enabled. (META: probably need to add/move this to the coding chapter)

Options are enabled by prepending + and disabled with -.

See also: this directive argument types and allowed location.

The available options are:

### 1.4.7.1 Enable

On by default, can be used to disable mod\_perl for a given VirtualHost. For example:

```
<VirtualHost ...>
  PerlOptions -Enable
</VirtualHost>
```

### 1.4.7.2 Clone

Share the parent Perl interpreter, but give the VirtualHost its own interpreter pool. For example should you wish to fine tune interpreter pools for a given virtual host:

```
<VirtualHost ...>
  PerlOptions +Clone
  PerlInterpStart 2
  PerlInterpMax 2
</VirtualHost>
```

This might be worthwhile in the case where certain hosts have their own sets of large-ish modules, used only in each host. By tuning each host to have its own pool, that host will continue to reuse the Perl allocations in their specific modules.

### 1.4.7.3 InheritSwitches

Off by default, can be used to have a `VirtualHost` inherit the value of the `PerlSwitches` from the parent server.

For instance, when cloning a Perl interpreter, to inherit the base Perl interpreter's `PerlSwitches` use:

```
<VirtualHost ...>
  PerlOptions +Clone +InheritSwitches
  ...
</VirtualHost>
```

### 1.4.7.4 Parent

Create a new parent Perl interpreter for the given `VirtualHost` and give it its own interpreter pool (implies the `Clone` option).

A common problem with `mod_perl 1.0` was the shared namespace between all code within the process. Consider two developers using the same server and each wants to run a different version of a module with the same name. This example will create two *parent* Perl interpreters, one for each `<VirtualHost>`, each with its own namespace and pointing to a different paths in `@INC`:

**META:** is `-Mlib` portable? (problems with `-Mlib` on Darwin/5.6.0?)

```
<VirtualHost ...>
  ServerName dev1
  PerlOptions +Parent
  PerlSwitches -Mlib=/home/dev1/lib/perl
</VirtualHost>

<VirtualHost ...>
  ServerName dev2
  PerlOptions +Parent
  PerlSwitches -Mlib=/home/dev2/lib/perl
</VirtualHost>
```

Remember that `+Parent` gives you a completely new Perl interpreters pool, so all your modifications to `@INC` and preloading of the modules should be done again. Consider using `PerlOptions +Clone` if you want to inherit from the parent Perl interpreter.

Or even for a given location, for something like "dirty" cgi scripts:

```
<Location /cgi-bin>
  PerlOptions +Parent
  PerlInterpMaxRequests 1
  PerlInterpStart 1
  PerlInterpMax 1
  PerlResponseHandler ModPerl::Registry
</Location>
```

will use a fresh interpreter with its own namespace to handle each request.

### 1.4.7.5 Perl\*Handler

Disable Perl\*Handlers, all compiled-in handlers are enabled by default. The option name is derived from the Perl\*Handler name, by stripping the Perl and Handler parts of the word. So PerlLogHandler becomes Log which can be used to disable PerlLogHandler:

```
PerlOptions -Log
```

Suppose one of the hosts does not want to allow users to configure PerlAuthenHandler, PerlAuthzHandler, PerlAccessHandler and <Perl> sections:

```
<VirtualHost ...>
  PerlOptions -Authen -Authz -Access -Sections
</VirtualHost>
```

Or maybe everything but the response handler:

```
<VirtualHost ...>
  PerlOptions None +Response
</VirtualHost>
```

### 1.4.7.6 AutoLoad

Resolve Perl\*Handlers at startup time, which includes loading the modules from disk if not already loaded.

In mod\_perl 1.0, configured Perl\*Handlers which are not a fully qualified subroutine names are resolved at request time, loading the handler module from disk if needed. In mod\_perl 2.0, configured Perl\*Handlers are resolved at startup time. By default, modules are not auto-loaded during startup-time resolution. It is possible to enable this feature with:

```
PerlOptions +Autoload
```

Consider this configuration:

```
PerlResponseHandler Apache::Magick
```

In this case, Apache::Magick is the package name, and the subroutine name will default to *handler*. If the Apache::Magick module is not already loaded, PerlOptions +Autoload will attempt to pull it in at startup time. With this option enabled you don't have to explicitly load the handler modules. For example you don't need to add:

```
PerlModule Apache::Magick
```

in our example.

Another way to preload only specific modules is to add `+` when configuring those, for example:

```
PerlResponseHandler +Apache::Magick
```

will automatically preload the `Apache::Magick` module.

### 1.4.7.7 GlobalRequest

Setup the global `$r` object for use with `Apache2->request`.

This setting is enabled by default during the `PerlResponseHandler` phase for sections configured as:

```
<Location ...>
    SetHandler perl-script
    ...
</Location>
```

but is not enabled by default for sections configured as:

```
<Location ...>
    SetHandler modperl
    ....
</Location>
```

And can be disabled with:

```
<Location ...>
    SetHandler perl-script
    PerlOptions -GlobalRequest
    ...
</Location>
```

Notice that if you need the global request object during other phases, you will need to explicitly enable it in the configuration file.

You can also set that global object from the handler code, like so:

```
sub handler {
    my $r = shift;
    Apache2::RequestUtil->request($r);
    ...
}
```

The `+GlobalRequest` setting is needed for example if you use older versions of `CGI.pm` to process the incoming request. Starting from version 2.93, `CGI.pm` optionally accepts `$r` as an argument to `new()`, like so:

```

sub handler {
    my $r = shift;
    my $q = CGI->new($r);
    ...
}

```

Remember that inside registry scripts you can always get `$r` at the beginning of the script, since it gets wrapped inside a subroutine and accepts `$r` as the first and the only argument. For example:

```

#!/usr/bin/perl
use CGI;
my $r = shift;
my $q = CGI->new($r);
...

```

of course you won't be able to run this under `mod_cgi`, so you may need to do:

```

#!/usr/bin/perl
use CGI;
my $q = $ENV{MOD_PERL} ? CGI->new(shift @_ ) : CGI->new();
...

```

in order to have the script running under `mod_perl` and `mod_cgi`.

### 1.4.7.8 ParseHeaders

Scan output for HTTP headers, same functionality as `mod_perl 1.0`'s `PerlSendHeader`, but more robust. This option is usually needs to be enabled for registry scripts which send the HTTP header with:

```
print "Content-type: text/html\n\n";
```

### 1.4.7.9 MergeHandlers

Turn on merging of `Perl*Handler` arrays. For example with a setting:

```

PerlFixupHandler Apache2::FixupA

<Location /inside>
    PerlFixupHandler Apache2::FixupB
</Location>

```

a request for `/inside` only runs `Apache2::FixupB` (`mod_perl 1.0` behavior). But with this configuration:

```

PerlFixupHandler Apache2::FixupA

<Location /inside>
    PerlOptions +MergeHandlers
    PerlFixupHandler Apache2::FixupB
</Location>

```

a request for `/inside` will run both `Apache2::FixupA` and `Apache2::FixupB` handlers.

### 1.4.7.10 SetupEnv

Set up environment variables for each request ala `mod_cgi`.

When this option is enabled, *mod\_perl* fiddles with the environment to make it appear as if the code is called under the `mod_cgi` handler. For example, the `$ENV{QUERY_STRING}` environment variable is initialized with the contents of `Apache2::args()`, and the value returned by `Apache2::server_hostname()` is put into `$ENV{SERVER_NAME}`.

But `%ENV` population is expensive. Those who have moved to the Perl Apache API no longer need this extra `%ENV` population, and can gain by disabling it. A code using the `CGI.pm` module require `PerlOptions +SetupEnv` because that module relies on a properly populated CGI environment table.

This option is enabled by default for sections configured as:

```
<Location ...>
    SetHandler perl-script
    ...
</Location>
```

Since this option adds an overhead to each request, if you don't need this functionality you can turn it off for a certain section:

```
<Location ...>
    SetHandler perl-script
    PerlOptions -SetupEnv
    ...
</Location>
```

or globally:

```
PerlOptions -SetupEnv
<Location ...>
    ...
</Location>
```

and then it'll affect the whole server. It can still be enabled for sections that need this functionality.

When this option is disabled you can still read environment variables set by you. For example when you use the following configuration:

```
PerlOptions -SetupEnv
PerlModule ModPerl::Registry
<Location /perl>
    PerlSetEnv TEST hi
    SetHandler perl-script
    PerlResponseHandler ModPerl::Registry
    Options +ExecCGI
</Location>
```

and you issue a request for this script:

```
setupenvoff.pl
-----
use Data::Dumper;
my $r = Apache2::RequestUtil->request();
$r->content_type('text/plain');
print Dumper(\%ENV);
```

you should see something like this:

```
$VAR1 = {
    'GATEWAY_INTERFACE' => 'CGI-Perl/1.1',
    'MOD_PERL' => 'mod_perl/2.0.1',
    'PATH' => 'bin:/usr/bin',
    'TEST' => 'hi'
};
```

Notice that we have got the value of the environment variable *TEST*.

## 1.4.8 PerlPassEnv

PerlPassEnv instructs mod\_perl to pass the environment variables you specify to your mod\_perl handlers. This is useful if you need to set the same environment variables for your shell as well as mod\_perl. For example if you had this in your `.bash_profile`:

```
export ORACLE_HOME=/oracle
```

And defined the following in your `httpd.conf`:

```
PerlPassEnv ORACLE_HOME
```

The your mod\_perl handlers would have access to the value via the standard Perl mechanism:

```
my $oracle_home = $ENV{'ORACLE_HOME'};
```

See also: this directive argument types and allowed location.

## 1.4.9 PerlPostConfigRequire

```
PerlPostConfigRequire /home/httpd/perl/lib/startup.pl
```

is equivalent to Perl's:

```
require "/home/httpd/perl/lib/startup.pl";
```

A PerlRequire filename argument can be absolute or relative to ServerRoot or a filepath in Perl's @INC.

You can pass one or more filenames as arguments to `PerlPostConfigRequire`:

```
PerlPostConfigRequire path1/startup.pl path2/startup.pl
```

`PerlPostConfigRequire` is used to load files with Perl code to be run at the server startup. It's not executed as soon as it is encountered, but as late as possible during the server startup.

Most of the time you should be using this directive. For example to preload some modules or run things at the server startup). Only if you need to load modules that introduce new configuration directives, used later in the configuration file you should use `PerlConfigRequire`.

As with any file with Perl code that gets `use()`'d or `require()`'d, it must return a *true* value. To ensure that this happens don't forget to add `1;` at the end of *startup.pl*.

See also: `PerlModule` and `PerlLoadModule`.

See also: this directive argument types and allowed location.

### ***1.4.10 PerlRequire***

`PerlRequire` does the same thing as `PerlPostConfigRequire`, but you have almost no control of when this code is going to be executed. Therefore you should be using either `PerlConfigRequire` (executes immediately) or `PerlPostConfigRequire` (executes just before the end of the server startup) instead. Most of the time you want to use the latter.

See also: this directive argument types and allowed location.

### ***1.4.11 PerlSetEnv***

`PerlSetEnv` allows you to specify system environment variables and pass them into your `mod_perl` handlers. These values are then available through the normal perl `%ENV` mechanisms. For example:

```
PerlSetEnv TEMPLATE_PATH /usr/share/templates
```

would create `$ENV{ 'TEMPLATE_PATH' }` and set it to */usr/share/templates*.

See also: this directive argument types and allowed location.

### ***1.4.12 PerlSetVar***

`PerlSetVar` allows you to pass variables into your `mod_perl` handlers from your *httpd.conf*. This method is preferable to using `PerlSetEnv` or Apache's `SetEnv` and `PassEnv` methods because of the overhead of having to populate `%ENV` for each request. An example of how this can be used is:

```
PerlSetVar foo bar
```

To retrieve the value of that variable in your Perl code you would use:

```
my $foo = $r->dir_config('foo');
```

In this example `$foo` would then hold the value `'bar'`. **NOTE:** that these directives are parsed at request time which is a slower method than using custom Apache configuration directives

See also: this directive argument types and allowed location.

### 1.4.13 PerlSwitches

Now you can pass any Perl's command line switches in *httpd.conf* using the `PerlSwitches` directive. For example to enable warnings and Taint checking add:

```
PerlSwitches -wT
```

As an alternative to using `use lib` in *startup.pl* to adjust `@INC`, now you can use the command line switch `-I` to do that:

```
PerlSwitches -I/home/stas/modperl
```

You could also use `-Mlib=/home/stas/modperl` which is the exact equivalent as `use lib`, but it's broken on certain platforms/version (e.g. Darwin/5.6.0). `use lib` is removing duplicated entries, whereas `-I` does not.

See also: this directive argument types and allowed location.

### 1.4.14 SetHandler

mod\_perl 2.0 provides two types of `SetHandler` handlers: `modperl` and `perl-script`. The `SetHandler` directive is only relevant for response phase handlers. It doesn't affect other phases.

See also: this directive argument types and allowed location.

#### 1.4.14.1 modperl

Configured as:

```
SetHandler modperl
```

The bare `mod_perl` handler type, which just calls the `Perl*Handler`'s callback function. If you don't need the features provided by the *perl-script* handler, with the `modperl` handler, you can gain even more performance. (This handler isn't available in mod\_perl 1.0.)

Unless the `Perl*Handler` callback, running under the `modperl` handler, is configured with:

```
PerlOptions +SetupEnv
```

or calls:

```
$r->subprocess_env;
```

in a void context with no arguments (which has the same effect as `PerlOptions +SetupEnv` for the handler that called it), only the following environment variables are accessible via `%ENV`:

- `MOD_PERL` and `MOD_PERL_API_VERSION` (always)
- `PATH` and `TZ` (if you had them defined in the shell or *httpd.conf*)

Therefore if you don't want to add the overhead of populating `%ENV`, when you simply want to pass some configuration variables from *httpd.conf*, consider using `PerlSetVar` and `PerlAddVar` instead of `PerlSetEnv` and `PerlPassEnv`. In your code you can retrieve the values using the `dir_config()` method. For example if you set in *httpd.conf*:

```
<Location /print_env2>
  SetHandler modperl
  PerlResponseHandler Apache2::VarTest
  PerlSetVar VarTest VarTestValue
</Location>
```

this value can be retrieved inside `Apache2::VarTest::handler()` with:

```
$r->dir_config('VarTest');
```

Alternatively use the Apache core directives `SetEnv` and `PassEnv`, which always populate `r->subprocess_env`, but this doesn't happen until the Apache *fixups* phase, which could be too late for your needs.

Notice also that this handler does not reset `%ENV` after each request's response phase, so if one response handler has changed `%ENV` without localizing the change, it'll affect other handlers running after it as well.

### 1.4.14.2 perl-script

Configured as:

```
SetHandler perl-script
```

Most `mod_perl` handlers use the *perl-script* handler. Among other things it does:

- `PerlOptions +GlobalRequest` is in effect only during the `PerlResponseHandler` phase unless:

```
PerlOptions -GlobalRequest
```

is specified.

- PerlOptions +SetupEnv is in effect unless:

```
PerlOptions -SetupEnv
```

is specified.

- STDIN and STDOUT get tied to the request object \$r, which makes possible to read from STDIN and print directly to STDOUT via CORE::print(), instead of implicit calls like \$r->puts().
- Several special global Perl variables are saved before the response handler is called and restored afterwards (similar to mod\_perl 1.0). This includes: %ENV, @INC, \$/, STDOUT's \$| and END blocks array (PL\_endav).
- Entries added to %ENV are passed on to the subprocess\_env table, and are thus accessible via r->subprocess\_env during the later PerlLogHandler and PerlCleanupHandler phases.

### 1.4.14.3 Examples

Let's demonstrate the differences between the modperl and the perl-script core handlers in the following example, which represents a simple mod\_perl response handler which prints out the environment variables as seen by it:

```
file:MyApache2/PrintEnv1.pm
-----
package MyApache2::PrintEnv1;
use strict;

use Apache2::RequestRec (); # for $r->content_type
use Apache2::RequestIO (); # for print
use Apache2::Const -compile => ':common';

sub handler {
    my $r = shift;

    $r->content_type('text/plain');
    for (sort keys %ENV){
        print "$_ => $ENV{$_}\n";
    }

    return Apache2::Const::OK;
}

1;
```

This is the required configuration:

```
PerlModule MyApache2::PrintEnv1
<Location /print_env1>
    SetHandler perl-script
    PerlResponseHandler MyApache2::PrintEnv1
</Location>
```

Now issue a request to *http://localhost/print\_env1* and you should see all the environment variables printed out.

Here is the same response handler, adjusted to work with the `modperl` core handler:

```
file:MyApache2/PrintEnv2.pm
-----
package MyApache2::PrintEnv2;
use strict;

use Apache2::RequestRec (); # for $r->content_type
use Apache2::RequestIO (); # for $r->print

use Apache2::Const -compile => ':common';

sub handler {
    my $r = shift;

    $r->content_type('text/plain');
    $r->subprocess_env;
    for (sort keys %ENV){
        $r->print("$_ => $ENV{$_}\n");
    }

    return Apache2::Const::OK;
}

1;
```

The configuration now will look as:

```
PerlModule MyApache2::PrintEnv2
<Location /print_env2>
    SetHandler modperl
    PerlResponseHandler MyApache2::PrintEnv2
</Location>
```

`MyApache2::PrintEnv2` cannot use `print()` and therefore uses `$r->print()` to generate a response. Under the `modperl` core handler `%ENV` is not populated by default, therefore `subprocess_env()` is called in a void context. Alternatively we could configure this section to do:

```
PerlOptions +SetupEnv
```

If you issue a request to *http://localhost/print\_env2*, you should see all the environment variables printed out as with *http://localhost/print\_env1*.

## 1.5 Server Life Cycle Handlers Directives

See Server life cycle.

### ***1.5.1 PerlOpenLogsHandler***

See PerlOpenLogsHandler.

### ***1.5.2 PerlPostConfigHandler***

See PerlPostConfigHandler.

### ***1.5.3 PerlChildInitHandler***

See PerlChildInitHandler.

### ***1.5.4 PerlChildExitHandler***

See PerlChildExitHandler.

## **1.6 Protocol Handlers Directives**

See Protocol handlers.

### ***1.6.1 PerlPreConnectionHandler***

See PerlPreConnectionHandler.

### ***1.6.2 PerlProcessConnectionHandler***

See PerlProcessConnectionHandler.

## **1.7 Filter Handlers Directives**

mod\_perl filters are described in the filter handlers tutorial, `Apache2::Filter` and `Apache2::FilterRec` manpages.

The following filter handler configuration directives are available:

### ***1.7.1 PerlInputFilterHandler***

See PerlInputFilterHandler.

### ***1.7.2 PerlOutputFilterHandler***

See PerlOutputFilterHandler.

### ***1.7.3 PerlSetInputFilter***

See PerlSetInputFilter.

### ***1.7.4 PerlSetOutputFilter***

See PerlSetInputFilter.

## **1.8 HTTP Protocol Handlers Directives**

See HTTP protocol handlers.

### ***1.8.1 PerlPostReadRequestHandler***

See PerlPostReadRequestHandler.

### ***1.8.2 PerlTransHandler***

See PerlTransHandler.

### ***1.8.3 PerlMapToStorageHandler***

See PerlMapToStorageHandler.

### ***1.8.4 PerlInitHandler***

See PerlInitHandler.

### ***1.8.5 PerlHeaderParserHandler***

See PerlHeaderParserHandler.

### ***1.8.6 PerlAccessHandler***

See PerlAccessHandler.

### ***1.8.7 PerlAuthenHandler***

See PerlAuthenHandler.

### ***1.8.8 PerlAuthzHandler***

See PerlAuthzHandler.

### ***1.8.9 PerlTypeHandler***

See PerlTypeHandler.

### ***1.8.10 PerlFixupHandler***

See PerlFixupHandler.

### ***1.8.11 PerlResponseHandler***

See PerlResponseHandler.

### ***1.8.12 PerlLogHandler***

See PerlLogHandler.

### ***1.8.13 PerlCleanupHandler***

See PerlCleanupHandler.

## **1.9 Threads Mode Specific Directives**

These directives are enabled only in a threaded mod\_perl+Apache combo:

### ***1.9.1 PerlInterpStart***

The number of interpreters to clone at startup time.

Default value: 3

See also: this directive argument types and allowed location.

## ***1.9.2 PerlInterpMax***

If all running interpreters are in use, `mod_perl` will clone new interpreters to handle the request, up until this number of interpreters is reached. when `PerlInterpMax` is reached, `mod_perl` will block (via `COND_WAIT()`) until one becomes available (signaled via `COND_SIGNAL()`).

Default value: 5

See also: this directive argument types and allowed location.

## ***1.9.3 PerlInterpMinSpare***

The minimum number of available interpreters this parameter will clone interpreters up to `PerlInterpMax`, before a request comes in.

Default value: 3

See also: this directive argument types and allowed location.

## ***1.9.4 PerlInterpMaxSpare***

`mod_perl` will throttle down the number of interpreters to this number as those in use become available.

Default value: 3

## ***1.9.5 PerlInterpMaxRequests***

The maximum number of requests an interpreter should serve, the interpreter is destroyed when the number is reached and replaced with a fresh clone.

Default value: 2000

See also: this directive argument types and allowed location.

## ***1.9.6 PerlInterpScope***

As mentioned, when a request in a threaded mpm is handled by `mod_perl`, an interpreter must be pulled from the interpreter pool. The interpreter is then only available to the thread that selected it, until it is released back into the interpreter pool. By default, an interpreter will be held for the lifetime of the request, equivalent to this configuration:

```
PerlInterpScope request
```

For example, if a `PerlAccessHandler` is configured, an interpreter will be selected before it is run and not released until after the logging phase.

Interpreters will be shared across sub-requests by default, however, it is possible to configure the interpreter scope to be per-sub-request on a per-directory basis:

```
PerlInterpScope subrequest
```

With this configuration, an autoindex generated page, for example, would select an interpreter for each item in the listing that is configured with a `Perl*Handler`.

It is also possible to configure the scope to be per-handler:

```
PerlInterpScope handler
```

For example if `PerlAccessHandler` is configured, an interpreter will be selected before running the handler, and put back immediately afterwards, before Apache moves onto the next phase. If a `PerlFixupHandler` is configured further down the chain, another interpreter will be selected and again put back afterwards, before `PerlResponseHandler` is run.

For protocol handlers, the interpreter is held for the lifetime of the connection. However, a C protocol module might hook into `mod_perl` (e.g. `mod_ftp`) and provide a `request_rec` record. In this case, the default scope is that of the request. Should a `mod_perl` handler want to maintain state for the lifetime of an ftp connection, it is possible to do so on a per-virtualhost basis:

```
PerlInterpScope connection
```

Default value: `request`

See also: this directive argument types and allowed location.

## 1.10 Debug Directives

### 1.10.1 *PerlTrace*

The `PerlTrace` is used for tracing the `mod_perl` execution. This directive is enabled when `mod_perl` is compiled with the `MP_TRACE=1` option.

To enable tracing, add to *httpd.conf*:

```
PerlTrace [level]
```

where `level` is either:

```
all
```

which sets maximum logging and debugging levels;

a combination of one or more option letters from the following list:

```

a Apache API interaction
c configuration for directive handlers
d directive processing
f filters
e environment variables
g globals management
h handlers
i interpreter pool management
m memory allocations
o I/O
r Perl runtime interaction
s Perl sections
t benchmark-ish timings

```

Tracing options add to the previous setting and don't override it. So for example:

```

PerlTrace c
...
PerlTrace f

```

will set tracing level first to 'c' and later to 'cf'. If you wish to override settings, unset any previous setting by assigning 0 (zero), like so:

```

PerlTrace c
...
PerlTrace 0
PerlTrace f

```

now the tracing level is set only to 'f'. You can't mix the number 0 with letters, it must be alone.

When `PerlTrace` is not specified, the tracing level will be set to the value of the `$ENV{MOD_PERL_TRACE}` environment variable.

See also: this directive argument types and allowed location.

## 1.11 mod\_perl Directives Argument Types and Allowed Location

The following table shows where in the configuration files `mod_perl` configuration directives are allowed to appear, what kind and how many arguments they expect:

General directives:

Directive	Arguments	Scope
<code>PerlSwitches</code>	ITERATE	SRV
<code>PerlRequire</code>	ITERATE	SRV
<code>PerlConfigRequire</code>	ITERATE	SRV
<code>PerlPostConfigRequire</code>	ITERATE	SRC
<code>PerlModule</code>	ITERATE	SRV
<code>PerlLoadModule</code>	RAW_ARGS	SRV
<code>PerlOptions</code>	ITERATE	DIR

PerlSetVar	TAKE2	DIR
PerlAddVar	ITERATE2	DIR
PerlSetEnv	TAKE2	DIR
PerlPassEnv	TAKE1	SRV
<Perl> Sections	RAW_ARGS	SRV
PerlTrace	TAKE1	SRV

### Handler assignment directives:

Directive	Arguments	Scope
PerlOpenLogsHandler	ITERATE	SRV
PerlPostConfigHandler	ITERATE	SRV
PerlChildInitHandler	ITERATE	SRV
PerlChildExitHandler	ITERATE	SRV
PerlPreConnectionHandler	ITERATE	SRV
PerlProcessConnectionHandler	ITERATE	SRV
PerlPostReadRequestHandler	ITERATE	SRV
PerlTransHandler	ITERATE	SRV
PerlMapToStorageHandler	ITERATE	SRV
PerlInitHandler	ITERATE	DIR
PerlHeaderParserHandler	ITERATE	DIR
PerlAccessHandler	ITERATE	DIR
PerlAuthenHandler	ITERATE	DIR
PerlAuthzHandler	ITERATE	DIR
PerlTypeHandler	ITERATE	DIR
PerlFixupHandler	ITERATE	DIR
PerlResponseHandler	ITERATE	DIR
PerlLogHandler	ITERATE	DIR
PerlCleanupHandler	ITERATE	DIR
PerlInputFilterHandler	ITERATE	DIR
PerlOutputFilterHandler	ITERATE	DIR
PerlSetInputFilter	ITERATE	DIR
PerlSetOutputFilter	ITERATE	DIR

### Perl Interpreter management directives:

Directive	Arguments	Scope
PerlInterpStart	TAKE1	SRV
PerlInterpMax	TAKE1	SRV
PerlInterpMinSpare	TAKE1	SRV
PerlInterpMaxSpare	TAKE1	SRV
PerlInterpMaxRequests	TAKE1	SRV
PerlInterpScope	TAKE1	DIR

### mod\_perl 1.0 back-compatibility directives:

Directive	Arguments	Scope
PerlHandler	ITERATE	DIR
PerlSendHeader	FLAG	DIR
PerlSetupEnv	FLAG	DIR
PerlTaintCheck	FLAG	SRV
PerlWarn	FLAG	SRV

The *Arguments* column represents the type of arguments directives accepts, where:

- **ITERATE**

Expects a list of arguments.

- **ITERATE2**

Expects one argument, followed by at least one or more arguments.

- **TAKE1**

Expects one argument only.

- **TAKE2**

Expects two arguments only.

- **FLAG**

One of On or Off (case insensitive).

- **RAW\_ARGS**

The function parses the command line by itself.

The *Scope* column shows the location the directives are allowed to appear in:

- **SRV**

Global configuration and `<VirtualHost>` (mnemonic: *SeRVer*). These directives are defined as `RSRC_CONF` in the source code.

- **DIR**

`<Directory>`, `<Location>`, `<Files>` and all their regular expression variants (mnemonic: *DIRectory*). These directives can also appear in `.htaccess` files. These directives are defined as `OR_ALL` in the source code.

These directives can also appear in the global server configuration and `<VirtualHost>`.

Apache specifies other allowed location types which are currently not used by the core mod\_perl directives and their definition can be found in *include/httpd\_config.h* (hint: search for RSRC\_CONF).

Also see Stacked Handlers.

## 1.12 Server Startup Options Retrieval

Inside *httpd.conf* one can do conditional configuration based on the define options passed at the server startup. For example:

```
<IfDefine PERLDB>
  <Perl>
    use Apache::DB ();
    Apache::DB->init;
  </Perl>

  <Location />
    PerlFixupHandler Apache::DB
  </Location>
</IfDefine>
```

So only when the server is started as:

```
% httpd C<-DPERLDB> ...
```

The configuration inside *IfDefine* will have an effect. If you want to have some configuration section to have an effect if a certain define wasn't defined use *!*, for example here is the opposite of the previous example:

```
<IfDefine !PERLDB>
  # ...
</IfDefine>
```

If you need to access any of the startup defines in the Perl code you use `Apache2::ServerUtil::exists_config_define()`. For example in a startup file you can say:

```
use Apache2::ServerUtil ();
if (Apache2::ServerUtil::exists_config_define("PERLDB")) {
  require Apache::DB;
  Apache::DB->init;
}
```

For example to check whether the server has been started in a single mode use:

```
if (Apache2::ServerUtil::exists_config_define("ONE_PROCESS")) {
  print "Running in a single mode";
}
```

### 1.12.1 MODPERL2 Define Option

When running under mod\_perl 2.0 a special configuration "define" symbol MODPERL2 is enabled internally, as if the server had been started with `-DMODPERL2`. For example this can be used to write a configuration file which needs to do something different whether it's running under mod\_perl 1.0 or 2.0:

```
<IfDefine MODPERL2>
  # 2.0 configuration
</IfDefine>
<IfDefine !MODPERL2>
  # else
</IfDefine>
```

From within Perl code this can be tested with `Apache2::ServerUtil::exists_config_define()`, for example:

```
use Apache2::ServerUtil ();
if (Apache2::ServerUtil::exists_config_define("MODPERL2")) {
  # some 2.0 specific code
}
```

## 1.13 Perl Interface to the Apache Configuration Tree

For now refer to the `Apache2::Directive` manpage and the test `t/response/TestApache2/confree.pm` in the mod\_perl source distribution.

META: need help to write the tutorial section on this with examples.

## 1.14 Adjusting @INC

You can always adjust contents of @INC before the server starts. There are several ways to do that.

- *startup.pl*

In the startup file you can use the `lib` pragma like so:

```
use lib qw(/home/httpd/project1/lib /tmp/lib);
use lib qw(/home/httpd/project2/lib);
```

- *httpd.conf*

In *httpd.conf* you can use the `PerlSwitches` directive to pass arguments to perl as you do from the command line, e.g.:

```
PerlSwitches -I/home/httpd/project1/lib -I/tmp/lib
PerlSwitches -I/home/httpd/project2/lib
```

### 1.14.1 PERL5LIB and PERLLIB Environment Variables

The effect of the PERL5LIB and PERLLIB environment variables on @INC is described in the *perlrun* manpage. mod\_perl 2.0 doesn't do anything special about them.

It's important to remind that both PERL5LIB and PERLLIB are ignored when the taint mode (`PerlSwitches -T`) is in effect. Since you want to make sure that your mod\_perl server is running under the taint mode, you can't use the PERL5LIB and PERLLIB environment variables.

However there is the *perl5lib* module on CPAN, which, if loaded, bypasses perl's security and will affect @INC. Use it only if you know what you are doing.

### 1.14.2 Modifying @INC on a Per-VirtualHost

If Perl used with mod\_perl was built with ithreads support one can specify different @INC values for different VirtualHosts, using a combination of `PerlOptions +Parent` and `PerlSwitches`. For example:

```
<VirtualHost ...>
  ServerName dev1
  PerlOptions +Parent
  PerlSwitches -I/home/dev1/lib/perl
</VirtualHost>

<VirtualHost ...>
  ServerName dev2
  PerlOptions +Parent
  PerlSwitches -I/home/dev2/lib/perl
</VirtualHost>
```

This technique works under any MPM with ithreads-enabled perl. It's just that under prefork your procs will be huge, because you will build a pool of interpreters in each process. While the same happens under threaded mpm, there you have many threads per process, so you need just 1 or 2 procs and therefore less memory will be used.

## 1.15 General Issues

### 1.16 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Stas Bekman [<http://stason.org/>]

## 1.17 Authors

- Doug MacEachern <dougm (at) covalent.net>
- Stas Bekman [<http://stason.org/>]

Only the major authors are listed above. For contributors see the Changes file.

## Table of Contents:

1	mod_perl 2.0 Server Configuration . . . . .	1
1.1	Description . . . . .	2
1.2	mod_perl configuration directives . . . . .	2
1.3	Enabling mod_perl . . . . .	2
1.4	Server Configuration Directives . . . . .	2
1.4.1	<Perl> Sections . . . . .	2
1.4.2	=pod, =over and =cut . . . . .	2
1.4.3	PerlAddVar . . . . .	3
1.4.4	PerlConfigRequire . . . . .	4
1.4.5	PerlLoadModule . . . . .	4
1.4.6	PerlModule . . . . .	4
1.4.7	PerlOptions . . . . .	5
1.4.7.1	Enable . . . . .	5
1.4.7.2	Clone . . . . .	5
1.4.7.3	InheritSwitches . . . . .	6
1.4.7.4	Parent . . . . .	6
1.4.7.5	Perl*Handler . . . . .	7
1.4.7.6	AutoLoad . . . . .	7
1.4.7.7	GlobalRequest . . . . .	8
1.4.7.8	ParseHeaders . . . . .	9
1.4.7.9	MergeHandlers . . . . .	9
1.4.7.10	SetupEnv . . . . .	10
1.4.8	PerlPassEnv . . . . .	11
1.4.9	PerlPostConfigRequire . . . . .	11
1.4.10	PerlRequire . . . . .	12
1.4.11	PerlSetEnv . . . . .	12
1.4.12	PerlSetVar . . . . .	12
1.4.13	PerlSwitches . . . . .	13
1.4.14	SetHandler . . . . .	13
1.4.14.1	modperl . . . . .	13
1.4.14.2	perl-script . . . . .	14
1.4.14.3	Examples . . . . .	15
1.5	Server Life Cycle Handlers Directives . . . . .	16
1.5.1	PerlOpenLogsHandler . . . . .	17
1.5.2	PerlPostConfigHandler . . . . .	17
1.5.3	PerlChildInitHandler . . . . .	17
1.5.4	PerlChildExitHandler . . . . .	17
1.6	Protocol Handlers Directives . . . . .	17
1.6.1	PerlPreConnectionHandler . . . . .	17
1.6.2	PerlProcessConnectionHandler . . . . .	17
1.7	Filter Handlers Directives . . . . .	17
1.7.1	PerlInputFilterHandler . . . . .	17
1.7.2	PerlOutputFilterHandler . . . . .	18
1.7.3	PerlSetInputFilter . . . . .	18

1.7.4 PerlSetOutputFilter . . . . .	18
1.8 HTTP Protocol Handlers Directives . . . . .	18
1.8.1 PerlPostReadRequestHandler . . . . .	18
1.8.2 PerlTransHandler . . . . .	18
1.8.3 PerlMapToStorageHandler . . . . .	18
1.8.4 PerlInitHandler . . . . .	18
1.8.5 PerlHeaderParserHandler . . . . .	18
1.8.6 PerlAccessHandler . . . . .	18
1.8.7 PerlAuthenHandler . . . . .	19
1.8.8 PerlAuthzHandler . . . . .	19
1.8.9 PerlTypeHandler . . . . .	19
1.8.10 PerlFixupHandler . . . . .	19
1.8.11 PerlResponseHandler . . . . .	19
1.8.12 PerlLogHandler . . . . .	19
1.8.13 PerlCleanupHandler . . . . .	19
1.9 Threads Mode Specific Directives . . . . .	19
1.9.1 PerlInterpStart . . . . .	19
1.9.2 PerlInterpMax . . . . .	20
1.9.3 PerlInterpMinSpare . . . . .	20
1.9.4 PerlInterpMaxSpare . . . . .	20
1.9.5 PerlInterpMaxRequests . . . . .	20
1.9.6 PerlInterpScope . . . . .	20
1.10 Debug Directives . . . . .	21
1.10.1 PerlTrace . . . . .	21
1.11 mod_perl Directives Argument Types and Allowed Location . . . . .	22
1.12 Server Startup Options Retrieval . . . . .	25
1.12.1 MODPERL2 Define Option . . . . .	26
1.13 Perl Interface to the Apache Configuration Tree . . . . .	26
1.14 Adjusting @INC . . . . .	26
1.14.1 PERL5LIB and PERLLIB Environment Variables . . . . .	27
1.14.2 Modifying @INC on a Per-VirtualHost . . . . .	27
1.15 General Issues . . . . .	27
1.16 Maintainers . . . . .	27
1.17 Authors . . . . .	28