1 Writing mod_perl Handlers and Scripts
1.1 Description

This chapter covers the mod_perl coding specifics, different from normal Perl coding. Most other perl coding issues are covered in the perl manpages and rich literature.

1.2 Prerequisites

1.3 Where the Methods Live

mod_perl 2.0 has all its methods spread across many modules. In order to use these methods the modules containing them have to be loaded first. If you don’t do that mod_perl will complain that it can’t find the methods in question. The module ModPerl::MethodLookup can be used to find out which modules need to be used.

1.4 Techniques

1.4.1 Method Handlers

In addition to function handlers method handlers can be used. Method handlers are useful when you want to write code that takes advantage of inheritance. To make the handler act as a method under mod_perl 2, use the method attribute.

See the Perl attributes manpage for details on the attributes syntax (perldoc attributes).

For example:

```perl
package Bird::Eagle;
@ISA = qw(Bird);

sub handler : method {
    my ($class_or_object, $r) = @_;
    ...;
}

sub new { bless (), __PACKAGE__ }
```

and then register it as:

```perl
PerlResponseHandler Bird::Eagle
```

When mod_perl sees that the handler has a method attribute, it passes two arguments to it: the calling object or a class, depending on how it was called, and the request object, as shown above.

If `Class->method` syntax is used for a Perl*Handler, e.g:
PerlResponseHandler Bird::Eagle->handler;

the :method attribute is not required.

In the preceding configuration example, the handler() method will be called as a class (static) method.

Also, you can use objects created at startup to call methods. For example:

```perl
<Perl>
  use Bird::Eagle;
  $Bird::Global::object = Bird::Eagle->new();
</Perl>
```

PerlResponseHandler $Bird::Global::object->handler

In this example, the handler() method will be called as an instance method on the global object $Bird::Global::object.

1.4.2 Cleaning up

It’s possible to arrange for cleanups to happen at the end of various phases. One can’t rely on END blocks to do the job, since these don’t get executed until the interpreter quits, with an exception to the Registry handlers.

Module authors needing to run cleanups after each HTTP request, should use PerlCleanupHandler.

Module authors needing to run cleanups at other times can always register a cleanup callback via cleanup_register on the pool object of choice. Here are some examples of its usage:

To run something at the server shutdown and restart use a cleanup handler registered on server_shutdown_cleanup_register() in startup.pl:

```perl
  use Apache2::ServerUtil ();
  use APR::Pool ();

  warn "parent pid is $$\n";
  Apache2::ServerUtil::server_shutdown_cleanup_register((\&cleanup);
  sub cleanup { warn "server cleanup in $$\n" }
```

This is usually useful when some server-wide cleanup should be performed when the server is stopped or restarted.

To run a cleanup at the end of each connection phase, assign a cleanup callback to the connection pool object:

```perl
  use Apache2::Connection ();
  use APR::Pool ();

  my $pool = $c->pool;
  $pool->cleanup_register((\&my_cleanup);
  sub my_cleanup { ... }
```
You can also create your own pool object, register a cleanup callback and it’ll be called when the object is destroyed:

```perl
use APR::Pool ();
{
    my @args = 1..3;
    my $pool = APR::Pool->new;
    $pool->cleanup_register(\&cleanup, \@args);
}

sub cleanup {
    my @args = @{ +shift };
    warn "cleanup was called with args: @args";
}
```

In this example the cleanup callback gets called, when $pool goes out of scope and gets destroyed. This is very similar to OO DESTROY method.

### 1.5 Goodies Toolkit

#### 1.5.1 Environment Variables

mod_perl sets the following environment variables:

- `$ENV{MOD_PERL}` - is set to the mod_perl version the server is running under. e.g.:

  ```perl
  mod_perl/2.000002
  ```

  If `$ENV{MOD_PERL}` doesn’t exist, most likely you are not running under mod_perl.

  ```perl
  die "I refuse to work without mod_perl!" unless exists $ENV{MOD_PERL};
  ```

  However to check which version is used it’s better to use the following technique:

  ```perl
  use mod_perl;
  use constant MP2 => ( exists $ENV{MOD_PERL_API_VERSION} and
                      $ENV{MOD_PERL_API_VERSION} >= 2 );
  
  # die "I want mod_perl 2.0!" unless MP2;
  ```

mod_perl passes (exports) the following shell environment variables (if they are set):

- `PATH` - Executables search path.
- `TZ` - Time Zone.

Any of these environment variables can be accessed via `%ENV`.  

4  

15 Feb 2014
1.5.2 Threaded MPM or not?

If the code needs to behave differently depending on whether it’s running under one of the threaded MPMs, or not, the class method Apache2::MPM->is_threaded can be used. For example:

```perl
use Apache2::MPM ();
if (Apache2::MPM->is_threaded) {
    require APR::OS;
    my $tid = APR::OS::current_thread_id();
    print "current thread id: $tid (pid: $$)"
} else {
    print "current process id: $$";
}
```

This code prints the current thread id if running under a threaded MPM, otherwise it prints the process id.

1.5.3 Writing MPM-specific Code

If you write a CPAN module it’s a bad idea to write code that won’t run under all MPMs, and developers should strive to write a code that works with all mpms. However it’s perfectly fine to perform different things under different mpms.

If you don’t develop CPAN modules, it’s perfectly fine to develop your project to be run under a specific MPM.

```perl
use Apache2::MPM ();
my $mpm = lc Apache2::MPM->show;
if ($mpm eq 'prefork') {
    # prefork-specific code
} elsif ($mpm eq 'worker') {
    # worker-specific code
} elsif ($mpm eq 'winnt') {
    # winnt-specific code
} else {
    # others...
}
```

1.6 Code Developing Nuances

1.6.1 Auto-Reloading Modified Modules with Apache2::Reload

META: need to port Apache2::Reload notes from the guide here. but the gist is:
PerlModule Apache2::Reload
PerlInitHandler Apache2::Reload
#PerlPreConnectionHandler Apache2::Reload
PerlSetVar ReloadAll Off
PerlSetVar ReloadModules "ModPerl::* Apache2::*"

Use:

PerlInitHandler Apache2::Reload

if you need to debug HTTP protocol handlers. Use:

PerlPreConnectionHandler Apache2::Reload

for any handlers.

Though notice that we have started to practice the following style in our modules:

package Apache2::Whatever;

use strict;
use warnings FATAL => 'all';

FATAL => ‘all’ escalates all warnings into fatal errors. So when Apache2::Whatever is modified and reloaded by Apache2::Reload the request is aborted. Therefore if you follow this very healthy style and want to use Apache2::Reload, flex the strictness by changing it to:

use warnings FATAL => 'all';
no warnings 'redefine';

but you probably still want to get the redefine warnings, but downgrade them to be non-fatal. The following will do the trick:

use warnings FATAL => 'all';
no warnings 'redefine';
use warnings 'redefine';

Perl 5.8.0 allows to do all this in one line:

use warnings FATAL => 'all', NONFATAL => 'redefine';

but if your code may be used with older perl versions, you probably don’t want to use this new functionality.

Refer to the perllexwarn manpage for more information.

1.7 Integration with Apache Issues

In the following sections we discuss the specifics of Apache behavior relevant to mod_perl developers.
### 1.7.1 HTTP Response Headers

#### 1.7.1.1 Generating HTTP Response Headers

The best approach for generating HTTP response headers is by using the mod_perl API. Some common
headers have dedicated methods, others are set by manipulating the headers_out table directly.

For example to set the *Content-type* header you should call `$r->content_type`:

```perl
use Apache2::RequestRec ();
$r->content_type('text/html');
```

To set a custom header *My-Header* you should call:

```perl
use Apache2::RequestRec ();
use APR::Table;
$r->headers_out->set(My-Header => "SomeValue");
```

If you are inside a registry script you can still access the Apache2::RequestRec object.

However you can choose a slower method of generating headers by just printing them out before printing any response. This will work only if PerlOptions +ParseHeaders is in effect. For example:

```perl
print "Content-type: text/html\n";
print "My-Header: SomeValue\n";
print "\n";
```

This method is slower since Apache needs to parse the text to identify certain headers it needs to know about. It also has several limitations which we will now discuss.

When using this approach you must make sure that the STDOUT filehandle is not set to flush the data after each print (which is set by the value of a special perl variable $|). Here we assume that STDOUT is the currently select() ed filehandle and $| affects it.

For example this code won’t work:

```perl
local $| = 1;
print "Content-type: text/html\n";
print "My-Header: SomeValue\n";
print "\n";
```

Having a true $| causes the first print() call to flush its data immediately, which is sent to the internal HTTP header parser, which will fail since it won’t see the terminating "\n\n". One solution is to make sure that STDOUT won’t flush immediately, like so:

```perl
local $| = 0;
print "Content-type: text/html\n";
print "My-Header: SomeValue\n";
print "\n";
```
Notice that we localize that change, so it won’t affect any other code.

If you send headers line by line and their total length is bigger than 8k, you will have the header parser problem again, since mod_perl will flush data when the 8k buffer gets full. In which case the solution is not to print the headers one by one, but to buffer them all in a variable and then print the whole set at once.

Notice that you don’t have any of these problems with mod_cgi, because it ignores any of the flush attempts by Perl. mod_cgi simply opens a pipe to the external process and reads any output sent from that process at once.

If you use $r to set headers as explained at the beginning of this section, you won’t encounter any of these problems.

Finally, if you don’t want Apache to send its own headers and you want to send your own set of headers (non-parsed headers handlers) use the $r->assbackwards method. Notice that registry handlers will do that for you if the script’s name start with the nph- prefix.

### 1.7.1.2 Forcing HTTP Response Headers Out

Apache 2.0 doesn’t provide a method to force HTTP response headers sending (what used to be done by send_http_header() in Apache 1.3). HTTP response headers are sent as soon as the first bits of the response body are seen by the special core output filter that generates these headers. When the response handler sends the first chunks of body it may be cached by the mod_perl internal buffer or even by some of the output filters. The response handler needs to flush the output in order to tell all the components participating in the sending of the response to pass the data out.

For example if the handler needs to perform a relatively long-running operation (e.g. a slow db lookup) and the client may timeout if it receives nothing right away, you may want to start the handler by setting the `Content-Type` header, following by an immediate flush:

```perl
sub handler {
    my $r = shift;
    $r->content_type('text/html');
    $r->rflush; # send the headers out
    $r->print(long_operation());
    return Apache2::Const::OK;
}
```

If this doesn’t work, check whether you have configured any third-party output filters for the resource in question. Improperly written filter may ignore the command to flush the data.

### 1.7.2 Sending HTTP Response Body

In mod_perl 2.0 a response body can be sent only during the response phase. Any attempts to do that in the earlier phases will fail with an appropriate explanation logged into the `error_log` file.
This happens due to the Apache 2.0 HTTP architecture specifics. One of the issues is that the HTTP response filters are not setup before the response phase.

### 1.7.3 Using Signal Handlers

3rd party Apache 2 modules should avoid using code relying on signals. This is because typical signal use is not thread-safe and modules which rely on signals may not work portably. Certain signals may still work for non-threaded mpm’s. For example `alarm()` can be used under prefork MPM, but it won’t work on any other MPM. Moreover the Apache developers don’t guarantee that the signals that currently happen to work will continue to do so in the future Apache releases. So use them at your own risk.

It should be possible to rework the code using signals to use an alternative solution, which works under threads. For example if you were using `alarm()` to trap potentially long running I/O, you can modify the I/O logic for select/poll usage (or if you use APR I/O then set timeouts on the apr pipes or sockets). For example, Apache 1.3 on Unix made blocking I/O calls and relied on the parent process to send the SIGALRM signal to break it out of the I/O after a timeout expired. With Apache 2.0, APR support for timeouts on I/O operations is used so that signals or other thread-unsafe mechanisms are not necessary.

CPU timeout handling is another example. It can be accomplished by modifying the computation logic to explicitly check for the timeout at intervals.

Talking about `alarm()` under prefork mpm, POSIX signals seem to work, but require Perl 5.8.x+. For example:

```perl
use POSIX qw(SIGALRM);
my $mask = POSIX::SigSet->new( SIGALRM );
my $action = POSIX::SigAction->new(sub { die "alarm" }, $mask);
my $oldaction = POSIX::SigAction->new();
POSIX::sigaction(SIGALRM, $action, $oldaction );
eval {
    alarm 2;
    sleep 10 # some real code should be here
    alarm 0;
};
POSIX::sigaction(SIGALRM, $oldaction); # restore original
warn "got alarm" if $@ and $@ =~ /alarm/;
```

For more details see: [http://search.cpan.org/dist/perl/ext/POSIX/POSIX.pod#POSIX::SigAction](http://search.cpan.org/dist/perl/ext/POSIX/POSIX.pod#POSIX::SigAction)

One could use the `$SIG{ALRM}` technique, working for 5.6.x+, but it works only under DSO modperl build. Moreover starting from 5.8.0 Perl delays signal delivery, making signals safe. This change may break previously working code. For more information please see: [http://search.cpan.org/dist/perl/od/Perl58delta.pod#Safe_Signals](http://search.cpan.org/dist/perl/od/Perl58delta.pod#Safe_Signals) and [http://search.cpan.org/dist/perl/perlipc.pod#Deferred_Signals._28Safe_Signals_29](http://search.cpan.org/dist/perl/perlipc.pod#Deferred_Signals._28Safe_Signals_29)

For example if you had the alarm code:
eval {
    local $SIG{ALRM} = sub { die "alarm" ;
    alarm 3;
    sleep 10; # in reality some real code should be here
    alarm 0;
    }
    die "the operation was aborted" if $0 and $0 =~ /alarm/;
}

It may not work anymore. Starting from 5.8.1 it’s possible to circumvent the safeness of signals, by setting:

$ENV{PERL_SIGNALS} = "unsafe";

as soon as you start your program (e.g. in the case of mod_perl in startup.pl). As of this writing, this workaround fails on MacOSX, POSIX signals must be used instead.

For more information please refer to:
http://search.cpan.org/dist/perl/pod/perl581delta.pod#Unsafe_signals_again_available and
http://search.cpan.org/dist/perl/pod/perlrun.pod#PERL_SIGNALS.

Though if you use perl 5.8.x+ it’s preferrable to use the POSIX API technique explained earlier in this section.

**1.8 Perl Specifics in the mod_perl Environment**

In the following sections we discuss the specifics of Perl behavior under mod_perl.

**1.8.1 BEGIN Blocks**

Perl executes BEGIN blocks as soon as possible, at the time of compiling the code. The same is true under mod_perl. However, since mod_perl normally only compiles scripts and modules once, either in the parent server (at the server startup) or once per-child (on the first request using a module), BEGIN blocks in that code will only be run once. As the perlmod manpage explains, once a BEGIN block has run, it is immediately undefined. In the mod_perl environment, this means that BEGIN blocks will not be run during the response to an incoming request unless that request happens to be the one that causes the compilation of the code, i.e. if it wasn’t loaded yet.

BEGIN blocks in modules and files pulled in via require() or use() will be executed:

- Only once, if pulled in by the parent process at the server startup.
- Once per each child process or Perl interpreter if not pulled in by the parent process.
- An additional time, once per each child process or Perl interpreter if the module is reloaded off disk again via Apache2::Reload.
- Unpredictable if you fiddle with %INC yourself.
The BEGIN blocks behavior is different in ModPerl::Registry and ModPerl::PerlRun handlers, and their subclasses.

### 1.8.2 CHECK and INIT Blocks

CHECK and INIT blocks run when the source code compilation is complete, but before the program starts. CHECK can mean "checkpoint" or "double-check" or even just "stop". INIT stands for "initialization". The difference is subtle; CHECK blocks are run just after the compilation ends, INIT just before the runtime begins. (Hence the -c command-line perl option runs CHECK blocks but not INIT blocks.)

Perl only calls these blocks during `perl_parse()`, which mod_perl calls once at startup time. Under threaded mpm, these blocks will be called once per parent perl interpreter startup. Therefore CHECK and INIT blocks don’t work after the server is started, for the same reason these code samples don’t work:

```perl
% perl -e 'eval qq(CHECK { print "ok\n" })'
% perl -e 'eval qq(INIT  { print "ok\n" })'
```

### 1.8.3 END Blocks

As the perlmod manpage explains, an END block is executed as late as possible, that is, when the interpreter exits. So for example mod_cgi will run its END blocks on each invocation, since on every invocation it starts a new interpreter and then kills it when the request processing is done.

In the mod_perl environment, the interpreter does not exit after serving a single request (unless it is configured to do so) and hence it will run its END blocks only when it exits, which usually happens during the server shutdown, but may also happen earlier than that (e.g. a process exits because it has served a MaxRequestsPerChild number of requests).

mod_perl does make a special case for scripts running under ModPerl::Registry and friends.

The Cleaning up section explains how to deal with cleanups for non-Registry handlers.

ModPerl::Global API: `special_list_register`, `special_list_call` and `special_list_clear`, internally used by registry handlers, can be used to run END blocks at arbitrary times.

### 1.8.4 Request-localized Globals

mod_perl 2.0 provides two types of SetHandler handlers: modperl and perl-script. Remember that the SetHandler directive is only relevant for the response phase handlers, it neither needed nor affects non-response phases.

Under the handler:
SetHandler perl-script

several special global Perl variables are saved before the handler is called and restored afterwards. This includes: %ENV, @INC, $/, STDOUT’s $ | and END blocks array (PL_endav).

Under:

SetHandler modperl

nothing is restored, so you should be especially careful to remember localize all special Perl variables so the local changes won’t affect other handlers.

1.8.5 exit

In the normal Perl code exit() is used to stop the program flow and exit the Perl interpreter. However under mod_perl we only want the stop the program flow without killing the Perl interpreter.

You should take no action if your code includes exit() calls and it’s OK to continue using them. mod_perl worries to override the exit() function with its own version which stops the program flow, and performs all the necessary cleanups, but doesn’t kill the server. This is done by overriding:

*CORE::GLOBAL::exit = \&ModPerl::Util::exit;

so if you mess up with *CORE::GLOBAL::exit yourself you better know what you are doing.

You can still call CORE::exit to kill the interpreter, again if you know what you are doing.

One caveat is when exit is called inside eval -- the ModPerl::Util::exit documentation explains how to deal with this situation.

1.9 ModPerl::Registry Handlers Family

1.9.1 A Look Behind the Scenes

If you have a CGI script test.pl:

#!/usr/bin/perl
print "Content-type: text/plain\n\n";
print "Hello";

a typical registry family handler turns it into something like:

package foo_bar_baz;
sub handler {
    local $0 = "/full/path/to/test.pl";
#line 1 test.pl
#!/usr/bin/perl
print "Content-type: text/plain\n\n";
print "Hello";
}
Turning it into an almost full-fledged mod_perl handler. The only difference is that it handles the return status for you. (META: more details on return status needed.)

It then executes it as:

```perl
foo_bar_baz::handler($r);
```

passing the \$r object as the only argument to the handler() function.

Depending on the used registry handler the package is made of the file path, the uri or anything else. Check the handler’s documentation to learn which method is used.

### 1.9.2 Getting the \$r Object

As explained in [A Look Behind the Scenes](#) the \$r object is always passed to the registry script’s special function handler as the first and the only argument, so you can get this object by accessing @_, since:

```perl
my $r = shift;
prient "Content-type: text/plain\n\n";
prient "Hello";
```

is turned into:

```perl
sub handler {
    my $r = shift;
    print "Content-type: text/plain\n\n";
    print "Hello";
}
```

behind the scenes. Now you can use \$r to call various mod_perl methods, e.g. rewriting the script as:

```perl
my $r = shift;
$r->content_type('text/plain');
$r->print();
```

If you are deep inside some code and can’t get to the entry point to reach for \$r, you can use Apache2->request.

### 1.10 Threads Coding Issues Under mod_perl

The following sections discuss threading issues when running mod_perl under a threaded MPM.

#### 1.10.1 Thread-environment Issues

The "only" thing you have to worry about your code is that it’s thread-safe and that you don’t use functions that affect all threads in the same process.
Perl 5.8.0 itself is thread-safe. That means that operations like `push()`, `map()`, `chomp()`, `=/`, `+=`, etc. are thread-safe. Operations that involve system calls, may or may not be thread-safe. It all depends on whether the underlying C libraries used by the perl functions are thread-safe.

For example the function `localtime()` is not thread-safe when the implementation of `asctime(3)` is not thread-safe. Other usually problematic functions include `readdir()`, `srand()`, etc.

Another important issue that shouldn’t be missed is what some people refer to as thread-locality. Certain functions executed in a single thread affect the whole process and therefore all other threads running inside that process. For example if you `chdir()` in one thread, all other thread now see the current working directory of that thread that `chdir()`’ed to that directory. Other functions with similar effects include `umask()`, `chroot()`, etc. Currently there is no cure for this problem. You have to find these functions in your code and replace them with alternative solutions which don’t incur this problem.

For more information refer to the `perltut` [http://perldoc.perl.org/perltut.html] manpage.

### 1.10.2 Deploying Threads

This is actually quite unrelated to mod_perl 2.0. You don’t have to know much about Perl threads, other than [Thread-environment Issues](#) to have your code properly work under threaded MPM mod_perl.

If you want to spawn your own threads, first of all study how the new ithreads Perl model works, by reading the `perltut`, [threads](http://search.cpan.org/search?query=threads) and [threads::shared](http://search.cpan.org/search?query=threads%3A%3Ashared) manpages.

Artur Bergman wrote an article which explains how to port pure Perl modules to work properly with Perl ithreads. Issues with `chdir()` and other functions that rely on shared process’ datastructures are discussed.[http://www.perl.com/lpt/a/2002/06/11/threads.html](http://www.perl.com/lpt/a/2002/06/11/threads.html)

### 1.10.3 Shared Variables

Global variables are only global to the interpreter in which they are created. Other interpreters from other threads can’t access that variable. Though it’s possible to make existing variables shared between several threads running in the same process by using the function `threads::shared::share()`. New variables can be shared by using the `shared` attribute when creating them. This feature is documented in the `threads::shared` [http://search.cpan.org/search?query=threads%3A%3Ashared](http://search.cpan.org/search?query=threads%3A%3Ashared) manpage.

### 1.11 Maintainers

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

Only the major authors are listed above. For contributors see the Changes file.
# Table of Contents:

1. **Writing mod_perl Handlers and Scripts** ........................................................................ 1
   1.1 Description........................................................................................................ 2
   1.2 Prerequisites..................................................................................................... 2
   1.3 Where the Methods Live .................................................................................. 2
   1.4 Techniques........................................................................................................ 2
      1.4.1 Method Handlers ..................................................................................... 2
      1.4.2 Cleaning up .............................................................................................. 3
   1.5 Goodies Toolkit ................................................................................................ 4
      1.5.1 Environment Variables ............................................................................ 4
      1.5.2 Threaded MPM or not? .............................................................................. 5
      1.5.3 Writing MPM-specific Code ...................................................................... 5
   1.6 Code Developing Nuances ................................................................................ 5
      1.6.1 Auto-Reloading Modified Modules with Apache2::Reload ....................... 5
   1.7 Integration with Apache Issues ......................................................................... 6
      1.7.1 HTTP Response Headers ............................................................................ 7
         1.7.1.1 Generating HTTP Response Headers .................................................. 7
         1.7.1.2 Forcing HTTP Response Headers Out ............................................... 8
      1.7.2 Sending HTTP Response Body .................................................................. 8
      1.7.3 Using Signal Handlers .............................................................................. 9
   1.8 Perl Specifics in the mod_perl Environment ....................................................... 10
      1.8.1 BEGIN Blocks ......................................................................................... 10
      1.8.2 CHECK and INIT Blocks ......................................................................... 11
      1.8.3 END Blocks ............................................................................................. 11
      1.8.4 Request-localized Globals ....................................................................... 11
      1.8.5 exit ........................................................................................................... 12
   1.9 ModPerl::Registry Handlers Family .................................................................... 12
      1.9.1 A Look Behind the Scenes ....................................................................... 12
      1.9.2 Getting the $r Object ............................................................................... 13
   1.10 Threads Coding Issues Under mod_perl ......................................................... 13
      1.10.1 Thread-environment Issues .................................................................... 13
      1.10.2 Deploying Threads .................................................................................. 14
      1.10.3 Shared Variables ...................................................................................... 14
   1.11 Maintainers ...................................................................................................... 14
   1.12 Authors .......................................................................................................... 15