User's guide

All you need to know about using mod_perl 2.0

Last modified Sun Feb 16 01:34:28 2014 GMT

Part VI: Performance

Part VII: Troubleshooting

1   Getting Your Feet Wet with mod_perl

1.1 Description

This chapter gives you the bare minimum information to get you started with mod_perl 2.0. For most people it's sufficient to get going.

1.2 Installation

If you are a Win32 user, please refer to the Win32 installation document.

First, download the mod_perl 2.0 source.

Before installing mod_perl, you need to check that you have the mod_perl 2.0 prerequisites **installed**. Apache and the right Perl version have to be built and installed **before** you can proceed with building mod_perl.

In this chapter we assume that httpd and all helper files were installed under *$HOME/httpd/prefork*, if your distribution doesn't install all the files under the same tree, please refer to the complete installation instructions.

Now, configure mod_perl:

```
% tar -xvzf mod_perl-2.x.xx.tar.gz
% cd modperl-2.0
% perl Makefile.PL MP_APXS=$HOME/httpd/prefork/bin/apxs
```

where `MP_APXS` is the full path to the `apxs` executable, normally found in the same directory as the `httpd` executable, but could be put in a different path as well.

Finally, build, test and install mod_perl:

```
% make && make test && make install
```

Become *root* before doing `make install` if installing system-wide.

If something goes wrong or you need to enable optional features please refer to the complete installation instructions.

1.3 Configuration

If you are a Win32 user, please refer to the Win32 configuration document.

Enable mod_perl built as DSO, by adding to *httpd.conf*:

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

There are many other configuration options which you can find in the configuration manual.

If you want to run mod_perl 1.0 code on mod_perl 2.0 server enable the compatibility layer:

```
PerlModule Apache2::compat
```

For more information see: Migrating from mod_perl 1.0 to mod_perl 2.0.

1.4 Server Launch and Shutdown

Apache is normally launched with `apachectl`:

```
% $HOME/httpd/prefork/bin/apachectl start
```

and shut down with:

```
% $HOME/httpd/prefork/bin/apachectl stop
```

Check *$HOME/httpd/prefork/logs/error_log* to see that the server has started and it's a right one. It should say something similar to:

```
[Fri Jul 22 09:39:55 2005] [notice] Apache/2.0.55-dev (Unix)
mod_ssl/2.0.55-dev OpenSSL/0.9.7e DAV/2 mod_perl/2.0.2-dev
Perl/v5.8.7 configured -- resuming normal operations
```

1.5 Registry Scripts

To enable registry scripts add the following to *httpd.conf*:

```
Alias /perl/ /home/httpd/httpd-2.0/perl/
<Location /perl/>
    SetHandler perl-script
    PerlResponseHandler ModPerl::Registry
    PerlOptions +ParseHeaders
    Options +ExecCGI
    Order allow,deny
    Allow from all
</Location>
```

and now assuming that we have the following script:

```
#!/usr/bin/perl
print "Content-type: text/plain\n\n";
print "mod_perl 2.0 rocks!\n";
```

saved in */home/httpd/httpd-2.0/perl/rock.pl*. Make the script executable and readable by everybody:

```
% chmod a+rx /home/httpd/httpd-2.0/perl/rock.pl
```

Of course the path to the script should be readable by the server too. In the real world you probably want to have a tighter permissions, but for the purpose of testing that things are working this is just fine.

Now restart the server and issue a request to *http://localhost/perl/rock.pl* and you should get the response:

```
mod_perl 2.0 rocks!
```

If that didn't work check the *error_log* file.

For more information on the registry scripts refer to the `ModPerl::Registry` manpage. (XXX: one day there will a tutorial on registry, should port it from 1.0's docs).

1.6 Handler Modules

Finally check that you can run mod_perl handlers. Let's write a response handler similar to the registry script from the previous section:

```
#file:MyApache2/Rocks.pm
#----------------------
package MyApache2::Rocks;

use strict;
use warnings;

use Apache2::RequestRec ();
use Apache2::RequestIO ();

use Apache2::Const -compile => qw(OK);

sub handler {
    my $r = shift;

    $r->content_type('text/plain');
    print "mod_perl 2.0 rocks!\n";

    return Apache2::Const::OK;
}
1;
```

Save the code in the file *MyApache2/Rocks.pm*, somewhere where mod_perl can find it. For example let's put it under */home/httpd/httpd-2.0/perl/MyApache2/Rocks.pm*, and we tell mod_perl that */home/httpd/httpd-2.0/perl/* is in `@INC`, via a startup file which includes just:

```
use lib qw(/home/httpd/httpd-2.0/perl);
1;
```

and loaded from *httpd.conf*:

```
PerlRequire /home/httpd/httpd-2.0/perl/startup.pl
```

Now we can configure our module in *httpd.conf*:

```
<Location /rocks>
    SetHandler perl-script
    PerlResponseHandler  MyApache2::Rocks
</Location>
```

Now restart the server and issue a request to *http://localhost/rocks* and you should get the response:

```
mod_perl 2.0 rocks!
```

If that didn't work check the *error_log* file.

1.7 Troubleshooting

If after reading the complete installation and configuration chapters you are still having problems, take a look at the troubleshooting sections. If the problem persist, please report them using the following guidelines.

1.8 Maintainers

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

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

1.9 Authors

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

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

2   Overview of mod_perl 2.0

2.1 Description

This chapter should give you a general idea about what mod_perl 2.0 is and how it differs from mod_perl 1.0. This chapter presents the new features of Apache 2.0, Perl 5.6.0 -- 5.8.0 and their influence on mod_perl 2.0. The new MPM models from Apache 2.0 are also discussed.

2.2 Version Naming Conventions

In order to keep things simple, here and in the rest of the documentation we refer to mod_perl 1.x series as mod_perl 1.0 and to 2.0.x series as mod_perl 2.0. Similarly we call Apache 1.3.x series as Apache 1.3 and 2.0.x as Apache 2.0. There is also Apache 2.1, which is a development track towards Apache 2.2.

2.3 Why mod_perl, The Next Generation

mod_perl was introduced in early 1996, both Perl and Apache have changed a great deal since that time. mod_perl has adjusted to both along the way over the past 4 and a half years or so using the same code base. Over this course of time, the mod_perl sources have become more and more difficult to maintain, in large part to provide compatibility between the many different flavors of Apache and Perl. And, compatibility across these versions and flavors is a more difficult goal for mod_perl to reach that a typical Apache or Perl module, since mod_perl reaches a bit deeper into the corners of Apache and Perl internals than most. Discussions of the idea to rewrite mod_perl as version 2.0 started in 1998, but never made it much further than an idea. When Apache 2.0 development was underway it became clear that a rewrite of mod_perl would be required to adjust to the new Apache architecture and API.

Of the many changes happening in Apache 2.0, the one which has the most significant impact on mod_perl is the introduction of threads to the overall design. Threads have been a part of Apache on the win32 side since the Apache port was introduced. The mod_perl port to win32 happened in version 1.00b1, released in June of 1997. This port enabled mod_perl to compile and run in a threaded windows environment, with one major caveat: only one concurrent mod_perl request could be handled at any given time. This was due to the fact that Perl did not introduce thread-safe interpreters until version 5.6.0, released in March of 2000. Contrary to popular belief, the "threads support" implemented in Perl 5.005 (released July 1998), did not make Perl thread-safe internally. Well before that version, Perl had the notion of "Multiplicity", which allowed multiple interpreter instances in the same process. However, these instances were not thread safe, that is, concurrent callbacks into multiple interpreters were not supported.

It just so happens that the release of Perl 5.6.0 was nearly at the same time as the first alpha version of Apache 2.0. The development of mod_perl 2.0 was underway before those releases, but as both Perl 5.6.0 and Apache 2.0 were reaching stability, mod_perl 2.0 was becoming more of a reality. In addition to the adjustments for threads and Apache 2.0 API changes, this rewrite of mod_perl is an opportunity to clean up the source tree. This includes both removing the old backward compatibility bandaids and building a smarter, stronger and faster implementation based on lessons learned over the 4.5 years since mod_perl was introduced.

The new version includes a mechanism for the automatic building of the Perl interface to Apache API, which allowed us to easily adjust mod_perl 2.0 to the ever changing Apache 2.0 API, during its development period. Another important feature is the `Apache::Test` framework, which was originally developed for mod_perl 2.0, but then was adopted by Apache 2.0 developers to test the core server features and third party modules. Moreover the tests written using the `Apache::Test` framework could be run with Apache 1.0 and 2.0, assuming that both supported the same features.

There are multiple other interesting changes that have already happened to mod_perl in version 2.0 and more will be developed in the future. Some of these are discussed in this chapter, others can be found in the rest of the mod_perl 2.0 documentation.

2.4 What's new in Apache 2.0

Apache 2.0 has introduced numerous new features and enhancements. Here are the most important new features:

- *Apache Portable Runtime* (APR)

  Apache 1.3 has been ported to a very large number of platforms including various flavors of unix, win32, os/2, the list goes on. However, in 1.3 there was no clear-cut, pre-designed portability layer for third-party modules to take advantage of. APR provides this API layer in a very clean way. APR assists a great deal with mod_perl portability. Combined with the portablity of Perl, mod_perl 2.0 needs only to implement a portable build system, the rest comes "for free". A Perl interface is provided for certain areas of APR, such as the shared memory abstraction, but the majority of APR is used by mod_perl "under the covers".

  The APR uses the concept of memory pools, which significantly simplifies the memory management code and reduces the possibility of having memory leaks, which always haunt C programmers.

- **I/O Filtering**

  Filtering of Perl modules output has been possible for years since tied filehandle support was added to Perl. There are several modules, such as `Apache2::Filter` and `Apache::OutputChain` which have been written to provide mechanisms for filtering the `STDOUT` stream. There are several of these modules because no one's approach has quite been able to offer the ease of use one would expect, which is due simply to limitations of the Perl tied filehandle design. Another problem is that these filters can only filter the output of other Perl modules. C modules in Apache 1.3 send data directly to the client and there is no clean way to capture this stream. Apache 2.0 has solved this problem by introducing a filtering API. With the baseline I/O stream tied to this filter mechansim, any module can filter the output of any other module, with any number of filters in between. Using this new feature things like SSL, data (de-)compression and other data manipulations are done very easily.

- *Multi Processing Model modules* (MPMs).

  In Apache 1.3 concurrent requests were handled by multiple processes, and the logic to manage these processes lived in one place, *http_main.c*, 7700 some odd lines of code. If Apache 1.3 is compiled on

a Win32 system large parts of this source file are redefined to handle requests using threads. Now suppose you want to change the way Apache 1.3 processes requests, say, into a DCE RPC listener. This is possible only by slicing and dicing *http_main.c* into more pieces or by redefining the *stan-dalone_main* function, with a `-DSTANDALONE_MAIN=your_function` compile time flag. Neither of which is a clean, modular mechanism.

Apache-2.0 solves this problem by introducing *Multi Processing Model modules*, better known as *MPMs*. The task of managing incoming requests is left to the MPMs, shrinking *http_main.c* to less than 500 lines of code. Now it's possible to write different processing modules specific to various platforms. For example the Apache 2.0 on Windows is much more efficient now, since it uses *mpm_winnt* which deploys the native Windows features.

Here is a partial list of major MPMs available as of this writing.

○ **prefork**

The *prefork* MPM emulates Apache 1.3's preforking model, where each request is handled by a different forked child process.

○ **worker**

The *worker* MPM implements a hybrid multi-process multi-threaded approach based on the *pthreads* standard. It uses one acceptor thread, multiple worker threads.

○ **mpmt_os2, netware, winnt and beos**

These MPMs also implement the hybrid multi-process/multi-threaded model, with each based on native OS thread implementations.

○ **perchild**

The *perchild* MPM is similar to the *worker* MPM, but is extended with a mechanism which allows mapping of requests to virtual hosts to a process running under the user id and group configured for that host. This provides a robust replacement for the *suexec* mechanism.

META: as of this writing this mpm is not working

On platforms that support more than one MPM, it's possible to switch the used MPMs as the need change. For example on Unix it's possible to start with a preforked module. Then when the demand is growing and the code matures, it's possible to migrate to a more efficient threaded MPM, assuming that the code base is capable of running in the threaded environment.

● **Protocol Modules**

Apache 1.3 is hardwired to speak only one protocol, HTTP. Apache 2.0 has moved to more of a "server framework" architecture making it possible to plugin handlers for protocols other than HTTP. The protocol module design also abstracts the transport layer so protocols such as SSL can be hooked into the server without requiring modifications to the Apache source code. This allows Apache to be extended much further than in the past, making it possible to add support for protocols such as FTP,

SMTP, RPC flavors and the like. The main advantage being that protocol plugins can take advantage of Apache's portability, process/thread management, configuration mechanism and plugin API.

- **Parsed Configuration Tree**

  When configuration files are read by Apache 1.3, it hands off the parsed text to module configuration directive handlers and discards that text afterwards. With Apache 2.0, the configuration files are first parsed into a tree structure, which is then walked to pass data down to the modules. This tree is then left in memory with an API for accessing it at request time. The tree can be quite useful for other modules. For example, in 1.3, mod_info has its own configuration parser and parses the configuration files each time you access it. With 2.0 there is already a parse tree in memory, which mod_info can then walk to output its information.

  If a mod_perl 1.0 module wants access to configuration information, there are two approaches. A module can "subclass" directive handlers, saving a copy of the data for itself, then returning **DECLINE_CMD** so the other modules are also handed the info. Or, the `$Apache2::PerlSections::Save` variable can be set to save <Perl> configuration in the `%Apache2::ReadConfig::` namespace. Both methods are rather kludgy, version 2.0 provides a Perl interface to the Apache configuration tree.

All these new features boost the Apache performance, scalability and flexibility. The APR helps the overall performance by doing lots of platform specific optimizations in the APR internals, and giving the developer the API which was already greatly optimized.

Apache 2.0 now includes special modules that can boost performance. For example the mod_mmap_static module loads webpages into the virtual memory and serves them directly avoiding the overhead of *open()* and *read()* system calls to pull them in from the filesystem.

The I/O layering is helping performance too, since now modules don't need to waste memory and CPU cycles to manually store the data in shared memory or *pnotes* in order to pass the data to another module, e.g., in order to provide response's gzip compression.

And of course a not least important impact of these features is the simplification and added flexibility for the core and third party Apache module developers.

2.5 What's new in Perl 5.6.0 - 5.8.0

As we have mentioned earlier Perl 5.6.0 is the minimum requirement for mod_perl 2.0. Though as we will see later certain new features work only with Perl 5.8.0 and higher.

These are the important changes in the recent Perl versions that had an impact on mod_perl. For a complete list of changes see the corresponding to the used version *perldelta* manpages (*http://perldoc.perl.org/perl56delta.html*, *http://perldoc.perl.org/perl561delta.html* and *http://perldoc.perl.org/perldelta.html*).

The 5.6 Perl generation has introduced the following features:

● The beginnings of support for running multiple interpreters concurrently in different threads. In conjunction with the perl_clone() API call, which can be used to selectively duplicate the state of any given interpreter, it is possible to compile a piece of code once in an interpreter, clone that interpreter one or more times, and run all the resulting interpreters in distinct threads. See the *perlembed* (*http://perldoc.perl.org/perlembed.html*) and *perl561delta* (*http://perldoc.perl.org/perl561delta.html*) manpages.

● The core support for declaring subroutine attributes, which is used by mod_perl 2.0's *method handlers*. See the *attributes* manpage.

● The *warnings* pragma, which allows to force the code to be super clean, via the setting:

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

which will abort any code that generates warnings. This pragma also allows a fine control over what warnings should be reported. See the *perllexwarn* (*http://perldoc.perl.org/perllexwarn.html*) manpage.

● Certain `CORE::` functions now can be overridden via `CORE::GLOBAL::` namespace. For example mod_perl now can override `CORE::exit()` via `CORE::GLOBAL::exit`. See the *perlsub* (*http://perldoc.perl.org/perlsub.html*) manpage.

● The `XSLoader` extension as a simpler alternative to `DynaLoader`. See the *XSLoader* manpage.

● The large file support. If you have filesystems that support "large files" (files larger than 2 gigabytes), you may now also be able to create and access them from Perl. See the *perl561delta* (*http://perldoc.perl.org/perl561delta.html*) manpage.

● Multiple performance enhancements were made. See the *perl561delta* (*http://perldoc.perl.org/perl561delta.html*) manpage.

● Numerous memory leaks were fixed. See the *perl561delta* (*http://perldoc.perl.org/perl561delta.html*) manpage.

● Improved security features: more potentially unsafe operations taint their results for improved security. See the *perlsec* (*http://perldoc.perl.org/perlsec.html*) and *perl561delta* (*http://perldoc.perl.org/perl561delta.html*) manpages.

● Available on new platforms: GNU/Hurd, Rhapsody/Darwin, EPOC.

Overall multiple bugs and problems very fixed in the Perl 5.6.1, so if you plan on running the 5.6 generation, you should run at least 5.6.1. It is possible that when this tutorial is printed 5.6.2 will be out.

The Perl 5.8.0 has introduced the following features:

- The introduced in 5.6.0 experimental PerlIO layer has been stabilized and become the default IO layer in 5.8.0. Now the IO stream can be filtered through multiple layers. See the *perlapio* (*http://perldoc.perl.org/perlapio.html*) and *perliol* (*http://perldoc.perl.org/perliol.html*) manpages.

  For example this allows mod_perl to inter-operate with the APR IO layer and even use the APR IO layer in Perl code. See the `APR::PerlIO` manpage.

  Another example of using the new feature is the extension of the `open()` functionality to create anonymous temporary files via:

  ```
  open my $fh, "+>", undef or die $!;
  ```

  That is a literal `undef()`, not an undefined value. See the `open()` entry in the *perlfunc* manpage (*http://perldoc.perl.org/functions/open.html*).

- More overridable via `CORE::GLOBAL::` keywords. See the *perlsub* (*http://perldoc.perl.org/perlsub.html*) manpage.

- The signal handling in Perl has been notoriously unsafe because signals have been able to arrive at inopportune moments leaving Perl in inconsistent state. Now Perl delays signal handling until it is safe.

- `File::Temp` was added to allow a creation of temporary files and directories in an easy, portable, and secure way. See the *File::Temp* manpage.

- A new command-line option, `-t` is available. It is the little brother of `-T`: instead of dying on taint violations, lexical warnings are given. This is only meant as a temporary debugging aid while securing the code of old legacy applications. **This is not a substitute for `-T`.** See the *perlrun* (*http://perldoc.perl.org/perlrun.html*) manpage.

  A new special variable `${^TAINT}` was introduced. It indicates whether taint mode is enabled. See the *perlvar* (*http://perldoc.perl.org/perlvar.html*) manpage.

- Threads implementation is much improved since 5.6.

- A much better support for Unicode.

- Numerous bugs and memory leaks fixed. For example now you can localize the tied `Apache::DBI` filehandles without leaking memory.

- Available on new platforms: AtheOS, Mac OS Classic, Mac OS X, MinGW, NCR MP-RAS, NonStop-UX, NetWare and UTS. The following platforms are again supported: BeOS, DYNIX/ptx, POSIX-BC, VM/ESA, z/OS (OS/390).

2.6 What's new in mod_perl 2.0

The new features introduced by Apache 2.0 and Perl 5.6 and 5.8 generations provide the base of the new mod_perl 2.0 features. In addition mod_perl 2.0 re-implements itself from scratch providing such new features as new build and testing framework. Let's look at the major changes since mod_perl 1.0.

2.6.1 Threads Support

In order to adapt to the Apache 2.0 threads architecture (for threaded MPMs), mod_perl 2.0 needs to use thread-safe Perl interpreters, also known as "ithreads" (Interpreter Threads). This mechanism can be enabled at compile time and ensures that each Perl interpreter uses its private `PerlInterpreter` structure for storing its symbol tables, stacks and other Perl runtime mechanisms. When this separation is engaged any number of threads in the same process can safely perform concurrent callbacks into Perl. This of course requires each thread to have its own `PerlInterpreter` object, or at least that each instance is only accessed by one thread at any given time.

The first mod_perl generation has only a single `PerlInterpreter`, which is constructed by the parent process, then inherited across the forks to child processes. mod_perl 2.0 has a configurable number of `PerlInterpreters` and two classes of interpreters, *parent* and *clone*. A *parent* is like that in mod_perl 1.0, where the main interpreter created at startup time compiles any pre-loaded Perl code. A *clone* is created from the parent using the Perl API *perl_clone()* (*http://perldoc.perl.org/perlapi.html#Cloning-an-interpreter*) function. At request time, *parent* interpreters are only used for making more *clones*, as the *clones* are the interpreters which actually handle requests. Care is taken by Perl to copy only mutable data, which means that no runtime locking is required and read-only data such as the syntax tree is shared from the *parent*, which should reduce the overall mod_perl memory footprint.

Rather than create a `PerlInterpreter` per-thread by default, mod_perl creates a pool of interpreters. The pool mechanism helps cut down memory usage a great deal. As already mentioned, the syntax tree is shared between all cloned interpreters. If your server is serving more than mod_perl requests, having a smaller number of PerlInterpreters than the number of threads will clearly cut down on memory usage. Finally and perhaps the biggest win is memory re-use: as calls are made into Perl subroutines, memory allocations are made for variables when they are used for the first time. Subsequent use of variables may allocate more memory, e.g. if a scalar variable needs to hold a longer string than it did before, or an array has new elements added. As an optimization, Perl hangs onto these allocations, even though their values "go out of scope". mod_perl 2.0 has a much better control over which PerlInterpreters are used for incoming requests. The interpreters are stored in two linked lists, one for available interpreters and another for busy ones. When needed to handle a request, one interpreter is taken from the head of the available list and put back into the head of the same list when done. This means if for example you have 10 interpreters configured to be cloned at startup time, but no more than 5 are ever used concurrently, those 5 continue to reuse Perl's allocations, while the other 5 remain much smaller, but ready to go if the need arises.

Various attributes of the pools are configurable using threads mode specific directives.

The interpreters pool mechanism has been abstracted into an API known as "tipool", *Thread Item Pool*. This pool can be used to manage any data structure, in which you wish to have a smaller number than the number of configured threads. For example a replacement for `Apache::DBI` based on the *tipool* will allow to reuse database connections between multiple threads of the same process.

2.6.2 Thread-environment Issues

While mod_perl itself is thread-safe, you may have issues with the thread-safety of your code. For more information refer to Threads Coding Issues Under mod_perl.

Another issue is that "global" variables are only global to the interpreter in which they are created. It's possible to share variables between several threads running in the same process. For more information see: Shared Variables.

2.6.3 Perl Interface to the APR and Apache APIs

As we have mentioned earlier, Apache 2.0 uses two APIs:

- the Apache Portable APR (APR) API, which implements a portable and efficient API to handle generically work with files, sockets, threads, processes, shared memory, etc.

- the Apache API, which handles issues specific to the web server.

In mod_perl 1.0, the Perl interface back into the Apache API and data structures was done piecemeal. As functions and structure members were found to be useful or new features were added to the Apache API, the XS code was written for them here and there.

mod_perl 2.0 generates the majority of XS code and provides thin wrappers where needed to make the API more Perlish. As part of this goal, nearly the entire APR and Apache API, along with their public data structures are covered from the get-go. Certain functions and structures which are considered "private" to Apache or otherwise un-useful to Perl aren't glued. Most of the API behaves just as it did in mod_perl 1.0, so users of the API will not notice the difference, other than the addition of many new methods. Where API has changed a special back compatibility module can be used.

In mod_perl 2.0 the APR API resides in the `APR::` namespace, and obviously the `Apache2::` namespace is mapped to the Apache API.

And in the case of `APR`, it is possible to use `APR` modules outside of Apache, for example:

```
% perl -MAPR -MAPR::UUID -le 'print APR::UUID->new->format'
b059a4b2-d11d-b211-bc23-d644b8ce0981
```

The mod_perl 2.0 generator is a custom suite of modules specifically tuned for gluing Apache and allows for complete control over *everything*, providing many possibilities none of *xsubpp*, *SWIG* or *Inline.pm* are designed to do. Advantages to generating the glue code include:

- Not tied tightly to xsubpp

- Easy adjustment to Apache 2.0 API/structure changes

- Easy adjustment to Perl changes (e.g., Perl 6)

- Ability to "discover" hookable third-party C modules.

- Cleanly take advantage of features in newer Perls

- Optimizations can happen across-the-board with one-shot

- Possible to AUTOLOAD XSUBs

- Documentation can be generated from code

- Code can be generated from documentation

2.7 Integration with 2.0 Filtering

The mod_perl 2.0 interface to the Apache filter API comes in two flavors. First, similar to the C API, where bucket brigades need to be manipulated. Second, streaming filtering, is much simpler than the C API, since it hides most of the details underneath. For a full discussion on filters and implementation examples refer to the Input and Output Filters chapter.

2.7.1 Other New Features

In addition to the already mentioned new features, the following are of a major importance:

- Apache 2.0 protocol modules are supported. Later we will see an example of a protocol module running on top of mod_perl 2.0.

- mod_perl 2.0 provides a very simply to use interface to the Apache filtering API. We will present a filter module example later on.

- A feature-full and flexible `Apache::Test` framework was developed especially for mod_perl testing. While used to test the core mod_perl features, it is used by third-party module writers to easily test their modules. Moreover `Apache::Test` was adopted by Apache and currently used to test both Apache 1.3, 2.0 and other ASF projects. Anything that runs top of Apache can be tested with `Apache::Test`, be the target written in Perl, C, PHP, etc.

- The support of the new MPMs model makes mod_perl 2.0 can scale better on wider range of platforms. For example if you've happened to try mod_perl 1.0 on Win32 you probably know that the requests had to be serialized, i.e. only a single request could be processed at a time, rendering the Win32 platform unusable with mod_perl as a heavy production service. Thanks to the new Apache MPM design, now mod_perl 2.0 can be used efficiently on Win32 platforms using its native *win32* MPM.

2.7.2 Optimizations

The rewrite of mod_perl gives us the chances to build a smarter, stronger and faster implementation based on lessons learned over the 4.5 years since mod_perl was introduced. There are optimizations which can be made in the mod_perl source code, some which can be made in the Perl space by optimizing its syntax tree and some a combination of both. In this section we'll take a brief look at some of the optimizations that are being considered.

The details of these optimizations from the most part are hidden from mod_perl users, the exception being that some will only be turned on with configuration directives. A few of which include:

- "Compiled" `Perl*Handlers`

- Inlined `Apache2::*.xs` calls

- Use of Apache pools for memory allocations

2.8 Maintainers

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

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

2.9 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.

3   Notes on the design and goals of mod_perl-2.0

3.1 Description

Notes on the design and goals of mod_perl-2.0.

We try to keep this doc in sync with the development, so some items discussed here were already implemented, while others are only planned. If you find some inconsistencies in this document please let the list know.

3.2 Introduction

In version 2.0 of mod_perl, the basic concept of 1.0 still applies:

```
Provide complete access to the Apache C API
via the Perl programming language.
```

Rather than "porting" mod_perl-1.0 to Apache 2.0, mod_perl-2.0 is being implemented as a complete re-write from scratch.

For a more detailed introduction and functionality overview, see Overview.

3.3 Interpreter Management

In order to support mod_perl in a multi-threaded environment, mod_perl-2.0 will take advantage of Perl's *ithreads* feature, new to Perl version 5.6.0. This feature encapsulates the Perl runtime inside a thread-safe *PerlInterpreter* structure. Each thread which needs to serve a mod_perl request will need its own *PerlInterpreter* instance.

Rather than create a one-to-one mapping of *PerlInterpreter* per-thread, a configurable pool of interpreters is managed by mod_perl. This approach will cut down on memory usage simply by maintaining a minimal number of intepreters. It will also allow re-use of allocations made within each interpreter by recycling those which have already been used. This was not possible in the 1.3.x model, where each child has its own interpreter and no control over which child Apache dispatches the request to.

The interpreter pool is only enabled if Perl is built with -Dusethreads otherwise, mod_perl will behave just as 1.0, using a single interpreter, which is only useful when Apache is configured with the prefork mpm.

When the server is started, a Perl interpreter is constructed, compiling any code specified in the configuration, just as 1.0 does. This interpreter is referred to as the "parent" interpreter. Then, for the number of *PerlInterpStart* configured, a (thread-safe) clone of the parent interpreter is made (via perl_clone()) and added to the pool of interpreters. This clone copies any writeable data (e.g. the symbol table) and shares the compiled syntax tree. From my measurements of a *startup.pl* including a few random modules:

```
use CGI ();
use POSIX ();
use IO ();
use SelfLoader ();
use AutoLoader ();
use B::Deparse ();
use B::Terse ();
use B ();
use B::C ();
```

The parent adds 6M size to the process, each clone adds less than half that size, ~2.3M, thanks to the shared syntax tree.

NOTE: These measurements were made prior to finding memory leaks related to perl_clone() in 5.6.0 and the GvSHARED optimization.

At request time, If any Perl*Handlers are configured, an available interpreter is selected from the pool. As there is a *conn_rec* and *request_rec* per thread, a pointer is saved in either the conn_rec->pool or request_rec->pool, which will be used for the lifetime of that request. For handlers that are called when threads are not running (PerlChild{Init,Exit}Handler), the parent interpreter is used. Several configuration directives control the interpreter pool management:

- **PerlInterpStart**

  The number of intepreters to clone at startup time.

- **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())

- **PerlInterpMinSpare**

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

- **PerlInterpMaxSpare**

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

- **PerlInterpMaxRequests**

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

- **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.

Intepreters will be shared across subrequests by default, however, it is possible to configure the intepreter scope to be per-subrequest 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
```

With this configuration, an interpreter will be selected before `PerlAccessHandlers` are run, and putback immediately afterwards, before Apache moves onto the authentication phase. If a `Perl-FixupHandler` is configured further down the chain, another interpreter will be selected and again putback 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
```

3.3.1 TIPool

The interpreter pool is implemented in terms of a "TIPool" (Thread Item Pool), a generic api which can be reused for other data such as database connections. A Perl interface will be provided for the *TIPool* mechanism, which, for example, will make it possible to share a pool of DBI connections.

3.3.2 Virtual Hosts

The interpreter management has been implemented in a way such that each `<VirtualHost>` can have its own parent Perl interpreter and/or MIP (Mod_perl Interpreter Pool). It is also possible to disable mod_perl for a given virtual host.

3.3.3 Further Enhancements

- The interpreter pool management could be moved into its own thread.

- A "garbage collector", which could also run in its own thread, examining the padlists of idle inter-preters and deciding to release and/or report large strings, array/hash sizes, etc., that Perl is keeping around as an optimization.

3.4 Hook Code and Callbacks

The code for hooking mod_perl in the various phases, including `Perl*Handler` directives is generated by the `ModPerl::Code` module. Access to all hooks will be provided by mod_perl in both the tradi-tional `Perl*Handler` configuration fashion and via dynamic registration methods (the ap_hook_* func-tions).

When a mod_perl hook is called for a given phase, the glue code has an index into the array of handlers, so it knows to return DECLINED right away if no handlers are configured, without entering the Perl runtime as 1.0 did. The handlers are also now stored in an apr_array_header_t, which is much lighter and faster than using a Perl AV, as 1.0 did. And more importantly, keeps us out of the Perl runtime until we're sure we need to be there.

`Perl*Handler`s are now "compiled", that is, the various forms of:

```
PerlResponseHandler MyModule->handler
# defaults to MyModule::handler or MyModule->handler
PerlResponseHandler MyModule
PerlResponseHandler $MyObject->handler
PerlResponseHandler 'sub { print "foo\n"; return OK }'
```

are only parsed once, unlike 1.0 which parsed every time the handler was used. There will also be an option to parse the handlers at startup time. Note: this feature is currently not enabled with threads, as each clone needs its own copy of Perl structures.

A "method handler" is now specified using the 'method' sub attribute, e.g.

```
sub handler : method {};
```

instead of 1.0's

```
sub handler ($$) {}
```

3.5 Perl interface to the Apache API and Data Structures

In 1.0, the Perl interface back into the Apache API and data structures was done piecemeal. As functions and structure members were found to be useful or new features were added to the Apache API, the xs code was written for them here and there.

The goal for 2.0 is to generate the majority of xs code and provide thin wrappers where needed to make the API more Perlish. As part of this goal, nearly the entire APR and Apache API, along with their public data structures is covered from the get-go. Certain functions and structures which are considered "private" to Apache or otherwise un-useful to Perl don't get glued.

The Apache header tree is parsed into Perl data structures which live in the generated `Apache2::FunctionTable` and `Apache2::StructureTable` modules. For example, the following function prototype:

```
AP_DECLARE(int) ap_meets_conditions(request_rec *r);
```

is parsed into the following Perl structure:

```
{
  'name' => 'ap_meets_conditions'
  'return_type' => 'int',
  'args' => [
    {
      'name' => 'r',
      'type' => 'request_rec *'
    }
  ],
},
```

and the following structure:

```
typedef struct {
    uid_t uid;
    gid_t gid;
} ap_unix_identity_t;
```

is parsed into:

```
{
  'type' => 'ap_unix_identity_t'
  'elts' => [
    {
      'name' => 'uid',
      'type' => 'uid_t'
    },
    {
      'name' => 'gid',
      'type' => 'gid_t'
    }
  ],
}
```

Similar is done for the mod_perl source tree, building `ModPerl::FunctionTable` and `ModPerl::StructureTable`.

Three files are used to drive these Perl structures into the generated xs code:

- **lib/ModPerl/function.map**

  Specifies which functions are made available to Perl, along with which modules and classes they reside in. Many functions will map directly to Perl, for example the following C code:

  ```
  static int handler (request_rec *r) {
      int rc = ap_meets_conditions(r);
      ...
  ```

  maps to Perl like so:

  ```
  sub handler {
      my $r = shift;
      my $rc = $r->meets_conditions;
   ...
  ```

  The function map is also used to dispatch Apache/APR functions to thin wrappers, rewrite arguments and rename functions which make the API more Perlish where applicable. For example, C code such as:

  ```
  char uuid_buf[APR_UUID_FORMATTED_LENGTH+1];
  apr_uuid_t uuid;
  apr_uuid_get(&uuid)
  apr_uuid_format(uuid_buf, &uuid);
  printf("uuid=%s\n", uuid_buf);
  ```

  is remapped to a more Perlish convention:

  ```
  printf "uuid=%s\n", APR::UUID->new->format;
  ```

- **lib/ModPerl/structure.map**

  Specifies which structures and members of each are made available to Perl, along with which modules and classes they reside in.

- **lib/ModPerl/type.map**

  This file defines how Apache/APR types are mapped to Perl types and vice-versa. For example:

  ```
  apr_int32_t => SvIV
  apr_int64_t => SvNV
  server_rec  => SvRV (Perl object blessed into the Apache2::ServerRec class)
  ```

3.5.1 Advantages to generating XS code

- Not tied tightly to xsubpp

- Easy adjustment to Apache 2.0 API/structure changes

- Easy adjustment to Perl changes (e.g., Perl 6)

- Ability to "discover" hookable third-party C modules.

- Cleanly take advantage of features in newer Perls

- Optimizations can happen across-the-board with one-shot

- Possible to AUTOLOAD XSUBs

- Documentation can be generated from code

- Code can be generated from documentation

3.5.2 Lvalue methods

A new feature to Perl 5.6.0 is *lvalue subroutines*, where the return value of a subroutine can be directly modified. For example, rather than the following code to modify the uri:

```
$r->uri($new_uri);
```

the same result can be accomplished with the following syntax:

```
$r->uri = $new_uri;
```

mod_perl-2.0 will support *lvalue subroutines* for all methods which access Apache and APR data structures.

3.6 Filter Hooks

mod_perl 2.0 provides two interfaces to filtering, a direct mapping to buckets and bucket brigades and a simpler, stream-oriented interface. This is discussed in the Chapter on filters.

3.7 Directive Handlers

mod_perl 1.0 provides a mechanism for Perl modules to implement first-class directive handlers, but requires an XS file to be generated and compiled. The 2.0 version provides the same functionality, but does not require the generated XS module (i.e. everything is implemented in pure Perl).

3.8 <Perl> Configuration Sections

The ability to write configuration in Perl carries over from 1.0, but but implemented much different internally. The mapping of a Perl symbol table fits cleanly into the new *ap_directive_t* API, unlike the hoop jumping required in mod_perl 1.0.

3.9 Protocol Module Support

Protocol module support is provided out-of-the-box, as the hooks and API are covered by the generated code blankets. Any functionality for assisting protocol modules should be folded back into Apache if possible.

3.10 mod_perl MPM

It will be possible to write an MPM (Multi-Processing Module) in Perl. mod_perl will provide a mod_perl_mpm.c framework which fits into the server/mpm standard convention. The rest of the functionality needed to write an MPM in Perl will be covered by the generated xs code blanket.

3.11 Build System

The biggest mess in 1.0 is mod_perl's Makefile.PL, the majority of logic has been broken down and moved to the `Apache2::Build` module. The *Makefile.PL* will construct an `Apache2::Build` object which will have all the info it needs to generate scripts and *Makefile*s that apache-2.0 needs. Regardless of what that scheme may be or change to, it will be easy to adapt to with build logic/variables/etc., divorced from the actual *Makefile*s and configure scripts. In fact, the new build will stay as far away from the Apache build system as possible. The module library (*libmodperl.so* or *libmodperl.a*) is built with as little help from Apache as possible, using only the `INCLUDEDIR` provided by *apxs*.

The new build system will also "discover" XS modules, rather than hard-coding the XS module names. This allows for switchabilty between static and dynamic builds, no matter where the xs modules live in the source tree. This also allows for third-party xs modules to be unpacked inside the mod_perl tree and built static without modification to the mod_perl Makefiles.

For platforms such as Win32, the build files are generated similar to how unix-flavor *Makefile*s are.

3.12 Test Framework

Similar to 1.0, mod_perl-2.0 provides a 'make test' target to exercise as many areas of the API and module features as possible.

The test framework in 1.0, like several other areas of mod_perl, was cobbled together over the years. mod_perl 2.0 provides a test framework that is usable not only for mod_perl, but for third-party `Apache2::*` modules and Apache itself. See `Apache::Test`.

3.13 CGI Emulation

As a side-effect of embedding Perl inside Apache and caching compiled code, mod_perl has been popular as a CGI accelerator. In order to provide a CGI-like environment, mod_perl must manage areas of the runtime which have a longer lifetime than when running under mod_cgi. For example, the `%ENV` environment variable table, `END` blocks, `@INC` include paths, etc.

CGI emulation is supported in mod_perl 2.0, but done so in a way that it is encapsulated in its own handler. Rather than 1.0 which uses the same response handler, regardless if the module requires CGI emulation or not. With an *ithreads* enabled Perl, it's also possible to provide more robust namespace protection.

Notice that `ModPerl::Registry` is used instead of 1.0's `Apache::Registry`, and similar for other registry groups. `ModPerl::RegistryCooker` makes it easy to write your own customizable registry handler.

3.14 Apache2::* Library

The majority of the standard `Apache2::*` modules in 1.0 are supported in 2.0. The main goal being that the non-core CGI emulation components of these modules are broken into small, re-usable pieces to subclass Apache::Registry like behavior.

3.15 Perl Enhancements

Most of the following items were projected for inclusion in perl 5.8.0, but that didn't happen. While these enhancements do not preclude the design of mod_perl-2.0, they could make an impact if they were implemented/accepted into the Perl development track.

3.15.1 GvSHARED

(Note: This item wasn't implemented in Perl 5.8.0)

As mentioned, the perl_clone() API will create a thread-safe interpreter clone, which is a copy of all mutable data and a shared syntax tree. The copying includes subroutines, each of which take up around 255 bytes, including the symbol table entry. Multiply that number times, say 1200, is around 300K, times 10 interpreter clones, we have 3Mb, times 20 clones, 6Mb, and so on. Pure perl subroutines must be copied, as the structure includes the `PADLIST` of lexical variables used within that subroutine. However, for XSUBs, there is no PADLIST, which means that in the general case, perl_clone() will copy the subroutine, but the structure will never be written to at runtime. Other common global variables, such as `@EXPORT` and `%EXPORT_OK` are built at compile time and never modified during runtime.

Clearly it would be a big win if XSUBs and such global variables were not copied. However, we do not want to introduce locking of these structures for performance reasons. Perl already supports the concept of a read-only variable, a flag which is checked whenever a Perl variable will be written to. A patch has been submitted to the Perl development track to support a feature known as `GvSHARED`. This mechanism allows XSUBs and global variables to be marked as shared, so perl_clone() will not copy these structures, but rather point to them.

3.15.2 Shared SvPVX

The string slot of a Perl scalar is known as the `SvPVX`. As Perl typically manages the string a variable points to, it must make a copy of it. However, it is often the case that these strings are never written to. It would be possible to implement copy-on-write strings in the Perl core with little performance overhead.

3.15.3 Compile-time method lookups

A known disadvantage to Perl method calls is that they are slower than direct function calls. It is possible to resolve method calls at compile time, rather than runtime, making method calls just as fast as subroutine calls. However, there is certain information required for method look ups that are only known at runtime. To work around this, compile-time hints can be used, for example:

```
 my Apache2::Request $r = shift;
```

Tells the Perl compiler to expect an object in the `Apache2::Request` class to be assigned to `$r`. A patch has already been submitted to use this information so method calls can be resolved at compile time. However, the implementation does not take into account sub-classing of the typed object. Since the mod_perl API consists mainly of methods, it would be advantageous to re-visit the patch to find an acceptable solution.

3.15.4 Memory management hooks

Perl has its own memory management system, implemented in terms of *malloc* and *free*. As an optimization, Perl will hang onto allocations made for variables, for example, the string slot of a scalar variable. If a variable is assigned, for example, a 5k chunk of HTML, Perl will not release that memory unless the variable is explicitly *undef*ed. It would be possible to modify Perl in such a way that the management of these strings are pluggable, and Perl could be made to allocate from an APR memory pool. Such a feature would maintain the optimization Perl attempts (to avoid malloc/free), but would greatly reduce the process size as pool resources are able to be re-used elsewhere.

3.15.5 Opcode hooks

Perl already has internal hooks for optimizing opcode trees (syntax tree). It would be quite possible for extensions to add their own optimizations if these hooks were plugable, for example, optimizing calls to *print*, so they directly call the Apache *ap_rwrite* function, rather than proxy via a *tied filehandle*.

Another optimization that was implemented is "inlined" XSUB calls. Perl has a generic opcode for calling subroutines, one which does not know the number of arguments coming into and being passed out of a subroutine. As the majority of mod_perl API methods have known in/out argument lists, mod_perl implements a much faster version of the Perl *pp_entersub* routine.

3.16 Maintainers

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

Doug MacEachern <dougm (at) covalent.net>

3.17 Authors

- **Doug MacEachern <dougm (at) covalent.net>**

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

4   Installing mod_perl 2.0

4.1 Description

This chapter provides an in-depth mod_perl 2.0 installation coverage.

4.2 Prerequisites

Before building mod_perl 2.0 you need to have its prerequisites installed. If you don't have them, download and install them first, using the information in the following sections. Otherwise proceed directly to the mod_perl building instructions.

The mod_perl 2.0 prerequisites are:

- **Apache**

  Apache 2.0 is required. mod_perl 2.0 **does not** work with Apache 1.3.

  Dynamic (DSO) mod_perl build requires Apache 2.0.47 or higher. Static build requires Apache 2.0.51 or higher.

- **Perl**
  - **Prefork MPM**

    Requires at least Perl version 5.6.1.

    You don't need to have threads-support enabled in Perl. If you do have it, it **must** be *ithreads* and not *5005threads*! If you have:

    ```
    % perl5.8.0 -V:use5005threads
    use5005threads='define';
    ```

    you must rebuild Perl without threads enabled or with -Dusethreads. Remember that threads-support slows things down and on some platforms it's unstable (e.g., FreeBSD), so don't enable it unless you really need it.

  - **64 bit Linux**

    If while running make test while building mod_perl 2 you get an error like this:

    ```
    /usr/bin/ld: /usr/local/lib/perl5/5.10.1/x86_64-linux/CORE/libperl.a(op.o): \
    relocation R_X86_64_32S against 'PL_sv_yes' can not be used when making a shared \
                    object; recompile with -fPIC
    /usr/local/lib/perl5/5.10.1/x86_64-linux/CORE/libperl.a: could not read symbols: Bad \
                    value
    ```

    You're likely on 64 bit Linux and will need to build Perl for that platform. You can do so by running Perl's Configure with the $CFLAGS environment variable and the -A and ccflags options. So if you normally build Perl with:

```
% ./Configure -des
```

You would instead configure with:

```
% CFLAGS='-m64 -mtune=nocona' ./Configure -des -A ccflags=-fPIC
```

○ **Threaded MPMs**

Require at least Perl version 5.8.0 with ithreads support built-in. That means that it should report:

```
% perl5.8.0 -V:useithreads -V:usemultiplicity
useithreads='define';
usemultiplicity='define';
```

If that's not what you see rebuild Perl with `-Dusethreads`.

○ **Static prefork build**

Perl with ithreads support version 5.6.1 or higher

Perl without ithreads support version 5.8.2 or higher

○ **Static non-prefork build**

Perl with ithreads support version 5.8.0 or higher

○ **threads.pm**

If you want to run applications that take benefit of Perl's *threads.pm* Perl version 5.8.1 or higher w/ithreads enabled is required. Perl 5.8.0's *threads.pm* doesn't work with mod_perl 2.0.

● **CPAN Perl Modules**

The mod_perl 2.0 test suite has several requirements on its own. If you don't satisfy them, the tests depending on these requirements will be skipped, which is OK, but you won't get to run these tests and potential problems, which may exhibit themselves in your own code, could be missed. We don't require them from `Makefile.PL`, which could have been automated the requirements installation, in order to have less dependencies to get mod_perl 2.0 installed.

Also if your code uses any of these modules, chances are that you will need to use at least the version numbers listed here.

○ **CGI.pm 3.11**
○ **Compress::Zlib 1.09**

Though the easiest way to satisfy all the dependencies is to install `Bundle::Apache2` available from CPAN.

4.2.1 Downloading Stable Release Sources

If you are going to install mod_perl on a production site, you want to use the officially released stable components. Since the latest stable versions change all the time you should check for the latest stable version at the listed below URLs:

- **Perl**

  Download from: *http://cpan.org/src/README.html*

  This   direct   link   which   symlinks   to   the   latest   release   should   work   too: *http://cpan.org/src/stable.tar.gz*.

  For the purpose of examples in this chapter we will use the package named *perl-5.8.x.tar.gz*, where *x* should be replaced with the real version number.

- **Apache**

  Download from: *http://www.apache.org/dist/httpd/*

  For the purpose of examples in this chapter we will use the package named *httpd-2.x.xx.tar.gz*, where *x.xx* should be replaced with the real version number.

4.2.2 Getting Bleeding Edge Sources

If you really know what you are doing you can use the cvs/svn versions of the components. Chances are that you don't want to them on a production site. You have been warned!

- **Perl**

  The cutting edge version of Perl (aka bleadperl or bleedperl) is only generally available through an rsync repository maintained by ActiveState:

  ```
  # (--delete to ensure a clean state)
  % rsync -acvz --delete --force \
    rsync://public.activestate.com/perl-current/ perl-current
  ```

  If you are re-building Perl after rsync-ing, make sure to cleanup first:

  ```
  % make distclean
  ```

  before running `./Configure`.

  You'll also want to install (at least) LWP if you want to fully test mod_perl. You can install LWP with `CPAN.pm` shell:

  ```
  % perl -MCPAN -e 'install("LWP")'
  ```

For more details on bleadperl, see *http://dev.perl.org/perl5/source.html*.

- **Apache**

See Development mod_perl 2.0 Source Distribution.

4.2.3 Configuring and Installing Prerequisites

If you don't have the prerequisites installed yet, install them now.

4.2.3.1 Perl

```
% cd perl-5.8.x
% ./Configure -des
```

If you need the threads support, run:

```
% ./Configure -des -Dusethreads
```

Most likely you don't want perl-support for threads enabled, in which case pass: `-Uusethreads` instead of `-Dusethreads`.

If you want to debug mod_perl segmentation faults, add the following *./Configure* options:

```
-Doptimize='-g' -Dusedevel
```

Now build it:

```
% make && make test && make install
```

4.2.3.2 Apache

You need to have Apache built and installed prior to building mod_perl, only if you intend build a DSO mod_perl. If you intend to build a statically linked Apache+mod_perl, you only need to have the Apache source available (mod_perl will build and install Apache for you), you should skip this step.

```
% cd httpd-2.x.xx
% ./configure --prefix=$HOME/httpd/prefork --with-mpm=prefork
% make && make install
```

Starting from 2.0.49, the Apache logging API escapes everything that goes to *error_log*, therefore if you're annoyed by this feature during the development phase (as your error messages will be all messed up) you can disable the escaping during the Apache build time:

```
% CFLAGS="-DAP_UNSAFE_ERROR_LOG_UNESCAPED" ./configure ...
```

Do **not** use that CFLAGS in production unless you know what you are doing.

4.3 Installing mod_perl from Binary Packages

As of this writing only the binaries for the Win32 platform are available, kindly prepared and maintained by Randy Kobes. See the documentation on Win32 binaries for details.

Some RPM packages can be found using rpmfind services, e.g.:

http://www.rpmfind.net/linux/rpm2html/search.php?query=mod_perl&submit=Search+... However if you have problems using them, you have to contact those who have created them.

4.4 Installing mod_perl from Source

Building from source is the best option, because it ensures a binary compatibility with Apache and Perl. However it's possible that your distribution provides a solid binary mod_perl 2.0 package.

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

4.4.1 Downloading the mod_perl Source

First download the mod_perl source.

- **Stable Release**

  Download from *http://perl.apache.org/download/* or your favorite CPAN mirror.

  This direct link which symlinks to the latest release should work too: *http://apache.org/dist/perl/mod_perl-2.0-current.tar.gz*.

  For the purpose of examples in this chapter we will use the package named *mod_perl-2.x.x.tar.gz*, where *x.x* should be replaced with the real version number.

  Open the package with:

  ```
  % tar -xvzf mod_perl-2.x.x.tar.gz
  ```

  or an equivalent command.

- **Development Version**

  See Development mod_perl 2.0 Source Distribution.

4.4.2 Configuring mod_perl

To build mod_perl, you **must** also use the same compiler that Perl was built with. You can find that out by running `perl -V` and looking at the `Compiler:` section.

Like any other Perl module, mod_perl is configured via the *Makefile.PL* file, but requires one or more configuration options:

```
% cd modperl-2.x.x
% perl Makefile.PL <options>
```

where *options* is an optional list of key/value pairs. These options can include all the usual options supported by `ExtUtils::MakeMaker` (e.g., `PREFIX`, `LIB`, etc.).

The following sections give the details about all the available options, but let's mention first an important one.

Configuration options are discussed in Build Options.

4.4.2.1 Dynamic mod_perl

Before you proceed, make sure that Apache 2.0 has been built and installed. mod_perl **cannot** be built before that.

It seems that most users use pre-packaged Apache installation, most of which tend to spread the Apache files across many directories (i.e. not using --enable-layout=Apache, which puts all the files under the same directory). If Apache 2.0 files are spread under different directories, you need to use at least the `MP_APXS` option, which should be set to a full path to the `apxs` executable. For example:

```
% perl Makefile.PL MP_APXS=/path/to/apxs
```

For example RedHat Linux system installs the `httpd` binary, the `apxs` and `apr-config` scripts (the latter two are needed to build mod_perl) all in different locations, therefore they configure mod_perl 2.0 as:

```
% perl Makefile.PL MP_APXS=/path/to/apxs \
  MP_APR_CONFIG=/another/path/to/apr-config <other options>
```

However a correctly built Apache shouldn't require the `MP_APR_CONFIG` option, since `MP_APXS` should provide the location of this script.

If however all Apache 2.0 files were installed under the same directory, mod_perl 2.0's build only needs to know the path to that directory, passed via the `MP_AP_PREFIX` option:

```
% perl Makefile.PL MP_AP_PREFIX=$HOME/httpd/prefork
```

4.4.2.2 Static mod_perl

Before you proceed make sure that Apache 2.0 has been downloaded and extracted. mod_perl **cannot** be built before that.

If this is an svn checkout and not an official distribution tarball, you need to first run:

```
% cd httpd-2.0
% ./buildconf
```

To enable statically linking mod_perl into Apache, use the `MP_USE_STATIC` flag like this:

```
% perl Makefile.PL MP_USE_STATIC=1 \
  MP_AP_PREFIX=$HOME/src/httpd-2.x \
  MP_AP_CONFIGURE="--with-mpm=prefork"
```

`MP_AP_PREFIX` **must** point to an extracted Apache 2.0 source tree.

This will configure Apache by passing `MP_AP_CONFIGURE` to Apache's *./configure* script.

Here is an example:

```
% cd ~/src
% tar -xvzf perl-5.8.x.tar.gz
% cd perl-5.8.x
% ./Configure -des
% make install
% cd ..
% tar -xvzf httpd-2.0.xx.tar.gz
% tar -xvzf mod_perl-2.x.x.tar.gz
% perl5.8.x Makefile.PL \
  MP_USE_STATIC=1 \
  MP_AP_PREFIX="$HOME/src/httpd-2.0.xx" \
  MP_AP_CONFIGURE="--with-mpm=prefork"
% make
% make test
% make install
% ./httpd -l | grep perl
   mod_perl.c
```

4.4.3 mod_perl Build Options

4.4.3.1 Boolean Build Options

The following options are boolean and can be set with `MP_XXX=1` or unset with `MP_XXX=0`, where XXX is the name of the option.

4.4.3.1.1 MP_PROMPT_DEFAULT

Accept default values for all would-be prompts.

4.4.3.1.2 MP_GENERATE_XS

Generate XS code from parsed source headers in *xs/tables/$httpd_version*. Default is 1, set to 0 to disable.

4.4.3.1.3 MP_USE_DSO

Build mod_perl as a DSO (*mod_perl.so*). This is the default.

4.4.3.1.4 MP_USE_STATIC

Build static mod_perl (*mod_perl.a*).

4.4.3.1.5 MP_STATIC_EXTS

Build `Apache2::*.xs` as static extensions.

4.4.3.1.6 MP_USE_GTOP

Link with *libgtop* and enable *libgtop* reporting.

4.4.3.1.7 MP_COMPAT_1X

`MP_COMPAT_1X=1` or a lack of it enables several mod_perl 1.0 back-compatibility features, which are deprecated in mod_perl 2.0. It's enabled by default, but can be disabled with `MP_COMPAT_1X=0` during the build process.

When this option is disabled, the following things will happen:

- Deprecated special variable, `$Apache2::__T` won't be available. Use `${^TAINT}` instead.

- *$ServerRoot* and *$ServerRoot/lib/perl* won't be appended to `@INC`. Instead use:

  ```
  PerlSwitches -I/path/to/server -I/path/to/server/lib/perl
  ```

  in *httpd.conf* or:

  ```
  use Apache2::ServerUtil ();
  use File::Spec::Functions qw(catfile);
  push @INC, catfile Apache2::ServerUtil::server_root, "";
  push @INC, catfile Apache2::ServerUtil::server_root, "lib/perl";
  ```

  in *startup.pl*.

- The following deprecated configuration directives won't be recognized by Apache:

  ```
  PerlSendHeader
  PerlSetupEnv
  PerlHandler
  PerlTaintCheck
  PerlWarn
  ```

  Use their 2.0 equivalents instead.

4.4.3.1.8 MP_DEBUG

Turn on debugging (`-g -lperld`) and tracing.

4.4.3.1.9 MP_MAINTAINER

Enable maintainer compile mode, which sets `MP_DEBUG=1` and adds the following `gcc` flags:

```
-DAP_DEBUG -Wall -Wmissing-prototypes -Wstrict-prototypes \
-Wmissing-declarations \
```

If gcc version 3.3.2+ is found, not compiling on OpenBSD, and `-Wdeclaration-after-state-ment` is not already part of the `gcc` flags add it.

To use this mode Apache must be build with `--enable-maintainer-mode`.

4.4.3.1.10 MP_TRACE

Enable tracing

4.4.3.2 Non-Boolean Build Options

set the non-boolean options with MP_XXX=value.

4.4.3.2.1 MP_APXS

Path to `apxs`. For example if you've installed Apache 2.0 under */home/httpd/httpd-2.0* as DSO, the default location would be */home/httpd/httpd-2.0/bin/apxs*.

4.4.3.2.2 MP_AP_CONFIGURE

The command-line arguments to pass to httpd's configure script.

4.4.3.2.3 MP_AP_PREFIX

Apache installation prefix, under which the *include/* directory with Apache C header files can be found. For example if you've installed Apache 2.0 in directory *\Apache2* on Win32, you should use:

```
MP_AP_PREFIX=\Apache2
```

If Apache is not installed yet, you can point to the Apache 2.0 source directory, but only after you've built or configured Apache in it. For example:

```
MP_AP_PREFIX=/home/stas/apache.org/httpd-2.0
```

Though in this case `make test` won't automatically find `httpd`, therefore you should run `t/TEST` instead and pass the location of `apxs` or `httpd`, e.g.:

```
% t/TEST -apxs /home/stas/httpd/prefork/bin/apxs
```

or

```
% t/TEST -httpd /home/stas/httpd/prefork/bin/httpd
```

4.4.3.2.4 MP_AP_DESTDIR

This option exists to make the lives of package maintainers easier. If you aren't a package manager you should not need to use this option.

Apache installation destination directory. This path will be prefixed to the installation paths for all Apache-specific files during `make install`. For instance, if Apache modules are normally installed into */path/to/httpd-2.0/modules/* and `MP_AP_DESTDIR` is set to */tmp/foo*, the *mod_perl.so* will be installed in:

```
/tmp/foo/path/to/httpd-2.0/modules/mod_perl.so
```

4.4.3.2.5 MP_APR_CONFIG

If APR wasn't installed under the same file tree as httpd, you may need to tell the build process where it can find the executable `apr-config`, which can then be used to figure out where the apr and aprutil *include/* and *lib/* directories can be found.

4.4.3.2.6 MP_CCOPTS

Add to compiler flags, e.g.:

```
MP_CCOPTS=-Werror
```

(Notice that `-Werror` will work only with the Perl version 5.7 and higher.)

4.4.3.2.7 MP_OPTIONS_FILE

Read build options from given file. e.g.:

```
MP_OPTIONS_FILE=~/.my_mod_perl2_opts
```

4.4.3.2.8 MP_APR_LIB

On Win32, in order to build the APR and APR::* modules so as to be independent of mod_perl.so, a static library is first built containing the needed functions these modules link into. The option

```
MP_APR_LIB=aprext
```

specifies the name that this library has. The default used is `aprext`. This option has no effect on platforms other than Win32, as they use a different mechanism to accomplish the decoupling of APR and APR::* from mod_perl.so.

4.4.3.3 mod_perl-specific Compiler Options

4.4.3.3.1 -DMP_IOBUFSIZE

Change the default mod_perl's 8K IO buffer size, e.g. to 16K:

```
MP_CCOPTS=-DMP_IOBUFSIZE=16384
```

4.4.3.4 mod_perl Options File

Options can also be specified in the file *makepl_args.mod_perl2* or *.makepl_args.mod_perl2*. The file can be placed under $ENV{HOME}, the root of the source package or its parent directory. So if you unpack the mod_perl source into */tmp/mod_perl-2.x/* and your home is */home/foo/*, the file will be searched in:

```
/tmp/mod_perl-2.x/makepl_args.mod_perl2
/tmp/makepl_args.mod_perl2
/home/foo/makepl_args.mod_perl2
/tmp/mod_perl-2.x/.makepl_args.mod_perl2
/tmp/.makepl_args.mod_perl2
/home/foo/.makepl_args.mod_perl2
```

If the file specified in `MP_OPTIONS_FILE` is found the *makepl_args.mod_perl2* will be ignored.

Options specified on the command line override those from *makepl_args.mod_perl2* and those from `MP_OPTIONS_FILE`.

If your terminal supports colored text you may want to set the environment variable `APACHE_TEST_COLOR` to 1 to enable the colored tracing which makes it easier to tell the reported errors and warnings, from the rest of the notifications.

4.4.4 Re-using Configure Options

Since mod_perl remembers what build options were used to build it if first place, you can use this knowledge to rebuild itself using the same options. Simply `chdir(1)` to the mod_perl source directory and run:

```
% cd modperl-2.x.
% perl -MApache2::Build -e rebuild
```

4.4.5 Compiling mod_perl

Next stage is to build mod_perl:

```
% make
```

4.4.6 Testing mod_perl

When mod_perl has been built, it's very important to test that everything works on your machine:

```
% make test
```

If something goes wrong with the test phase and want to figure out how to run individual tests and pass various options to the test suite, see the corresponding sections of the bug reporting guidelines or the Apache::Test Framework tutorial.

4.4.7 Installing mod_perl

Once the test suite has passed, it's a time to install mod_perl.

```
% make install
```

If you install mod_perl system wide, you probably need to become *root* prior to doing the installation:

```
% su
# make install
```

4.5 If Something Goes Wrong

If something goes wrong during the installation, try to repeat the installation process from scratch, while verifying all the steps with this document.

If the problem persists report the problem.

4.6 Maintainers

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

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

4.7 Authors

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

- Doug MacEachern <dougm (at) covalent.net>

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

5 mod_perl 2.0 Server Configuration

5.1 Description

This chapter provides an in-depth mod_perl 2.0 configuration details.

5.2 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.

5.3 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.

5.4 Server Configuration Directives

5.4.1 `<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.

5.4.2 `=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.

5.4.3 `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:

```
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.

5.4.4 `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 `PerlPostConfigRe-quire`.

One of the reasons for avoding 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.

5.4.5 `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.

5.4.6 `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.

5.4.7 *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:

5.4.7.1 **Enable**

On by default, can be used to disable mod_perl for a given `VirtualHost`. For example:

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

5.4.7.2 **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.

5.4.7.3 `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>
```

5.4.7.4 `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.

5.4.7.5 `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 `Perl-LogHandler` 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>
```

5.4.7.6 `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.

5.4.7.7 `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.

5.4.7.8 `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";
```

5.4.7.9 `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.

5.4.7.10 `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*.

5.4.8 `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.

5.4.9 `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.

5.4.10 *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.

5.4.11 *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.

5.4.12 *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.

5.4.13 `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.

5.4.14 `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.

5.4.14.1 `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.

5.4.14.2 `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.

5.4.14.3 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*.

5.5 Server Life Cycle Handlers Directives

See Server life cycle.

5.5.1 `PerlOpenLogsHandler`

See `PerlOpenLogsHandler`.

5.5.2 `PerlPostConfigHandler`

See `PerlPostConfigHandler`.

5.5.3 `PerlChildInitHandler`

See `PerlChildInitHandler`.

5.5.4 `PerlChildExitHandler`

See `PerlChildExitHandler`.

5.6 Protocol Handlers Directives

See Protocol handlers.

5.6.1 `PerlPreConnectionHandler`

See `PerlPreConnectionHandler`.

5.6.2 `PerlProcessConnectionHandler`

See `PerlProcessConnectionHandler`.

5.7 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:

5.7.1 `PerlInputFilterHandler`

See `PerlInputFilterHandler`.

5.7.2 `PerlOutputFilterHandler`

See `PerlOutputFilterHandler`.

5.7.3 `PerlSetInputFilter`

See `PerlSetInputFilter`.

5.7.4 `PerlSetOutputFilter`

See `PerlSetInputFilter`.

5.8 HTTP Protocol Handlers Directives

See HTTP protocol handlers.

5.8.1 `PerlPostReadRequestHandler`

See `PerlPostReadRequestHandler`.

5.8.2 `PerlTransHandler`

See `PerlTransHandler`.

5.8.3 `PerlMapToStorageHandler`

See `PerlMapToStorageHandler`.

5.8.4 `PerlInitHandler`

See `PerlInitHandler`.

5.8.5 `PerlHeaderParserHandler`

See `PerlHeaderParserHandler`.

5.8.6 `PerlAccessHandler`

See `PerlAccessHandler`.

5.8.7 `PerlAuthenHandler`

See `PerlAuthenHandler`.

5.8.8 `PerlAuthzHandler`

See `PerlAuthzHandler`.

5.8.9 `PerlTypeHandler`

See `PerlTypeHandler`.

5.8.10 `PerlFixupHandler`

See `PerlFixupHandler`.

5.8.11 `PerlResponseHandler`

See `PerlResponseHandler`.

5.8.12 `PerlLogHandler`

See `PerlLogHandler`.

5.8.13 `PerlCleanupHandler`

See `PerlCleanupHandler`.

5.9 Threads Mode Specific Directives

These directives are enabled only in a threaded mod_perl+Apache combo:

5.9.1 `PerlInterpStart`

The number of interpreters to clone at startup time.

Default value: 3

See also: this directive argument types and allowed location.

5.9.2 `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.

5.9.3 `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.

5.9.4 `PerlInterpMaxSpare`

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

Default value: 3

5.9.5 `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.

5.9.6 `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 `PerlFix-upHandler` 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.

5.10 Debug Directives

5.10.1 *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.

5.11 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
  -------------------------------------------
  PerlSwitches                 ITERATE    SRV
  PerlRequire                  ITERATE    SRV
  PerlConfigRequire            ITERATE    SRV
  PerlPostConfigRequire        ITERATE    SRC
  PerlModule                   ITERATE    SRV
  PerlLoadModule               RAW_ARGS   SRV
  PerlOptions                  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.

5.12 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";
}
```

5.12.1 `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
}
```

5.13 Perl Interface to the Apache Configuration Tree

For now refer to the Apache2::Directive manpage and the test *t/response/TestApache2/conftree.pm* in the mod_perl source distribution.

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

5.14 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
    ```

5.14.1 `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 (`Perl-Switches -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.

5.14.2 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.

5.15 General Issues

5.16 Maintainers

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

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

5.17 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.

6   Apache Server Configuration Customization in Perl

6.1 Description

This chapter explains how to create custom Apache configuration directives in Perl.

6.2 Incentives

mod_perl provides several ways to pass custom configuration information to the modules.

The simplest way to pass custom information from the configuration file to the Perl module is to use the `PerlSetVar` and `PerlAddVar` directives. For example:

```
PerlSetVar Secret "Matrix is us"
```

and in the mod_perl code this value can be retrieved as:

```
my $secret = $r->dir_config("Secret");
```

Another alternative is to add custom configuration directives. There are several reasons for choosing this approach:

- When the expected value is not a simple argument, but must be supplied using a certain syntax, Apache can verify at startup time that this syntax is valid and abort the server start up if the syntax is invalid.

- Custom configuration directives are faster because their values are parsed at the startup time, whereas `PerlSetVar` and `PerlAddVar` values are parsed at the request time.

- It's possible that some other modules have accidentally chosen to use the same key names but for absolutely different needs. So the two now can't be used together. Of course this collision can be avoided if a unique to your module prefix is used in the key names. For example:

    ```
    PerlSetVar ApacheFooSecret "Matrix is us"
    ```

Finally, modules can be configured in pure Perl using `<Perl> Sections` or a startup file, by simply modifying the global variables in the module's package. This approach could be undesirable because it requires a use of globals, which we all try to reduce. A bigger problem with this approach is that you can't have different settings for different sections of the site (since there is only one version of a global variable), something that the previous two approaches easily achieve.

6.3 Creating and Using Custom Configuration Directives

In mod_perl 2.0, adding new configuration directives is a piece of cake, because it requires no XS code and *Makefile.PL*, needed in case of mod_perl 1.0. In mod_perl 2.0, custom directives are implemented in pure Perl.

Here is a very basic module that declares two new configuration directives: `MyParameter`, which accepts one or more arguments, and `MyOtherParameter` which accepts a single argument. `MyParameter` validates that its arguments are valid strings.

```
#file:MyApache2/MyParameters.pm
#---------------------------
package MyApache2::MyParameters;

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

use Apache::Test;
use Apache::TestUtil;

use Apache2::Const -compile => qw(OR_ALL ITERATE);

use Apache2::CmdParms ();
use Apache2::Module ();
use Apache2::Directive ();

my @directives = (
    {
     name         => 'MyParameter',
     func         => __PACKAGE__ . '::MyParameter',
     req_override => Apache2::Const::OR_ALL,
     args_how     => Apache2::Const::ITERATE,
     errmsg       => 'MyParameter Entry1 [Entry2 ... [EntryN]]',
    },
    {
     name         => 'MyOtherParameter',
    },
);
Apache2::Module::add(__PACKAGE__, \@directives);

sub MyParameter {
    my ($self, $parms, @args) = @_;
    $self->{MyParameter} = \@args;

    # validate that the arguments are strings
    for (@args) {
        unless (/^\w+$/) {
            my $directive = $parms->directive;
            die sprintf "error: MyParameter at %s:%d expects " .
                "string arguments: ('$_' is not a string)\n",

                $directive->filename, $directive->line_num;
        }
    }
}
1;
```

And here is how to use it in *httpd.conf*:

```
# first load the module so Apache will recognize the new directives
PerlLoadModule MyApache2::MyParameters

MyParameter one two three
MyOtherParameter Foo
<Location /perl>
  MyParameter eleven twenty
  MyOtherParameter Bar
</Location>
```

The following sections discuss this and more advanced modules in detail.

A minimal configuration module is comprised of three groups of elements:

- **An array @directives for declaring the new directives and their behavior.**
- **A call to Apache2::Module::add() to register the new directives with apache.**
- **A subroutine per each new directive, which is called when the directive is seen**

6.3.1 @directives

@directives is an array of hash references. Each hash represents a separate new configuration directive. In our example we had:

```
my @directives = (
    {
     name         => 'MyParameter',
     func         => __PACKAGE__ . '::MyParameter',
     req_override => Apache2::Const::OR_ALL,
     args_how     => Apache2::Const::ITERATE,
     errmsg       => 'MyParameter Entry1 [Entry2 ... [EntryN]]',
    },
    {
     name         => 'MyOtherParameter',
    },
);
```

This structure declares two new directives: MyParameter and MyOtherParameter. You have to declare at least the name of the new directive, which is how we have declared the MyOtherParameter directive. mod_perl will fill in the rest of the configuration using the defaults described next.

These are the attributes that can be used to define the directives behavior: *name*, *func*, *args_how*, *req_override* and *errmsg*. They are discussed in the following sections.

It is worth noting that in previous versions of mod_perl, it was necessary to call this variable @APACHE_MODULE_COMMANDS. It is not the case anymore, and we consistently use the name @directives in the documentation for clarity. It can be named anything at all.

6.3.1.1 `name`

This is the only required attribute. And it declares the name of the new directive as it'll be used in *httpd.conf*.

6.3.1.2 `func`

The *func* attribute expects a reference to a function or a function name. This function is called by httpd every time it encounters the directive that is described by this entry while parsing the configuration file. Therefore it's invoked once for every instance of the directive at the server startup, and once per request per instance in the *.htaccess* file.

This function accepts two or more arguments, depending on the *args_how* attribute's value.

This attribute is optional. If not supplied, mod_perl will try to use a function in the current package whose name is the same as of the directive in question. In our example with `MyOtherParameter`, mod_perl will use:

```
__PACKAGE__ . '::MyOtherParameter'
```

as a name of a subroutine and it anticipates that it exists in that package.

6.3.1.3 `req_override`

The attribute defines the valid scope in which this directive can appear. There are several constants which map onto the corresponding Apache macros. These constants should be imported from the `Apache2::Const` package.

For example, to use the `OR_ALL` constant, which allows directives to be defined anywhere, first, it needs to be imported:

```
use Apache2::Const -compile => qw(OR_ALL);
```

and then assigned to the *req_override* attribute:

```
req_override => Apache2::Const::OR_ALL,
```

It's possible to combine several options using the unary operators. For example, the following setting:

```
req_override => Apache2::Const::RSRC_CONF | Apache2::Const::ACCESS_CONF
```

will allow the directive to appear anywhere in *httpd.conf*, but forbid it from ever being used in *.htaccess* files:

This attribute is optional. If not supplied, the default value of `Apache2::Const::OR_ALL` is used.

6.3.1.4 `args_how`

Directives can receive zero, one or many arguments. In order to help Apache validate that the number of arguments is valid, the *args_how* attribute should be set to the desired value. Similar to the *req_override* attribute, the `Apache2::Const` package provides a special `:cmd_how` constants group which maps to the corresponding Apache macros. There are several constants to choose from.

In our example, the directive `MyParameter` accepts one or more arguments, therefore we have the `Apache2::Const::ITERATE` constant:

```
args_how => Apache2::Const::ITERATE,
```

This attribute is optional. If not supplied, the default value of `Apache2::Const::TAKE1` is used.

6.3.1.5 `errmsg`

The *errmsg* attribute provides a short but succinct usage statement that summarizes the arguments that the directive takes. It's used by Apache to generate a descriptive error message, when the directive is configured with a wrong number of arguments.

In our example, the directive `MyParameter` accepts one or more arguments, therefore we have chosen the following usage string:

```
errmsg => 'MyParameter Entry1 [Entry2 ... [EntryN]]',
```

This attribute is optional. If not supplied, the default value of will be a string based on the directive's *name* and *args_how* attributes.

6.3.1.6 `cmd_data`

Sometimes it is useful to pass information back to the directive handler callback. For instance, if you use the *func* parameter to specify the same callback for two different directives you might want to know which directive is being called currently. To do this, you can use the *cmd_data* parameter, which allows you to store arbitrary strings for later retrieval from your directive handler. For instance:

```
my @directives = (
    {
     name         => '<Location',
     # func defaults to Location()
     req_override => Apache2::Const::RSRC_CONF,
     args_how     => Apache2::Const::RAW_ARGS,
    },
    {
     name         => '<LocationMatch',
     func         => Location,
     req_override => Apache2::Const::RSRC_CONF,
     args_how     => Apache2::Const::RAW_ARGS,
     cmd_data     => '1',
    },
);
```

Here, we are using the `Location()` function to process both the `Location` and `LocationMatch` directives. In the `Location()` callback we can check the data in the *cmd_data* slot to see whether the directive being processed is `LocationMatch` and alter our logic accordingly. How? Through the `info()` method exposed by the `Apache2::CmdParms` class.

```
use Apache2::CmdParms ();

sub Location {

  my ($cfg, $parms, $data) = @_;

  # see if we were called via LocationMatch
  my $regex = $parms->info;

  # continue along
}
```

In case you are wondering, `Location` and `LocationMatch` were chosen for a reason - this is exactly how httpd core handles these two directives.

6.3.2 Registering the new directives

Once the `@directives` array is populated, it needs to be registered with apache using `Apache2::Module::add()`

```
Apache2::Module::add(__PACKAGE__, \@directives);
```

6.3.3 Directive Scope Definition Constants

The *req_override* attribute specifies the configuration scope in which it's valid to use a given configuration directive. This attribute's value can be any of or a combination of the following constants:

(these constants are declared in *httpd-2.0/include/http_config.h*.)

6.3.3.1 `Apache2::Const::OR_NONE`

The directive cannot be overridden by any of the `AllowOverride` options.

6.3.3.2 `Apache2::Const::OR_LIMIT`

The directive can appear within directory sections, but not outside them. It is also allowed within *.htaccess* files, provided that `AllowOverride Limit` is set for the current directory.

6.3.3.3 `Apache2::Const::OR_OPTIONS`

The directive can appear anywhere within *httpd.conf*, as well as within *.htaccess* files provided that `AllowOverride Options` is set for the current directory.

6.3.3.4 `Apache2::Const::OR_FILEINFO`

The directive can appear anywhere within *httpd.conf*, as well as within *.htaccess* files provided that `AllowOverride FileInfo` is set for the current directory.

6.3.3.5 `Apache2::Const::OR_AUTHCFG`

The directive can appear within directory sections, but not outside them. It is also allowed within *.htaccess* files, provided that `AllowOverride AuthConfig` is set for the current directory.

6.3.3.6 `Apache2::Const::OR_INDEXES`

The directive can appear anywhere within *httpd.conf*, as well as within *.htaccess* files provided that `AllowOverride Indexes` is set for the current directory.

6.3.3.7 `Apache2::Const::OR_UNSET`

META: details? "unset a directive (in Allow)"

6.3.3.8 `Apache2::Const::ACCESS_CONF`

The directive can appear within directory sections. The directive is not allowed in *.htaccess* files.

6.3.3.9 `Apache2::Const::RSRC_CONF`

The directive can appear in *httpd.conf* outside a directory section (`<Directory>`, `<Location>` or `<Files>`; also `<FilesMatch>` and kin). The directive is not allowed in *.htaccess* files.

6.3.3.10 `Apache2::Const::EXEC_ON_READ`

Force directive to execute a command which would modify the configuration (like including another file, or `IFModule`).

Normally, Apache first parses the configuration tree and then executes the directives it has encountered (e.g., `SetEnv`). But there are directives that must be executed during the initial parsing, either because they affect the configuration tree (e.g., `Include` may load extra configuration) or because they tell Apache about new directives (e.g., `IfModule` or `PerlLoadModule`, may load a module, which installs handlers for new directives). These directives must have the `Apache2::Const::EXEC_ON_READ` turned on.

6.3.3.11 `Apache2::Const::OR_ALL`

The directive can appear anywhere. It is not limited in any way.

6.3.4 Directive Callback Subroutine

Depending on the value of the *args_how* attribute the callback subroutine, specified with the *func* attribute, will be called with two or more arguments. The first two arguments are always $self and $parms. A typical callback function which expects a single value (Apache2::Const::TAKE1) might look like the following:

```
sub MyParam {
    my ($self, $parms, $arg) = @_;
    $self->{MyParam} = $arg;
}
```

In this function we store the passed single value in the configuration object, using the directive's name (assuming that it was MyParam) as the key.

Let's look at the subroutine arguments in detail:

1. $self is the current container's configuration object.

   This configuration object is a reference to a hash, in which you can store arbitrary key/value pairs. When the directive callback function is invoked it may already include several key/value pairs inserted by other directive callbacks or during the SERVER_CREATE and DIR_CREATE functions, which will be explained later.

   Usually the callback function stores the passed argument(s), which later will be read by SERVER_MERGE and DIR_MERGE, which will be explained later, and of course at request time.

   The convention is use the name of the directive as the hash key, where the received values are stored. The value can be a simple scalar, or a reference to a more complex structure. So for example you can store a reference to an array, if there is more than one value to store.

   This object can be later retrieved at request time via:

   ```
   my $dir_cfg = $self->get_config($s, $r->per_dir_config);
   ```

   You can retrieve the server configuration object via:

   ```
   my $srv_cfg = $self->get_config($s);
   ```

   if invoked inside the virtual host, the virtual host's configuration object will be returned.

2. $parms is an Apache2::CmdParms object from which you can retrieve various other information about the configuration. For example to retrieve the server object:

   ```
   my $s = $parms->server;
   ```

   See Apache2::CmdParms for more information.

3. The rest of the arguments whose number depends on the *args_how*'s value are covered in the next section.

6.3.5 *Directive Syntax Definition Constants*

The following values of the *args_how* attribute define how many arguments and what kind of arguments directives can accept. These values are constants that can be imported from the `Apache2::Const` package (`:cmd_how constants group`).

For example:

```
use Apache2::Const -compile => qw(TAKE1 TAKE23);
```

6.3.5.1 `Apache2::Const::NO_ARGS`

The directive takes no arguments. The callback will be invoked once each time the directive is encountered. For example:

```
sub MyParameter {
    my ($self, $parms) = @_;
    $self->{MyParameter}++;
}
```

6.3.5.2 `Apache2::Const::TAKE1`

The directive takes a single argument. The callback will be invoked once each time the directive is encountered, and its argument will be passed as the third argument. For example:

```
sub MyParameter {
    my ($self, $parms, $arg) = @_;
    $self->{MyParameter} = $arg;
}
```

6.3.5.3 `Apache2::Const::TAKE2`

The directive takes two arguments. They are passed to the callback as the third and fourth arguments. For example:

```
sub MyParameter {
    my ($self, $parms, $arg1, $arg2) = @_;
    $self->{MyParameter} = {$arg1 => $arg2};
}
```

6.3.5.4 `Apache2::Const::TAKE3`

This is like `Apache2::Const::TAKE1` and `Apache2::Const::TAKE2`, but the directive takes three mandatory arguments. For example:

```
sub MyParameter {
    my ($self, $parms, @args) = @_;
    $self->{MyParameter} = \@args;
}
```

6.3.5.5 `Apache2::Const::TAKE12`

This directive takes one mandatory argument, and a second optional one. This can be used when the second argument has a default value that the user may want to override. For example:

```
sub MyParameter {
    my ($self, $parms, $arg1, $arg2) = @_;
    $self->{MyParameter} = {$arg1 => $arg2||'default'};
}
```

6.3.5.6 `Apache2::Const::TAKE23`

`Apache2::Const::TAKE23` is just like `Apache2::Const::TAKE12`, except now there are two mandatory arguments and an optional third one.

6.3.5.7 `Apache2::Const::TAKE123`

In the `Apache2::Const::TAKE123` variant, the first argument is mandatory and the other two are optional. This is useful for providing defaults for two arguments.

6.3.5.8 `Apache2::Const::ITERATE`

`Apache2::Const::ITERATE` is used when a directive can take an unlimited number of arguments. The callback is invoked repeatedly with a single argument, once for each argument in the list. It's done this way for interoperability with the C API, which doesn't have the flexible argument passing that Perl provides. For example:

```
sub MyParameter {
    my ($self, $parms, $args) = @_;
    push @{ $self->{MyParameter} }, $arg;
}
```

6.3.5.9 `Apache2::Const::ITERATE2`

`Apache2::Const::ITERATE2` is used for directives that take a mandatory first argument followed by a list of arguments to be applied to the first. A familiar example is the `AddType` directive, in which a series of file extensions are applied to a single MIME type:

```
AddType image/jpeg JPG JPEG JFIF jfif
```

Apache will invoke your callback once for each item in the list. Each time Apache runs your callback, it passes the routine the constant first argument (*"image/jpeg"* in the example above), and the current item in the list (*"JPG"* the first time around, *"JPEG"* the second time, and so on). In the example above, the configuration processing routine will be run a total of four times.

For example:

```
sub MyParameter {
    my ($self, $parms, $key, $val) = @_;
    push @{ $self->{MyParameter}{$key} }, $val;
}
```

6.3.5.10 `Apache2::Const::RAW_ARGS`

An *args_how* of `Apache2::Const::RAW_ARGS` instructs Apache to turn off parsing altogether. Instead it simply passes your callback function the line of text following the directive. Leading and trailing whitespace is stripped from the text, but it is not otherwise processed. Your callback can then do whatever processing it wishes to perform.

This callback receives three arguments (similar to `Apache2::Const::TAKE1`), the third of which is a string-valued scalar containing the remaining text following the directive line.

```
sub MyParameter {
    my ($self, $parms, $val) = @_;
    # process $val
}
```

If this mode is used to implement a custom "container" directive, the attribute *req_override* needs to OR `Apache2::Const::EXEC_ON_READ`. e.g.:

```
req_override => Apache2::Const::OR_ALL | Apache2::Const::EXEC_ON_READ,
```

META: complete the details, which are new to 2.0.

To retrieve the contents of a custom "container" directive, use the `Apache2::Directive` object's methods `as_hash` or `as_string`:

```
sub MyParameter {
    my ($self, $parms, $val) = @_;
    my $directive = $parms->directive;
    my $content = $directive->as_string;
}
```

There is one other trick to making configuration containers work. In order to be recognized as a valid directive, the *name* attribute must contain the leading <. This token will be stripped by the code that handles the custom directive callbacks to Apache. For example:

```
name     => '<MyContainer',
```

One other trick that is not required, but can provide some more user friendliness is to provide a handler for the container end token. In our example, the Apache configuration gears will never see the `</MyContainer>` token, as our `Apache2::Const::RAW_ARGS` handler will read in that line and stop reading when it is seen. However in order to catch cases in which the `</MyContainer>` text appears without a preceding `<MyContainer>` opening section, we need to turn the end token into a directive that simply reports an error and exits. For example:

```
{
  name         => '</MyContainer>',
  func         => __PACKAGE__ . "::MyContainer_END",
  errmsg       => 'end of MyContainer without beginning?',
  args_how     => Apache2::Const::NO_ARGS,
  req_override => Apache2::Const::OR_ALL,
},
...
my $EndToken = "</MyContainer>";
sub MyContainer_END {
    die "$EndToken outside a <MyContainer> container\n";
}
```

Now, should the server administrator misplace the container end token, the server will not start, complaining with this error message:

```
Syntax error on line 54 of httpd.conf:
</MyContainer> outside a <MyContainer> container
```

6.3.5.11 `Apache2::Const::FLAG`

When `Apache2::Const::FLAG` is used, Apache will only allow the argument to be one of two values, `On` or `Off`. This string value will be converted into an integer, `1` if the flag is `On`, `0` if it is `Off`. If the configuration argument is anything other than `On` or `Off`, Apache will complain:

```
Syntax error on line 73 of httpd.conf:
MyFlag must be On or Off
```

For example:

```
sub MyFlag {
    my ($self, $parms, $arg) = @_;
    $self->{MyFlag} = $arg; # 1 or 0
}
```

6.3.6 Enabling the New Configuration Directives

As seen in the first example, the module needs to be loaded before the new directives can be used. A special directive `PerlLoadModule` is used for this purpose. For example:

```
PerlLoadModule MyApache2::MyParameters
```

This directive is similar to `PerlModule`, but it require()'s the Perl module immediately, causing an early mod_perl startup. After loading the module it let's Apache know of the new directives and installs the callbacks to be called when the corresponding directives are encountered.

6.3.7 Creating and Merging Configuration Objects

By default mod_perl creates a simple hash to store each container's configuration values, which are populated by directive callbacks, invoked when the *httpd.conf* and the *.htaccess* files are parsed and the corresponding directive are encountered. It's possible to pre-populate the hash entries when the data structure is

created, e.g., to provide reasonable default values for cases where they weren't set in the configuration file. To accomplish that the optional `SERVER_CREATE` and `DIR_CREATE` functions can be supplied.

When a request is mapped to a container, Apache checks if that container has any ancestor containers. If that's the case, it allows mod_perl to call special merging functions, which decide whether configurations in the parent containers should be inherited, appended or overridden in the child container. The custom configuration module can supply custom merging functions `SERVER_MERGE` and `DIR_MERGE`, which can override the default behavior. If these functions are not supplied the following default behavior takes place: The child container inherits its parent configuration, unless it specifies its own and then it overrides its parent configuration.

6.3.7.1 `SERVER_CREATE`

`SERVER_CREATE` is called once for the main server, and once more for each virtual host defined in *httpd.conf*. It's called with two arguments: `$class`, the package name it was created in and `$parms` the already familiar `Apache2::CmdParms` object. The object is expected to return a reference to a blessed hash, which will be used by configuration directives callbacks to set the values assigned in the configuration file. But it's possible to preset some values here:

For example, in the following example the object assigns a default value, which can be overridden during merge if a the directive was used to assign a custom value:

```
package MyApache2::MyParameters;
...
use Apache2::Module ();
use Apache2::CmdParms ();
my @directives = (...);
Apache2::Module::add(__PACKAGE__, \@directives);
...
sub SERVER_CREATE {
    my ($class, $parms) = @_;
    return bless {
        name => __PACKAGE__,
    }, $class;
}
```

To retrieve that value later, you can use:

```
use Apache2::Module ();
...
my $srv_cfg = Apache2::Module::get_config('MyApache2::MyParameters', $s);
print $srv_cfg->{name};
```

If a request is made to a resource inside a virtual host, `$srv_cfg` will contain the object of the virtual host's server. To reach the main server's configuration object use:

```
use Apache2::Module ();
use Apache2::ServerRec ();
use Apache2::ServerUtil ();
...
if ($s->is_virtual) {
    my $base_srv_cfg = Apache2::Module::get_config('MyApache2::MyParameters',
                                                   Apache2::ServerUtil->server);
    print $base_srv_cfg->{name};
}
```

If the function SERVER_CREATE is not supplied by the module, a function that returns a blessed into the current package reference to a hash is used.

6.3.7.2 `SERVER_MERGE`

During the configuration parsing virtual hosts are given a chance to inherit the configuration from the main host, append to or override it. The SERVER_MERGE subroutine can be supplied to override the default behavior, which simply overrides the main server's configuration.

The custom subroutine accepts two arguments: $base, a blessed reference to the main server configuration object, and $add, a blessed reference to a virtual host configuration object. It's expected to return a blessed object after performing the merge of the two objects it has received. Here is the skeleton of a merging function:

```
sub merge {
    my ($base, $add) = @_;
    my %mrg = ();
    # code to merge %$base and %$add
    return bless \%mrg, ref($base);
}
```

The section Merging at Work provides an extensive example of a merging function.

6.3.7.3 `DIR_CREATE`

Similarly to SERVER_CREATE, this optional function, is used to create an object for the directory resource. If the function is not supplied mod_perl will use an empty hash variable as an object.

Just like SERVER_CREATE, it's called once for the main server and one more time for each virtual host. In addition it'll be called once more for each resource (<Location>, <Directory> and others). All this happens during the startup. At request time it might be called for each parsed *.htaccess* file and for each resource defined in it.

The DIR_CREATE function's skeleton is identical to SERVER_CREATE. Here is an example:

```
package MyApache2::MyParameters;
...
use Apache2::Module ();
use Apache2::CmdParms ();
my @directives = (...);
Apache2::Module::add(__PACKAGE__, \@directives);
...
```

```
sub DIR_CREATE {
    my ($class, $parms) = @_;
    return bless {
        foo => 'bar',
    }, $class;
}
```

To retrieve that value later, you can use:

```
use Apache2::Module ();
...
my $dir_cfg = Apache2::Module::get_config('MyApache2::MyParameters',
                                          $s, $r->per_dir_config);
print $dir_cfg->{foo};
```

The only difference in the retrieving the directory configuration object. Here the third argument `$r->per_dir_config` tells `Apache2::Module` to get the directory configuration object.

6.3.7.4 `DIR_MERGE`

Similarly to `SERVER_MERGE`, `DIR_MERGE` merges the ancestor and the current node's directory configuration objects. At the server startup `DIR_MERGE` is called once for each virtual host. At request time, the merging of the objects of resources, their sub-resources and the virtual host/main server merge happens. Apache caches the products of merges, so you may see certain merges happening only once.

The section Merging Order Consequences discusses in detail the merging order.

The section Merging at Work provides an extensive example of a merging function.

6.4 Examples

6.4.1 Merging at Work

In the following example we are going to demonstrate in details how merging works, by showing various merging techniques.

Here is an example Perl module, which, when loaded, installs four custom directives into Apache.

```
#file:MyApache2/CustomDirectives.pm
#------------------------------
package MyApache2::CustomDirectives;

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

use Apache2::CmdParms ();
use Apache2::Module ();
use Apache2::ServerUtil ();

use Apache2::Const -compile => qw(OK);
```

```perl
my @directives = (
    { name => 'MyPlus' },
    { name => 'MyList' },
    { name => 'MyAppend' },
    { name => 'MyOverride' },
);
Apache2::Module::add(__PACKAGE__, \@directives);

sub MyPlus     { set_val('MyPlus',     @_) }
sub MyAppend   { set_val('MyAppend',   @_) }
sub MyOverride { set_val('MyOverride', @_) }
sub MyList     { push_val('MyList',    @_) }

sub DIR_MERGE    { merge(@_) }
sub SERVER_MERGE { merge(@_) }

sub set_val {
    my ($key, $self, $parms, $arg) = @_;
    $self->{$key} = $arg;
    unless ($parms->path) {
        my $srv_cfg = Apache2::Module::get_config($self,
                                                  $parms->server);
        $srv_cfg->{$key} = $arg;
    }
}

sub push_val {

    my ($key, $self, $parms, $arg) = @_;
    push @{ $self->{$key} }, $arg;
    unless ($parms->path) {
        my $srv_cfg = Apache2::Module::get_config($self,
                                                  $parms->server);
        push @{ $srv_cfg->{$key} }, $arg;
    }
}

sub merge {
    my ($base, $add) = @_;

    my %mrg = ();
    for my $key (keys %$base, keys %$add) {
        next if exists $mrg{$key};
        if ($key eq 'MyPlus') {
            $mrg{$key} = ($base->{$key}||0) + ($add->{$key}||0);
        }
        elsif ($key eq 'MyList') {
            push @{ $mrg{$key} },
                @{ $base->{$key}||[] }, @{ $add->{$key}||[] };
        }
        elsif ($key eq 'MyAppend') {
            $mrg{$key} = join " ", grep defined, $base->{$key},
                                                 $add->{$key};
        }
        else {
            # override mode
            $mrg{$key} = $base->{$key} if exists $base->{$key};
```

```
            $mrg{$key} = $add->{$key}  if exists $add->{$key};
        }
    }

    return bless \%mrg, ref($base);
}

1;
__END__
```

It's probably a good idea to specify all the attributes for the `@directives` entries, but here for simplicity we have only assigned to the *name* directive, which is a must. Since all our directives take a single argument, `Apache2::Const::TAKE1`, the default *args_how*, is what we need. We also allow the directives to appear anywhere, so `Apache2::Const::OR_ALL`, the default for *req_override*, is good for us as well.

We use the same callback for the directives `MyPlus`, `MyAppend` and `MyOverride`, which simply assigns the specified value to the hash entry with the key of the same name as the directive.

The `MyList` directive's callback stores the value in the list, a reference to which is stored in the hash, again using the name of the directive as the key. This approach is usually used when the directive is of type `Apache2::Const::ITERATE`, so you may have more than one value of the same kind inside a single container. But in our example we choose to have it of the type `Apache2::Const::TAKE1`.

In both callbacks in addition to storing the value in the current *directory* configuration, if the value is configured in the main server or the virtual host (which is when `$parms->path` is false), we also store the data in the same way in the server configuration object. This is done in order to be able to query the values assigned at the server and virtual host levels, when the request is made to one of the sub-resources. We will show how to access that information in a moment.

Finally we use the same merge function for merging directory and server configuration objects. For the key `MyPlus` (remember we have used the same key name as the name of the directive), the merging function performs, the obvious, summation of the ancestor's merged value (base) and the current resource's value (add). `MyAppend` joins the values into a string, `MyList` joins the lists and finally `MyOverride` (the default) overrides the value with the current one if any. Notice that all four merging methods take into account that the values in the ancestor or the current configuration object might be unset, which is the case when the directive wasn't used by all ancestors or for the current resource.

At the end of the merging, a blessed reference to the merged hash is returned. The reference is blessed into the same class, as the base or the add objects, which is `MyApache2::CustomDirectives` in our example. That hash is used as the merged ancestor's object for a sub-resource of the resource that has just undergone merging.

Next we supply the following *httpd.conf* configuration section, so we can demonstrate the features of this example:

```
PerlLoadModule MyApache2::CustomDirectives
MyPlus 5
MyList      "MainServer"
MyAppend    "MainServer"
```

```
MyOverride "MainServer"
Listen 8081
<VirtualHost _default_:8081>
    MyPlus 2
    MyList      "VHost"
    MyAppend    "VHost"
    MyOverride "VHost"
    <Location /custom_directives_test>
        MyPlus 3
        MyList      "Dir"
        MyAppend    "Dir"
        MyOverride "Dir"
        SetHandler modperl
        PerlResponseHandler MyApache2::CustomDirectivesTest
    </Location>
    <Location /custom_directives_test/subdir>
        MyPlus 1
        MyList      "SubDir"
        MyAppend    "SubDir"
        MyOverride "SubDir"
    </Location>
</VirtualHost>
<Location /custom_directives_test>
    SetHandler modperl
    PerlResponseHandler MyApache2::CustomDirectivesTest
</Location>
```

`PerlLoadModule` loads the Perl module `MyApache2::CustomDirectives` and then installs a new Apache module named `MyApache2::CustomDirectives`, using the callbacks provided by the Perl module. In our example functions `SERVER_CREATE` and `DIR_CREATE` aren't provided, so by default an empty hash will be created to represent the configuration object for the merging functions. If we don't provide merging functions, Apache will simply skip the merging. Though you must provide a callback function for each directive you add.

After installing the new module, we add a virtual host container, containing two resources (which at other times called locations, directories, sections, etc.), one being a sub-resource of the other, plus one another resource which resides in the main server.

We assign different values in all four containers, but the last one. Here we refer to the four containers as *MainServer*, *VHost*, *Dir* and *SubDir*, and use these names as values for all configuration directives, but `MyPlus`, to make it easier understand the outcome of various merging methods and the merging order. In the last container used by `<Location /custom_directives_test>`, we don't specify any directives so we can verify that all the values are inherited from the main server.

For all three resources we are going to use the same response handler, which will dump the values of configuration objects that in its reach. As we will see that different resources will see see certain things identically, while others differently. So here it the handler:

```
#file:MyApache2/CustomDirectivesTest.pm
#---------------------------------
package MyApache2::CustomDirectivesTest;

use strict;
```

```perl
use warnings FATAL => 'all';

use Apache2::RequestRec ();
use Apache2::RequestIO ();
use Apache2::ServerRec ();
use Apache2::ServerUtil ();
use Apache2::Module ();

use Apache2::Const -compile => qw(OK);

sub get_config {
    Apache2::Module::get_config('MyApache2::CustomDirectives', @_);
}

sub handler {
    my ($r) = @_;
    my %secs = ();

    $r->content_type('text/plain');

    my $s = $r->server;
    my $dir_cfg = get_config($s, $r->per_dir_config);
    my $srv_cfg = get_config($s);

    if ($s->is_virtual) {
        $secs{"1: Main Server"}  = get_config(Apache2::ServerUtil->server);
        $secs{"2: Virtual Host"} = $srv_cfg;
        $secs{"3: Location"}     = $dir_cfg;
    }
    else {
        $secs{"1: Main Server"}  = $srv_cfg;
        $secs{"2: Location"}     = $dir_cfg;
     }

    $r->printf("Processing by %s.\n",

        $s->is_virtual ? "virtual host" : "main server");

    for my $sec (sort keys %secs) {
        $r->print("\nSection $sec\n");
        for my $k (sort keys %{ $secs{$sec}||{} }) {
            my $v = exists $secs{$sec}->{$k}
                ? $secs{$sec}->{$k}
                : 'UNSET';
            $v = '[' . (join ", ", map {qq{"$_"}} @$v) . ']'
                if ref($v) eq 'ARRAY';
            $r->printf("%-10s : %s\n", $k, $v);
        }
    }

    return Apache2::Const::OK;
}

1;
__END__
```

The handler is relatively simple. It retrieves the current resource (directory) and the server's configuration objects. If the server is a virtual host, it also retrieves the main server's configuration object. Once these objects are retrieved, we simply dump the contents of these objects, so we can verify that our merging worked correctly. Of course we nicely format the data that we print, taking a special care of array references, which we know is the case with the key *MyList*, but we use a generic code, since Perl tells us when a reference is a list.

It's a show time. First we issue a request to a resource residing in the main server:

```
% GET http://localhost:8002/custom_directives_test/

Processing by main server.

Section 1: Main Server
MyAppend   : MainServer
MyList     : ["MainServer"]
MyOverride : MainServer
MyPlus     : 5

Section 2: Location
MyAppend   : MainServer
MyList     : ["MainServer"]
MyOverride : MainServer
MyPlus     : 5
```

Since we didn't have any directives in that resource's configuration, we confirm that our merge worked correctly and the directory configuration object contains the same data as its ancestor, the main server. In this case the merge has simply inherited the values from its ancestor.

The next request is for the resource residing in the virtual host:

```
% GET http://localhost:8081/custom_directives_test/

Processing by virtual host.

Section 1: Main Server
MyAppend   : MainServer
MyList     : ["MainServer"]
MyOverride : MainServer
MyPlus     : 5

Section 2: Virtual Host
MyAppend   : MainServer VHost
MyList     : ["MainServer", "VHost"]
MyOverride : VHost
MyPlus     : 7

Section 3: Location
MyAppend   : MainServer VHost Dir
MyList     : ["MainServer", "VHost", "Dir"]
MyOverride : Dir
MyPlus     : 10
```

That's where the real fun starts. We can see that the merge worked correctly in the virtual host, and so it did inside the `<Location>` resource. It's easy to see that `MyAppend` and `MyList` are correct, the same for `MyOverride`. For `MyPlus`, we have to work harder and perform some math. Inside the virtual host we have main(5)+vhost(2)=7, and inside the first resource vhost_merged(7)+resource(3)=10.

So far so good, the last request is made to the sub-resource of the resource we have requested previously:

```
% GET http://localhost:8081/custom_directives_test/subdir/

Processing by virtual host.

Section 1: Main Server
MyAppend   : MainServer
MyList     : ["MainServer"]
MyOverride : MainServer
MyPlus     : 5

Section 2: Virtual Host
MyAppend   : MainServer VHost
MyList     : ["MainServer", "VHost"]
MyOverride : VHost
MyPlus     : 7

Section 3: Location
MyAppend   : MainServer VHost Dir SubDir
MyList     : ["MainServer", "VHost", "Dir", "SubDir"]
MyOverride : SubDir
MyPlus     : 11
```

No surprises here. By comparing the configuration sections and the outcome, it's clear that the merging is correct for most directives. The only harder verification is for `MyPlus`, all we need to do is to add 1 to 10, which was the result we saw in the previous request, or to do it from scratch, summing up all the ancestors of this sub-resource: 5+2+3+1=11.

6.4.1.1 Merging Entries Whose Values Are References

When merging entries whose values are references and not scalars, it's important to make a deep copy and not a shallow copy, when the references gets copied. In our example we merged two references to lists, by explicitly extracting the values of each list:

```
push @{ $mrg{$key} },
    @{ $base->{$key}||[] }, @{ $add->{$key}||[] };
```

While seemingly the following snippet is doing the same:

```
$mrg{$key} = $base->{$key};
push @{ $mrg{$key} }, @{ $add->{$key}||[] };
```

it won't do what you expect if the same merge (with the same `$base` and `$add` arguments) is called more than once, which is the case in certain cases. What happens in the latter implementation, is that the first line makes both $mrg{$key} and $base->{$key} point to the same reference. When the second line expands the @{ $mrg{$key} }, it also affects @{ $base->{$key} }. Therefore when the

same merge is called second time, the $base argument is not the same anymore.

Certainly we could workaround this problem in the mod_perl core, by freezing the arguments before the merge call and restoring them afterwards, but this will incur a performance hit. One simply has to remember that the arguments and the references they point to, should stay unmodified through the function call, and then the right code can be supplied.

6.4.1.2 Merging Order Consequences

Sometimes the merging logic can be influenced by the order of merging. It's desirable that the logic will work properly regardless of the merging order.

In Apache 1.3 the merging was happening in the following order:

```
(((base_srv -> vhost) -> section) -> subsection)
```

Whereas as of this writing Apache 2.0 performs:

```
((base_srv -> vhost) -> (section -> subsection))
```

A product of subsections merge (which happen during the request) is merged with the product of the server and virtual host merge (which happens at the startup time). This change was done to improve the configuration merging performance.

So for example, if you implement a directive MyExp which performs the exponential: $mrg=$base**$add, and let's say there directive is used four times in *httpd.conf*:

```
MyExp 5
<VirtualHost _default_:8001>
    MyExp 4
    <Location /section>
        MyExp 3
    </Location>
    <Location /section/subsection>
        MyExp 2
    </Location>
```

The merged configuration for a request *http://localhost:8001/section/subsection* will see:

```
(5 ** 4) ** (3 ** 2)  = 1.45519152283669e+25
```

under Apache 2.0, whereas under Apache 1.3 the result would be:

```
( (5 ** 4) ** 3) ** 2 = 5.96046447753906e+16
```

which is not quite the same.

Chances are that your merging rules work identically, regardless of the merging order. But you should be aware of this behavior.

6.5 Maintainers

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

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

6.6 Authors

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

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

7   Writing mod_perl Handlers and Scripts

7.1 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.

7.2 Prerequisites

7.3 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.

7.4 Techniques

7.4.1 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:

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

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

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

and then register it as:

```
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>
    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`.

7.4.2 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*:

```
#PerlPostConfigRequire startup.pl
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:

```
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:

```
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.

7.5 Goodies Toolkit

7.5.1 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.:

    ```
    mod_perl/2.000002
    ```

    If $ENV{MOD_PERL} doesn't exist, most likely you are not running under mod_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:

    ```
    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.

7.5.2 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:

```
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.

7.5.3 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.

```
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...
}
```

7.6 Code Developing Nuances

7.6.1 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.

7.7 Integration with Apache Issues

In the following sections we discuss the specifics of Apache behavior relevant to mod_perl developers.

7.7.1 HTTP Response Headers

7.7.1.1 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`:

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

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

```
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.

Howerever 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:

```
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:

```
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:

```
local $| = 0;
print "Content-type: text/html\n";
print "My-Header: SomeValue\n";
print "\n";
```

Notice that we `local()`ize 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.

7.7.1.2 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:

```
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.

7.7.2 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.

7.7.3 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 mpms. For example `alarm()` can be used under prefork MPM, but it won't work on any other MPM. Moreover the Apache developers don'tq 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:

```
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.

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/pod/perl58delta.pod#Safe_Signals and
http://search.cpan.org/dist/perl/pod/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 $@ and $@ =~ /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.

7.8 Perl Specifics in the mod_perl Environment

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

7.8.1 *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.

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

7.8.3 *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.

7.8.4 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.

7.8.5 `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.

7.9 `ModPerl::Registry` Handlers Family

7.9.1 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:

```
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.

7.9.2 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:

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

is turned into:

```
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:

```
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`.

7.10 Threads Coding Issues Under mod_perl

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

7.10.1 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 *perlthrtut* (*http://perldoc.perl.org/perlthrtut.html*) manpage.

7.10.2 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 *perlthrtut*, *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*.

7.10.3 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*) manpage.

7.11 Maintainers

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

-

7.12 Authors

-

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

8   Cooking Recipes

8.1 Description

As the chapter's title implies, here you will find ready-to-go mod_perl 2.0 recipes.

If you know a useful recipe, not yet listed here, please post it to the mod_perl mailing list and we will add it here.

8.2 Sending Cookies in REDIRECT Response (ModPerl::Registry)

```
use CGI::Cookie ();
use Apache2::RequestRec ();
use APR::Table ();

use Apache2::Const -compile => qw(REDIRECT);

my $location = "http://example.com/final_destination/";

sub handler {
    my $r = shift;

    my $cookie = CGI::Cookie->new(-name  => 'mod_perl',
                                  -value => 'awesome');

    $r->err_headers_out->add('Set-Cookie' => $cookie);
    $r->headers_out->set(Location => $location);
    $r->status(Apache2::Const::REDIRECT);

    return Apache2::Const::REDIRECT;
}
1;
```

8.3 Sending Cookies in REDIRECT Response (handlers)

```
use CGI::Cookie ();
use Apache2::RequestRec ();
use APR::Table ();

use Apache2::Const -compile => qw(REDIRECT);

my $location = "http://example.com/final_destination/";

sub handler {
    my $r = shift;

    my $cookie = CGI::Cookie->new(-name  => 'mod_perl',
                                  -value => 'awesome');

    $r->err_headers_out->add('Set-Cookie' => $cookie);
    $r->headers_out->set(Location => $location);
```

```
    return Apache2::Const::REDIRECT;
}
1;
```

note that this example differs from the Registry example only in that it does not attempt to fiddle with `$r->status()` - `ModPerl::Registry` uses `$r->status()` as a hack, but handlers should never manipulate the status field in the request record.

8.4 Sending Cookies Using libapreq2

```
use Apache2::Request ();
use Apache2::RequestRec ();
use Apache2::Const -compile => qw(OK);

use APR::Table ();
use APR::Request::Cookie ();

sub handler {
    my $r = shift;
    my $req = $r->pool();

    my $cookie = APR::Request::Cookie->new($req, name => "foo", value => time(), path => '/cookie');

    $r->err_headers_out->add('Set-Cookie' => $cookie->as_string);

    $r->content_type("text/plain");
    $r->print("Testing....");

    return Apache2::Const::OK;
}
```

8.5 Maintainers

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

● Stas Bekman [http://stason.org/]

8.6 Authors

● Stas Bekman [http://stason.org/]

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

9   Porting Apache:: Perl Modules from mod_perl 1.0 to 2.0

9.1 Description

This document describes the various options for porting a mod_perl 1.0 Apache module so that it runs on a Apache 2.0 / mod_perl 2.0 server. It's also helpful to those who start developing mod_perl 2.0 handlers.

Developers who need to port modules using XS code, should also read about porting Apache:: XS modules.

There is also: Porting CPAN modules to mod_perl 2.0 Status.

9.2 Introduction

In the vast majority of cases, a perl Apache module that runs under mod_perl 1.0 will **not** run under mod_perl 2.0 without at least some degree of modification.

Even a very simple module that does not in itself need any changes will at least need the mod_perl 2.0 Apache modules loaded, because in mod_perl 2.0 basic functionality, such as access to the request object and returning an HTTP status, is not found where, or implemented how it used to be in mod_perl 1.0.

Most real-life modules will in fact need to deal with the following changes:

- methods that have moved to a different (new) package

- methods that must be called differently (due to changed prototypes)

- methods that have ceased to exist (functionality provided in some other way)

Do not be alarmed! One way to deal with all of these issues is to load the `Apache2::compat` compatibility layer bundled with mod_perl 2.0. This magic spell will make almost any 1.0 module run under 2.0 without further changes. It is by no means the solution for every case, however, so please read carefully the following discussion of this and other options.

There are three basic options for porting. Let's take a quick look at each one and then discuss each in more detail.

1. **Run the module on 2.0 under `Apache2::compat` with no further changes**

   As we have said mod_perl 2.0 ships with a module, `Apache2::compat`, that provides a complete drop-in compatibility layer for 1.0 modules. `Apache2::compat` does the following:

   - Loads all the mod_perl 2.0 Apache2:: modules

   - Adjusts method calls where the prototype has changed

   - Provides Perl implementation for methods that no longer exist in 2.0

The drawback to using `Apache2::compat` is the performance hit, which can be significant.

Authors of CPAN and other publicly distributed modules should not use `Apache2::compat` since this forces its use in environments where the administrator may have chosen to optimize memory use by making all code run natively under 2.0.

2. **Modify the module to run only under 2.0**

   If you are not interested in providing backwards compatibility with mod_perl 1.0, or if you plan to leave your 1.0 module in place and develop a new version compatible with 2.0, you will need to make changes to your code. How significant or widespread the changes are depends largely of course on your existing code.

   Several sections of this document provide detailed information on how to rewrite your code for mod_perl 2.0 Several tools are provided to help you, and it should be a relatively painless task and one that you only have to do once.

3. **Modify the module so that it runs under both 1.0 and 2.0**

   You need to do this if you want to keep the same version number for your module, or if you distribute your module on CPAN and want to maintain and release just one codebase.

   This is a relatively simple enhancement of option (2) above. The module tests to see which version of mod_perl is in use and then executes the appropriate method call.

The following sections provide more detailed information and instructions for each of these three porting strategies.

9.3 Using `Apache2::porting`

META: to be written. this is a new package which makes chunks of this doc simpler. for now see the `Apache2::porting` manpage.

9.4 Using the `Apache2::compat` Layer

The `Apache2::compat` module tries to hide the changes in API prototypes between version 1.0 and 2.0 of mod_perl, and implements "virtual methods" for the methods and functions that actually no longer exist.

`Apache2::compat` is extremely easy to use. Either add at the very beginning of startup.pl:

```
  use Apache2::compat;
```

or add to httpd.conf:

```
PerlModule Apache2::compat
```

That's all there is to it. Now you can run your 1.0 module unchanged.

Remember, however, that using `Apache2::compat` will make your module run slower. It can create a larger memory footprint than you need and it implements functionality in pure Perl that is provided in much faster XS in mod_perl 1.0 as well as in 2.0. This module was really designed to assist in the transition from 1.0 to 2.0. Generally you will be better off if you port your code to use the mod_perl 2.0 API.

It's also especially important to repeat that `CPAN module developers are requested not to use this module in their code`, since this takes the control over performance away from users.

9.5 Porting a Perl Module to Run under mod_perl 2.0

Note: API changes are listed in the mod_perl 1.0 backward compatibility document.

The following sections will guide you through the steps of porting your modules to mod_perl 2.0.

9.5.1 Using `ModPerl::MethodLookup` to Discover Which mod_perl 2.0 Modules Need to Be Loaded

It would certainly be nice to have our mod_perl 1.0 code run on the mod_perl 2.0 server unmodified. So first of all, try your luck and test the code.

It's almost certain that your code won't work when you try, however, because mod_perl 2.0 splits functionality across many more modules than version 1.0 did, and you have to load these modules before the methods that live in them can be used. So the first step is to figure out which these modules are and `use()` them.

The `ModPerl::MethodLookup` module provided with mod_perl 2.0 allows you to find out which module contains the functionality you are looking for. Simply provide it with the name of the mod_perl 1.0 method that has moved to a new module, and it will tell you what the module is.

For example, let's say we have a mod_perl 1.0 code snippet:

```
$r->content_type('text/plain');
$r->print("Hello cruel world!");
```

If we run this, mod_perl 2.0 will complain that the method `content_type()` can't be found. So we use `ModPerl::MethodLookup` to figure out which module provides this method. We can just run this from the command line:

```
% perl -MModPerl::MethodLookup -e print_method content_type
```

This prints:

```
to use method 'content_type' add:
        use Apache2::RequestRec ();
```

We do what it says and add this `use()` statement to our code, restart our server (unless we're using `Apache2::Reload`), and mod_perl will no longer complain about this particular method.

Since you may need to use this technique quite often you may want to `define an alias`. Once defined the last command line lookup can be accomplished with:

```
% lookup content_type
```

`ModPerl::MethodLookup` also provides helper functions for finding `which methods are defined in a given module`, or `which methods can be invoked on a given object`.

9.5.1.1 Handling Methods Existing In More Than One Package

Some methods exists in several classes. For example this is the case with the `print()` method. We know the drill:

```
% lookup print
```

This prints:

```
There is more than one class with method 'print'
try one of:
      use Apache2::RequestIO ();
      use Apache2::Filter ();
```

So there is more than one package that has this method. Since we know that we call the `print()` method with the `$r` object, it must be the `Apache2::RequestIO` module that we are after. Indeed, loading this module solves the problem.

9.5.1.2 Using **ModPerl::MethodLookup** Programmatically

The issue of picking the right module, when more than one matches, can be resolved when using `ModPerl::MethodLookup` programmatically -- `lookup_method` accepts an object as an optional second argument, which is used if there is more than one module that contains the method in question. `ModPerl::MethodLookup` knows that `Apache2::RequestIO` and and `Apache2::Filter` expect an object of type `Apache2::RequestRec` and type `Apache2::Filter` respectively. So in a program running under mod_perl we can call:

```
ModPerl::MethodLookup::lookup_method('print', $r);
```

Now only one module will be matched.

This functionality can be used in `AUTOLOAD`, for example, although most users will not have a need for this robust of solution.

9.5.1.3 Pre-loading All mod_perl 2.0 Modules

Now if you use a wide range of methods and functions from the mod_perl 1.0 API, the process of finding all the modules that need to be loaded can be quite frustrating. In this case you may find the function `preload_all_modules()` to be the right tool for you. This function preloads **all** mod_perl 2.0 modules, implementing their API in XS.

While useful for testing and development, it is not recommended to use this function in production systems. Before going into production you should remove the call to this function and load only the modules that are used, in order to save memory.

CPAN module developers should **not** be tempted to call this function from their modules, because it prevents the user of their module from optimizing her system's memory usage.

9.5.2 Handling Missing and Modified mod_perl 1.0 Methods and Functions

The mod_perl 2.0 API is modeled even more closely upon the Apache API than was mod_perl version 1.0. Just as the Apache 2.0 API is substantially different from that of Apache 1.0, therefore, the mod_perl 2.0 API is quite different from that of mod_perl 1.0. Unfortunately, this means that certain method calls and functions that were present in mod_perl version 1.0 are missing or modified in mod_perl 2.0.

If mod_perl 2.0 tells you that some method is missing and it can't be found using ModPerl::Method-Lookup, it's most likely because the method doesn't exist in the mod_perl 2.0 API. It's also possible that the method does still exist, but nevertheless it doesn't work, since its usage has changed (e.g. its prototype has changed, or it requires different arguments, etc.).

In either of these cases, refer to the backwards compatibility document for an exhaustive list of API calls that have been modified or removed.

9.5.2.1 Methods that No Longer Exist

Some methods that existed in mod_perl 1.0 simply do not exist anywhere in version 2.0 and you must therefore call a different method o methods to get the functionality you want.

For example, suppose we have a mod_perl 1.0 code snippet:

```
$r->log_reason("Couldn't open the session file: $@");
```

If we try to run this under mod_perl 2.0 it will complain about the call to `log_reason()`. But when we use `ModPerl::MethodLookup` to see which module to load in order to call that method, nothing is found:

```
% perl -MModPerl::MethodLookup -le \
  'print((ModPerl::MethodLookup::lookup_method(shift))[0])' \
  log_reason
```

This prints:

```
don't know anything about method 'log_reason'
```

Looks like we are calling a non-existent method! Our next step is to refer to the backwards compatibility document, wherein we find that as we suspected, the method `log_reason()` no longer exists, and that instead we should use the other standard logging functions provided by the `Apache2::Log` module.

9.5.2.2 Methods Whose Usage Has Been Modified

Some methods still exist, but their usage has been modified, and your code must call them in the new fashion or it will generate an error. Most often the method call requires new or different arguments.

For example, say our mod_perl 1.0 code said:

```
$parsed_uri = Apache2::URI->parse($r, $r->uri);
```

This code causes mod_perl 2.0 to complain first about not being able to load the method `parse()` via the package Apache2::URI. We use the tools described above to discover that the package containing our method has moved and change our code to load and use `APR::URI`:

```
$parsed_uri = APR::URI->parse($r, $r->uri);
```

But we still get an error. It's a little cryptic, but it gets the point across:

```
p is not of type APR::Pool at /path/to/OurModule.pm line 9.
```

What this is telling us is that the method `parse` requires an APR::Pool object as its first argument. (Some methods whose usage has changed emit more helpful error messages prefixed with "Usage: ...") So we change our code to:

```
$parsed_uri = APR::URI->parse($r->pool, $r->uri);
```

and all is well in the world again.

9.5.3 *Requiring a specific mod_perl version.*

To require a module to run only under 2.0, simply add:

```
use mod_perl 2.0;
```

META: In fact, before 2.0 is released you really have to say:

```
use mod_perl 1.99;
```

And you can even require a specific version (for example when a certain API has been added only starting from that version). For example to require version 1.99_08, you can say:

```
use mod_perl 1.9908;
```

9.5.4 Should the Module Name Be Changed?

If it is not possible to make your code run under both mod_perl versions (see below), you will have to maintain two separate versions of your own code. While you can change the name of the module for the new version, it's best to try to preserve the name and use some workarounds.

META: need to discuss this more.

9.5.5 Using `Apache2::compat` As a Tutorial

Even if you have followed the recommendation and eschewed use of the Apache2::compat module, you may find it useful to learn how the API has been changed and how to modify your own code. Simply look at the Apache2::compat source code and see how the functionality should be implemented in mod_perl 2.0.

For example, mod_perl 2.0 doesn't provide the Apache->gensym method. As we can see if we look at the Apache2/compat.pm source, the functionality is now available via the core Perl module Symbol and its gensym() function. (Since mod_perl 2.0 works only with Perl versions 5.6 and higher, and Symbol.pm is included in the core Perl distribution since version 5.6.0, there was no reason to keep providing Apache->gensym.)

So if the original code looked like:

```
my $fh = Apache->gensym;
open $fh, $file or die "Can't open $file: $!";
```

in order to port it mod_perl 2.0 we can write:

```
my $fh = Symbol::gensym;
open $fh, $file or die "Can't open $file: $!";
```

Or we can even skip loading Symbol.pm, since under Perl version 5.6 and higher we can just do:

```
open my $fh, $file or die "Can't open $file: $!";
```

9.5.6 How `Apache::MP3` was Ported to mod_perl 2.0

Apache::MP3 is an elaborate application that uses a lot of mod_perl API. After porting it, I have realized that if you go through the notes or even better try to do it by yourself, referring to the notes only when in trouble, you will most likely be able to port any other mod_perl 1.0 module to run under mod_perl 2.0. So here the log of what I have done while doing the porting.

Please notice that this tutorial should be considered as-is and I'm not claiming that I have got everything polished, so if you still find problems, that's absolutely OK. What's important is to try to learn from the process, so you can attack other modules on your own.

I've started to work with `Apache::MP3` version 3.03 which you can retrieve from Lincoln's CPAN directory: http://search.cpan.org/CPAN/authors/id/L/LD/LDS/Apache-MP3-3.03.tar.gz Even though by the time you'll read this there will be newer versions available it's important that you use the same version as a starting point, since if you don't, the notes below won't make much sense.

9.5.6.1 Preparations

First of all, I scratched most of mine *httpd.conf* and *startup.pl* leaving the bare minimum to get mod_perl started. This is needed to ensure that once I've completed the porting, the module will work correct on other users systems. For example if my *httpd.conf* and *startup.pl* were loading some other modules, which in turn may load modules that a to-be-ported module may rely on, the ported module may work for me, but once released, it may not work for others. It's the best to create a new *httpd.conf* when doing the porting putting only the required bits of configuration into it.

9.5.6.1.1 httpd.conf

Next, I configure the `Apache2::Reload` module, so we don't have to constantly restart the server after we modify `Apache::MP3`. In order to do that add to *httpd.conf*:

```
PerlModule Apache2::Reload
PerlInitHandler Apache2::Reload
PerlSetVar ReloadAll Off
PerlSetVar ReloadModules "ModPerl::* Apache2::*"
PerlSetVar ReloadConstantRedefineWarnings Off
```

You can refer to `the Apache2::Reload manpage` for more information if you aren't familiar with this module. The part:

```
PerlSetVar ReloadAll Off
PerlSetVar ReloadModules "ModPerl::* Apache::*"
```

tells `Apache2::Reload` to monitor only modules in the `ModPerl::` and `Apache::` namespaces. So `Apache::MP3` will be monitored. If your module is named `Foo::Bar`, make sure to include the right pattern for the `ReloadModules` directive. Alternatively simply have:

```
PerlSetVar ReloadAll On
```

which will monitor all modules in `%INC`, but will be a bit slower, as it'll have to `stat(3)` many more modules on each request.

Finally, `Apache::MP3` uses constant subroutines. Because of that you will get lots of warnings every time the module is modified, which I wanted to avoid. I can safely shut those warnings off, since I'm not going to change those constants. Therefore I've used the setting

```
PerlSetVar ReloadConstantRedefineWarnings Off
```

If you do change those constants, refer to the section on `ReloadConstantRedefineWarnings` .

Next I configured `Apache::MP3`. In my case I've followed the `Apache::MP3` documentation, created a directory *mp3/* under the server document root and added the corresponding directives to *httpd.conf*.

Now my *httpd.conf* looked like this:

```
#file:httpd.conf
#---------------
Listen 127.0.0.1:8002
#... standard Apache configuration bits omitted ...

LoadModule perl_module modules/mod_perl.so

PerlSwitches -wT

PerlRequire "/home/httpd/2.0/perl/startup.pl"

PerlModule Apache2::Reload
PerlInitHandler Apache2::Reload
PerlSetVar ReloadAll Off
PerlSetVar ReloadModules "ModPerl::* Apache::*"
PerlSetVar ReloadConstantRedefineWarnings Off

AddType audio/mpeg      mp3 MP3
AddType audio/playlist m3u M3U
AddType audio/x-scpls  pls PLS
AddType application/x-ogg ogg OGG
<Location /mp3>
  SetHandler perl-script
  PerlResponseHandler Apache::MP3
  PerlSetVar PlaylistImage playlist.gif
  PerlSetVar StreamBase http://localhost:8002
  PerlSetVar BaseDir /mp3
</Location>
```

9.5.6.1.2 startup.pl

Since chances are that no mod_perl 1.0 module will work out of box without at least preloading some modules, I've enabled the `Apache2::compat` module. Now my *startup.pl* looked like this:

```
#file:startup.pl
#---------------
use lib qw(/home/httpd/2.0/perl);
use Apache2::compat;
```

9.5.6.1.3 Apache/MP3.pm

Before I even started porting `Apache::MP3`, I've added the warnings pragma to *Apache/MP3.pm* (which wasn't there because mod_perl 1.0 had to work with Perl versions prior to 5.6.0, which is when the `warnings` pragma was added):

```
#file:apache_mp3_prep.diff
--- Apache/MP3.pm.orig 2003-06-03 18:44:21.000000000 +1000
+++ Apache/MP3.pm      2003-06-03 18:44:47.000000000 +1000
@@ -4,2 +4,5 @@
 use strict;
+use warnings;
+no warnings 'redefine';  # XXX: remove when done with porting
+
```

From now on, I'm going to use unified diffs which you can apply using `patch(1)`. Though you may have to refer to its manpage on your platform since the usage flags may vary. On linux I'd apply the above patch as:

```
% cd ~/perl/blead-ithread/lib/site_perl/5.9.0/
% patch -p0 < apache_mp3_prep.diff
```

(note: I've produced the above patch and one more below with `diff -u1`, to avoid the RCS Id tag geting into this document. Normally I produce diffs with `diff -u` which uses the default context of 3.)

assuming that *Apache/MP3.pm* is located in the directory *~/perl/blead-ithread/lib/site_perl/5.9.0/*.

I've enabled the `warnings` pragma even though I did have warnings turned globally in *httpd.conf* with:

```
PerlSwitches -wT
```

it's possible that some badly written module has done:

```
$^W = 0;
```

without localizing the change, affecting other code. Also notice that the *taint* mode was enabled from *httpd.conf*, something that you shouldn't forget to do.

I have also told the `warnings` pragma not to complain about redefined subs via:

```
no warnings 'redefine';  # XXX: remove when done with porting
```

I will remove that code, once porting is completed.

At this point I was ready to start the porting process and I have started the server.

```
% hup2
```

I'm using the following aliases to save typing:

```
alias err2      "tail -f ~/httpd/prefork/logs/error_log"
alias acc2      "tail -f ~/httpd/prefork/logs/access_log"
alias stop2     "~/httpd/prefork/bin/apachectl stop"
alias start2    "~/httpd/prefork/bin/apachectl start"
alias restart2  "~/httpd/prefork/bin/apachectl restart"
alias graceful2 "~/httpd/prefork/bin/apachectl graceful"
alias hup2 "stop2; sleep 3; start2; err2"
```

(I also have a similar set of aliases for mod_perl 1.0)

9.5.6.2 Porting with `Apache2::compat`

I have configured my server to listen on port 8002, so I issue a request http://localhost:8002/mp3/ in one console:

```
% lynx --dump http://localhost:8002/mp3/
```

keeping the *error_log* open in the other:

```
% err2
```

which expands to:

```
% tail -f ~/httpd/prefork/logs/error_log
```

When the request is issued, the *error_log* file tells me:

```
[Thu Jun 05 15:29:45 2003] [error] [client 127.0.0.1]
Usage: Apache2::RequestRec::new(classname, c, base_pool=NULL)
at .../Apache/MP3.pm line 60.
```

Looking at the code:

```
58: sub handler ($$) {
59:   my $class = shift;
60:   my $obj = $class->new(@_) or die "Can't create object: $!";
```

The problem is that handler wasn't invoked as method, but had `$r` passed to it (we can tell because `new()` was invoked as `Apache2::RequestRec::new()`, whereas it should have been `Apache::MP3::new()`. Why *Apache::MP3* wasn't passed as the first argument? I go to the mod_perl 1.0 backward compatibility document and find that method handlers are now marked using the *method* subroutine attribute. So I modify the code:

```
--- Apache/MP3.pm.0    2003-06-05 15:29:19.000000000 +1000
+++ Apache/MP3.pm      2003-06-05 15:38:41.000000000 +1000
@@ -55,7 +55,7 @@
 my $NO  = '^(no|false)$';  # regular expression
 my $YES = '^(yes|true)$';  # regular expression

-sub handler ($$) {
+sub handler : method {
    my $class = shift;
    my $obj = $class->new(@_) or die "Can't create object: $!";
    return $obj->run();
```

and issue the request again (no server restart needed).

This time we get a bunch of looping redirect responses, due to a bug in mod_dir which kicks in to handle the existing dir and messing up with `$r->path_info` keeping it empty at all times. I thought I could work around this by not having the same directory and location setting, e.g. by moving the location to be */songs/* while keeping the physical directory with mp3 files as *$DocumentRoot/mp3/*, but `Apache::MP3`

won't let you do that. So a solution suggested by Justin Erenkrantz is to simply shortcut that piece of code with:

```
--- Apache/MP3.pm.1       2003-06-06 14:50:59.000000000 +1000
+++ Apache/MP3.pm         2003-06-06 14:51:11.000000000 +1000
@@ -253,7 +253,7 @@
   my $self = shift;
   my $dir = shift;

-  unless ($self->r->path_info){
+  unless ($self->r->path_info eq ''){
      #Issue an external redirect if the dir isn't tailed with a '/'
      my $uri = $self->r->uri;
      my $query = $self->r->args;
```

which is equivalent to removing this code, until the bug is fixed (it was still there as of Apache 2.0.46). But the module still works without this code, because if you issue a request to */mp3* (w/o trailing slash) mod_dir, will do the redirect for you, replacing the code that we just removed. In any case this got me past this problem.

Since I have turned on the warnings pragma now I was getting loads of *uninitialized value* warnings from `$r->dir_config()` whose return value were used without checking whether they are defined or not. But you'd get them with mod_perl 1.0 as well, so they are just an example of not-so clean code, not really a relevant obstacle in my pursuit to port this module to mod_perl 2.0. Unfortunately they were cluttering the log file so I had to fix them. I've defined several convenience functions:

```
sub get_config {
    my $val = shift->r->dir_config(shift);
    return defined $val ? $val : '';
}

sub config_yes { shift->get_config(shift) !~ /$YES/oi; }
sub config_no  { shift->get_config(shift) !~ /$NO/oi; }
```

and replaced them as you can see in this patch: *code/apache_mp3_2.diff*:

```
--- Apache/MP3.pm.2       2003-06-06 15:17:22.000000000 +1000
+++ Apache/MP3.pm         2003-06-06 15:16:21.000000000 +1000
@@ -55,6 +55,14 @@
 my $NO  = '^(no|false)$';  # regular expression
 my $YES = '^(yes|true)$';  # regular expression

+sub get_config {
+    my $val = shift->r->dir_config(shift);
+    return defined $val ? $val : '';
+}
+
+sub config_yes { shift->get_config(shift) !~ /$YES/oi; }
+sub config_no  { shift->get_config(shift) !~ /$NO/oi; }
+
 sub handler : method {
   my $class = shift;
   my $obj = $class->new(@_) or die "Can't create object: $!";
@@ -70,7 +78,7 @@
```

```
   my @lang_tags;
   push @lang_tags,split /,\s+/,$r->header_in('Accept-language')
     if $r->header_in('Accept-language');
-  push @lang_tags,$r->dir_config('DefaultLanguage') || 'en-US';
+  push @lang_tags,$new->get_config('DefaultLanguage') || 'en-US';

   $new->{'lh'} ||=
     Apache::MP3::L10N->get_handle(@lang_tags)
@@ -343,7 +351,7 @@
     my $file = $subr->filename;
     my $type = $subr->content_type;
     my $data = $self->fetch_info($file,$type);
-    my $format = $self->r->dir_config('DescriptionFormat');
+    my $format = $self->get_config('DescriptionFormat');
     if ($format) {
       $r->print('#EXTINF:' , $data->{seconds} , ',');
       (my $description = $format) =~ s{%([atfglncrdmsqS%])}
@@ -1204,7 +1212,7 @@
 # get fields to display in list of MP3 files
 sub fields {
   my $self = shift;
-  my @f = split /\W+/,$self->r->dir_config('Fields');
+  my @f = split /\W+/,$self->get_config('Fields');
   return map { lc $_  } @f if @f;           # lower case
   return qw(title artist duration bitrate); # default
 }
@@ -1340,7 +1348,7 @@
 sub get_dir {
   my $self = shift;
   my ($config,$default) = @_;
-  my $dir = $self->r->dir_config($config) || $default;
+  my $dir = $self->get_config($config) || $default;
   return $dir if $dir =~ m!^/!;        # looks like a path
   return $dir if $dir =~ m!^\w+://!;  # looks like a URL
   return $self->default_dir . '/' . $dir;
@@ -1348,22 +1356,22 @@

 # return true if downloads are allowed from this directory
 sub download_ok {
-  shift->r->dir_config('AllowDownload') !~ /$NO/oi;
+  shift->config_no('AllowDownload');
 }

 # return true if streaming is allowed from this directory
 sub stream_ok {
-  shift->r->dir_config('AllowStream') !~ /$NO/oi;
+  shift->config_no('AllowStream');
 }

 # return true if playing locally is allowed
 sub playlocal_ok {
-  shift->r->dir_config('AllowPlayLocally') =~ /$YES/oi;
+  shift->config_yes('AllowPlayLocally');
 }

 # return true if we should check that the client can accomodate streaming
 sub check_stream_client {
```

```
-    shift->r->dir_config('CheckStreamClient') =~ /$YES/oi;
+    shift->config_yes('CheckStreamClient');
  }

  # return true if client can stream
@@ -1378,48 +1386,48 @@

  # whether to read info for each MP3 file (might take a long time)
  sub read_mp3_info {
-    shift->r->dir_config('ReadMP3Info') !~ /$NO/oi;
+    shift->config_no('ReadMP3Info');
  }

  # whether to time out streams
  sub stream_timeout {
-    shift->r->dir_config('StreamTimeout') || 0;
+    shift->get_config('StreamTimeout') || 0;
  }

  # how long an album list is considered so long we should put buttons
  # at the top as well as the bottom
-sub file_list_is_long { shift->r->dir_config('LongList') || 10 }
+sub file_list_is_long { shift->get_config('LongList') || 10 }

  sub home_label {
    my $self = shift;
-    my $home = $self->r->dir_config('HomeLabel') ||
+    my $home = $self->get_config('HomeLabel') ||
        $self->x('Home');
    return lc($home) eq 'hostname' ? $self->r->hostname : $home;
  }

  sub path_style {  # style for the path to parent directories
-    lc(shift->r->dir_config('PathStyle')) || 'staircase';
+    lc(shift->get_config('PathStyle')) || 'staircase';
  }

  # where is our cache directory (if any)
  sub cache_dir    {
    my $self = shift;
-    return unless my $dir  = $self->r->dir_config('CacheDir');
+    return unless my $dir  = $self->get_config('CacheDir');
    return $self->r->server_root_relative($dir);
  }

  # columns to display
-sub subdir_columns {shift->r->dir_config('SubdirColumns') || SUBDIRCOLUMNS   }
-sub playlist_columns {shift->r->dir_config('PlaylistColumns') || PLAYLISTCOLUMNS }
+sub subdir_columns {shift->get_config('SubdirColumns') || SUBDIRCOLUMNS   }
+sub playlist_columns {shift->get_config('PlaylistColumns') || PLAYLISTCOLUMNS }

  # various configuration variables
-sub default_dir   { shift->r->dir_config('BaseDir') || BASE_DIR  }
+sub default_dir   { shift->get_config('BaseDir') || BASE_DIR  }
  sub stylesheet     { shift->get_dir('Stylesheet', STYLESHEET)      }
  sub parent_icon    { shift->get_dir('ParentIcon',PARENTICON)       }
  sub cd_list_icon  {
```

```
   my $self   = shift;
   my $subdir = shift;
-  my $image = $self->r->dir_config('CoverImageSmall') || COVERIMAGESMALL;
+  my $image = $self->get_config('CoverImageSmall') || COVERIMAGESMALL;
   my $directory_specific_icon = $self->r->filename."/$subdir/$image";
   return -e $directory_specific_icon
     ? join ("/",$self->r->uri,escape($subdir),$image)
@@ -1427,7 +1435,7 @@
 }
 sub playlist_icon {
   my $self = shift;
-  my $image = $self->r->dir_config('PlaylistImage') || PLAYLISTIMAGE;
+  my $image = $self->get_config('PlaylistImage') || PLAYLISTIMAGE;
   my $directory_specific_icon = $self->r->filename."/$image";
 warn $directory_specific_icon;
   return -e $directory_specific_icon
@@ -1444,7 +1452,7 @@
 sub cd_icon {
   my $self = shift;
   my $dir = shift;
-  my $coverimg = $self->r->dir_config('CoverImage') || COVERIMAGE;
+  my $coverimg = $self->get_config('CoverImage') || COVERIMAGE;
   if (-e "$dir/$coverimg") {
     $coverimg;
   } else {
@@ -1453,7 +1461,7 @@
 }
 sub missing_comment {
   my $self = shift;
-  my $missing = $self->r->dir_config('MissingComment');
+  my $missing = $self->get_config('MissingComment');
   return if $missing eq 'off';
   $missing = $self->lh->maketext('unknown') unless $missing;
   $missing;
@@ -1464,7 +1472,7 @@
   my $self = shift;
   my $data = shift;
   my $description;
-  my $format = $self->r->dir_config('DescriptionFormat');
+  my $format = $self->get_config('DescriptionFormat');
   if ($format) {
     ($description = $format) =~ s{%([atfglncrdmsqS%])}
       {$1 eq '%' ? '%'
@@ -1495,7 +1503,7 @@
     }
   }

-  if ((my $basename = $r->dir_config('StreamBase')) && !$self->is_localnet()) {
+  if ((my $basename = $self->get_config('StreamBase')) && !$self->is_localnet()) {
     $basename =~ s!http://!http://$auth_info! if $auth_info;
     return $basename;
   }
@@ -1536,7 +1544,7 @@
 sub is_localnet {
   my $self = shift;
   return 1 if $self->is_local;  # d'uh
-  my @local = split /\s+/,$self->r->dir_config('LocalNet') or return;
```

```
+   my @local = split /\s+/,$self->get_config('LocalNet') or return;

    my $remote_ip = $self->r->connection->remote_ip . '.';
    foreach (@local) {
```

, it was 194 lines long so I didn't inline it here, but it was quick to create with a few regexes search-n-replace manipulations in xemacs.

Now I have the browsing of the root *mp3/* directory and its sub-directories working. If I click on *'Fetch'* of a particular song it works too. However if I try to *'Stream'* a song, I get a 500 response with error_log telling me:

```
[Fri Jun 06 15:33:33 2003] [error] [client 127.0.0.1] Bad arg length
for Socket::unpack_sockaddr_in, length is 31, should be 16 at
.../5.9.0/i686-linux-thread-multi/Socket.pm line 370.
```

It would be certainly nice for *Socket.pm* to use `Carp::carp()` instead of `warn()` so we will know where in the `Apache::MP3` code this problem was triggered. However reading the *Socket.pm* manpage reveals that `sockaddr_in()` in the list context is the same as calling an explicit `unpack_sock-addr_in()`, and in the scalar context it's calling `pack_sockaddr_in()`. So I have found `sock-addr_in` was the only *Socket.pm* function used in `Apache::MP3` and I have found this code in the function `is_local()`:

```
my $r = $self->r;
my ($serverport,$serveraddr) = sockaddr_in($r->connection->local_addr);
my ($remoteport,$remoteaddr) = sockaddr_in($r->connection->remote_addr);
return $serveraddr eq $remoteaddr;
```

Since something is wrong with function calls `$r->connection->local_addr` and/or `$r->connection->remote_addr` and I referred to the mod_perl 1.0 backward compatibility document and found the relevant entry on these two functions. Indeed the API have changed. Instead of returning a packed `SOCKADDR_IN` string, Apache now returns an `APR::SockAddr` object, which I can query to get the bits of information I'm interested in. So I applied this patch:

```
--- Apache/MP3.pm.3      2003-06-06 15:36:15.000000000 +1000
+++ Apache/MP3.pm        2003-06-06 15:56:32.000000000 +1000
@@ -1533,10 +1533,9 @@
 # allows the player to fast forward, pause, etc.
 sub is_local {
    my $self = shift;
-   my $r = $self->r;
-   my ($serverport,$serveraddr) = sockaddr_in($r->connection->local_addr);
-   my ($remoteport,$remoteaddr) = sockaddr_in($r->connection->remote_addr);
-   return $serveraddr eq $remoteaddr;
+   my $c = $self->r->connection;
+   require APR::SockAddr;
+   return $c->local_addr->ip_get eq $c->remote_addr->ip_get;
 }

   # Check if the requesting client is on the local network, as defined by
```

And voila, the streaming option now works. I get a warning on *'Use of uninitialized value'* on line 1516 though, but again this is unrelated to the porting issues, just a flow logic problem, which wasn't triggered without the warnings mode turned on. I have fixed it with:

```
--- Apache/MP3.pm.4      2003-06-06 15:57:15.000000000 +1000
+++ Apache/MP3.pm        2003-06-06 16:04:48.000
@@ -1492,7 +1492,7 @@
   my $suppress_auth = shift;
   my $r = $self->r;

-  my $auth_info;
+  my $auth_info = '';
   # the check for auth_name() prevents an anno
   # the apache server log when authentication
   if ($r->auth_name && !$suppress_auth) {
@@ -1509,10 +1509,9 @@
   }

   my $vhost = $r->hostname;
-  unless ($vhost) {
-    $vhost = $r->server->server_hostname;
-    $vhost .= ':' . $r->get_server_port unless
-  }
+  $vhost = $r->server->server_hostname unless
+  $vhost .= ':' . $r->get_server_port unless $
+
   return "http://${auth_info}${vhost}";
 }
```

This completes the first part of the porting. I have tried to use all the visible functions of the interface and everything seemed to work and I haven't got any warnings logged. Certainly I may have missed some usage patterns which may be still problematic. But this is good enough for this tutorial.

9.5.6.3 Getting Rid of the `Apache2::compat` Dependency

The final stage is going to get rid of `Apache2::compat` since this is a CPAN module, which must not load `Apache2::compat` on its own. I'm going to make `Apache::MP3` work with mod_perl 2.0 all by itself.

The first step is to comment out the loading of `Apache2::compat` in *startup.pl*:

```
#file:startup.pl
#---------------
use lib qw(/home/httpd/2.0/perl);
#use Apache2::compat ();
```

9.5.6.4 Ensuring that `Apache2::compat` is not loaded

The second step is to make sure that `Apache2::compat` doesn't get loaded indirectly, through some other module. So I've added this line of code to *Apache/MP3.pm*:

```
--- Apache/MP3.pm.5      2003-06-06 16:17:50.000000000 +1000
+++ Apache/MP3.pm        2003-06-06 16:21:14.000000000 +1000
@@ -3,2 +3,6 @@

+BEGIN {
+    die "Apache2::compat is loaded loaded" if $INC{'Apache2/compat.pm'};
+}
+
 use strict;
```

and indeed, even though I've commented out the loading of `Apache2::compat` from *startup.pl*, this module was still getting loaded. I knew that because the request to */mp3* were failing with the error message:

```
Apache2::compat is loaded loaded at ...
```

There are several ways to find the guilty party, you can `grep(1)` for it in the perl libraries, you can override `CORE::GLOBAL::require()` in *startup.pl*:

```
BEGIN {
  use Carp;
  *CORE::GLOBAL::require = sub {
      Carp::cluck("Apache2::compat is loaded") if $_[0] =~ /compat/;
      CORE::require(@_);
  };
}
```

or you can modify *Apache2/compat.pm* and make it print the calls trace when it gets compiled:

```
--- Apache2/compat.pm.orig   2003-06-03 16:11:07.000000000 +1000
+++ Apache2/compat.pm        2003-06-03 16:11:58.000000000 +1000
@@ -1,5 +1,9 @@
 package Apache2::compat;

+BEGIN {
+    use Carp;
+    Carp::cluck("Apache2::compat is loaded by");
+}
```

I've used this last technique, since it's the safest one to use. Remember that `Apache2::compat` can also be loaded with:

```
    do "Apache2/compat.pm";
```

in which case, neither `grep(1)`'ping for `Apache2::compat`, nor overriding `require()` will do the job.

When I've restarted the server and tried to use `Apache::MP3` (I wasn't preloading it at the server startup since I wanted the server to start normally and cope with problem when it's running), the *error_log* had an entry:

```
Apache2::compat is loaded by at .../Apache2/compat.pm line 6
  Apache2::compat::BEGIN() called at .../Apache2/compat.pm line 8
  eval {...} called at .../Apache2/compat.pm line 8
  require Apache2/compat.pm called at .../5.9.0/CGI.pm line 169
  require CGI.pm called at .../site_perl/5.9.0/Apache/MP3.pm line 8
  Apache::MP3::BEGIN() called at .../Apache2/compat.pm line 8
```

(I've trimmed the whole paths of the libraries and the trace itself, to make it easier to understand.)

We could have used `Carp::carp()` which would have told us only the fact that `Apache2::compat` was loaded by `CGI.pm`, but by using `Carp::cluck()` we've obtained the whole stack backtrace so we also can learn which module has loaded `CGI.pm`.

Here I've learned that I had an old version of `CGI.pm` (2.89) which automatically loaded `Apache2::compat` (which should be never done by CPAN modules). Once I've upgraded `CGI.pm` to version 2.93 and restarted the server, `Apache2::compat` wasn't getting loaded any longer.

9.5.6.5 Installing the `ModPerl::MethodLookup` Helper

Now that `Apache2::compat` is not loaded, I need to deal with two issues: modules that need to be loaded and APIs that have changed.

For the second issue I'll have to refer to the the mod_perl 1.0 backward compatibility document.

But the first issue can be easily worked out using `ModPerl::MethodLookup`. As explained in the section Using `ModPerl::MethodLookup` Programmatically I've added the `AUTOLOAD` code to my *startup.pl* so it'll automatically lookup the packages that I need to load based on the request method and the object type.

So now my *startup.pl* looked like:

```
#file:startup.pl
#---------------
use lib qw(/home/httpd/2.0/perl);

{
  package ModPerl::MethodLookupAuto;
  use ModPerl::MethodLookup;

  use Carp;
  sub handler {

      # look inside mod_perl:: Apache2:: APR:: ModPerl:: excluding DESTROY
      my $skip = '^(?!DESTROY$';
      *UNIVERSAL::AUTOLOAD = sub {
          my $method = $AUTOLOAD;
          return if $method =~ /DESTROY/;
          my ($hint, @modules) =
              ModPerl::MethodLookup::lookup_method($method, @_);
          $hint ||= "Can't find method $AUTOLOAD";
          croak $hint;
      };
```

```
      return 0;
   }
}
1;
```

and I add to my *httpd.conf*:

```
PerlChildInitHandler ModPerl::MethodLookupAuto
```

9.5.6.6 Adjusting the code to run under mod_perl 2

I restart the server and off I go to complete the second porting stage.

The first error that I've received was:

```
[Fri Jun 06 16:28:32 2003] [error] failed to resolve handler 'Apache::MP3'
[Fri Jun 06 16:28:32 2003] [error] [client 127.0.0.1] Can't locate
object method "boot" via package "mod_perl" at .../Apache/Constants.pm
line 8. Compilation failed in require at .../Apache/MP3.pm line 12.
```

I go to line 12 and find the following code:

```
use Apache::Constants qw(:common REDIRECT HTTP_NO_CONTENT
                         DIR_MAGIC_TYPE HTTP_NOT_MODIFIED);
```

Notice that I did have mod_perl 1.0 installed, so the `Apache::Constant` module from mod_perl 1.0 couldn't find the `boot()` method which doesn't exist in mod_perl 2.0. If you don't have mod_perl 1.0 installed the error would simply say, that it can't find *Apache/Constants.pm* in `@INC`. In any case, we are going to replace this code with mod_perl 2.0 equivalent:

```
--- Apache/MP3.pm.6     2003-06-06 16:33:05.000000000 +1000
+++ Apache/MP3.pm       2003-06-06 17:03:43.000000000 +1000
@@ -9,7 +9,9 @@
 use warnings;
 no warnings 'redefine';  # XXX: remove when done with porting

-use Apache::Constants qw(:common REDIRECT HTTP_NO_CONTENT DIR_MAGIC_TYPE HTTP_NOT_MODIFIED);
+use Apache2::Const -compile => qw(:common REDIRECT HTTP_NO_CONTENT
+                                  DIR_MAGIC_TYPE HTTP_NOT_MODIFIED);
+
 use Apache::MP3::L10N;
 use IO::File;
 use Socket 'sockaddr_in';
```

and I also had to adjust the constants, since what used to be `OK`, now has to be `Apache2::Const::OK`, mainly because in mod_perl 2.0 there is an enormous amount of constants (coming from Apache and APR) and most of them are grouped in `Apache2::` or `APR::` namespaces. The `Apache2::Const` and `APR::Const` manpage provide more information on available constants.

This search and replace accomplished the job:

```
  % perl -pi -e 's/return\s(OK|DECLINED|FORBIDDEN| \
    REDIRECT|HTTP_NO_CONTENT|DIR_MAGIC_TYPE| \
    HTTP_NOT_MODIFIED)/return Apache2::$1/xg' Apache/MP3.pm
```

As you can see the regex explicitly lists all constants that were used in `Apache::MP3`. Your situation may vary. Here is the patch: *code/apache_mp3_7.diff*:

```
--- Apache/MP3.pm.7     2003-06-06 17:04:27.000000000 +1000
+++ Apache/MP3.pm       2003-06-06 17:13:26.000000000 +1000
@@ -129,7 +129,7 @@
   my $self = shift;

   $self->r->send_http_header( $self->html_content_type );
-   return OK if $self->r->header_only;
+   return Apache::OK if $self->r->header_only;

   print start_html(
                    -lang => $self->lh->language_tag,
@@ -246,20 +246,20 @@
      $self->send_playlist(\@matches);
    }

-    return OK;
+    return Apache::OK;
  }

  # this is called to generate a playlist for selected files
  if (param('Play Selected')) {
-    return HTTP_NO_CONTENT unless my @files = param('file');
+    return Apache::HTTP_NO_CONTENT unless my @files = param('file');
    my $uri = dirname($r->uri);
    $self->send_playlist([map { "$uri/$_" } @files]);
-    return OK;
+    return Apache::OK;
  }

  # otherwise don't know how to deal with this
  $self->r->log_reason('Invalid parameters -- possible attempt to circumvent checks.');
-  return FORBIDDEN;
+  return Apache::FORBIDDEN;
 }

 # this generates the top-level directory listing
@@ -273,7 +273,7 @@
    my $query = $self->r->args;
    $query = "?" . $query if defined $query;
    $self->r->header_out(Location => "$uri/$query");
-    return REDIRECT;
+    return Apache::REDIRECT;
  }

  return $self->list_directory($dir);
@@ -289,9 +289,9 @@

  if ($is_audio && !$self->download_ok) {
    $self->r->log_reason('File downloading is forbidden');
-    return FORBIDDEN;
+    return Apache::FORBIDDEN;
```

```
     } else {
-      return DECLINED;  # allow Apache to do its standard thing
+      return Apache::DECLINED;  # allow Apache to do its standard thing
     }

 }
@@ -302,17 +302,17 @@
    my $self = shift;
    my $r = $self->r;

-    return DECLINED unless -e $r->filename;  # should be $r->finfo
+    return Apache::DECLINED unless -e $r->filename;  # should be $r->finfo

    unless ($self->stream_ok) {
      $r->log_reason('AllowStream forbidden');
-      return FORBIDDEN;
+      return Apache::FORBIDDEN;
    }

    if ($self->check_stream_client and !$self->is_stream_client) {
      my $useragent = $r->header_in('User-Agent');
      $r->log_reason("CheckStreamClient is true and $useragent is not a streaming client");
-      return FORBIDDEN;
+      return Apache::FORBIDDEN;
    }

    return $self->send_stream($r->filename,$r->uri);
@@ -322,12 +322,12 @@
 sub send_playlist {
   my $self = shift;
   my ($urls,$shuffle) = @_;
-   return HTTP_NO_CONTENT unless @$urls;
+   return Apache::HTTP_NO_CONTENT unless @$urls;
   my $r = $self->r;
   my $base = $self->stream_base;

   $r->send_http_header('audio/mpegurl');
-   return OK if $r->header_only;
+   return Apache::OK if $r->header_only;

   # local user
   my $local = $self->playlocal_ok && $self->is_local;
@@ -377,7 +377,7 @@
       $r->print ("$base$_?$stream_parms$CRLF");
     }
   }
-   return OK;
+   return Apache::OK;
 }

 sub stream_parms {
@@ -468,7 +468,7 @@
   my $self = shift;
   my $dir  = shift;

-   return DECLINED unless -d $dir;
+   return Apache::DECLINED unless -d $dir;

   my $last_modified = (stat(_))[9];
```

```
@@ -478,15 +478,15 @@
    my ($time, $ver) = $check =~ /^([a-f0-9]+)-([0-9.]+)$/;

    if ($check eq '*' or (hex($time) == $last_modified and $ver == $VERSION)) {
-      return HTTP_NOT_MODIFIED;
+      return Apache::HTTP_NOT_MODIFIED;
    }
  }

- return DECLINED unless my ($directories,$mp3s,$playlists,$txtfiles)
+ return Apache::DECLINED unless my ($directories,$mp3s,$playlists,$txtfiles)
    = $self->read_directory($dir);

  $self->r->send_http_header( $self->html_content_type );
- return OK if $self->r->header_only;
+ return Apache::OK if $self->r->header_only;

  $self->page_top($dir);
  $self->directory_top($dir);
@@ -514,7 +514,7 @@
  print hr                         unless %$mp3s;
  print "\n\n";
  $self->directory_bottom($dir);
- return OK;
+ return Apache::OK;
 }

 # print the HTML at the top of the page
@@ -1268,8 +1268,8 @@

  my $mime = $r->content_type;
  my $info = $self->fetch_info($file,$mime);
- return DECLINED unless $info;  # not a legit mp3 file?
- my $fh = $self->open_file($file) || return DECLINED;
+ return Apache::DECLINED unless $info;  # not a legit mp3 file?
+ my $fh = $self->open_file($file) || return Apache::DECLINED;
  binmode($fh);  # to prevent DOS text-mode foolishness

  my $size = -s $file;
@@ -1317,7 +1317,7 @@
  $r->print("Content-Length: $size$CRLF");
  $r->print("Content-Type: $mime$CRLF");
  $r->print("$CRLF");
- return OK if $r->header_only;
+ return Apache::OK if $r->header_only;

  if (my $timeout = $self->stream_timeout) {
    my $seconds  = $info->{seconds};
@@ -1330,12 +1330,12 @@
      $bytes -= $b;
      $r->print($data);
    }
-    return OK;
+    return Apache::OK;
  }

  # we get here for untimed transmits
  $r->send_fd($fh);
```

```
-  return OK;
+  return Apache::OK;
 }

 # called to open the MP3 file
```

.

I had to manually fix the `DIR_MAGIC_TYPE` constant which didn't fit the regex pattern:

```
--- Apache/MP3.pm.8     2003-06-06 17:24:33.000000000 +1000
+++ Apache/MP3.pm       2003-06-06 17:26:29.000000000 +1000
@@ -1055,7 +1055,7 @@

     my $mime = $self->r->lookup_file("$dir/$d")->content_type;

-     push(@directories,$d) if !$seen{$d}++ && $mime eq DIR_MAGIC_TYPE;
+     push(@directories,$d) if !$seen{$d}++ && $mime eq Apache2::Const::DIR_MAGIC_TYPE;

     # .m3u files should be configured as audio/playlist MIME types in your apache .conf file
     push(@playlists,$d) if $mime =~ m!^audio/(playlist|x-mpegurl|mpegurl|x-scpls)$!;
```

And I move on, the next error is:

```
[Fri Jun 06 17:28:00 2003] [error] [client 127.0.0.1]
Can't locate object method "header_in" via package
"Apache2::RequestRec" at .../Apache2/MP3.pm line 85.
```

The porting document quickly reveals me that `header_in()` and its brothers `header_out()` and `err_header_out()` are R.I.P. and that I have to use the corresponding functions `headers_in()`, `headers_out()` and `err_headers_out()` which are available in mod_perl 1.0 API as well.

So I adjust the code to use the new API:

```
% perl -pi -e 's|header_in\((.*?)\)|headers_in->{$1}|g' Apache/MP3.pm
% perl -pi -e 's|header_out\((.*?)\s*=>\s*(.*?)\);|headers_out->{$1} = $2;|g' Apache/MP3.pm
```

which results in this patch: *code/apache_mp3_9.diff*:

```
--- Apache/MP3.pm.9     2003-06-06 17:27:45.000000000 +1000
+++ Apache/MP3.pm       2003-06-06 17:55:14.000000000 +1000
@@ -82,8 +82,8 @@
   $new->{'r'} ||= $r if $r;

   my @lang_tags;
- push @lang_tags,split /,\s+/,$r->header_in('Accept-language')
-    if $r->header_in('Accept-language');
+ push @lang_tags,split /,\s+/,$r->headers_in->{'Accept-language'}
+    if $r->headers_in->{'Accept-language'};
   push @lang_tags,$new->get_config('DefaultLanguage') || 'en-US';

   $new->{'lh'} ||=
@@ -272,7 +272,7 @@
     my $uri = $self->r->uri;
     my $query = $self->r->args;
     $query = "?" . $query if defined $query;
-     $self->r->header_out(Location => "$uri/$query");
```

```
+    $self->r->headers_out->{Location} = "$uri/$query";
     return Apache::REDIRECT;
    }

@@ -310,7 +310,7 @@
    }

    if ($self->check_stream_client and !$self->is_stream_client) {
-      my $useragent = $r->header_in('User-Agent');
+      my $useragent = $r->headers_in->{'User-Agent'};
      $r->log_reason("CheckStreamClient is true and $useragent is not a streaming client");
      return Apache::FORBIDDEN;
    }
@@ -472,9 +472,9 @@

    my $last_modified = (stat(_))[9];

-    $self->r->header_out('ETag' => sprintf("%lx-%s", $last_modified, $VERSION));
+    $self->r->headers_out->{'ETag'} = sprintf("%lx-%s", $last_modified, $VERSION);

-    if (my $check = $self->r->header_in("If-None-Match")) {
+    if (my $check = $self->r->headers_in->{"If-None-Match"}) {
      my ($time, $ver) = $check =~ /^([a-f0-9]+)-([0-9.]+)$/;

      if ($check eq '*' or (hex($time) == $last_modified and $ver == $VERSION)) {
@@ -1283,8 +1283,8 @@
    my $genre       = $info->{genre} || $self->lh->maketext('unknown');

    my $range = 0;
-    $r->header_in("Range")
-      and $r->header_in("Range") =~ m/bytes=(\d+)/
+    $r->headers_in->{"Range"}
+      and $r->headers_in->{"Range"} =~ m/bytes=(\d+)/
      and $range = $1
      and seek($fh,$range,0);

@@ -1383,11 +1383,11 @@
 # return true if client can stream
 sub is_stream_client {
    my $r = shift->r;
- $r->header_in('Icy-MetaData')   # winamp/xmms
-   || $r->header_in('Bandwidth')   # realplayer
-     || $r->header_in('Accept') =~ m!\baudio/mpeg\b!  # mpg123 and others
-       || $r->header_in('User-Agent') =~ m!^NSPlayer/!  # Microsoft media player
-         || $r->header_in('User-Agent') =~ m!^xmms/!;
+ $r->headers_in->{'Icy-MetaData'}   # winamp/xmms
+   || $r->headers_in->{'Bandwidth'}   # realplayer
+     || $r->headers_in->{'Accept'} =~ m!\baudio/mpeg\b!  # mpg123 and others
+       || $r->headers_in->{'User-Agent'} =~ m!^NSPlayer/!  # Microsoft media player
+         || $r->headers_in->{'User-Agent'} =~ m!^xmms/!;
 }

 # whether to read info for each MP3 file (might take a long time)
```

.

On the next error `ModPerl::MethodLookup`'s `AUTOLOAD` kicks in. Instead of complaining:

```
[Fri Jun 06 18:35:53 2003] [error] [client 127.0.0.1]
Can't locate object method "FETCH" via package "APR::Table"
at .../Apache/MP3.pm line 85.
```

I now get:

```
[Fri Jun 06 18:36:35 2003] [error] [client 127.0.0.1]
to use method 'FETCH' add:
      use APR::Table ();
at .../Apache/MP3.pm line 85
```

So I follow the suggestion and load `APR::Table()`:

```
--- Apache/MP3.pm.10    2003-06-06 17:57:54.000000000 +1000
+++ Apache/MP3.pm       2003-06-06 18:37:33.000000000 +1000
@@ -9,6 +9,8 @@
 use warnings;
 no warnings 'redefine';  # XXX: remove when done with porting

+use APR::Table ();
+
 use Apache2::Const -compile => qw(:common REDIRECT HTTP_NO_CONTENT
                                   DIR_MAGIC_TYPE HTTP_NOT_MODIFIED);
```

I continue issuing the request and adding the missing modules again and again till I get no more complaints. During this process I've added the following modules:

```
--- Apache/MP3.pm.11    2003-06-06 18:38:47.000000000 +1000
+++ Apache/MP3.pm       2003-06-06 18:39:10.000000000 +1000
@@ -9,6 +9,14 @@
 use warnings;
 no warnings 'redefine';  # XXX: remove when done with porting

+use Apache2::Connection ();
+use Apache2::SubRequest ();
+use Apache2::Access ();
+use Apache2::RequestIO ();
+use Apache2::RequestUtil ();
+use Apache2::RequestRec ();
+use Apache2::ServerUtil ();
+use Apache2::Log;
 use APR::Table ();

 use Apache2::Const -compile => qw(:common REDIRECT HTTP_NO_CONTENT
```

The AUTOLOAD code helped me to trace the modules that contain the existing APIs, however I still have to deal with APIs that no longer exist. Rightfully the helper code says that it doesn't know which module defines the method: `send_http_header()` because it no longer exists in Apache 2.0 vocabulary:

```
[Fri Jun 06 18:40:34 2003] [error] [client 127.0.0.1]
Don't know anything about method 'send_http_header'
at .../Apache/MP3.pm line 498
```

So I go back to the porting document and find the relevant entry. In 2.0 lingo, we just need to set the `content_type()`:

```
--- Apache/MP3.pm.12     2003-06-06 18:43:42.000000000 +1000
+++ Apache/MP3.pm        2003-06-06 18:51:23.000000000 +1000
@@ -138,7 +138,7 @@
 sub help_screen {
   my $self = shift;

-  $self->r->send_http_header( $self->html_content_type );
+  $self->r->content_type( $self->html_content_type );
   return Apache2::Const::OK if $self->r->header_only;

   print start_html(
@@ -336,7 +336,7 @@
   my $r = $self->r;
   my $base = $self->stream_base;

-  $r->send_http_header('audio/mpegurl');
+  $r->content_type('audio/mpegurl');
   return Apache2::Const::OK if $r->header_only;

   # local user
@@ -495,7 +495,7 @@
   return Apache2::Const::DECLINED unless my ($directories,$mp3s,$playlists,$txtfiles)
     = $self->read_directory($dir);

-  $self->r->send_http_header( $self->html_content_type );
+  $self->r->content_type( $self->html_content_type );
   return Apache2::Const::OK if $self->r->header_only;

   $self->page_top($dir);
```

also I've noticed that there was this code:

```
return Apache2::Const::OK if $self->r->header_only;
```

This technique is no longer needed in 2.0, since Apache 2.0 automatically discards the body if the request is of type `HEAD` -- the handler should still deliver the whole body, which helps to calculate the content-length if this is relevant to play nicer with proxies. So you may decide not to make a special case for `HEAD` requests.

At this point I was able to browse the directories and play files via most options without relying on `Apache2::compat`.

There were a few other APIs that I had to fix in the same way, while trying to use the application, looking at the *error_log* referring to the porting document and applying the suggested fixes. I'll make sure to send all these fixes to Lincoln Stein, so the new versions will work correctly with mod_perl 2.0. I also had to fix other `Apache::MP3::` files, which come as a part of the `Apache-MP3` distribution, pretty much using the same techniques explained here. A few extra fixes of interest in `Apache::MP3` were:

- **`send_fd()`**

  As of this writing we don't have this function in the core, because Apache 2.0 doesn't have it (it's in `Apache2::compat` but implemented in a slow way). However we may provide one in the future. Currently one can use the function `sendfile()` which requires a filename as an argument and not the file descriptor. So I have fixed the code:

  ```
  -     if($r->request($r->uri)->content_type eq 'audio/x-scpls'){
  -        open(FILE,$r->filename) || return 404;
  -        $r->send_fd(\*FILE);
  -        close(FILE);
  +
  +     if($r->content_type eq 'audio/x-scpls'){
  +        $r->sendfile($r->filename) || return Apache2::Const::NOT_FOUND;
  ```

- **`log_reason`**

  `log_reason` is now `log_error`:

  ```
  - $self->r->log_reason('Invalid parameters -- possible attempt to circumvent checks.');
  + $r->log_error('Invalid parameters -- possible attempt to circumvent checks.')
  ;
  ```

I have found the porting process to be quite interesting, especially since I have found several bugs in Apache 2.0 and documented a few undocumented API changes. It was also fun, because I've got to listen to mp3 files when I did things right, and was getting silence in my headphones and a visual irritation in the form of *error_log* messages when I didn't ;)

9.6 Porting a Module to Run under both mod_perl 2.0 and mod_perl 1.0

Sometimes code needs to work with both mod_perl versions. For example this is the case with CPAN module developers who wish to continue to maintain a single code base, rather than supplying two separate implementations.

9.6.1 Making Code Conditional on Running mod_perl Version

In this case you can test for which version of mod_perl your code is running under and act appropriately.

To continue our example above, let's say we want to support opening a filehandle in both mod_perl 2.0 and mod_perl 1.0. Our code can make use of the environment variable `$ENV{MOD_PERL_API_VERSION}`

```
use mod_perl;
use constant MP2 => ( exists $ENV{MOD_PERL_API_VERSION} and
                      $ENV{MOD_PERL_API_VERSION} >= 2 );
# ...
require Symbol if MP2;
# ...

my $fh = MP2 ? Symbol::gensym : Apache->gensym;
open $fh, $file or die "Can't open $file: $!";
```

Some modules, like `CGI.pm` may work under mod_perl and without it, and will want to use the mod_perl 1.0 API if that's available, or mod_perl 2.0 API otherwise. So the following idiom could be used for this purpose.

```
use constant MP_GEN => $ENV{MOD_PERL}
    ? { ( exists $ENV{MOD_PERL_API_VERSION} and
          $ENV{MOD_PERL_API_VERSION} >= 2 ) ? 2 : 1 }
    : 0;
```

It sets the constant `MP_GEN` to 0 if mod_perl is not available, to 1 if running under mod_perl 1.0 and 2 for mod_perl 2.0.

Here's another way to find out the mod_perl version. In the server configuration file you can use a special configuration "define" symbol `MODPERL2`, which is magically enabled internally, as if the server had been started with `-DMODPERL2`.

```
# in httpd.conf
<IfDefine MODPERL2>
    # 2.0 configuration
</IfDefine>
<IfDefine !MODPERL2>
    # else
</IfDefine>
```

From within Perl code this can be tested with `Apache2::exists_config_define()`. For example, we can use this method to decide whether or not to call `$r->send_http_header()`, which no longer exists in mod_perl 2.0:

```
sub handler {
    my $r = shift;
    $r->content_type('text/html');
    $r->send_http_header() unless Apache2::exists_config_define("MODPERL2");
    ...
}
```

Relevant links to other places in the porting documents:

- mod_perl 1.0 and 2.0 Constants Coexistence

9.6.2 *Method Handlers*

Method handlers in mod_perl are declared using the *'method'* attribute. However if you want to have the same code base for mod_perl 1.0 and 2.0 applications, whose handler has to be a method, you will need to do the following trick:

```
sub handler_mp1 ($$)    { ... }
sub handler_mp2 : method { ... }
*handler = MP2 ? \&handler_mp2 : \&handler_mp1;
```

Note that this requires at least Perl 5.6.0, the *:method* attribute is not supported by older Perl versions, which will fail to compile such code.

Here are two complete examples. The first example implements `MyApache2::Method` which has a single method that works for both mod_perl generations:

The configuration:

```
PerlModule MyApache2::Method
<Location /method>
    SetHandler perl-script
    PerlHandler MyApache2::Method->handler
</Location>
```

The code:

```
#file:MyApache2/Method.pm
package MyApache2::Method;

# PerlModule MyApache2::Method
# <Location /method>
#     SetHandler perl-script
#     PerlHandler MyApache2::Method->handler
#  </Location>

use strict;
use warnings;

use mod_perl;
use constant MP2 => ( exists $ENV{MOD_PERL_API_VERSION} and
                      $ENV{MOD_PERL_API_VERSION >= 2 );

BEGIN {
    if (MP2) {
        require Apache2::RequestRec;
        require Apache2::RequestIO;
        require Apache2::Const;
        Apache2::Const->import(-compile => 'OK');
    }
    else {
        require Apache;
        require Apache::Constants;
        Apache::Constants->import('OK');
    }
```

```
   }

   sub handler_mp1 ($$)      { &run }
   sub handler_mp2 : method { &run }
   *handler = MP2 ? \&handler_mp2 : \&handler_mp1;

   sub run {
       my ($class, $r) = @_;
       MP2 ? $r->content_type('text/plain')
           : $r->send_http_header('text/plain');
       print "$class was called\n";
       return MP2 ? Apache2::Const::OK : Apache::Constants::OK;

   }
```

Here are two complete examples. The second example implements `MyApache2::Method2`, which is very similar to `MyApache2::Method`, but uses separate methods for mod_perl 1.0 and 2.0 servers.

The configuration is the same:

```
   PerlModule MyApache2::Method2
   <Location /method2>
       SetHandler perl-script
       PerlHandler MyApache2::Method2->handler
   </Location>
```

The code:

```
   #file:MyApache2/Method2.pm
   package MyApache2::Method2;

   # PerlModule MyApache2::Method
   # <Location /method>
   #      SetHandler perl-script
   #      PerlHandler MyApache2::Method->handler
   #  </Location>

   use strict;
   use warnings;

   use mod_perl;
   use constant MP2 => ( exists $ENV{MOD_PERL_API_VERSION} and
                         $ENV{MOD_PERL_API_VERSION >= 2 );

   BEGIN {
       warn "running $ENV{MOD_PERL_API_VERSION}\n";
       if (MP2) {
           require Apache2::RequestRec;
           require Apache2::RequestIO;
           require Apache2::Const;
           Apache2::Const->import(-compile => 'OK');
       }
       else {
           require Apache;
           require Apache::Constants;
           Apache::Constants->import('OK');
```

```
       }
  }

  sub handler_mp1 ($$)      { &mp1 }
  sub handler_mp2 : method { &mp2 }

  *handler = MP2 ? \&handler_mp2 : \&handler_mp1;

  sub mp1 {
      my ($class, $r) = @_;
      $r->send_http_header('text/plain');
      $r->print("mp1: $class was called\n");

      return Apache::Constants::OK();
  }

  sub mp2 {
      my ($class, $r) = @_;
      $r->content_type('text/plain');
      $r->print("mp2: $class was called\n");
      return Apache2::Const::OK();
  }
```

Assuming that mod_perl 1.0 is listening on port 8001 and mod_perl 2.0 on 8002, we get the following results:

```
  % lynx --source http://localhost:8001/method
  MyApache2::Method was called

  % lynx --source http://localhost:8001/method2
  mp1: MyApache2::Method2 was called

  % lynx --source http://localhost:8002/method
  MyApache2::Method was called

  % lynx --source http://localhost:8002/method2
  mp2: MyApache2::Method2 was called
```

9.7 The Conflict of mp1 vs mp2 vs mp22 vs ... vs mpNN

META: should something be said here?

9.7.1 Distributors

Distributors should mark the different generations of mod_perl core as conflicting, so only one version can be installed using the binary package. Users requiring more than one installation should do a manual install.

In order to have any of the 3rd party modperl modules installed users need to have the correct modperl package installed. So there is no need to mark the 3rd party modules as conflicting, since their most important prerequisite (the modperl-core) is already handling that.

Of course packagers can decide to make the two generation packages as non-conflicting, by building all mp2 core and 3rd party modules into *Apache2/* subdir, in which case the two will always co-exist. But this is not the most logical approach since 99% of users will want only one generation of mod_perl core and 3rd party modules.

9.8 Maintainers

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

Stas Bekman [http://stason.org/]

9.9 Authors

- Nick Tonkin <nick (at) tonkinresolutions.com>

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

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

10   A Reference to mod_perl 1.0 to mod_perl 2.0 Migration.

10.1 Description

This chapter is a reference for porting code and configuration files from mod_perl 1.0 to mod_perl 2.0.

To learn about the porting process you should first read about porting Perl modules (and may be about porting XS modules).

As it will be explained in details later, loading `Apache2::compat` at the server startup, should make the code running properly under 1.0 work under mod_perl 2.0. If you want to port your code to mod_perl 2.0 or writing from scratch and not concerned about backwards compatibility, this document explains what has changed compared to mod_perl 1.0.

Several configuration directives were changed, renamed or removed. Several APIs have changed, renamed, removed, or moved to new packages. Certain functions while staying exactly the same as in mod_perl 1.0, now reside in different packages. Before using them you need to find out those packages and load them.

You should be able to find the destiny of the functions that you cannot find any more or which behave differently now under the package names the functions belong in mod_perl 1.0.

10.2 Configuration Files Porting

To migrate the configuration files to the mod_perl 2.0 syntax, you may need to do certain adjustments. Several configuration directives are deprecated in 2.0, but still available for backwards compatibility with mod_perl 1.0 unless 2.0 was built with `MP_COMPAT_1X=0`. If you don't need the backwards compatibility consider using the directives that have replaced them.

10.2.1 `PerlHandler`

`PerlHandler` was replaced with `PerlResponseHandler`.

10.2.2 `PerlScript`

`PerlScript` was replaced with `PerlRequire`. `PerlRequire` is available in mod_perl 1.0, since 1997.

10.2.3 `PerlSendHeader`

`PerlSendHeader` was replaced with `PerlOptions +/-ParseHeaders` directive.

```
PerlSendHeader On  => PerlOptions +ParseHeaders
PerlSendHeader Off => PerlOptions -ParseHeaders
```

10.2.4 `PerlSetupEnv`

`PerlSetupEnv` was replaced with `PerlOptions +/-SetupEnv` directive.

```
PerlSetupEnv On  => PerlOptions +SetupEnv
PerlSetupEnv Off => PerlOptions -SetupEnv
```

10.2.5 `PerlTaintCheck`

The taint mode now can be turned on with `PerlSwitches`:

```
PerlSwitches -T
```

As with standard Perl, by default the taint mode is disabled and once enabled cannot be turned off inside the code.

10.2.6 `PerlWarn`

Warnings now can be enabled globally with `PerlSwitches`:

```
PerlSwitches -w
```

10.2.7 `PerlFreshRestart`

`PerlFreshRestart` is a mod_perl 1.0 legacy and doesn't exist in mod_perl 2.0. A full teardown and startup of interpreters is done on restart.

If you need to use the same *httpd.conf* for 1.0 and 2.0, use:

```
<IfDefine !MODPERL2>
    PerlFreshRestart
</IfDefine>
```

10.2.8 `$Apache::Server::StrictPerlSections`

In mod_perl 2.0, `<Perl>` sections errors are now always fatal. Any error in them will cause an immediate server startup abort, dumping the error to STDERR. To avoid this, `eval {}` can be used to trap errors and ignore them. In mod_perl 1.0, `strict` was somewhat of a misnomer.

10.2.9 `$Apache::Server::SaveConfig`

`$Apache::Server::SaveConfig` has been renamed to `$Apache2::PerlSections::Save`. see `<Perl>` sections for more information on this global variable.

10.2.10 Apache Configuration Customization

mod_perl 2.0 has slightly changed the mechanism for adding custom configuration directives and now also makes it easy to access an Apache parsed configuration tree's values.

META: add to the config tree access when it'll be written.

10.2.11 `@INC` Manipulation

- **Directories Added Automatically to `@INC`**

    Only if mod_perl was built with `MP_COMPAT_1X=1`, two directories: *$ServerRoot* and *$Server-Root/lib/perl* are pushed onto `@INC`. *$ServerRoot* is as defined by the `ServerRoot` directive in *httpd.conf*.

- **`PERL5LIB` and `PERLLIB` Environment Variables**

    mod_perl 2.0 doesn't do anything special about `PERL5LIB` and `PERLLIB` Environment Variables. If `-T` is in effect these variables are ignored by Perl. There are several other ways to adjust `@INC`.

10.3 Server Startup

mod_perl 1.0 was always running its startup code as soon as it was encountered. In mod_perl 2.0, it is not always the case. Refer to the mod_perl 2.0 startup process section for details.

10.4 Code Porting

mod_perl 2.0 is trying hard to be back compatible with mod_perl 1.0. However some things (mostly APIs) have been changed. In order to gain a complete compatibilty with 1.0 while running under 2.0, you should load the compatibility module as early as possible:

```
use Apache2::compat;
```

at the server startup. And unless there are forgotten things or bugs, your code should work without any changes under 2.0 series.

However, unless you want to keep the 1.0 compatibility, you should try to remove the compatibility layer and adjust your code to work under 2.0 without it. You want to do it mainly for the performance improvement.

This document explains what APIs have changed and what new APIs should be used instead.

Finally, 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. The module `ModPerl::MethodLookup` can be used to find out which modules need to be used. This module also provides a function `preload_all_modules()` that will load all mod_perl 2.0 modules, implementing their API in XS, which is useful when one starts to port their mod_perl 1.0 code, though preferrably avoided in the produc-

tion environment if you want to save memory.

10.5 `Apache::Registry, Apache::PerlRun` and Friends

`Apache::Registry, Apache::PerlRun` and other modules from the registry family now live in the `ModPerl::` namespace. In mod_perl 2.0 we put mod_perl specific functionality into the `ModPerl::` namespace, similar to `APR::` and `Apache2::` which are used for libapr and Apache, respectively.

`ModPerl::Registry` (and others) doesn't `chdir()` into the script's dir like `Apache::Registry` does, because `chdir()` affects the whole process under threads. If you need this functionality use `ModPerl::RegistryPrefork` or `ModPerl::PerlRunPrefork`.

Otherwise `ModPerl::Registry` modules are configured and used similarly to `Apache::Registry` modules. Refer to one of the following manpages for more information:
`ModPerl::RegistryCooker, ModPerl::Registry, ModPerl::RegistryBB` and
`ModPerl::PerlRun`.

10.5.1 *`ModPerl::RegistryLoader`*

In mod_perl 1.0 it was only possible to preload scripts as `Apache::Registry` handlers. In 2.0 the loader can use any of the registry classes to preload into. The old API works as before, but new options can be passed. See the `ModPerl::RegistryLoader` manpage for more information.

10.6 `Apache::Constants`

`Apache::Constants` has been replaced by three classes:

- **`Apache2::Const`**

  Apache constants

- **`APR::Const`**

  Apache Portable Runtime constants

- **`ModPerl::Const`**

  mod_perl specific constants

See the manpages of the respective modules to figure out which constants they provide.

META: add the info how to perform the transition. XXX: may be write a script, which can tell you how to port the constants to 2.0? Currently `Apache2::compat` doesn't provide a complete back compatibility layer.

10.6.1 mod_perl 1.0 and 2.0 Constants Coexistence

If the same codebase is used for both mod_perl generations, the following technique can be used for using constants:

```
package MyApache2::Foo;

use strict;
use warnings;

use mod_perl;
use constant MP2 => ( exists $ENV{MOD_PERL_API_VERSION} and
                      $ENV{MOD_PERL_API_VERSION} >= 2 );


BEGIN {
    if (MP2) {
        require Apache2::Const;
        Apache2::Const->import(-compile => qw(OK DECLINED));
    }
    else {
        require Apache::Constants;
        Apache::Constants->import(qw(OK DECLINED));
    }
}

sub handler {
    # ...
    return MP2 ? Apache2::Const::OK : Apache::Constants::OK;
}
1;
```

Notice that if you don't use the idiom:

```
    return MP2 ? Apache2::Const::OK : Apache::Constants::OK;
```

but something like the following:

```
sub handler1 {
    ...
    return Apache::Constants::OK();
}
sub handler2 {
    ...
    return Apache2::Const::OK();
}
```

You need to add `()`. If you don't do that, let's say that you run under mod_perl 2.0, perl will complain about mod_perl 1.0 constant:

```
Bareword "Apache::Constants::OK" not allowed while "strict subs" ...
```

Adding `()` prevents this warning.

10.6.2 Deprecated Constants

`REDIRECT` and similar constants have been deprecated in Apache for years, in favor of the `HTTP_*` names (they no longer exist Apache 2.0). mod_perl 2.0 API performs the following aliasing behind the scenes:

```
    NOT_FOUND      => 'HTTP_NOT_FOUND',
    FORBIDDEN      => 'HTTP_FORBIDDEN',
    AUTH_REQUIRED  => 'HTTP_UNAUTHORIZED',
    SERVER_ERROR   => 'HTTP_INTERNAL_SERVER_ERROR',
    REDIRECT       => 'HTTP_MOVED_TEMPORARILY',
```

but we suggest moving to use the `HTTP_*` names. For example if running in mod_perl 1.0 compatibility mode, change:

```
  use Apache::Constants qw(REDIRECT);
```

to:

```
  use Apache::Constants qw(HTTP_MOVED_TEMPORARILY);
```

This will work in both mod_perl generations.

10.6.3 `SERVER_VERSION()`

`Apache::Constants::SERVER_VERSION()` has been replaced with `Apache2::ServerUtil::get_server_version()`.

10.6.4 `export()`

`Apache::Constants::export()` has no replacement in 2.0 as it's not needed.

10.7 Issues with Environment Variables

There are several thread-safety issues with setting environment variables.

Environment variables set during request time won't be seen by C code. See the DBD::Oracle issue for possible workarounds.

Forked processes (including backticks) won't see CGI emulation environment variables. (META: This will hopefully be resolved in the future, it's documented in modperl_env.c:modperl_env_magic_set_all.)

10.8 Special Environment Variables

10.8.1 *$ENV{GATEWAY_INTERFACE}*

The environment variable $ENV{GATEWAY_INTERFACE} is not special in mod_perl 2.0, but the same as any other CGI environment variables, i.e. it'll be enabled only if PerlOptions +SetupEnv is enabled and its value would be the default:

```
CGI/1.1
```

or anything else Apache decides to set it to, but not:

```
CGI-Perl/1.1
```

Instead use $ENV{MOD_PERL} (available in both mod_perl generations), which is set to the mod_perl version, like so:

```
mod_perl/2.000002
```

Therefore in order to check whether you are running under mod_perl, you'd say:

```
if ($ENV{MOD_PERL}) { ... }
```

To check for a specific version it's better to use $ENV{MOD_PERL_API_VERSION}

```
use mod_perl;
use constant MP2 => ( exists $ENV{MOD_PERL_API_VERSION} and
                      $ENV{MOD_PERL_API_VERSION} >= 2 );
```

10.9 `Apache::` Methods

10.9.1 *Apache->request*

Apache->request has been replaced with Apache2::RequestUtil::request().

Apache2::RequestUtil->request usage should be avoided under mod_perl 2.0 $r should be passed around as an argument instead (or in the worst case maintain your own global variable). Since your application may run under threaded mpm, the Apache2::RequestUtil->request usage involves storage and retrieval from the thread local storage, which is expensive.

It's possible to use $r even in CGI scripts running under Registry modules, without breaking the mod_cgi compatibility. Registry modules convert a script like:

```
print "Content-type: text/plain";
print "Hello";
```

into something like:

```
package Foo;
sub handler {
   print "Content-type: text/plain\n\n";
   print "Hello";
   return Apache2::Const::OK;
}
```

where the `handler()` function always receives `$r` as an argument, so if you change your script to be:

```
my $r;
$r = shift if $ENV{MOD_PERL};
if ($r) {
    $r->content_type('text/plain');
}
else {
    print "Content-type: text/plain\n\n";
}
print "Hello"
```

it'll really be converted into something like:

```
package Foo;
sub handler {
    my $r;
    $r = shift if $ENV{MOD_PERL};
    if ($r) {
        $r->content_type('text/plain');
    }
    else {
        print "Content-type: text/plain\n\n";
    }
    print "Hello"
    return Apache2::Const::OK;
}
```

The script works under both mod_perl and mod_cgi.

For example CGI.pm 2.93 or higher accepts `$r` as an argument to its `new()` function. So does `CGI::Cookie::fetch` from the same distribution.

Moreover, user's configuration may preclude from `Apache2::RequestUtil->request` being available at run time. For any location that uses `Apache2::RequestUtil->request` and uses `SetHandler modperl`, the configuration should either explicitly enable this feature:

```
<Location ...>
    SetHandler modperl
    PerlOptions +GlobalRequest
    ...
</Location>
```

It's already enabled for `SetHandler perl-script`:

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

This configuration makes `Apache2::RequestUtil->request` available **only** during the response phase (`PerlResponseHandler`). Other phases can make `Apache2::RequestUtil->request` available, by explicitly setting it in the handler that has an access to `$r`. For example the following skeleton for an *authen* phase handler makes the `Apache2::RequestUtil->request` available in the calls made from it:

```
package MyApache2::Auth;

# PerlAuthenHandler MyApache2::Auth

use Apache2::RequestUtil ();
#...
sub handler {
    my $r = shift;
    Apache2::RequestUtil->request($r);
    # do some calls that rely on Apache2::RequestUtil->request being available
    #...
}
```

10.9.2 *Apache->define*

`Apache->define` has been replaced with
`Apache2::ServerUtil::exists_config_define()`.

10.9.3 *Apache->can_stack_handlers*

`Apache->can_stack_handlers` is no longer needed, as mod_perl 2.0 can always stack handlers.

10.9.4 *Apache->untaint*

`Apache->untaint` has moved to `Apache2::ModPerl::Util::untaint()` and now is a function, rather a class method. It'll will untaint all its arguments. You shouldn't be using this function unless you know what you are doing. Refer to the *perlsec* manpage for more information.

`Apache2::compat` provides the backward compatible with mod_perl 1.0 implementation.

10.9.5 *Apache->get_handlers*

To get handlers for the server level, mod_perl 2.0 code should use
`Apache2::ServerUtil::get_handlers()`:

```
$s->get_handlers(...);
```

or:

```
Apache2::ServerUtil->server->get_handlers(...);
```

`Apache->get_handlers` is avalable via `Apache2::compat`.

See also `Apache2::RequestUtil::get_handlers()`.

10.9.6 *Apache->push_handlers*

To push handlers at the server level, mod_perl 2.0 code should use
`Apache2::ServerUtil::push_handlers()`:

```
$s->push_handlers(...);
```

or:

```
Apache2::ServerUtil->server->push_handlers(...);
```

`Apache->push_handlers` is avalable via `Apache2::compat`.

See also `Apache2::RequestUtil::push_handlers()`.

10.9.7 *Apache->set_handlers*

To set handlers at the server level, mod_perl 2.0 code should use
`Apache2::ServerUtil::set_handlers()`:

```
$s->set_handlers(...);
```

or:

```
Apache2::ServerUtil->server->set_handlers(...);
```

`Apache->set_handlers` is avalable via `Apache2::compat`.

To reset the list of handlers, instead of doing:

```
$r->set_handlers(PerlAuthenHandler => [ \&OK ]);
```

do:

```
$r->set_handlers(PerlAuthenHandler => []);
```

or

```
  $r->set_handlers(PerlAuthenHandler => undef);
```

See also `Apache2::RequestUtil::set_handlers()`.

10.9.8 *Apache->httpd_conf*

Apache->httpd_conf is now `$s->add_config`:

```
  require Apache2::ServerUtil;
  Apache2::ServerUtil->server->add_config(['require valid-user']);
```

Apache->httpd_conf is avalable via `Apache2::compat`.

See also `Apache2::RequestUtil::add_config()`.

10.9.9 *Apache->unescape_url_info*

Apache->unescape_url_info is not available in mod_perl 2.0 API. Use `CGI::Util::unescape` instead (http://search.cpan.org/dist/CGI.pm/CGI/Util.pm).

It is also available via `Apache2::compat` for backwards compatibility.

10.9.10 *Apache::exit()*

Apache::exit() has been replaced with `ModPerl::Util::exit()`.

10.9.11 *Apache::gensym()*

Since Perl 5.6.1 filehandlers are autovivified and there is no need for `Apache::gensym()` function, since now it can be done with:

```
  open my $fh, "foo" or die $!;
```

Though the C function `modperl_perl_gensym()` is available for XS/C extensions writers.

10.9.12 *Apache::log_error()*

Apache::log_error() is not available in mod_perl 2.0 API. You can use `Apache2::Log::log_error()`:

```
  Apache2::ServerUtil->server->log_error
```

instead. See the `Apache2::Log` manpage.

10.9.13 `Apache->warn`

`$Apache->warn` has been removed and exists only in `Apache2::compat`. Choose another `Apache2::Log` method.

10.9.14 `Apache::warn`

`$Apache::warn` has been removed and exists only in `Apache2::compat`. Choose another `Apache2::Log` method.

10.9.15 `Apache::module()`

`Apache::module()` has been replaced with the function `Apache2::Module::loaded()`, which now accepts a single argument: the module name.

10.10 `Apache::` Variables

10.10.1 `$Apache::__T`

`$Apache::__T` is deprecated in mod_perl 2.0. Use `${^TAINT}` instead.

10.11 `Apache::Module::` Methods

10.11.1 `Apache::Module->top_module`

`Apache::Module->top_module` has been replaced with the function
`Apache2::Module::top_module()`.

10.11.2 `Apache::Module->get_config`

`Apache::Module->get_config` has been replaced with the function
`Apache2::Module::get_config()`.

10.12 `Apache::ModuleConfig::` Methods

10.12.1 `Apache::ModuleConfig->get`

`Apache::ModuleConfig->get` has been replaced with the function
`Apache2::Module::get_config()`.

10.13 `Apache::Server::` Methods and Variables

10.13.1 *`$Apache::Server::CWD`*

`$Apache::Server::CWD` is deprecated and exists only in `Apache2::compat`.

10.13.2 *`$Apache::Server::AddPerlVersion`*

`$Apache::Server::AddPerlVersion` is deprecated and exists only in `Apache2::compat`.

10.13.3 *`$Apache::Server::Starting` and `$Apache::Server::ReStarting`*

`$Apache::Server::Starting` and `$Apache::Server::ReStarting` were replaced by `Apache2::ServerUtil::restart_count()`. Though both exist in `Apache2::compat`.

10.13.4 *`Apache::Server->warn`*

`Apache::Server->warn` has been removed and exists only in `Apache2::compat`. Choose another `Apache2::Log` method.

10.14 Server Object Methods

10.14.1 *`$s->register_cleanup`*

`$s->register_cleanup` has been replaced with `APR::Pool::cleanup_register()` which accepts the pool object as the first argument instead of the server object, a callback function as a second and data variable as the optional third argument. If that data argument was provided it is then passed to the callback function when the time comes for the pool object to get destroyed.

```
use Apache2::ServerUtil ();
sub cleanup_callback {
    my $data = shift;
    # your code comes here
    return Apache2::Const::OK;
}
$s->pool->cleanup_register(\&cleanup_callback, $data);
```

See also `PerlChildExitHandler`.

In order to register a cleanup handler to be run only once when the main server (not each child process) shuts down, you can register a cleanup handler with `server_shutdown_cleanup_register()`.

10.14.2 `$s->uid`

See the next entry.

10.14.3 `$s->gid`

apache-1.3 had server_rec records for server_uid and server_gid. httpd-2.0 doesn't have them, because in httpd-2.0 the directives User and Group are platform specific. And only UNIX supports it: http://httpd.apache.org/docs-2.0/mod/mpm_common.html#user

It's possible to emulate mod_perl 1.0 API doing:

```
sub Apache2::Server::uid { $< }
sub Apache2::Server::gid { $( }
```

but the problem is that if the server is started as *root*, but its child processes are run under a different user-name, e.g. *nobody*, at the startup the above function will report the uid and gid values of *root* and not *nobody*, i.e. at startup it won't be possible to know what the User and Group settings are in *httpd.conf*.

META: though we can probably access the parsed config tree and try to fish these values from there. The real problem is that these values won't be available on all platforms and therefore we should probably not support them and let developers figure out how to code around it (e.g. by using $< and $()).

10.15 Request Object Methods

10.15.1 `$r->print`

```
$r->print($foo);
```

or

```
print $foo;
```

no longer accepts a reference to a scalar as it did in mod_perl 1.0. This optimisation is not needed in the mod_perl 2.0's implementation of print.

10.15.2 `$r->cgi_env`

See the next item

10.15.3 `$r->cgi_var`

$r->cgi_env and $r->cgi_var should be replaced with $r->subprocess_env, which works identically in both mod_perl generations.

10.15.4 *$r->current_callback*

$r->current_callback is now simply a ModPerl::Util::current_callback and can be called for any of the phases, including those where $r simply doesn't exist.

Apache2::compat implements $r->current_callback for backwards compatibility.

10.15.5 *$r->cleanup_for_exec*

$r->cleanup_for_exec wasn't a part of the mp1 core API, but lived in a 3rd party module Apache2::SubProcess. That module's functionality is now a part of mod_perl 2.0 API. But Apache 2.0 doesn't need this function any longer.

Apache2::compat implements $r->cleanup_for_exec for backwards compatibility as a NOOP.

See also the Apache2::SubProcess manpage.

10.15.6 *$r->get_remote_host*

get_remote_host() is now invoked on the connection object:

```
  use Apache2::Connection;
  $r->connection->get_remote_host();
```

$r->get_remote_host is available through Apache2::compat.

10.15.7 *$r->content*

See the next item.

10.15.8 *$r->args in an Array Context*

$r->args in 2.0 returns the query string without parsing and splitting it into an array. You can also set the query string by passing a string to this method.

$r->content and $r->args in an array context were mistakes that never should have been part of the mod_perl 1.0 API. There are multiple reason for that, among others:

- does not handle multi-value keys

- does not handle multi-part content types

- does not handle chunked encoding

- slurps $r->headers_in->{'content-length'} into a single buffer (bad for performance, memory bloat, possible dos attack, etc.)

- in general duplicates functionality (and does so poorly) that is done better in `Apache2::Request`.

- if one wishes to simply read POST data, there is the more modern filter API, along with continued support for `read(STDIN, ...)` and `$r->read($buf, $r->headers_in->{'content-length'})`

You could use `CGI.pm` or the code in `Apache2::compat` (it's slower).

However, now that `Apache2::Request` has been ported to mod_perl 2.0 you can use it instead and reap the benefits of the fast C implementations of these functions. For documentation on its uses, please see:

http://httpd.apache.org/apreq

10.15.9 $r->chdir_file

`chdir()` cannot be used in the threaded environment, therefore `$r->chdir_file` is not in the mod_perl 2.0 API.

For more information refer to: Threads Coding Issues Under mod_perl.

10.15.10 $r->is_main

`$r->is_main` is not part of the mod_perl 2.0 API. Use `!$r->main` instead.

Refer to the `Apache2::RequestRec` manpage.

10.15.11 $r->filename

When a new `$r->filename` is assigned Apache 2.0 doesn't update the finfo structure like it did in Apache 1.3. If the old behavior is desired Apache2::compat's overriding can be used. Otherwise one should explicitly update the finfo struct when desired as explained in the `filename` API entry.

10.15.12 $r->finfo

As Apache 2.0 doesn't provide an access to the stat structure, but hides it in the opaque object `$r->finfo` now returns an `APR::Finfo` object. You can then invoke the `APR::Finfo` accessor methods on it.

It's also possible to adjust the mod_perl 1.0 code using Apache2::compat's overriding. For example:

```
use Apache2::compat;
Apache2::compat::override_mp2_api('Apache2::RequestRec::finfo');
my $is_writable = -w $r->finfo;
Apache2::compat::restore_mp2_api('Apache2::RequestRec::finfo');
```

which internally does just the following:

```
 stat $r->filename and return \*_;
```

So may be it's easier to just change the code to use this directly, so the above example can be adjusted to be:

```
 my $is_writable = -w $r->filename;
```

with the performance penalty of an extra `stat()` system call. If you don't want this extra call, you'd have to write:

```
 use APR::Finfo;
 use Apache2::RequestRec;
 use APR::Const -compile => qw(WWRITE);
 my $is_writable = $r->finfo->protection & APR::WWRITE,
```

See the `APR::Finfo` manpage for more information.

10.15.13 *$r->notes*

Similar to `headers_in()`, `headers_out()` and `err_headers_out()` in mod_perl 2.0, `$r->notes()` returns an `APR::Table` object, which can be used as a tied hash or calling its *get()* / *set()* / *add()* / *unset()* methods.

It's also possible to adjust the mod_perl 1.0 code using `Apache2::compat`'s overriding:

```
 use Apache2::compat;
 Apache2::compat::override_mp2_api('Apache2::RequestRec::notes');
 $r->notes($key => $val);
 $val = $r->notes($key);
 Apache2::compat::restore_mp2_api('Apache2::RequestRec::notes');
```

See the `Apache2::RequestRec` manpage.

10.15.14 *$r->header_in*

See $r->err_header_out.

10.15.15 *$r->header_out*

See $r->err_header_out.

10.15.16 *$r->err_header_out*

`header_in()`, `header_out()` and `err_header_out()` are not available in 2.0. Use `headers_in()`, `headers_out()` and `err_headers_out()` instead (which should be used in 1.0 as well). For example you need to replace:

```
$r->err_header_out("Pragma" => "no-cache");
```

with:

```
$r->err_headers_out->{'Pragma'} = "no-cache";
```

See the Apache2::RequestRec manpage.

10.15.17 *$r->register_cleanup*

Similarly to $s->register_cleanup, $r->register_cleanup has been replaced with APR::Pool::cleanup_register() which accepts the pool object as the first argument instead of the request object. e.g.:

```
sub cleanup_callback { my $data = shift; ... }
$r->pool->cleanup_register(\&cleanup_callback, $data);
```

where the last argument $data is optional, and if supplied will be passed as the first argument to the callback function.

See the APR::Pool manpage.

10.15.18 *$r->post_connection*

$r->post_connection has been replaced with:

```
$r->connection->pool->cleanup_register();
```

See the APR::Pool manpage.

10.15.19 *$r->request*

Use Apache2::RequestUtil->request.

10.15.20 *$r->send_fd*

mod_perl 2.0 provides a new method sendfile() instead of send_fd, so if your code used to do:

```
open my $fh, "<$file" or die "$!";
$r->send_fd($fh);
close $fh;
```

now all you need is:

```
$r->sendfile($file);
```

There is also a compatibility implementation of send_fd in pure perl in Apache2::compat.

XXX: later we may provide a direct access to the real send_fd. That will be possible if we figure out how to portably convert PerlIO/FILE into apr_file_t (with help of apr_os_file_put, which expects a native file-handle, so I'm not sure whether this will work on win32).

10.15.21 $r->send_http_header

This method is not needed in 2.0, though available in `Apache2::compat`. 2.0 handlers only need to set the *Content-type* via `$r->content_type($type)`.

10.15.22 $r->server_root_relative

This method was replaced with `Apache2::ServerUtil::server_root_relative()` function and its first argument is a *pool* object. For example:

```
# during request
$conf_dir = Apache2::server_root_relative($r->pool, 'conf');
# during startup
$conf_dir = Apache2::server_root_relative($s->pool, 'conf');
```

Note that the old form

```
my $conf_dir = Apache->server_root_relative('conf');
```

is no longer valid - `server_root_relative()` must be explicitly passed a pool.

The old functionality is available with `Apache2::compat`.

10.15.23 $r->hard_timeout

See $r->kill_timeout.

10.15.24 $r->reset_timeout

See $r->kill_timeout.

10.15.25 $r->soft_timeout

See $r->kill_timeout.

10.15.26 $r->kill_timeout

The functions `$r->hard_timeout`, `$r->reset_timeout`, `$r->soft_timeout` and `$r->kill_timeout` aren't needed in mod_perl 2.0. `Apache2::compat` implements these functions for backwards compatibility as NOOPs.

10.15.27 $r->set_byterange

See $r->each_byterange.

10.15.28 $r->each_byterange

The functions $r->set_byterange and $r->each_byterange aren't in the Apache 2.0 API, and therefore don't exist in mod_perl 2.0. The byterange serving functionality is now implemented in the ap_byterange_filter, which is a part of the core http module, meaning that it's automatically taking care of serving the requested ranges off the normal complete response. There is no need to configure it. It's executed only if the appropriate request headers are set. These headers aren't listed here, since there are several combinations of them, including the older ones which are still supported. For a complete info on these see *modules/http/http_protocol.c*.

10.16 `Apache::Connection`

10.16.1 $connection->auth_type

The record *auth_type* doesn't exist in the Apache 2.0's connection struct. It exists only in the request record struct. The new accessor in 2.0 API is $r->ap_auth_type.

Apache2::compat provides a back compatibility method, though it relies on the availability of the global Apache->request, which requires the configuration to have:

```
  PerlOptions +GlobalRequest
```

to set it up for earlier stages than response handler.

10.16.2 $connection->user

This method is deprecated in mod_perl 1.0 and $r->user should be used instead for both mod_perl generations. $r->user() method is available since mod_perl version 1.24_01.

10.16.3 $connection->local_addr

See $connection->remote_addr

10.16.4 $connection->remote_addr

$c->local_addr and $c->remote_addr return an APR::SockAddr object and you can use this object's methods to retrieve the wanted bits of information, so if you had a code like:

```
  use Socket 'sockaddr_in';
  my $c = $r->connection;
  my ($serverport, $serverip) = sockaddr_in($c->local_addr);
  my ($remoteport, $remoteip) = sockaddr_in($c->remote_addr);
```

now it'll be written as:

```
require APR::SockAddr;
my $c = $r->connection;
my $serverport = $c->local_addr->port;
my $serverip   = $c->local_addr->ip_get;
my $remoteport = $c->remote_addr->port;
my $remoteip   = $c->remote_addr->ip_get;
```

It's also possible to adjust the code using Apache2::compat's overriding:

```
use Socket 'sockaddr_in';
use Apache2::compat;

Apache2::compat::override_mp2_api('Apache2::Connection::local_addr');
my ($serverport, $serverip) = sockaddr_in($r->connection->local_addr);
Apache2::compat::restore_mp2_api('Apache2::Connection::local_addr');

Apache2::compat::override_mp2_api('Apache::Connection::remote_addr');
my ($remoteport, $remoteip) = sockaddr_in($r->connection->remote_addr);
Apache2::compat::restore_mp2_api('Apache::Connection::remote_addr');
```

10.17 `Apache::File`

The methods from mod_perl 1.0's module `Apache::File` have been either moved to other packages or removed.

10.17.1 `new( )`, `open( )` and `close( )`

The methods `new()`, `open()` and `close()` were removed. See the back compatibility implementation in the module `Apache2::compat`.

Because of that some of the idioms have changes too. If previously you were writing:

```
my $fh = Apache::File->new($r->filename)
    or return Apache::DECLINED;
# Slurp the file (hopefully it's not too big).
my $content = do { local $/; <$fh> };
close $fh;
```

Now, you would write that using `Apache2::RequestUtil::slurp_filename()`:

```
use Apache2::RequestUtil ();
my $content = ${ $r->slurp_filename() };
```

10.17.2 `tmpfile( )`

The method `tmpfile()` was removed since Apache 2.0 doesn't have the API for this method anymore.

See `File::Temp`, or the back compatibility implementation in the module `Apache2::compat`.

With Perl v5.8.0 you can create anonymous temporary files:

```
open $fh, "+>", undef or die $!;
```

That is a literal `undef`, not an undefined value.

10.18 `Apache::Util`

A few `Apache2::Util` functions have changed their interface.

10.18.1 *`Apache::Util::size_string()`*

`Apache::Util::size_string()` has been replaced with `APR::String::format_size()`, which returns formatted strings of only 4 characters long.

10.18.2 *`Apache::Util::escape_uri()`*

`Apache::Util::escape_uri()` has been replaced with `Apache2::Util::escape_path()` and requires a pool object as a second argument. For example:

```
$escaped_path = Apache2::Util::escape_path($path, $r->pool);
```

10.18.3 *`Apache::Util::unescape_uri()`*

`Apache::Util::unescape_uri()` has been replaced with
`Apache2::URI::unescape_url()`.

10.18.4 *`Apache::Util::escape_html()`*

`Apache::Util::escape_html` is not available in mod_perl 2.0. Use `HTML::Entities` instead (http://search.cpan.org/dist/HTML-Parser/lib/HTML/Entities.pm).

It's also available via `Apache2::compat` for backwards compatibility.

10.18.5 *`Apache::Util::parsedate()`*

`Apache::Util::parsedate()` has been replaced with `APR::Date::parse_http()`.

10.18.6 *`Apache::Util::ht_time()`*

`Apache2::Util::ht_time()` now requires a `pool` object as a first argument.

For example:

```
use Apache2::Util ();
$fmt = '%a, %d %b %Y %H:%M:%S %Z';
$gmt = 1;
$fmt_time = Apache2::Util::ht_time($r->pool, time(), $fmt, $gmt);
```

See the Apache2::Util manpage.

It's also possible to adjust the mod_perl 1.0 code using Apache2::compat's overriding.

For example:

```
use Apache2::compat;
Apache2::compat::override_mp2_api('Apache2::Util::ht_time');
$fmt_time = Apache2::Util::ht_time(time(), $fmt, $gmt);
Apache2::compat::restore_mp2_api('Apache2::Util::ht_time');
```

10.18.7 *Apache::Util::validate_password()*

Apache::Util::validate_password() has been replaced with APR::Util::pass-word_validate(). For example:

```
my $ok = Apache2::Util::password_validate("stas", "ZeO.RAc3iYvpA");
```

10.19 *Apache::URI*

10.19.1 *Apache::URI->parse($r, [$uri])*

parse() and its associated methods have moved into the APR::URI package. For example:

```
my $curl = $r->construct_url;
APR::URI->parse($r->pool, $curl);
```

See the APR::URI manpage.

10.19.2 *unparse( )*

Other than moving to the package APR::URI, unparse is now protocol-agnostic. Apache won't use *http* as the default protocol if *hostname* was set, but *scheme* wasn't not. So the following code:

```
# request http://localhost.localdomain:8529/TestAPI::uri
my $parsed = $r->parsed_uri;
$parsed->hostname($r->get_server_name);
$parsed->port($r->get_server_port);
print $parsed->unparse;
```

prints:

```
//localhost.localdomain:8529/TestAPI::uri
```

forcing you to make sure that the scheme is explicitly set. This will do the right thing:

```
# request http://localhost.localdomain:8529/TestAPI::uri
my $parsed = $r->parsed_uri;
$parsed->hostname($r->get_server_name);
$parsed->port($r->get_server_port);
$parsed->scheme('http');
print $parsed->unparse;
```

prints:

```
http://localhost.localdomain:8529/TestAPI::uri
```

See the APR::URI manpage for more information.

It's also possible to adjust the behavior to be mod_perl 1.0 compatible using Apache2::compat's overriding, in which case unparse() will transparently set *scheme* to *http*.

```
# request http://localhost.localdomain:8529/TestAPI::uri
Apache2::compat::override_mp2_api('APR::URI::unparse');
my $parsed = $r->parsed_uri;
# set hostname, but not the scheme
$parsed->hostname($r->get_server_name);
$parsed->port($r->get_server_port);
print $parsed->unparse;
Apache2::compat::restore_mp2_api('APR::URI::unparse');
```

prints:

```
http://localhost.localdomain:8529/TestAPI::uri
```

10.20 Miscellaneous

10.20.1 Method Handlers

In mod_perl 1.0 the method handlers could be specified by using the ($$) prototype:

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

sub handler ($$) {
    my ($class, $r) = @_;
    ...;
}
```

mod_perl 2.0 doesn't handle callbacks with ($$) prototypes differently than other callbacks (as it did in mod_perl 1.0), mainly because several callbacks in 2.0 have more arguments than just $r, so the ($$) prototype doesn't make sense anymore. Therefore if you want your code to work with both mod_perl

generations and you can allow the luxury of:

```
require 5.6.0;
```

or if you need the code to run only on mod_perl 2.0, use the *method* subroutine attribute. (The subroutine attributes are supported in Perl since version 5.6.0.)

Here is the same example rewritten using the *method* subroutine attribute:

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

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

See the *attributes* manpage.

If `Class->method` syntax is used for a `Perl*Handler`, the `:method` attribute is not required.

The porting tutorial provides examples on how to use the same code base under both mod_perl generations when the handler has to be a method.

10.20.2 Stacked Handlers

Both mod_perl 1.0 and 2.0 support the ability to register more than one handler in each runtime phase, a feature known as stacked handlers. For example,

```
PerlAuthenHandler My::First My::Second
```

The behavior of stacked Perl handlers differs between mod_perl 1.0 and 2.0. In 2.0, mod_perl respects the run-type of the underlying hook - it does not run all configured Perl handlers for each phase but instead behaves in the same way as Apache does when multiple handlers are configured, respecting (or ignoring) the return value of each handler as it is called.

See Stacked Handlers for a complete description of each hook and its run-type.

10.21 **Apache::src**

For those who write 3rd party modules using XS, this module was used to supply mod_perl specific include paths, defines and other things, needed for building the extensions. mod_perl 2.0 makes things transparent with `ModPerl::MM`.

Here is how to write a simple *Makefile.PL* for modules wanting to build XS code against mod_perl 2.0:

```
use mod_perl 2.0;
use ModPerl::MM ();

ModPerl::MM::WriteMakefile(
    NAME => "Foo",
);
```

and everything will be done for you.

META: we probably will have a compat layer at some point.

META: move this section to the devel/porting and link there instead

10.22 `Apache::Table`

`Apache::Table` has been renamed to `APR::Table`.

10.23 `Apache::SIG`

`Apache::SIG` currently exists only `Apache2::compat` and it does nothing.

10.24 `Apache::StatINC`

`Apache::StatINC` has been replaced by `Apache2::Reload`, which works for both mod_perl generations. To migrate to `Apache2::Reload` simply replace:

```
PerlInitHandler Apache::StatINC
```

with:

```
PerlInitHandler Apache2::Reload
```

However `Apache2::Reload` provides an extra functionality, covered in the module's manpage.

10.25 Maintainers

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

● Stas Bekman [http://stason.org/]

10.26 Authors

● Stas Bekman [http://stason.org/]

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

11   Introducing mod_perl Handlers

11.1 Description

This chapter provides an introduction into mod_perl handlers.

11.2 What are Handlers?

Apache distinguishes between numerous phases for which it provides hooks (because the C functions are called *ap_hook_<phase_name>*) where modules can plug various callbacks to extend and alter the default behavior of the webserver. mod_perl provides a Perl interface for most of the available hooks, so mod_perl modules writers can change the Apache behavior in Perl. These callbacks are usually referred to as *handlers* and therefore the configuration directives for the mod_perl handlers look like: `Perl-FooHandler`, where `Foo` is one of the handler names. For example `PerlResponseHandler` configures the response callback.

A typical handler is simply a perl package with a *handler* subroutine. For example:

```
file:MyApache2/CurrentTime.pm
---------------------------
package MyApache2::CurrentTime;

use strict;
use warnings;

use Apache2::RequestRec ();
use Apache2::RequestIO ();

use Apache2::Const -compile => qw(OK);

sub handler {
    my $r = shift;

    $r->content_type('text/plain');
    $r->print("Now is: " . scalar(localtime) . "\n");

    return Apache2::Const::OK;
}
1;
```

This handler simply returns the current date and time as a response.

Since this is a response handler, we configure it as a such in *httpd.conf*:

```
PerlResponseHandler MyApache2::CurrentTime
```

Since the response handler should be configured for a specific location, let's write a complete configuration section:

```
PerlModule MyApache2::CurrentTime
<Location /time>
    SetHandler modperl
    PerlResponseHandler MyApache2::CurrentTime
</Location>
```

Now when a request is issued to *http://localhost/time* this response handler is executed and a response that includes the current time is returned to the client.

11.3 Handler Return Values

Different handler groups are supposed to return different values.

Make sure that you **always** explicitly return a wanted value and don't rely on the result of last expression to be used as the return value -- things will change in the future and you won't know why things aren't working anymore.

The only value that can be returned by all handlers is `Apache2::Const::OK`, which tells Apache that the handler has successfully finished its execution.

`Apache2::Const::DECLINED` is another return value that indicates success, but it's only relevant for phases of type `RUN_FIRST`.

HTTP handlers may also return `Apache2::Const::DONE` which tells Apache to stop the normal HTTP request cycle and fast forward to the `PerlLogHandler`, followed by `PerlCleanupHandler`. HTTP handlers may return any HTTP status, which similarly to `Apache2::Const::DONE` will cause an abort of the request cycle, by also will be interpreted as an error. Therefore you don't want to return `Apache2::Const::HTTP_OK` from your HTTP response handler, but `Apache2::Const::OK` and Apache will send the `200 OK` status by itself.

Filter handlers return `Apache2::Const::OK` to indicate that the filter has successfully finished. If the return value is `Apache2::Const::DECLINED`, mod_perl will read and forward the data on behalf of the filter. Please notice that this feature is specific to mod_perl. If there is some problem with obtaining or sending the bucket brigades, or the buckets in it, filters need to return the error returned by the method that tried to manipulate the bucket brigade or the bucket. Normally it'd be an `APR::` constant.

Protocol handler return values aren't really handled by Apache, the handler is supposed to take care of any errors by itself. The only special case is the `PerlPreConnectionHandler` handler, which, if returning anything but `Apache2::Const::OK` or `Apache2::Const::DONE`, will prevent from `Perl-ConnectionHandler` to be run. `PerlPreConnectionHandler` handlers should always return `Apache2::Const::OK`.

11.4 mod_perl Handlers Categories

The mod_perl handlers can be divided by their application scope in several categories:

- **Server life cycle**
  - **PerlOpenLogsHandler**
  - **PerlPostConfigHandler**
  - **PerlChildInitHandler**
  - **PerlChildExitHandler**
- **Protocols**

○ **PerlPreConnectionHandler**
○ **PerlProcessConnectionHandler**
● **Filters**
○ **PerlInputFilterHandler**
○ **PerlOutputFilterHandler**
● **HTTP Protocol**
○ **PerlPostReadRequestHandler**
○ **PerlTransHandler**
○ **PerlMapToStorageHandler**
○ **PerlInitHandler**
○ **PerlHeaderParserHandler**
○ **PerlAccessHandler**
○ **PerlAuthenHandler**
○ **PerlAuthzHandler**
○ **PerlTypeHandler**
○ **PerlFixupHandler**
○ **PerlResponseHandler**
○ **PerlLogHandler**
○ **PerlCleanupHandler**

11.5 Stacked Handlers

For each phase there can be more than one handler assigned (also known as *hooks*, because the C functions are called *ap_hook_<phase_name>*). Phases' behavior varies when there is more then one handler registered to run for the same phase. The following table specifies each handler's behavior in this situation:

```
    Directive                      Type
    ------------------------------------
PerlOpenLogsHandler            RUN_ALL
PerlPostConfigHandler          RUN_ALL
PerlChildInitHandler           VOID
PerlChildExitHandler           VOID

PerlPreConnectionHandler       RUN_ALL
PerlProcessConnectionHandler   RUN_FIRST

PerlPostReadRequestHandler     RUN_ALL
PerlTransHandler               RUN_FIRST
PerlMapToStorageHandler        RUN_FIRST
PerlInitHandler                RUN_ALL
PerlHeaderParserHandler        RUN_ALL
PerlAccessHandler              RUN_ALL
PerlAuthenHandler              RUN_FIRST
PerlAuthzHandler               RUN_FIRST
PerlTypeHandler                RUN_FIRST
PerlFixupHandler               RUN_ALL
PerlResponseHandler            RUN_FIRST
PerlLogHandler                 RUN_ALL
```

```
PerlCleanupHandler            RUN_ALL

PerlInputFilterHandler        VOID
PerlOutputFilterHandler       VOID
```

Note: `PerlChildExitHandler` and `PerlCleanupHandler` are not real Apache hooks, but to mod_perl users they behave as all other hooks.

And here is the description of the possible types:

11.5.1 *VOID*

Handlers of the type `VOID` will be *all* executed in the order they have been registered disregarding their return values. Though in mod_perl they are expected to return `Apache2::Const::OK`.

11.5.2 *RUN_FIRST*

Handlers of the type `RUN_FIRST` will be executed in the order they have been registered until the first handler that returns something other than `Apache2::Const::DECLINED`. If the return value is `Apache2::Const::DECLINED`, the next handler in the chain will be run. If the return value is `Apache2::Const::OK` the next phase will start. In all other cases the execution will be aborted.

11.5.3 *RUN_ALL*

Handlers of the type `RUN_ALL` will be executed in the order they have been registered until the first handler that returns something other than `Apache2::Const::OK` or `Apache2::Const::DECLINED`.

For C API declarations see *include/ap_config.h*, which includes other types which aren't exposed by mod_perl handlers.

Also see mod_perl Directives Argument Types and Allowed Location

11.6 Hook Ordering (Position)

The following constants specify how the new hooks (handlers) are inserted into the list of hooks when there is at least one hook already registered for the same phase.

META: Not working yet.

META: need to verify the following:

- **APR::Const::HOOK_REALLY_FIRST**

  run this hook first, before ANYTHING.

- **`APR::Const::HOOK_FIRST`**

  run this hook first.

- **`APR::Const::HOOK_MIDDLE`**

  run this hook somewhere.

- **`APR::Const::HOOK_LAST`**

  run this hook after every other hook which is defined.

- **`APR::Const::HOOK_REALLY_LAST`**

  run this hook last, after EVERYTHING.

META: more information in mod_example.c talking about position/predecessors, etc.

11.7 Bucket Brigades

Apache 2.0 allows multiple modules to filter both the request and the response. Now one module can pipe its output as an input to another module as if another module was receiving the data directly from the TCP stream. The same mechanism works with the generated response.

With I/O filtering in place, simple filters, like data compression and decompression, can be easily implemented and complex filters, like SSL, are now possible without needing to modify the the server code which was the case with Apache 1.3.

In order to make the filtering mechanism efficient and avoid unnecessary copying, while keeping the data abstracted, the *Bucket Brigades* technology was introduced. It's also used in protocol handlers.

A bucket represents a chunk of data. Buckets linked together comprise a brigade. Each bucket in a brigade can be modified, removed and replaced with another bucket. The goal is to minimize the data copying where possible. Buckets come in different types, such as files, data blocks, end of stream indicators, pools, etc. To manipulate a bucket one doesn't need to know its internal representation.

The stream of data is represented by bucket brigades. When a filter is called it gets passed the brigade that was the output of the previous filter. This brigade is then manipulated by the filter (e.g., by modifying some buckets) and passed to the next filter in the stack.

The following figure depicts an imaginary bucket brigade:

bucket brigades

The figure tries to show that after the presented bucket brigade has passed through several filters some buckets were removed, some modified and some added. Of course the handler that gets the brigade cannot tell the history of the brigade, it can only see the existing buckets in the brigade.

Bucket brigades are discussed in detail in the protocol handlers and I/O filtering chapters.

11.8 Maintainers

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

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

11.9 Authors

-

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

12   Server Life Cycle Handlers

12.1 Description

This chapter discusses server life cycle and the mod_perl handlers participating in it.

12.2 Server Life Cycle

The following diagram depicts the Apache 2.0 server life cycle and highlights which handlers are available to mod_perl 2.0:

server life cycle

Apache 2.0 starts by parsing the configuration file. After the configuration file is parsed, the `PerlOpen-LogsHandler` handlers are executed if any. After that it's a turn of `PerlPostConfigHandler` handlers to be run. When the *post_config* phase is finished the server immediately restarts, to make sure that it can survive graceful restarts after starting to serve the clients.

When the restart is completed, Apache 2.0 spawns the workers that will do the actual work. Depending on the used MPM, these can be threads, processes or a mixture of both. For example the *worker* MPM spawns a number of processes, each running a number of threads. When each child process is started `PerlChildInitHandler` handlers are executed. Notice that they are run for each starting process, not a thread.

From that moment on each working thread processes connections until it's killed by the server or the server is shutdown.

12.2.1 Startup Phases Demonstration Module

Let's look at the following example that demonstrates all the startup phases:

```
#file:MyApache2/StartupLog.pm
#--------------------------
package MyApache2::StartupLog;

use strict;
use warnings;

use Apache2::Log ();
use Apache2::ServerUtil ();

use Fcntl qw(:flock);
use File::Spec::Functions;

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

my $log_path = catfile Apache2::ServerUtil::server_root,
    "logs", "startup_log";
my $log_fh;

sub open_logs {
    my ($conf_pool, $log_pool, $temp_pool, $s) = @_;
```

```
    $s->warn("opening the log file: $log_path");
    open $log_fh, ">>$log_path" or die "can't open $log_path: $!";
    my $oldfh = select($log_fh); $| = 1; select($oldfh);

    say("process $$ is born to reproduce");
    return Apache2::Const::OK;
}

sub post_config {
    my ($conf_pool, $log_pool, $temp_pool, $s) = @_;
    say("configuration is completed");
    return Apache2::Const::OK;
}

sub child_init {
    my ($child_pool, $s) = @_;
    say("process $$ is born to serve");
    return Apache2::Const::OK;

}

sub child_exit {
    my ($child_pool, $s) = @_;
    say("process $$ now exits");
    return Apache2::Const::OK;
}

sub say {
    my ($caller) = (caller(1))[3] =~ /([^:]+)$/;
    if (defined $log_fh) {
        flock $log_fh, LOCK_EX;
        printf $log_fh "[%s] - %-11s: %s\n",
            scalar(localtime), $caller, $_[0];
        flock $log_fh, LOCK_UN;
    }
    else {
        # when the log file is not open
        warn __PACKAGE__ . " says: $_[0]\n";
    }
}

my $parent_pid = $$;
END {
    my $msg = "process $$ is shutdown";
    $msg .= "\n". "-" x 20 if $$ == $parent_pid;
    say($msg);
}

1;
```

And the *httpd.conf* configuration section:

```
<IfModule prefork.c>
  StartServers        4
  MinSpareServers     4
  MaxSpareServers     4
```

```
  MaxClients            10
  MaxRequestsPerChild   0
</IfModule>

PerlModule              MyApache2::StartupLog
PerlOpenLogsHandler     MyApache2::StartupLog::open_logs
PerlPostConfigHandler   MyApache2::StartupLog::post_config
PerlChildInitHandler    MyApache2::StartupLog::child_init
PerlChildExitHandler    MyApache2::StartupLog::child_exit
```

When we perform a server startup followed by a shutdown, the *logs/startup_log* is created if it didn't exist already (it shares the same directory with *error_log* and other standard log files), and each stage appends to that file its log information. So when we perform:

```
% bin/apachectl start && bin/apachectl stop
```

the following is getting logged to *logs/startup_log*:

```
[Sun Jun  6 01:50:06 2004] - open_logs  : process 24189 is born to reproduce
[Sun Jun  6 01:50:06 2004] - post_config: configuration is completed
[Sun Jun  6 01:50:07 2004] - END        : process 24189 is shutdown
--------------------
[Sun Jun  6 01:50:08 2004] - open_logs  : process 24190 is born to reproduce
[Sun Jun  6 01:50:08 2004] - post_config: configuration is completed
[Sun Jun  6 01:50:09 2004] - child_init : process 24192 is born to serve
[Sun Jun  6 01:50:09 2004] - child_init : process 24193 is born to serve
[Sun Jun  6 01:50:09 2004] - child_init : process 24194 is born to serve
[Sun Jun  6 01:50:09 2004] - child_init : process 24195 is born to serve
[Sun Jun  6 01:50:10 2004] - child_exit : process 24193 now exits
[Sun Jun  6 01:50:10 2004] - END        : process 24193 is shutdown
[Sun Jun  6 01:50:10 2004] - child_exit : process 24194 now exits
[Sun Jun  6 01:50:10 2004] - END        : process 24194 is shutdown
[Sun Jun  6 01:50:10 2004] - child_exit : process 24195 now exits
[Sun Jun  6 01:50:10 2004] - child_exit : process 24192 now exits
[Sun Jun  6 01:50:10 2004] - END        : process 24192 is shutdown
[Sun Jun  6 01:50:10 2004] - END        : process 24195 is shutdown
[Sun Jun  6 01:50:10 2004] - END        : process 24190 is shutdown
--------------------
```

First of all, we can clearly see that Apache always restart itself after the first *post_config* phase is over. The logs show that the *post_config* phase is preceded by the *open_logs* phase. Only after Apache has restarted itself and has completed the *open_logs* and *post_config* phase again, the *child_init* phase is run for each child process. In our example we have had the setting `StartServers=4`, therefore you can see four child processes were started.

Finally you can see that on server shutdown, the *child_exit* phase is run for each child process and the `END` `{}` block is executed by the parent process and each of the child processes. This is because that `END` block was inherited from the parent on fork.

However the presented behavior varies from MPM to MPM. This demonstration was performed using prefork mpm. Other MPMs like winnt, may run *open_logs* and *post_config* more than once. Also the END blocks may be run more times, when threads are involved. You should be very careful when designing features relying on the phases covered in this chapter if you plan support multiple MPMs. The only thing

that's sure is that you will have each of these phases run at least once.

Apache also specifies the *pre_config* phase, which is executed before the configuration files are parsed, but this is of no use to mod_perl, because mod_perl is loaded only during the configuration phase.

Now let's discuss each of the mentioned startup handlers and their implementation in the `MyApache2::StartupLog` module in detail.

12.2.2 *PerlOpenLogsHandler*

The *open_logs* phase happens just before the *post_config* phase.

Handlers registered by `PerlOpenLogsHandler` are usually used for opening module-specific log files (e.g., httpd core and mod_ssl open their log files during this phase).

At this stage the `STDERR` stream is not yet redirected to *error_log*, and therefore any messages to that stream will be printed to the console the server is starting from (if such exists).

This phase is of type `RUN_ALL`.

The handler's configuration scope is `SRV`.

Arguments

The *open_logs* handler is passed four arguments: the configuration pool, the logging stream pool, the temporary pool and the main server object.

The pool arguments are:

- `$conf_pool` is the main process sub-pool, therefore its life-span is the same as the main process's one. The main process is a sub-pool of the global pool.

- `$log_pool` is a global pool's sub-pool, therefore its life-span is the same as the Apache program's one.

  META: what is it good for if it lives the same life as conf pool?

- `$temp_pool` is a `$conf_pool` subpool, created before the config phase, lives through the open_logs phase and get destroyed after the post_config phase. So you will want to use that pool for doing anything that can be discarded before the requests processing starts.

All three pool arguments are instances of `APR::Pool`.

`$s` is the base server object (an instance of `Apache2::ServerRec`).

Return

The handler should return `Apache2::Const::OK` if it completes successfully.

Examples

```
sub open_logs {
    my ($conf_pool, $log_pool, $temp_pool, $s) = @_;

    $s->warn("opening the log file: $log_path");
    open $log_fh, ">>$log_path" or die "can't open $log_path: $!";
    my $oldfh = select($log_fh); $| = 1; select($oldfh);

    say("process $$ is born to reproduce");
    return Apache2::Const::OK;
}
```

In our example the handler opens a log file for appending and sets the filehandle to unbuffered mode. It then logs the fact that it's running in the parent process.

As you've seen in the example this handler is configured by adding to the top level of *httpd.conf*:

```
PerlOpenLogsHandler MyApache2::StartupLog::open_logs
```

This handler can be executed only by the main server. If you want to traverse the configured virtual hosts, you can accomplish that using a simple loop. For example to print out the configured port numbers do:

```
use Apache2::ServerRec ();
# ...
sub open_logs {
    my ($conf_pool, $log_pool, $temp_pool, $s) = @_;

    my $port = $s->port;
    warn "base port: $port\n";
    for (my $vs = $s->next; $vs; $vs = $vs->next) {
        my $port = $vs->port;
        warn "vhost port: $port\n";
    }
    return Apache2::Const::OK;
}
```

12.2.3 *PerlPostConfigHandler*

The *post_config* phase happens right after Apache has processed the configuration files, before any child processes were spawned (which happens at the *child_init* phase).

This phase can be used for initializing things to be shared between all child processes. You can do the same in the startup file, but in the *post_config* phase you have an access to a complete configuration tree (via `Apache2::Directive`).

This phase is of type `RUN_ALL`.

The handler's configuration scope is `SRV`.

Arguments

Arguments are exactly as for `PerlOpenLogsHandler`.

Return

If the handler completes successfully it should return `Apache2::Const::OK`.

Examples

In our `MyApache2::StartupLog` example we used the *post_config()* handler:

```
sub post_config {
    my ($conf_pool, $log_pool, $temp_pool, $s) = @_;
    say("configuration is completed");
    return Apache2::Const::OK;
}
```

As you can see, its arguments are identical to the *open_logs* phase's handler. In this example handler we don't do much, but logging that the configuration was completed and returning right away.

As you've seen in the example this handler is configured by adding to *httpd.conf*:

```
PerlPostConfigHandler MyApache2::StartupLog::post_config
```

Everything that applies to `PerlOpenLogsHandler` identically applies to this handler.

The `add_version_component()` includes another useful example.

12.2.4 `PerlChildInitHandler`

The *child_init* phase happens immediately after the child process is spawned. Each child process (not a thread!) will run the hooks of this phase only once in their life-time.

In the prefork MPM this phase is useful for initializing any data structures which should be private to each process. For example `Apache::DBI` pre-opens database connections during this phase and `Apache2::Resource` sets the process' resources limits.

This phase is of type `VOID`.

The handler's configuration scope is `SRV`.

Arguments

The *child_init()* handler is passed two arguments: the child process pool (`APR::Pool`) and the server object (`Apache2::ServerRec`).

Return

If the handler completes successfully it should return `Apache2::Const::OK`.

Examples

In our `MyApache2::StartupLog` example we used the *child_init()* handler:

```
sub child_init {
    my ($child_pool, $s) = @_;
    say("process $$ is born to serve");
    return Apache2::Const::OK;
}
```

The example handler logs the pid of the child process it's run in and returns.

As you've seen in the example this handler is configured by adding to *httpd.conf*:

```
PerlChildInitHandler  MyApache2::StartupLog::child_init
```

12.2.5 `PerlChildExitHandler`

Opposite to the *child_init* phase, the *child_exit* phase is executed before the child process exits. Notice that it happens only when the process exits, not the thread (assuming that you are using a threaded mpm).

This phase is of type `RUN_ALL`.

The handler's configuration scope is `SRV`.

Arguments

The *child_exit()* handler accepts two arguments: the child process pool (`APR::Pool`) and the server object (`Apache2::ServerRec`).

Return

If the handler completes successfully it should return `Apache2::Const::OK`.

Examples

In our `MyApache2::StartupLog` example we used the *child_exit()* handler:

```
sub child_exit {
    my ($child_pool, $s) = @_;
    say("process $$ now exits");
    return Apache2::Const::OK;
}
```

The example handler logs the pid of the child process it's run in and returns.

As you've seen in the example this handler is configured by adding to *httpd.conf*:

```
PerlChildExitHandler  MyApache2::StartupLog::child_exit
```

12.3 Apache Command-line Commands

Some notes on how Apache start/restart Apache commands affect mod_perl.

META: not sure this is the best place for this section, but start some notes here.

Apache re-parses *httpd.conf* at least once for **each** of the following commands (and will run any mod_perl code found in it).

- **httpd -k start**

    No special issues here.

    Apache start and immediately restarts itself.

- **httpd -k restart**

    This will abort any processed requests and restart the server.

    All kind of problems could be encountered here, including segfaults and other kind of crashes. This is because when the `SIGTERM` signal is sent, things in process will be aborted.

    Avoid using this method.

    Alternatively `httpd -k restart` can be executed `kill -HUP HTTPD_PID`.

- **httpd -k graceful**

    No issues here. Apache starts and restarts itself just like with `start`, but it waits for the existing requests to finish before killing them.

    Alternatively `httpd -k graceful` can be executed `kill -USR1 HTTPD_PID`.

- **httpd -k stop**

    Similarly to `httpd -k restart` you may encounter all kind of issues here, due to the `SIGTERM` signal.

12.4 mod_perl Startup

The following sections discuss the specifics of the mod_perl startup.

12.4.1 Start Immediately Restarts

As explained in the Server Life Cycle section, on start Apache normally runs the server configuration phase, followed by `PerlOpenLogsHandler` and `PerlPostConfigHandler` phases, then immediately restarts itself. Therefore everything running at the server startup is executed twice. During the restart, Perl is completely destroyed and started again.

12.4.2 When Does perl Start To Run

If Apache is started as 'httpd -t' (equivalent to 'apachectl configtest') or as 'httpd -S', it will run only the configuration phase and exit. Depending on your configuration file, it may or may not start perl. See the details below.

During the normal startup, mod_perl 2.0 postpones the startup of perl until after the configuration phase is over, to allow the usage of the `PerlSwitches` directive, which can't be used after Perl is started.

After the configuration phase is over, as the very first thing during the `post_config phase`, mod_perl starts perl and runs any registered `PerlRequire` and `PerlModule` entries.

At the very end of the `post_config phase` any registrered `PerlPostConfigRequire` entries are run.

When any of the following configuration directives is encountered (during the configuration phase) mod_perl 2.0 is forced to start as soon as they are encountered (as these options require a running perl):

- `PerlLoadModule`

- `<Perl> section`

- `PerlConfigRequire`

Therefore if you want to trigger an early Perl startup, you could add an empty `<Perl> section` in *httpd.conf*:

```
<Perl>
# trigger an early Perl startup
</Perl>
```

right after loading the mod_perl module, if you are using DSO, or just before your mod_perl configuration section, if you're using a static mod_perl build. But most likely you want to use the `PerlConfigRequire` instead.

12.4.3 Startup File

A startup file with Perl code to be executed at the server startup can be loaded using `PerlPostConfigRequire`. For example:

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

It's used to adjust Perl modules search paths in `@INC`, pre-load commonly used modules, pre-compile constants, etc. Here is a typical *startup.pl* for mod_perl 2.0:

```
#file:startup.pl
#---------------

use lib qw(/home/httpd/perl);

# enable if the mod_perl 1.0 compatibility is needed
# use Apache2::compat ();

# preload all mp2 modules
# use ModPerl::MethodLookup;
# ModPerl::MethodLookup::preload_all_modules();

use ModPerl::Util (); #for CORE::GLOBAL::exit

use Apache2::RequestRec ();
use Apache2::RequestIO ();
use Apache2::RequestUtil ();

use Apache2::ServerRec ();
use Apache2::ServerUtil ();
use Apache2::Connection ();
use Apache2::Log ();

use APR::Table ();

use ModPerl::Registry ();

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

1;
```

In this file `@INC` in adjusted to include non-standard directories with Perl modules:

```
use lib qw(/home/httpd/perl);
```

If you need to use the backwards compatibility layer load:

```
use Apache2::compat ();
```

Next we preload the commonly used mod_perl 2.0 modules and precompile common constants.

Finally as usual the *startup.pl* file must be terminated with `1;`.

12.4.4 Dealing with Restarts

Ideally the code running at the server startup shouldn't be affected by the apache restart. If however this is not the case, you can use `Apache2::ServerUtil::restart_count`.

12.5 Maintainers

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

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

12.6 Authors

-

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

13   Protocol Handlers

13.1 Description

This chapter explains how to implement Protocol (Connection) Handlers in mod_perl.

13.2 Connection Cycle Phases

As we saw earlier, each child server (be it a thread or a process) is engaged in processing connections. Each connection may be served by different connection protocols, e.g., HTTP, POP3, SMTP, etc. Each connection may include more than one request, e.g., several HTTP requests can be served over a single connection, when several images are requested for the same webpage.

The following diagram depicts the connection life cycle and highlights which handlers are available to mod_perl 2.0:

connection cycle

When a connection is issued by a client, it's first run through `PerlPreConnectionHandler` and then passed to the `PerlProcessConnectionHandler`, which generates the response. When `PerlProcessConnectionHandler` is reading data from the client, it can be filtered by connection input filters. The generated response can be also filtered though connection output filters. Filters are usually used for modifying the data flowing though them, but can be used for other purposes as well (e.g., logging interesting information). For example the following diagram shows the connection cycle mapped to the time scale:

connection cycle timing

The arrows show the program control. In addition, the black-headed arrows also show the data flow. This diagram matches an interactive protocol, where a client send something to the server, the server filters the input, processes it and send it out through output filters. This cycle is repeated till the client or the server don't tell each other to go away or abort the connection. Before the cycle starts any registered pre_connection handlers are run.

Now let's discuss each of the `PerlPreConnectionHandler` and `PerlProcessConnection-Handler` handlers in detail.

13.2.1 PerlPreConnectionHandler

The *pre_connection* phase happens just after the server accepts the connection, but before it is handed off to a protocol module to be served. It gives modules an opportunity to modify the connection as soon as possible and insert filters if needed. The core server uses this phase to setup the connection record based on the type of connection that is being used. mod_perl itself uses this phase to register the connection input and output filters.

In mod_perl 1.0 during code development `Apache::Reload` was used to automatically reload modified since the last request Perl modules. It was invoked during `post_read_request`, the first HTTP request's phase. In mod_perl 2.0 *pre_connection* is the earliest phase, so if we want to make sure that all

modified Perl modules are reloaded for any protocols and its phases, it's the best to set the scope of the Perl interpreter to the lifetime of the connection via:

```
PerlInterpScope connection
```

and invoke the `Apache2::Reload` handler during the *pre_connection* phase. However this develop-ment-time advantage can become a disadvantage in production--for example if a connection, handled by HTTP protocol, is configured as `KeepAlive` and there are several requests coming on the same connec-tion and only one handled by mod_perl and the others by the default images handler, the Perl interpreter won't be available to other threads while the images are being served.

This phase is of type `RUN_ALL`.

The handler's configuration scope is `SRV`, because it's not known yet which resource the request will be mapped to.

Arguments

A *pre_connection* handler is passed a `connection record` as its argument:

```
sub handler {
    my $c = shift;
    # ...
    return Apache2::Const::OK;
}
```

[META: There is another argument passed (the actual client socket), but it is currently an undef]

Return

The handler should return `Apache2::Const::OK` if it completes successfully or `Apache2::Const::FORBIDDEN` if the request is forbidden.

Examples

Here is a useful *pre_connection* phase example: provide a facility to block remote clients by their IP, before too many resources were consumed. This is almost as good as a firewall blocking, as it's executed before Apache has started to do any work at all.

`MyApache2::BlockIP2` retrieves client's remote IP and looks it up in the black list (which should certainly live outside the code, e.g. dbm file, but a hardcoded list is good enough for our example).

```
#file:MyApache2/BlockIP2.pm
#------------------------
package MyApache2::BlockIP2;

use strict;
use warnings;

use Apache2::Connection ();

use Apache2::Const -compile => qw(FORBIDDEN OK);
```

```
my %bad_ips = map {$_ => 1} qw(127.0.0.1 10.0.0.4);

sub handler {
    my Apache2::Connection $c = shift;

    my $ip = $c->remote_ip;
    if (exists $bad_ips{$ip}) {
        warn "IP $ip is blocked\n";
        return Apache2::Const::FORBIDDEN;
    }

    return Apache2::Const::OK;
}

1;
```

This all happens during the *pre_connection* phase:

```
PerlPreConnectionHandler MyApache2::BlockIP2
```

If a client connects from a blacklisted IP, Apache will simply abort the connection without sending any reply to the client, and move on to serving the next request.

13.2.2 *PerlProcessConnectionHandler*

The *process_connection* phase is used to process incoming connections. Only protocol modules should assign handlers for this phase, as it gives them an opportunity to replace the standard HTTP processing with processing for some other protocols (e.g., POP3, FTP, etc.).

This phase is of type RUN_FIRST.

The handler's configuration scope is SRV. Therefore the only way to run protocol servers different than the core HTTP is inside dedicated virtual hosts.

Arguments

A *process_connection* handler is passed a connection record object as its only argument.

A socket object can be retrieved from the connection record object.

Return

The handler should return Apache2::Const::OK if it completes successfully.

Examples

Here is a simplified handler skeleton:

```
sub handler {
    my ($c) = @_;
    my $sock = $c->client_socket;
    $sock->opt_set(APR::Const::SO_NONBLOCK, 0);
    # ...
    return Apache2::Const::OK;
}
```

Most likely you'll need to set the socket to perform blocking IO. On some platforms (e.g. Linux) Apache gives us a socket which is set for blocking, on other platforms (.e.g. Solaris) it doesn't. Unless you know which platforms your application will be running on, always explicitly set it to the blocking IO mode as in the example above. Alternatively, you could query whether the socket is already set to a blocking IO mode with help of the opt_get() method.

Now let's look at the following two examples of connection handlers. The first using the connection socket to read and write the data and the second using bucket brigades to accomplish the same and allow for connection filters to do their work.

13.2.2.1 Socket-based Protocol Module

To demonstrate the workings of a protocol module, we'll take a look at the MyApache2::EchoSocket module, which simply echoes the data read back to the client. In this module we will use the implementation that works directly with the connection socket and therefore bypasses connection filters if any.

A protocol handler is configured using the PerlProcessConnectionHandler directive and we will use the Listen and <VirtualHost> directives to bind to the non-standard port **8010**:

```
Listen 8010
<VirtualHost _default_:8010>
    PerlModule                    MyApache2::EchoSocket
    PerlProcessConnectionHandler MyApache2::EchoSocket
</VirtualHost>
```

MyApache2::EchoSocket is then enabled when starting Apache:

```
panic% httpd
```

And we give it a whirl:

```
panic% telnet localhost 8010
Trying 127.0.0.1...
Connected to localhost (127.0.0.1).
Escape character is '^]'.
Hello
Hello

fOo BaR
fOo BaR

Connection closed by foreign host.
```

Here is the code:

```
#file:MyApache2/EchoSocket.pm
#---------------------------
package MyApache2::EchoSocket;

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

use Apache2::Connection ();
use APR::Socket ();

use Apache2::Const -compile => 'OK';
use APR::Const     -compile => 'SO_NONBLOCK';

use constant BUFF_LEN => 1024;

sub handler {
    my $c = shift;
    my $sock = $c->client_socket;

    # set the socket to the blocking mode
    $sock->opt_set(APR::Const::SO_NONBLOCK => 0);

    while ($sock->recv(my $buff, BUFF_LEN)) {
        last if $buff =~ /^[\r\n]+$/;
        $sock->send($buff);
    }

    Apache2::Const::OK;
}
1;
```

The example handler starts with the standard *package* declaration and of course, `use strict;`. As with all `Perl*Handlers`, the subroutine name defaults to *handler*. However, in the case of a protocol handler, the first argument is not a `request_rec`, but a `conn_rec` blessed into the `Apache2::Connection` class. We have direct access to the client socket via `Apache2::Connection`'s *client_socket* method. This returns an object, blessed into the `APR::Socket` class. Before using the socket, we make sure that it's set to perform blocking IO, by using the `APR::Const::SO_NONBLOCK` constant, compiled earlier.

Inside the recv/send loop, the handler attempts to read `BUFF_LEN` bytes from the client socket into the `$buff` buffer. The handler breaks the loop if nothing was read (EOF) or if the buffer contains nothing but new line character(s), which is how we know to abort the connection in the interactive mode.

If the handler receives some data, it sends it unmodified back to the client with the `APR::Socket::send()` method. When the loop is finished the handler returns `Apache2::Const::OK`, telling Apache to terminate the connection. As mentioned earlier since this handler is working directly with the connection socket, no filters can be applied.

13.2.2.2 Bucket Brigades-based Protocol Module

Now let's look at the same module, but this time implemented by manipulating bucket brigades, and which runs its output through a connection output filter that turns all uppercase characters into their lowercase equivalents.

The following configuration defines a virtual host listening on port 8011 and which enables the `MyApache2::EchoBB` connection handler, which will run its output through `MyApache2::EchoBB::lowercase_filter` filter:

```
Listen 8011
<VirtualHost _default_:8011>
    PerlModule                  MyApache2::EchoBB
    PerlProcessConnectionHandler MyApache2::EchoBB
    PerlOutputFilterHandler     MyApache2::EchoBB::lowercase_filter
</VirtualHost>
```

As before we start the httpd server:

```
panic% httpd
```

And try the new connection handler in action:

```
panic% telnet localhost 8011
Trying 127.0.0.1...
Connected to localhost (127.0.0.1).
Escape character is '^]'.
Hello
hello

fOo BaR
foo bar

Connection closed by foreign host.
```

As you can see the response part this time was all in lower case, because of the output filter.

And here is the implementation of the connection and the filter handlers.

```
#file:MyApache2/EchoBB.pm
#-----------------------
package MyApache2::EchoBB;

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

use Apache2::Connection ();
use APR::Socket ();
use APR::Bucket ();
use APR::Brigade ();
use APR::Error ();
use APR::Status ();

use APR::Const     -compile => qw(SUCCESS SO_NONBLOCK);
```

```
use Apache2::Const -compile => qw(OK MODE_GETLINE);

sub handler {
    my $c = shift;

    $c->client_socket->opt_set(APR::Const::SO_NONBLOCK => 0);

    my $bb_in  = APR::Brigade->new($c->pool, $c->bucket_alloc);
    my $bb_out = APR::Brigade->new($c->pool, $c->bucket_alloc);

    my $last = 0;
    while (1) {
        my $rc = $c->input_filters->get_brigade($bb_in,
                                                Apache2::Const::MODE_GETLINE);
        last if APR::Status::is_EOF($rc);
        die APR::Error::strerror($rc) unless $rc == APR::Const::SUCCESS;

        while (!$bb_in->is_empty) {
            my $b = $bb_in->first;

            $b->remove;

            if ($b->is_eos) {
                $bb_out->insert_tail($b);
                last;

            }

            if ($b->read(my $data)) {
                $last++ if $data =~ /^[\r\n]+$/;
                # could do some transformation on data here
                $b = APR::Bucket->new($bb_out->bucket_alloc, $data);
            }

            $bb_out->insert_tail($b);
        }

        my $fb = APR::Bucket::flush_create($c->bucket_alloc);
        $bb_out->insert_tail($fb);
        $c->output_filters->pass_brigade($bb_out);
        last if $last;
    }

    $bb_in->destroy;
    $bb_out->destroy;

    Apache2::Const::OK;
}

use base qw(Apache2::Filter);
use constant BUFF_LEN => 1024;

sub lowercase_filter : FilterConnectionHandler {
    my $filter = shift;

    while ($filter->read(my $buffer, BUFF_LEN)) {
        $filter->print(lc $buffer);
```

```
    }

    return Apache2::Const::OK;
}

1;
```

For the purpose of explaining how this connection handler works, we are going to simplify the handler. The whole handler can be represented by the following pseudo-code:

```
while ($bb_in = get_brigade()) {
    while ($b_in = $bb_in->get_bucket()) {
        $b_in->read(my $data);
        # do something with data
        $b_out = new_bucket($data);

        $bb_out->insert_tail($b_out);
    }
    $bb_out->insert_tail($flush_bucket);
    pass_brigade($bb_out);
}
```

The handler receives the incoming data via bucket bridges, one at a time in a loop. It then process each bridge, by retrieving the buckets contained in it, reading the data in, then creating new buckets using the received data, and attaching them to the outgoing brigade. When all the buckets from the incoming bucket brigade were transformed and attached to the outgoing bucket brigade, a flush bucket is created and added as the last bucket, so when the outgoing bucket brigade is passed out to the outgoing connection filters, it won't be buffered but sent to the client right away.

It's possible to make the flushing code simpler, by using a dedicated method `fflush()` that does just that -- flushing of the bucket brigade. It replaces 3 lines of code:

```
        my $fb = APR::Bucket::flush_create($c->bucket_alloc);
        $bb_out->insert_tail($fb);
        $c->output_filters->pass_brigade($bb_out);
```

with just one line:

```
        $c->output_filters->fflush($bb_out);
```

If you look at the complete handler, the loop is terminated when one of the following conditions occurs: an error happens, the end of stream status code (`EOF`) has been received (no more input at the connection) or when the received data contains nothing but new line characters which we used to to tell the server to terminate the connection.

Now that you've learned how to move buckets from one brigade to another, let's see how the presented handler can be reimplemented using a single bucket brigade. Here is the modified code:

```
sub handler {
    my $c = shift;

    $c->client_socket->opt_set(APR::Const::SO_NONBLOCK, 0);
```

```
      my $bb = APR::Brigade->new($c->pool, $c->bucket_alloc);

      while (1) {
          my $rc = $c->input_filters->get_brigade($bb,
                                                  Apache2::Const::MODE_GETLINE);
          last if APR::Status::is_EOF($rc);
          die APR::Error::strerror($rc) unless $rc == APR::Const::SUCCESS;

          for (my $b = $bb->first; $b; $b = $bb->next($b)) {

              last if $b->is_eos;

              if ($b->read(my $data)) {
                  last if $data =~ /^[\r\n]+$/;
                  my $nb = APR::Bucket->new($bb->bucket_alloc, $data);
                  # head->...->$nb->$b ->...->tail
                  $b->insert_before($nb);
                  $b->remove;
              }
          }

          $c->output_filters->fflush($bb);
      }

      $bb->destroy;

      Apache2::Const::OK;
  }
```

This code is shorter and simpler. Since it sends out the same bucket brigade it got from the incoming filters, it only needs to replace buckets that get modified, which is probably the only tricky part here. The code:

```
                    # head->...->$nb->$b ->...->tail
                    $b->insert_before($nb);
                    $b->remove;
```

inserts a new bucket in front of the currently processed bucket, so that when the latter removed the former takes place of the latter.

Notice that this handler could be much simpler, since we don't modify the data. We could simply pass the whole brigade unmodified without even looking at the buckets. But from this example you can see how to write a connection handler where you actually want to read and/or modify the data. To accomplish that modification simply add a code that transforms the data which has been read from the bucket before it's inserted to the outgoing brigade.

We will skip the filter discussion here, since we are going to talk in depth about filters in the dedicated to filters tutorial. But all you need to know at this stage is that the data sent from the connection handler is filtered by the outgoing filter and which transforms it to be all lowercase.

And here is the simplified version of this handler, which doesn't attempt to do any transformation, but simply passes the data though:

```
sub handler {
    my $c = shift;

    $c->client_socket->opt_set(APR::Const::SO_NONBLOCK => 0);

    my $bb = APR::Brigade->new($c->pool, $c->bucket_alloc);

    while (1) {
        my $rc = $c->input_filters->get_brigade($bb,
                                           Apache2::Const::MODE_GETLINE);
        last if APR::Status::is_EOF($rc);
        die APR::Error::strerror($rc) unless $rc == APR::Const::SUCCESS;

        $c->output_filters->fflush($bb);
    }

    $bb->destroy;

    Apache2::Const::OK;
}
```

Since the simplified handler no longer has the condition:

```
$last++ if $data =~ /^[\r\n]+$/;
```

which was used to know when to break from the external `while(1)` loop, it will not work in the interactive mode, because when telnet is used we always end the line with `/[\r\n]/`, which will always send data back to the protocol handler and the condition:

```
last if $bb->is_empty;
```

will never be true. However, this latter version works fine when the client is a script and when it stops sending data, our shorter handler breaks out of the loop.

So let's do one more tweak and make the last version work in the interactive telnet mode without manipulating each bucket separately. This time we will use `flatten()` to slurp all the data from all the buckets, which saves us the explicit loop over the buckets in the brigade. The handler now becomes:

```
sub handler {
    my $c = shift;

    $c->client_socket->opt_set(APR::Const::SO_NONBLOCK => 0);

    my $bb = APR::Brigade->new($c->pool, $c->bucket_alloc);

    while (1) {
        my $rc = $c->input_filters->get_brigade($bb,
                                           Apache2::Const::MODE_GETLINE);
        last if APR::Status::is_EOF($rc);
        die APR::Error::strerror($rc) unless $rc == APR::Const::SUCCESS;

        next unless $bb->flatten(my $data);
        $bb->cleanup;
        last if $data =~ /^[\r\n]+$/;
```

```
        # could transform data here
        my $b = APR::Bucket->new($bb->bucket_alloc, $data);
        $bb->insert_tail($b);

        $c->output_filters->fflush($bb);
    }

    $bb->destroy;

    Apache2::Const::OK;
}
```

Notice, that once we slurped the data in the buckets, we had to strip the brigade of its buckets, since we re-used the same brigade to send the data out. We used `cleanup()` to get rid of the buckets.

13.3 Examples

Following are some practical examples.

META: If you have written an interesting, but not too complicated module, which others can learn from, please submit a pod to the mailing list so we can include it here.

13.3.1 Command Server

The `MyApache2::CommandServer` example is based on the example in the "TCP Servers with IO::Socket" section of the *perlipc* manpage. Of course, we don't need `IO::Socket` since Apache takes care of those details for us. The rest of that example can still be used to illustrate implementing a simple text protocol. In this case, one where a command is sent by the client to be executed on the server side, with results sent back to the client.

The `MyApache2::CommandServer` handler will support four commands: `motd`, `date`, `who` and `quit`. These are probably not commands which can be exploited, but should we add such commands, we'll want to limit access based on ip address/hostname, authentication and authorization. Protocol handlers need to take care of these tasks themselves, since we bypass the HTTP protocol handler.

Here is the whole module:

```
package MyApache2::CommandServer;

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

use Apache2::Connection ();
use Apache2::RequestRec ();
use Apache2::RequestUtil ();
use Apache2::HookRun ();
use Apache2::Access ();
use APR::Socket ();

use Apache2::Const -compile => qw(OK DONE DECLINED);
```

```perl
my @cmds = qw(motd date who quit);
my %commands = map { $_, \&{$_} } @cmds;

sub handler {
    my $c = shift;
    my $socket = $c->client_socket;

    if ((my $rc = login($c)) != Apache2::Const::OK) {
        $socket->send("Access Denied\n");
        return $rc;
    }

    $socket->send("Welcome to " . __PACKAGE__ .
                  "\nAvailable commands: @cmds\n");

    while (1) {
        my $cmd;
        next unless $cmd = getline($socket);

        if (my $sub = $commands{$cmd}) {
            last unless $sub->($socket) == Apache2::Const::OK;
        }
        else {
            $socket->send("Commands: @cmds\n");
        }
    }


    return Apache2::Const::OK;
}

sub login {
    my $c = shift;

    my $r = Apache2::RequestRec->new($c);
    $r->location_merge(__PACKAGE__);

    for my $method (qw(run_access_checker run_check_user_id
                       run_auth_checker)) {
        my $rc = $r->$method();

        if ($rc != Apache2::Const::OK and $rc != Apache2::Const::DECLINED) {
            return $rc;
        }

        last unless $r->some_auth_required;

        unless ($r->user) {
            my $socket = $c->client_socket;
            my $username = prompt($socket, "Login");
            my $password = prompt($socket, "Password");

            $r->set_basic_credentials($username, $password);
        }
    }

    return Apache2::Const::OK;
```

```
    }

    sub getline {
        my $socket = shift;

        my $line;
        $socket->recv($line, 1024);
        return unless $line;
        $line =~ s/[\r\n]*$//;

        return $line;
    }

    sub prompt {
        my ($socket, $msg) = @_;

        $socket->send("$msg: ");
        getline($socket);
    }

    sub motd {
        my $socket = shift;

        open my $fh, '/etc/motd' or return;
        local $/;
        $socket->send(scalar <$fh>);
        close $fh;

        return Apache2::Const::OK;
    }

    sub date {
        my $socket = shift;

        $socket->send(scalar(localtime) . "\n");

        return Apache2::Const::OK;
    }

    sub who {
        my $socket = shift;

        # make -T happy
        local $ENV{PATH} = "/bin:/usr/bin";

        $socket->send(scalar 'who');

        return Apache2::Const::OK;
    }

    sub quit { Apache2::Const::DONE }

    1;
    __END__
```

Next, let's explain what this module does in details.

As with all `PerlProcessConnectionHandlers`, we are passed an `Apache2::Connection` object as the first argument. Again, we will be directly accessing the client socket via the *client_socket* method. The *login* subroutine is called to check if access by this client should be allowed. This routine makes up for what we lost with the core HTTP protocol handler bypassed.

First we call the `Apache2::RequestRec new()` method, which returns a *request_rec* object, just like that, which is passed at request time to HTTP protocol `Perl*Handlers` and returned by the subrequest API methods, *lookup_uri* and *lookup_file*. However, this "fake request" does not run handlers for any of the phases, it simply returns an object which we can use to do that ourselves.

The `location_merge()` method is passed the `location` for this request, it will look up the `<Location>` section that matches the given name and merge it with the default server configuration. For example, should we only wish to allow access to this server from certain locations:

```
<Location MyApache2::CommandServer>
    Order Deny,Allow
    Deny from all
    Allow from 10.*
</Location>
```

The `location_merge()` method only looks up and merges the configuration, we still need to apply it. This is done in *for* loop, iterating over three methods: `run_access_checker()`, `run_check_user_id()` and `run_auth_checker()`. These methods will call directly into the Apache functions that invoke module handlers for these phases and will return an integer status code, such as `Apache2::Const::OK`, `Apache2::Const::DECLINED` or `Apache2::Const::FORBID-DEN`. If *run_access_check* returns something other than `Apache2::Const::OK` or `Apache2::Const::DECLINED`, that status will be propagated up to the handler routine and then back up to Apache. Otherwise, the access check passed and the loop will break unless `some_auth_required()` returns true. This would be false given the previous configuration example, but would be true in the presence of a `require` directive, such as:

```
<Location MyApache2::CommandServer>
    Order Deny,Allow
    Deny from all
    Allow from 10.*
    Require user dougm
</Location>
```

Given this configuration, `some_auth_required()` will return true. The `user()` method is then called, which will return false if we have not yet authenticated. A `prompt()` utility is called to read the username and password, which are then injected into the `headers_in()` table using the `set_basic_credentials()` method. The *Authenticate* field in this table is set to a *base64* encoded value of the username:password pair, exactly the same format a browser would send for *Basic authentication*. Next time through the loop *run_check_user_id* is called, which will in turn invoke any authentication handlers, such as *mod_auth*. When *mod_auth* calls the `ap_get_basic_auth_pw()` API function (as all `Basic` auth modules do), it will get back the username and password we injected. If we fail authentication a `401` status code is returned which we propagate up. Otherwise, authorization handlers are run via `run_auth_checker()`. Authorization handlers normally need the *user* field of the `request_rec`

for its checks and that field was filled in when *mod_auth* called `ap_get_basic_auth_pw()`.

Provided login is a success, a welcome message is printed and main request loop entered. Inside the loop the `getline()` function returns just one line of data, with newline characters stripped. If the string sent by the client is in our command table, the command is then invoked, otherwise a usage message is sent. If the command does not return `Apache2::Const::OK`, we break out of the loop.

Let's use this configuration:

```
Listen 8085
<VirtualHost _default_:8085>
    PerlProcessConnectionHandler MyApache2::CommandServer

    <Location MyApache2::CommandServer>
        Order Deny,Allow
        Allow from 127.0.0.1
        Require user dougm
        Satisfy any
        AuthUserFile /tmp/basic-auth
    </Location>
</VirtualHost>
```

Since we are using `mod_auth` directives here, you need to make sure that it's available and loaded for this example to work as explained.

The auth file can be created with the help of `htpasswd` utility coming bundled with the Apache server. For example to create a file */tmp/basic-auth* and add a password entry for user *dougm* with password *foobar* we do:

```
% htpasswd -bc /tmp/basic-auth dougm foobar
```

Now we are ready to try the command server:

```
% telnet localhost 8085
Trying 127.0.0.1...
Connected to localhost (127.0.0.1).
Escape character is '^]'.
Login: dougm
Password: foobar
Welcome to MyApache2::CommandServer
Available commands: motd date who quit
motd
Have a lot of fun...
date
Mon Mar 12 19:20:10 PST 2001
who
dougm    tty1     Mar 12 00:49
dougm    pts/0    Mar 12 11:23
dougm    pts/1    Mar 12 14:08
dougm    pts/2    Mar 12 17:09
quit
Connection closed by foreign host.
```

13.4 CPAN Modules

Some of the CPAN modules that implement mod_perl 2.0 protocols:

- **`Apache::SMTP` - An SMTP server**

  http://search.cpan.org/dist/Apache-SMTP/

13.5 Maintainers

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

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

13.6 Authors

-

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

14 HTTP Handlers

14.1 Description

This chapter explains how to implement the HTTP protocol handlers in mod_perl.

14.2 HTTP Request Handler Skeleton

All HTTP Request handlers have the following structure:

```
package MyApache2::MyHandlerName;

# load modules that are going to be used
use ...;

# compile (or import) constants
use Apache2::Const -compile => qw(OK);

sub handler {
    my $r = shift;

    # handler code comes here

    return Apache2::Const::OK; # or another status constant
}
1;
```

First, the package is declared. Next, the modules that are going to be used are loaded and constants compiled.

The handler itself coming next and usually it receives the only argument: the `Apache2::RequestRec` object. If the handler is declared as a method handler :

```
sub handler : method {
    my ($class, $r) = @_;
```

the handler receives two arguments: the class name and the `Apache2::RequestRec` object.

The handler ends with a return code and the file is ended with `1;` to return true when it gets loaded.

14.3 HTTP Request Cycle Phases

Those familiar with mod_perl 1.0 will find the HTTP request cycle in mod_perl 2.0 to be almost identical to the mod_perl 1.0's model. The different things are:

- a new directive `PerlMapToStorageHandler` was added to match the new phase *map_to_storage* added by Apache 2.0.

- the `PerlHandler` directive has been renamed to `PerlResponseHandler` to better match the corresponding Apache phase name (*response*).

- the *response* phase now includes filtering.

The following diagram depicts the HTTP request life cycle and highlights which handlers are available to mod_perl 2.0:

HTTP cycle

From the diagram it can be seen that an HTTP request is processed by 12 phases, executed in the following order:

1. **PerlPostReadRequestHandler (PerlInitHandler)**
2. **PerlTransHandler**
3. **PerlMapToStorageHandler**
4. **PerlHeaderParserHandler (PerlInitHandler)**
5. **PerlAccessHandler**
6. **PerlAuthenHandler**
7. **PerlAuthzHandler**
8. **PerlTypeHandler**
9. **PerlFixupHandler**
10. **PerlResponseHandler**
11. **PerlLogHandler**
12. **PerlCleanupHandler**

It's possible that the cycle will not be completed if any of the phases terminates it, usually when an error happens. In that case Apache skips to the logging phase (mod_perl executes all registered `PerlLogHandler` handlers) and finally the cleanup phase happens.

Notice that when the response handler is reading the input data it can be filtered through request input filters, which are preceded by connection input filters if any. Similarly the generated response is first run through request output filters and eventually through connection output filters before it's sent to the client. We will talk about filters in detail later in the dedicated to filters chapter.

Before discussing each handler in detail remember that if you use the stacked handlers feature all handlers in the chain will be run as long as they return `Apache2::Const::OK` or `Apache2::Const::DECLINED`. Because stacked handlers is a special case. So don't be surprised if you've returned `Apache2::Const::OK` and the next handler was still executed. This is a feature, not a bug.

Now let's discuss each of the mentioned handlers in detail.

14.3.1 *PerlPostReadRequestHandler*

The *post_read_request* phase is the first request phase and happens immediately after the request has been read and HTTP headers were parsed.

This phase is usually used to do processing that must happen once per request. For example `Apache2::Reload` is usually invoked at this phase to reload modified Perl modules.

This phase is of type `RUN_ALL`.

The handler's configuration scope is `SRV`, because at this phase the request has not yet been associated with a particular filename or directory.

Arguments

See the HTTP Request Handler Skeleton for a description of handler arguments.

Return

See Stacked Handlers for a description of handler return codes.

Examples

Now, let's look at an example. Consider the following registry script:

```
#file:touch.pl
#-------------
use strict;
use warnings;

use Apache2::ServerUtil ();
use Apache2::RequestIO ();
use File::Spec::Functions qw(catfile);

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

my $conf_file = catfile Apache2::ServerUtil::server_root,
  "conf", "httpd.conf";

printf "$conf_file is %0.2f minutes old\n", 60*24*(-M $conf_file);
```

This registry script is supposed to print when the last time *httpd.conf* has been modified, compared to the start of the request process time. If you run this script several times you might be surprised that it reports the same value all the time. Unless the request happens to be served by a recently started child process which will then report a different value. But most of the time the value won't be reported correctly.

This happens because the `-M` operator reports the difference between file's modification time and the value of a special Perl variable `$^T`. When we run scripts from the command line, this variable is always set to the time when the script gets invoked. Under mod_perl this variable is getting preset once when the child process starts and doesn't change since then, so all requests see the same time, when operators like `-M`, `-C` and `-A` are used.

Armed with this knowledge, in order to make our code behave similarly to the command line programs we need to reset `$^T` to the request's start time, before `-M` is used. We can change the script itself, but what if we need to do the same change for several other scripts and handlers? A simple `PerlPostRead-`

`RequestHandler` handler, which will be executed as the very first thing of each requests, comes handy here:

```
#file:MyApache2/TimeReset.pm
#--------------------------
package MyApache2::TimeReset;

use strict;
use warnings;

use Apache2::RequestRec ();

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

sub handler {
    my $r = shift;
    $^T = $r->request_time;
    return Apache2::Const::OK;
}
1;
```

We could do:

```
$^T = time();
```

But to make things more efficient we use `$r->request_time` since the request object `$r` already stores the request's start time, so we get it without performing an additional system call.

To enable it just add to *httpd.conf*:

```
PerlPostReadRequestHandler MyApache2::TimeReset
```

either to the global section, or to the `<VirtualHost>` section if you want this handler to be run only for a specific virtual host.

14.3.2 PerlTransHandler

The *translate* phase is used to perform the manipulation of a request's URI. If no custom handler is provided, the server's standard translation rules (e.g., `Alias` directives, mod_rewrite, etc.) will be used. A `PerlTransHandler` handler can alter the default translation mechanism or completely override it. This is also a good place to register new handlers for the following phases based on the URI. `PerlMap-ToStorageHandler` is to be used to override the URI to filename translation.

This phase is of type `RUN_FIRST`.

The handler's configuration scope is `SRV`, because at this phase the request has not yet been associated with a particular filename or directory.

Arguments

See the HTTP Request Handler Skeleton for a description of handler arguments.

Return

See Stacked Handlers for a description of handler return codes.

Examples

There are many useful things that can be performed at this stage. Let's look at the example handler that rewrites request URIs, similar to what mod_rewrite does. For example, if your web-site was originally made of static pages, and now you have moved to a dynamic page generation chances are that you don't want to change the old URIs, because you don't want to break links for those who link to your site. If the URI:

```
http://example.com/news/20021031/09/index.html
```

is now handled by:

```
http://example.com/perl/news.pl?date=20021031;id=09;page=index.html
```

the following handler can do the rewriting work transparent to *news.pl*, so you can still use the former URI mapping:

```
#file:MyApache2/RewriteURI.pm
#--------------------------
package MyApache2::RewriteURI;

use strict;
use warnings;

use Apache2::RequestRec ();

use Apache2::Const -compile => qw(DECLINED);

sub handler {
    my $r = shift;

    my ($date, $id, $page) = $r->uri =~ m|^/news/(\d+)/(\d+)/(.*)|;
    $r->uri("/perl/news.pl");
    $r->args("date=$date;id=$id;page=$page");

    return Apache2::Const::DECLINED;
}
1;
```

The handler matches the URI and assigns a new URI via $r->uri() and the query string via $r->args(). It then returns Apache2::Const::DECLINED, so the next translation handler will get invoked, if more rewrites and translations are needed.

Of course if you need to do a more complicated rewriting, this handler can be easily adjusted to do so.

To configure this module simply add to *httpd.conf*:

```
PerlTransHandler +MyApache2::RewriteURI
```

14.3.3 *PerlMapToStorageHandler*

The *map_to_storage* phase is used to perform the translation of a request's URI into a corresponding file-name. If no custom handler is provided, the server will try to walk the filesystem trying to find what file or directory corresponds to the request's URI. Since usually mod_perl handler don't have corresponding files on the filesystem, you will want to shortcut this phase and save quite a few CPU cycles.

This phase is of type `RUN_FIRST`.

The handler's configuration scope is `SRV`, because at this phase the request has not yet been associated with a particular filename or directory.

Arguments

See the HTTP Request Handler Skeleton for a description of handler arguments.

Return

See Stacked Handlers for a description of handler return codes.

Examples

For example if you don't want Apache to try to attempt to translate URI into a filename, just add a handler:

```
PerlMapToStorageHandler MyApache2::NoTranslation
```

using the following code:

```
#file:MyApache2/NoTranslation.pm
#----------------------------
package MyApache2::NoTranslation;

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

use Apache2::Const -compile => qw(OK);

sub handler {
    my $r = shift;

    # skip ap_directory_walk stat() calls
    return Apache2::Const::OK;
}
1;
```

But this can be done from *httpd.conf* too!

```
PerlMapToStorageHandler Apache2::Const::OK
```

If you haven't already compiled `Apache2::Const::OK` elsewhere, you should add:

```
<Perl>
    use Apache2::Const -compile => qw(OK);
</Perl>
```

Apache also uses this phase to handle `TRACE` requests. So if you shortcut it, `TRACE` calls will be not handled. In case you need to handle such, you may rewrite it as:

```
#file:MyApache2/NoTranslation2.pm
#-----------------------------
package MyApache2::NoTranslation2;

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

use Apache2::RequestRec ();

use Apache2::Const -compile => qw(DECLINED OK M_TRACE);

sub handler {
    my $r = shift;

    return Apache2::Const::DECLINED
        if $r->method_number == Apache2::Const::M_TRACE;

    # skip ap_directory_walk stat() calls
    return Apache2::Const::OK;
}
1;
```

BTW, the HTTP TRACE method asks a web server to echo the contents of the request back to the client for debugging purposes. i.e., the complete request, including HTTP headers, is returned in the entity-body of a TRACE response. Attackers may abuse HTTP TRACE functionality to gain access to information in HTTP headers such as cookies and authentication data. In the presence of other cross-domain vulnerabilities in web browsers, sensitive header information could be read from any domains that support the HTTP TRACE method.

Another way to prevent the core translation is to set `$r->filename()` to some value, which can also be done in the `PerlTransHandler`, if you are already using it.

14.3.4 PerlHeaderParserHandler

The *header_parser* phase is the first phase to happen after the request has been mapped to its `<Location>` (or an equivalent container). At this phase the handler can examine the request headers and to take a special action based on these. For example this phase can be used to block evil clients targeting certain resources, while little resources were wasted so far.

This phase is of type `RUN_ALL`.

The handler's configuration scope is `DIR`.

Arguments

See the HTTP Request Handler Skeleton for a description of handler arguments.

Return

See Stacked Handlers for a description of handler return codes.

Examples

This phase is very similar to `PerlPostReadRequestHandler`, with the only difference that it's run after the request has been mapped to the resource. Both phases are useful for doing something once per request, as early as possible. And usually you can take any `PerlPostReadRequestHandler` and turn it into `PerlHeaderParserHandler` by simply changing the directive name in *httpd.conf* and moving it inside the container where it should be executed. Moreover, because of this similarity mod_perl provides a special directive `PerlInitHandler` which if found outside resource containers behaves as `PerlPostReadRequestHandler`, otherwise as `PerlHeaderParserHandler`.

You already know that Apache handles the `HEAD`, `GET`, `POST` and several other HTTP methods. But did you know that you can invent your own HTTP method as long as there is a client that supports it. If you think of emails, they are very similar to HTTP messages: they have a set of headers and a body, sometimes a multi-part body. Therefore we can develop a handler that extends HTTP by adding a support for the `EMAIL` method. We can enable this protocol extension and push the real content handler during the `PerlHeaderParserHandler` phase:

```
<Location /email>
    PerlHeaderParserHandler MyApache2::SendEmail
</Location>
```

and here is the `MyApache2::SendEmail` handler:

```
#file:MyApache2/SendEmail.pm
#-------------------------
package MyApache2::SendEmail;

use strict;
use warnings;

use Apache2::RequestRec ();
use Apache2::RequestIO ();
use Apache2::RequestUtil ();
use Apache2::ServerUtil ();
use Apache2::ServerRec ();
use Apache2::Process ();
use APR::Table ();

use Apache2::Const -compile => qw(DECLINED OK);
```

```perl
use constant METHOD        => 'EMAIL';
use constant SMTP_HOSTNAME => "localhost";

sub handler {
    my $r = shift;

    return Apache2::Const::DECLINED unless $r->method eq METHOD;

    $r->server->method_register(METHOD);
    $r->handler("perl-script");
    $r->push_handlers(PerlResponseHandler => \&send_email_handler);

    return Apache2::Const::OK;
}

sub send_email_handler {
    my $r = shift;

    my %headers = map {$_ => $r->headers_in->get($_)}
               qw(To From Subject);

        my $content = content($r);

    my $status = send_email(\%headers, \$content);

    $r->content_type('text/plain');
    $r->print($status ? "ACK" : "NACK");
    return Apache2::Const::OK;
}

sub send_email {
    my ($rh_headers, $r_body) = @_;

    require MIME::Lite;
    MIME::Lite->send("smtp", SMTP_HOSTNAME, Timeout => 60);

    my $msg = MIME::Lite->new(%$rh_headers, Data => $$r_body);
    #warn $msg->as_string;
    $msg->send;
}

use APR::Brigade ();
use APR::Bucket ();

use Apache2::Const -compile => qw(MODE_READBYTES);
use APR::Const     -compile => qw(SUCCESS BLOCK_READ);

use constant IOBUFSIZE => 8192;

sub content {
    my $r = shift;

    my $bb = APR::Brigade->new($r->pool, $r->connection->bucket_alloc);

    my $data = '';
    my $seen_eos = 0;
    do {
```

```
        $r->input_filters->get_brigade($bb,
                        Apache2::Const::MODE_READBYTES,
            APR::Const::BLOCK_READ, IOBUFSIZE);

        for (my $b = $bb->first; $b; $b = $bb->next($b)) {
            if ($b->is_eos) {

                $seen_eos++;
                last;
            }

            if ($b->read(my $buf)) {
                $data .= $buf;
            }

            $b->remove; # optimization to reuse memory
        }
    } while (!$seen_eos);

    $bb->destroy;

    return $data;
}

1;
```

Let's get the less interesting code out of the way. The function content() grabs the request body. The function send_email() sends the email over SMTP. You should adjust the constant SMTP_HOSTNAME to point to your outgoing SMTP server. You can replace this function with your own if you prefer to use a different method to send email.

Now to the more interesting functions. The function handler() returns immediately and passes the control to the next handler if the request method is not equal to EMAIL (set in the METHOD constant):

```
        return Apache2::Const::DECLINED unless $r->method eq METHOD;
```

Next it tells Apache that this new method is a valid one and that the perl-script handler will do the processing.

```
        $r->server->method_register(METHOD);
        $r->handler("perl-script");
```

Finally it pushes the function send_email_handler() to the PerlResponseHandler list of handlers:

```
        $r->push_handlers(PerlResponseHandler => \&send_email_handler);
```

The function terminates the header_parser phase by:

```
        return Apache2::Const::OK;
```

All other phases run as usual, so you can reuse any HTTP protocol hooks, such as authentication and fixup phases.

When the response phase starts `send_email_handler()` is invoked, assuming that no other response handlers were inserted before it. The response handler consists of three parts. Retrieve the email headers `To`, `From` and `Subject`, and the body of the message:

```
my %headers = map {$_ => $r->headers_in->get($_)}
        qw(To From Subject);
my $content = $r->content;
```

Then send the email:

```
my $status = send_email(\%headers, \$content);
```

Finally return to the client a simple response acknowledging that email has been sent and finish the response phase by returning `Apache2::Const::OK`:

```
$r->content_type('text/plain');
$r->print($status ? "ACK" : "NACK");
return Apache2::Const::OK;
```

Of course you will want to add extra validations if you want to use this code in production. This is just a proof of concept implementation.

As already mentioned when you extend an HTTP protocol you need to have a client that knows how to use the extension. So here is a simple client that uses `LWP::UserAgent` to issue an `EMAIL` method request over HTTP protocol:

```
#file:send_http_email.pl
#----------------------
#!/usr/bin/perl

use strict;
use warnings;

require LWP::UserAgent;

my $url = "http://localhost:8000/email/";

my %headers = (
    From    => 'example@example.com',
    To      => 'example@example.com',
    Subject => '3 weeks in Tibet',
);

my $content = <<EOI;
I didn't have an email software,
but could use HTTP so I'm sending it over HTTP
EOI
```

```
my $headers = HTTP::Headers->new(%headers);
my $req = HTTP::Request->new("EMAIL", $url, $headers, $content);
my $res = LWP::UserAgent->new->request($req);
print $res->is_success ? $res->content : "failed";
```

most of the code is just a custom data. The code that does something consists of four lines at the very end. Create `HTTP::Headers` and `HTTP::Request` object. Issue the request and get the response. Finally print the response's content if it was successful or just *"failed"* if not.

Now save the client code in the file *send_http_email.pl*, adjust the *To* field, make the file executable and execute it, after you have restarted the server. You should receive an email shortly to the address set in the *To* field.

14.3.5 *PerlInitHandler*

When configured inside any container directive, except `<VirtualHost>`, this handler is an alias for `PerlHeaderParserHandler` described earlier. Otherwise it acts as an alias for `PerlPostRead-RequestHandler` described earlier.

It is the first handler to be invoked when serving a request.

This phase is of type `RUN_ALL`.

Arguments

See the HTTP Request Handler Skeleton for a description of handler arguments.

Return

See Stacked Handlers for a description of handler return codes.

Examples

The best example here would be to use `Apache2::Reload` which takes the benefit of this directive. Usually `Apache2::Reload` is configured as:

```
PerlInitHandler Apache2::Reload
PerlSetVar ReloadAll Off
PerlSetVar ReloadModules "MyApache2::*"
```

which during the current HTTP request will monitor and reload all `MyApache2::*` modules that have been modified since the last HTTP request. However if we move the global configuration into a `<Location>` container:

```
<Location /devel>
    PerlInitHandler Apache2::Reload
    PerlSetVar ReloadAll Off
    PerlSetVar ReloadModules "MyApache2::*"
    SetHandler perl-script
    PerlResponseHandler ModPerl::Registry
    Options +ExecCGI
</Location>
```

`Apache2::Reload` will reload the modified modules, only when a request to the */devel* namespace is issued, because `PerlInitHandler` plays the role of `PerlHeaderParserHandler` here.

14.3.6 PerlAccessHandler

The *access_checker* phase is the first of three handlers that are involved in what's known as AAA: Authentication, Authorization, and Access control.

This phase can be used to restrict access from a certain IP address, time of the day or any other rule not connected to the user's identity.

This phase is of type `RUN_ALL`.

The handler's configuration scope is `DIR`.

Arguments

See the HTTP Request Handler Skeleton for a description of handler arguments.

Return

See Stacked Handlers for a description of handler return codes.

Examples

The concept behind access checker handler is very simple, return `Apache2::Const::FORBIDDEN` if the access is not allowed, otherwise return `Apache2::Const::OK`.

The following example handler denies requests made from IPs on the blacklist.

```
#file:MyApache2/BlockByIP.pm
#------------------------
package MyApache2::BlockByIP;

use strict;
use warnings;

use Apache2::RequestRec ();
use Apache2::Connection ();

use Apache2::Const -compile => qw(FORBIDDEN OK);

my %bad_ips = map {$_ => 1} qw(127.0.0.1 10.0.0.4);
```

```
sub handler {
    my $r = shift;

    return exists $bad_ips{$r->connection->remote_ip}
        ? Apache2::Const::FORBIDDEN
        : Apache2::Const::OK;
}

1;
```

The handler retrieves the connection's IP address, looks it up in the hash of blacklisted IPs and forbids the access if found. If the IP is not blacklisted, the handler returns control to the next access checker handler, which may still block the access based on a different rule.

To enable the handler simply add it to the container that needs to be protected. For example to protect an access to the registry scripts executed from the base location */perl* add:

```
<Location /perl/>
    SetHandler perl-script
    PerlResponseHandler ModPerl::Registry
    PerlAccessHandler MyApache2::BlockByIP
    Options +ExecCGI
</Location>
```

It's important to notice that `PerlAccessHandler` can be configured for any subsection of the site, no matter whether it's served by a mod_perl response handler or not. For example to run the handler from our example for all requests to the server simply add to *httpd.conf*:

```
<Location />
    PerlAccessHandler MyApache2::BlockByIP
</Location>
```

14.3.7 PerlAuthenHandler

The *check_user_id* (*authen*) phase is called whenever the requested file or directory is password protected. This, in turn, requires that the directory be associated with `AuthName`, `AuthType` and at least one `require` directive.

This phase is usually used to verify a user's identification credentials. If the credentials are verified to be correct, the handler should return `Apache2::Const::OK`. Otherwise the handler returns `Apache2::Const::HTTP_UNAUTHORIZED` to indicate that the user has not authenticated success-fully. When Apache sends the HTTP header with this code, the browser will normally pop up a dialog box that prompts the user for login information.

This phase is of type `RUN_FIRST`.

The handler's configuration scope is `DIR`.

Arguments

See the HTTP Request Handler Skeleton for a description of handler arguments.

Return

See Stacked Handlers for a description of handler return codes.

Examples

The following handler authenticates users by asking for a username and a password and lets them in only if the length of a string made from the supplied username and password and a single space equals to the secret length, specified by the constant `SECRET_LENGTH`.

```
#file:MyApache2/SecretLengthAuth.pm
#-------------------------------
package MyApache2::SecretLengthAuth;

use strict;
use warnings;

use Apache2::Access ();
use Apache2::RequestUtil ();

use Apache2::Const -compile => qw(OK DECLINED HTTP_UNAUTHORIZED);

use constant SECRET_LENGTH => 14;

sub handler {
    my $r = shift;

    my ($status, $password) = $r->get_basic_auth_pw;
    return $status unless $status == Apache2::Const::OK;

    return Apache2::Const::OK
        if SECRET_LENGTH == length join " ", $r->user, $password;

    $r->note_basic_auth_failure;
    return Apache2::Const::HTTP_UNAUTHORIZED;
}

1;
```

First the handler retrieves the status of the authentication and the password in plain text. The status will be set to `Apache2::Const::OK` only when the user has supplied the username and the password credentials. If the status is different, we just let Apache handle this situation for us, which will usually challenge the client so it'll supply the credentials.

Note that `get_basic_auth_pw()` does a few things behind the scenes, which are important to understand if you plan on implementing your own authentication mechanism that does not use `get_basic_auth_pw()`. First, is checks the value of the configured `AuthType` for the request, making sure it is `Basic`. Then it makes sure that the Authorization (or Proxy-Authorization) header is formatted for `Basic` authentication. Finally, after isolating the user and password from the header, it

populates the *ap_auth_type* slot in the request record with `Basic`. For the first and last parts of this process, mod_perl offers an API. `$r->auth_type` returns the configured authentication type for the current request - whatever was set via the `AuthType` configuration directive. `$r->ap_auth_type` populates the *ap_auth_type* slot in the request record, which should be done after it has been confirmed that the request is indeed using `Basic` authentication. (Note: `$r->ap_auth_type` was `$r->connection->auth_type` in the mod_perl 1.0 API.)

Once we know that we have the username and the password supplied by the client, we can proceed with the authentication. Our authentication algorithm is unusual. Instead of validating the username/password pair against a password file, we simply check that the string built from these two items plus a single space is `SECRET_LENGTH` long (14 in our example). So for example the pair *mod_perl/rules* authenticates correctly, whereas *secret/password* does not, because the latter pair will make a string of 15 characters. Of course this is not a strong authentication scheme and you shouldn't use it for serious things, but it's fun to play with. Most authentication validations simply verify the username/password against a database of valid pairs, usually this requires the password to be encrypted first, since storing passwords in clear is a bad idea.

Finally if our authentication fails the handler calls note_basic_auth_failure() and returns `Apache2::Const::HTTP_UNAUTHORIZED`, which sets the proper HTTP response headers that tell the client that its user that the authentication has failed and the credentials should be supplied again.

It's not enough to enable this handler for the authentication to work. You have to tell Apache what authentication scheme to use (`Basic` or `Digest`), which is specified by the `AuthType` directive, and you should also supply the `AuthName` -- the authentication realm, which is really just a string that the client usually uses as a title in the pop-up box, where the username and the password are inserted. Finally the `Require` directive is needed to specify which usernames are allowed to authenticate. If you set it to `valid-user` any username will do.

Here is the whole configuration section that requires users to authenticate before they are allowed to run the registry scripts from */perl/*:

```
<Location /perl/>
    SetHandler perl-script
    PerlResponseHandler ModPerl::Registry
    PerlAuthenHandler MyApache2::SecretLengthAuth
    Options +ExecCGI

    AuthType Basic
    AuthName "The Gate"
    Require valid-user
</Location>
```

Just like `PerlAccessHandler` and other mod_perl handlers, `PerlAuthenHandler` can be configured for any subsection of the site, no matter whether it's served by a mod_perl response handler or not. For example to use the authentication handler from the last example for any requests to the site, simply use:

```
<Location />
    PerlAuthenHandler MyApache2::SecretLengthAuth
    AuthType Basic
    AuthName "The Gate"
    Require valid-user
</Location>
```

14.3.8 PerlAuthzHandler

The *auth_checker* (*authz*) phase is used for authorization control. This phase requires a successful authentication from the previous phase, because a username is needed in order to decide whether a user is authorized to access the requested resource.

As this phase is tightly connected to the authentication phase, the handlers registered for this phase are only called when the requested resource is password protected, similar to the auth phase. The handler is expected to return `Apache2::Const::DECLINED` to defer the decision, `Apache2::Const::OK` to indicate its acceptance of the user's authorization, or `Apache2::Const::HTTP_UNAUTHORIZED` to indicate that the user is not authorized to access the requested document.

This phase is of type `RUN_FIRST`.

The handler's configuration scope is `DIR`.

Arguments

See the HTTP Request Handler Skeleton for a description of handler arguments.

Return

See Stacked Handlers for a description of handler return codes.

Examples

Here is the `MyApache2::SecretResourceAuthz` handler which grants access to certain resources only to certain users who have already properly authenticated:

```
#file:MyApache2/SecretResourceAuthz.pm
#----------------------------------
package MyApache2::SecretResourceAuthz;

use strict;
use warnings;

use Apache2::Access ();
use Apache2::RequestUtil ();

use Apache2::Const -compile => qw(OK HTTP_UNAUTHORIZED);

my %protected = (
    'admin'  => ['stas'],
    'report' => [qw(stas boss)],
);
```

```
sub handler {
    my $r = shift;

    my $user = $r->user;
    if ($user) {
        my ($section) = $r->uri =~ m|^/company/(\w+)/|;
        if (defined $section && exists $protected{$section}) {
            my $users = $protected{$section};
            return Apache2::Const::OK if grep { $_ eq $user } @$users;
        }
        else {
            return Apache2::Const::OK;
        }
    }

    $r->note_basic_auth_failure;
    return Apache2::Const::HTTP_UNAUTHORIZED;
}

1;
```

This authorization handler is very similar to the authentication handler from the previous section. Here we rely on the previous phase to get users authenticated, and now as we have the username we can make decisions whether to let the user access the resource it has asked for or not. In our example we have a simple hash which maps which users are allowed to access what resources. So for example anything under */company/admin/* can be accessed only by the user *stas*, */company/report/* can be accessed by users *stas* and *boss*, whereas any other resources under */company/* can be accessed by everybody who has reached so far. If for some reason we don't get the username, we or the user is not authorized to access the resource the handler does the same thing as it does when the authentication fails, i.e, calls:

```
$r->note_basic_auth_failure;
return Apache2::Const::HTTP_UNAUTHORIZED;
```

The configuration is similar to the one in the previous section, this time we just add the `PerlAuthzHandler` setting. The rest doesn't change.

```
Alias /company/ /home/httpd/httpd-2.0/perl/
<Location /company/>
    SetHandler perl-script
    PerlResponseHandler ModPerl::Registry
    PerlAuthenHandler MyApache2::SecretLengthAuth
    PerlAuthzHandler  MyApache2::SecretResourceAuthz
    Options +ExecCGI

    AuthType Basic
    AuthName "The Secret Gate"
    Require valid-user
</Location>
```

And if you want to run the authentication and authorization for the whole site, simply add:

```
<Location />
    PerlAuthenHandler MyApache2::SecretLengthAuth
    PerlAuthzHandler  MyApache2::SecretResourceAuthz
    AuthType Basic
    AuthName "The Secret Gate"
    Require valid-user
</Location>
```

14.3.9 PerlTypeHandler

The *type_checker* phase is used to set the response MIME type (`Content-type`) and sometimes other bits of document type information like the document language.

For example `mod_autoindex`, which performs automatic directory indexing, uses this phase to map the filename extensions to the corresponding icons which will be later used in the listing of files.

Of course later phases may override the mime type set in this phase.

This phase is of type `RUN_FIRST`.

The handler's configuration scope is `DIR`.

Arguments

See the HTTP Request Handler Skeleton for a description of handler arguments.

Return

See Stacked Handlers for a description of handler return codes.

Examples

The most important thing to remember when overriding the default *type_checker* handler, which is usually the mod_mime handler, is that you have to set the handler that will take care of the response phase and the response callback function or the code won't work. mod_mime does that based on `SetHandler` and `AddHandler` directives, and file extensions. So if you want the content handler to be run by mod_perl, set either:

```
$r->handler('perl-script');
$r->set_handlers(PerlResponseHandler => \&handler);
```

or:

```
$r->handler('modperl');
$r->set_handlers(PerlResponseHandler => \&handler);
```

depending on which type of response handler is wanted.

Writing a `PerlTypeHandler` handler which sets the content-type value and returns `Apache2::Const::DECLINED` so that the default handler will do the rest of the work, is not a good idea, because mod_mime will probably override this and other settings.

Therefore it's the easiest to leave this stage alone and do any desired settings in the *fixups* phase.

14.3.10 *PerlFixupHandler*

The *fixups* phase is happening just before the content handling phase. It gives the last chance to do things before the response is generated. For example in this phase mod_env populates the environment with variables configured with *SetEnv* and *PassEnv* directives.

This phase is of type `RUN_ALL`.

The handler's configuration scope is `DIR`.

Arguments

See the HTTP Request Handler Skeleton for a description of handler arguments.

Return

See Stacked Handlers for a description of handler return codes.

Examples

The following fixup handler example tells Apache at run time which handler and callback should be used to process the request based on the file extension of the request's URI.

```
#file:MyApache2/FileExtDispatch.pm
#-------------------------------
package MyApache2::FileExtDispatch;

use strict;
use warnings;

use Apache2::RequestIO ();
use Apache2::RequestRec ();
use Apache2::RequestUtil ();

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

use constant HANDLER  => 0;
use constant CALLBACK => 1;

my %exts = (
    cgi => ['perl-script',    \&cgi_handler],
    pl  => ['modperl',        \&pl_handler ],
    tt  => ['perl-script',    \&tt_handler ],
    txt => ['default-handler', undef       ],
);
```

```perl
sub handler {
    my $r = shift;

    my ($ext) = $r->uri =~ /\.(\w+)$/;
    $ext = 'txt' unless defined $ext and exists $exts{$ext};

    $r->handler($exts{$ext}->[HANDLER]);

    if (defined $exts{$ext}->[CALLBACK]) {
        $r->set_handlers(PerlResponseHandler => $exts{$ext}->[CALLBACK]);
    }

    return Apache2::Const::OK;
}

sub cgi_handler { content_handler($_[0], 'cgi') }
sub pl_handler  { content_handler($_[0], 'pl')  }

sub tt_handler  { content_handler($_[0], 'tt')  }

sub content_handler {
    my ($r, $type) = @_;

    $r->content_type('text/plain');
    $r->print("A handler of type '$type' was called");

    return Apache2::Const::OK;
}

1;
```

In the example we have used the following mapping.

```perl
my %exts = (
    cgi => ['perl-script',    \&cgi_handler],
    pl  => ['modperl',        \&pl_handler ],
    tt  => ['perl-script',    \&tt_handler ],
    txt => ['default-handler', undef        ],
);
```

So that *.cgi* requests will be handled by the `perl-script` handler and the `cgi_handler()` callback, *.pl* requests by `modperl` and `pl_handler()`, *.tt* (template toolkit) by `perl-script` and the `tt_handler()`, finally *.txt* request by the `default-handler` handler, which requires no callback.

Moreover the handler assumes that if the request's URI has no file extension or it does, but it's not in its mapping, the `default-handler` will be used, as if the *txt* extension was used.

After doing the mapping, the handler assigns the handler:

```perl
$r->handler($exts{$ext}->[HANDLER]);
```

and the callback if needed:

```
    if (defined $exts{$ext}->[CALLBACK]) {
        $r->set_handlers(
                        PerlResponseHandler => $exts{$ext}->[CALLBACK]);
    }
```

In this simple example the callback functions don't do much but calling the same content handler which simply prints the name of the extension if handled by mod_perl, otherwise Apache will serve the other files using the default handler. In real world you will use callbacks to real content handlers that do real things.

Here is how this handler is configured:

```
Alias /dispatch/ /home/httpd/httpd-2.0/htdocs/
<Location /dispatch/>
    PerlFixupHandler MyApache2::FileExtDispatch
</Location>
```

Notice that there is no need to specify anything, but the fixup handler. It applies the rest of the settings dynamically at run-time.

14.3.11 *PerlResponseHandler*

The *handler* (*response*) phase is used for generating the response. This is arguably the most important phase and most of the existing Apache modules do most of their work at this phase.

This is the only phase that requires two directives under mod_perl. For example:

```
<Location /perl>
    SetHandler perl-script
    PerlResponseHandler MyApache2::WorldDomination
</Location>
```

`SetHandler` set to `perl-script` or `modperl` tells Apache that mod_perl is going to handle the response generation. `PerlResponseHandler` tells mod_perl which callback is going to do the job.

This phase is of type `RUN_FIRST`.

The handler's configuration scope is `DIR`.

Arguments

See the HTTP Request Handler Skeleton for a description of handler arguments.

Return

See Stacked Handlers for a description of handler return codes.

Examples

Most of the `Apache::` modules on CPAN are dealing with this phase. In fact most of the developers spend the majority of their time working on handlers that generate response content.

Let's write a simple response handler, that just generates some content. This time let's do something more interesting than printing *"Hello world"*. Let's write a handler that prints itself:

```
#file:MyApache2/Deparse.pm
#----------------------
package MyApache2::Deparse;

use strict;
use warnings;

use Apache2::RequestRec ();
use Apache2::RequestIO ();
use B::Deparse ();

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

sub handler {
    my $r = shift;

    $r->content_type('text/plain');
    $r->print('sub handler ', B::Deparse->new->coderef2text(\&handler));

    return Apache2::Const::OK;
}
1;
```

To enable this handler add to *httpd.conf*:

```
<Location /deparse>
    SetHandler modperl
    PerlResponseHandler MyApache2::Deparse
</Location>
```

Now when the server is restarted and we issue a request to *http://localhost/deparse* we get the following response:

```
sub handler {
    package MyApache2::Deparse;
    use warnings;
    use strict 'refs';
    my $r = shift @_;
    $r->content_type('text/plain');
    $r->print('sub handler ', 'B::Deparse'->new->coderef2text(\&handler));
    return 0;
}
```

If you compare it to the source code, it's pretty much the same code. `B::Deparse` is fun to play with!

14.3.12 PerlLogHandler

The *log_transaction* phase happens no matter how the previous phases have ended up. If one of the earlier phases has aborted a request, e.g., failed authentication or 404 (file not found) errors, the rest of the phases up to and including the response phases are skipped. But this phase is always executed.

By this phase all the information about the request and the response is known, therefore the logging handlers usually record this information in various ways (e.g., logging to a flat file or a database).

This phase is of type `RUN_ALL`.

The handler's configuration scope is `DIR`.

Arguments

See the HTTP Request Handler Skeleton for a description of handler arguments.

Return

See Stacked Handlers for a description of handler return codes.

Examples

Imagine a situation where you have to log requests into individual files, one per user. Assuming that all requests start with */~username/*, so it's easy to categorize requests by the username. Here is the log handler that does that:

```
#file:MyApache2/LogPerUser.pm
#--------------------------
package MyApache2::LogPerUser;

use strict;
use warnings;

use Apache2::RequestRec ();
use Apache2::Connection ();

use Fcntl qw(:flock);
use File::Spec::Functions qw(catfile);

use Apache2::Const -compile => qw(OK DECLINED);

sub handler {
    my $r = shift;

    my ($username) = $r->uri =~ m|^/~([^/]+)|;
    return Apache2::Const::DECLINED unless defined $username;

    my $entry = sprintf qq(%s [%s] "%s" %d %d\n),
        $r->connection->remote_ip, scalar(localtime),
        $r->uri, $r->status, $r->bytes_sent;

    my $log_path = catfile Apache2::ServerUtil::server_root,
```

```
        "logs", "$username.log";
    open my $fh, ">>$log_path" or die "can't open $log_path: $!";
    flock $fh, LOCK_EX;
    print $fh $entry;
    close $fh;

    return Apache2::Const::OK;
}
1;
```

First the handler tries to figure out what username the request is issued for, if it fails to match the URI, it simply returns `Apache2::Const::DECLINED`, letting other log handlers to do the logging. Though it could return `Apache2::Const::OK` since all other log handlers will be run anyway.

Next it builds the log entry, similar to the default *access_log* entry. It's comprised of remote IP, the current time, the uri, the return status and how many bytes were sent to the client as a response body.

Finally the handler appends this entry to the log file for the user the request was issued for. Usually it's safe to append short strings to the file without being afraid of messing up the file, when two files attempt to write at the same time, but just to be on the safe side the handler exclusively locks the file before performing the writing.

To configure the handler simply enable the module with the `PerlLogHandler` directive, for the desired URI namespace (starting with : /~ in our example):

```
<LocationMatch "^/~">
    SetHandler perl-script
    PerlResponseHandler ModPerl::Registry
    PerlLogHandler MyApache2::LogPerUser
    Options +ExecCGI
</LocationMatch>
```

After restarting the server and issuing requests to the following URIs:

```
http://localhost/~stas/test.pl
http://localhost/~eric/test.pl
http://localhost/~stas/date.pl
```

The `MyApache2::LogPerUser` handler will append to *logs/stas.log*:

```
127.0.0.1 [Sat Aug 31 01:50:38 2002] "/~stas/test.pl" 200 8
127.0.0.1 [Sat Aug 31 01:50:40 2002] "/~stas/date.pl" 200 44
```

and to *logs/eric.log*:

```
127.0.0.1 [Sat Aug 31 01:50:39 2002] "/~eric/test.pl" 200 8
```

It's important to notice that `PerlLogHandler` can be configured for any subsection of the site, no matter whether it's served by a mod_perl response handler or not. For example to run the handler from our example for all requests to the server, simply add to *httpd.conf*:

```
<Location />
    PerlLogHandler MyApache2::LogPerUser
</Location>
```

Since the `PerlLogHandler` phase is of type `RUN_ALL`, all other logging handlers will be called as well.

14.3.13 PerlCleanupHandler

There is no *cleanup* Apache phase, it exists only inside mod_perl. It is used to execute some code immediately after the request has been served (the client went away) and before the request object is destroyed.

There are several usages for this use phase. The obvious one is to run a cleanup code, for example removing temporarily created files. The less obvious is to use this phase instead of `PerlLogHandler` if the logging operation is time consuming. This approach allows to free the client as soon as the response is sent.

This phase is of type `RUN_ALL`.

The handler's configuration scope is `DIR`.

Arguments

See the HTTP Request Handler Skeleton for a description of handler arguments.

Return

See Stacked Handlers for a description of handler return codes.

Examples

There are two ways to register and run cleanup handlers:

1. **Using the `PerlCleanupHandler` phase**

   ```
   PerlCleanupHandler MyApache2::Cleanup
   ```

   or:

   ```
   $r->push_handlers(PerlCleanupHandler => \&cleanup);
   ```

   This method is identical to all other handlers.

   In this technique the `cleanup()` callback accepts `$r` as its only argument.

2. **Using `cleanup_register()` acting on the request object's pool**

   Since a request object pool is destroyed at the end of each request, we can use `cleanup_register` to register a cleanup callback which will be executed just before the pool is destroyed. For example:

```
        $r->pool->cleanup_register(\&cleanup, $arg);
```

The important difference from using the `PerlCleanupHandler` handler, is that here you can pass an optional arbitrary argument to the callback function, and no `$r` argument is passed by default. Therefore if you need to pass any data other than `$r` you may want to use this technique.

Here is an example where the cleanup handler is used to delete a temporary file. The response handler is running `ls -l` and stores the output in temporary file, which is then used by `$r->sendfile` to send the file's contents. We use `push_handlers()` to push `PerlCleanupHandler` to unlink the file at the end of the request.

```perl
#file:MyApache2/Cleanup1.pm
#-----------------------
package MyApache2::Cleanup1;

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

use File::Spec::Functions qw(catfile);

use Apache2::RequestRec ();
use Apache2::RequestIO ();
use Apache2::RequestUtil ();

use Apache2::Const -compile => qw(OK DECLINED);
use APR::Const    -compile => 'SUCCESS';

my $file = catfile "/tmp", "data";

sub handler {
    my $r = shift;

    $r->content_type('text/plain');

    local @ENV{qw(PATH BASH_ENV)};
    qx(/bin/ls -l > $file);

    my $status = $r->sendfile($file);
    die "sendfile has failed" unless $status == APR::Const::SUCCESS;

    $r->push_handlers(PerlCleanupHandler => \&cleanup);

    return Apache2::Const::OK;
}

sub cleanup {
    my $r = shift;

    die "Can't find file: $file" unless -e $file;
    unlink $file or die "failed to unlink $file";

    return Apache2::Const::OK;
}
1;
```

Next we add the following configuration:

```
<Location /cleanup1>
    SetHandler modperl
    PerlResponseHandler MyApache2::Cleanup1
</Location>
```

Now when a request to */cleanup1* is made, the contents of the current directory will be printed and once the request is over the temporary file is deleted.

This response handler has a problem of running in a multi-process environment, since it uses the same file, and several processes may try to read/write/delete that file at the same time, wrecking havoc. We could have appended the process id `$$` to the file's name, but remember that mod_perl 2.0 code may run in the threaded environment, meaning that there will be many threads running in the same process and the `$$` trick won't work any longer. Therefore one really has to use this code to create unique, but predictable, file names across threads and processes:

```
sub unique_id {
    require Apache2::MPM;
    require APR::OS;
    return Apache2::MPM->is_threaded
        ? "$$." . ${ APR::OS::current_thread_id() }
        : $$;
}
```

In the threaded environment it will return a string containing the process ID, followed by a thread ID. In the non-threaded environment only the process ID will be returned. However since it gives us a predictable string, they may still be a non-satisfactory solution. Therefore we need to use a random string. We can either either Perl's `rand`, some CPAN module or the APR's `APR::UUID`:

```
sub unique_id {
  require APR::UUID;
  return APR::UUID->new->format;
}
```

Now the problem is how do we tell the cleanup handler what file should be cleaned up? We could have stored it in the `$r->notes` table in the response handler and then retrieve it in the cleanup handler. However there is a better way - as mentioned earlier, we can register a callback for request pool cleanup, and when using this method we can pass an arbitrary argument to it. Therefore in our case we choose to pass the file name, based on random string. Here is a better version of the response and cleanup handlers, that uses this technique:

```
#file:  MyApache2/Cleanup2.pm
#-------------------------
package MyApache2::Cleanup2;

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

use File::Spec::Functions qw(catfile);

use Apache2::RequestRec ();
use Apache2::RequestIO ();
```

```
use Apache2::RequestUtil ();
use APR::UUID ();
use APR::Pool ();

use Apache2::Const -compile => qw(OK DECLINED);
use APR::Const    -compile => 'SUCCESS';

my $file_base = catfile "/tmp", "data-";

sub handler {
        my $r = shift;

    $r->content_type('text/plain');
    my $file = $file_base . APR::UUID->new->format;

    local @ENV{qw(PATH BASH_ENV)};
    qx(/bin/ls -l > $file);

    my $status = $r->sendfile($file);
    die "sendfile has failed" unless $status == APR::Const::SUCCESS;

    $r->pool->cleanup_register(\&cleanup, $file);

    return Apache2::Const::OK;
}

sub cleanup {
    my $file = shift;

    die "Can't find file: $file" unless -e $file;
    unlink $file or die "failed to unlink $file";

    return Apache2::Const::OK;
}
1;
```

Similarly to the first handler, we add the configuration:

```
<Location /cleanup2>
    SetHandler modperl
    PerlResponseHandler MyApache2::Cleanup2
</Location>
```

And now when requesting */cleanup2* we still get the same output -- the listing of the current directory -- but this time this code will work correctly in the multi-processes/multi-threaded environment and temporary files get cleaned up as well.

14.3.13.1 Possible Caveats

`PerlCleanupHandler` may fail to be completed on server shutdown/graceful restart since Apache will kill the registered handlers via SIGTERM, before they had a chance to run or even in the middle of its execution. See: http://marc.theaimsgroup.com/?t=106387845200003&r=1&w=2   http://marc.theaims-group.com/?l=apache-modperl-dev&m=106427616108596&w=2

14.4 Miscellaneous Issues

14.4.1 Handling HEAD Requests

In order to avoid the overhead of sending the data to the client when the request is of type HEAD in mod_perl 1.0 we used to return early from the handler:

```
 return Apache2::Const::OK if $r->header_only;
```

This logic should not be used in mod_perl 2.0, because Apache 2.0 automatically discards the response body for HEAD requests. It expects the full body to generate the correct set of response headers, if you don't send the body you may encounter problems.

(You can also read the comment in for `ap_http_header_filter()` in *modules/http/http_protocol.c* in the Apache 2.0 source.)

14.4.2 `Content-Length` Response Header

You may encounter some issues with the C-L (`Content-Length`) header. Some of them are discussed here.

- **The special case of `Content-Length: 0`**

  Since Apache proclaims itself governor of the C-L header via the C-L filter (ap_content_length_filter at *httpd-2.0/server/protocol.c*), for the most part GET and HEAD behave exactly the same. However, when Apache sees a HEAD request with a C-L header of zero it takes special action and removes the C-L header. This is done to protect against handlers that called `$r->header_only` (which was ok in 1.3 but is not in 2.0). Therefore, GET and HEAD behave identically, except when the content handler (and/or filters) end up sending no content. For more details refer to the lengthy comments in `ap_http_header_filter()` in *httpd-2.0/modules/http/http_protocol.c*).

  For more discussion on why it is important to get HEAD requests right, see these threads from the mod_perl list:

  ```
   http://marc.theaimsgroup.com/?l=apache-modperl&m=108647669726915&w=2
   http://marc.theaimsgroup.com/?t=109122984600001&r=1&w=2
  ```

  as well as this bug report from mozilla, which shows how HEAD requests are used in the wild:

  ```
   http://bugzilla.mozilla.org/show_bug.cgi?id=245447
  ```

- **Not getting `Content-Length` header with `HEAD` requests**

  Even though the spec says that content handlers should send an identical response for GET and HEAD requests, some folks try to avoid the overhead of generating the response body, which Apache is going to discard anyway for HEAD requests. The following discussion assumes that we deal with a HEAD request.

When Apache sees EOS and no headers and no response body were sent, `ap_content_length_filter()` (*httpd-2.0/server/protocol.c*) sets C-L to 0. Later on `ap_http_header_filter()` (*httpd-2.0/modules/http/http_protocol.c*) removes the C-L header for the HEAD requests.

The workaround is to force the sending of the response headers, before `EOS` was sent (which happens when the response handler returns). The simplest solution is to use rflush():

```
if ($r->header_only) { # HEAD
    $body_len = calculate_body_len();
    $r->set_content_length($body_len);
    $r->rflush;
}
else {                  # GET
    # generate and send the body
}
```

now if the handler sets the C-L header it'll be delivered to the client unmodified.

14.5 Misc Notes

These items will need to be extended and integrated in this or other HTTP related documents:

- **front-end back-end setup: mod_proxy+X-Forwarded-For**

  apache-1.3:

  frontend: mod_proxy_add_forward http://develooper.com/code/mpaf/

  backend: mod_rpaf (reverse proxy add forward): http://stderr.net/apache/rpaf/

  apache-2.x:

  frontend: mod_proxy

  backend: mod_rpaf: http://stderr.net/apache/rpaf/

14.6 Extending HTTP Protocol

Extending HTTP under mod_perl is a trivial task. Look at the example of adding a new method `EMAIL` for details.

14.7 HTTP Status Codes

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers. A feature of HTTP is the typing and negotiation

of data representation, allowing systems to be built independently of the data being transferred.

HTTP 1.0 is described in Requests For Comments (RFC) 1945. HTTP 1.1 is the latest version of the specifications and as of this writing HTTP 1.1 is covered in RFC 2616.

When writing mod_perl applications, usually only a small subset of HTTP response codes is used, but sometimes you need to know others as well. We will give a short description of each code and you will find the extended explanation in the appropriate RFC. (Section 9 in RFC 1945 and section 10 in RFC 2616). You can always find the latest link to these RFCs at the Web Consortium site, *http://www.w3.org/Protocols/*.

While HTTP 1.1 is widely supported, HTTP 1.0 still remains the mainstream standard. Therefore we will supply a summary of the both versions including the corresponding Apache constants.

In mod_perl these constants can be accessed the `Apache::Constants` package (e.g., to access the HTTP_OK constant use `Apache::Constants::HTTP_OK`). See the `Apache::Constants` manpage for more information.

In mod_perl2 these constants can be accessed the `Apache2::Const` package (e.g., to access the HTTP_OK constant use `Apache2::Const::HTTP_OK`). See the `Apache2::Const` manpage for more information.

14.7.1 HTTP 1.0 Status Codes

- Successful 2xx:

```
200 HTTP_OK                 OK
201 HTTP_CREATED            Created
202 HTTP_ACCEPTED           Accepted
204 HTTP_NO_CONTENT         No Content
```

- Redirection 3xx:

```
301 HTTP_MOVED_PERMANENTLY    Multiple Choices
302 HTTP_MOVED_TEMPORARILY    Moved Permanently
303 HTTP_SEE_OTHER            Moved Temporarily
304 HTTP_NOT_MODIFIED         Not Modified
```

- Client Error 4xx:

```
400 HTTP_BAD_REQUEST        Bad Request
401 HTTP_UNAUTHORIZED       Unauthorized
403 HTTP_FORBIDDEN          Forbidden
404 HTTP_NOT_FOUND          Not Found
```

- Server Error 5xx:

```
500 HTTP_INTERNAL_SERVER_ERROR Internal Server Error
501 HTTP_NOT_IMPLEMENTED     Not Implemented
502 HTTP_BAD_GATEWAY         Bad Gateway
503 HTTP_SERVICE_UNAVAILABLE  Service UnavailableStatus Codes
```

14.7.2 HTTP 1.1 Status Codes

- Informational 1xx:

```
100 HTTP_CONTINUE                       Continue
101 HTTP_SWITCHING_PROTOCOLS            Switching Protocols
```

- Successful 2xx:

```
200 HTTP_OK                             OK
201 HTTP_CREATED                        Created
202 HTTP_ACCEPTED                       Accepted
203 HTTP_NON_AUTHORITATIVE              Non-Authoritative Information
204 HTTP_NO_CONTENT                     No Content
205 HTTP_RESET_CONTENT                  Reset Content
206 HTTP_PARTIAL_CONTENT                Partial Content
```

- Redirection 3xx:

```
300 HTTP_MULTIPLE_CHOICES               Multiple Choices
301 HTTP_MOVED_PERMANENTLY              Moved Permanently
302 HTTP_MOVED_TEMPORARILY              Found
303 HTTP_SEE_OTHER                      See Other
304 HTTP_NOT_MODIFIED                   Not Modified
305 HTTP_USE_PROXY                      Use Proxy
306                                     (Unused)
307 HTTP_TEMPORARY_REDIRECT             Temporary Redirect
```

- Client Error 4xx:

```
400 HTTP_BAD_REQUEST                    Bad Request
401 HTTP_UNAUTHORIZED                   Unauthorized
402 HTTP_PAYMENT_REQUIRED               Payment Required
403 HTTP_FORBIDDEN                      Forbidden
404 HTTP_NOT_FOUND                      Not Found
405 HTTP_METHOD_NOT_ALLOWED             Method Not Allowed
406 HTTP_NOT_ACCEPTABLE                 Not Acceptable
407 HTTP_PROXY_AUTHENTICATION_REQUIRED  Proxy Authentication Required
408 HTTP_REQUEST_TIMEOUT                Request Timeout
409 HTTP_CONFLICT                       Conflict
410 HTTP_GONE                           Gone
411 HTTP_LENGTH REQUIRED                Length Required
412 HTTP_PRECONDITION_FAILED            Precondition Failed
413 HTTP_REQUEST_ENTITY_TOO_LARGE       Request Entity Too Large
414 HTTP_REQUEST_URI_TOO_LARGE          Request-URI Too Long
415 HTTP_UNSUPPORTED_MEDIA_TYPE         Unsupported Media Type
416 HTTP_RANGE_NOT_SATISFIABLE          Requested Range Not Satisfiable
417 HTTP_EXPECTATION_FAILED             Expectation Failed
```

- Server Error 5xx:

```
500 HTTP_INTERNAL_SERVER_ERROR          Internal Server Error
501 HTTP_NOT IMPLEMENTED                Not Implemented
502 HTTP_BAD_GATEWAY                    Bad Gateway
503 HTTP_SERVICE_UNAVAILABLE            Service Unavailable
504 HTTP_GATEWAY_TIME_OUT               Gateway Timeout
505 HTTP_VERSION_NOT_SUPPORTED          HTTP Version Not Supported
```

14.7.3 References

All the information related to web protocols can be found at the World Wide Web Consortium site, *http://www.w3.org/Protocols/*.

There are many mirrors of the RFCs all around the world. One of the good starting points might be *http://www.rfc-editor.org/*.

The Eagle Book provided much of the HTTP constants material shown here *http://www.modperl.com/book/chapters/ch9.html#The_Apache_Constants_Class*

14.8 Maintainers

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

● The mod_perl development team and numerous contributors.

14.9 Authors

● Stas Bekman [http://stason.org/]

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

15   Input and Output Filters

15.1 Description

This chapter discusses mod_perl's input and output filter handlers.

If all you need is to lookup the filtering API proceed directly to the `Apache2::Filter` and `Apache2::FilterRec` manpages.

15.2 Introducing Filters

You certainly already know how filters work, because you encounter filters so often in real life. If you are unfortunate to live in smog-filled cities like Saigon or Bangkok you are probably used to wear a dust filter mask:

dust mask

If you are smoker, chances are that you smoke cigarettes with filters:

cigarrette filter

If you are a coffee gourmand, you have certainly tried a filter coffee:

coffee machine

The shower that you use, may have a water filter:

shower filter

When the sun is too bright, you protect your eyes by wearing sun goggles with UV filter:

sun goggles

If are a photographer you can't go a step without using filter lenses:

photo camera

If you love music, you might be unaware of it, but your super-modern audio system is literally loaded with various electronic filters:

LP player

There are many more places in our lives where filters are used. The purpose of all filters is to apply some transformation to what's coming into the filter, letting something different out of the filter. Certainly in some cases it's possible to modify the source itself, but that makes things unflexible, and but most of the time we have no control over the source. The advantage of using filters to modify something is that they can be replaced when requirements change Filters also can be stacked, which allows us to make each filter do simple transformations. For example by combining several different filters, we can apply multiple transformations. In certain situations combining several filters of the same kind let's us achieve a better quality output.

The mod_perl filters are not any different, they receive some data, modify it and send it out. In the case of filtering the output of the response handler, we could certainly change the response handler's logic to do something different, since we control the response handler. But this may make the code unnecessary complex. If we can apply transformations to the response handler's output, it certainly gives us more flexibility and simplifies things. For example if a response needs to be compressed before sent out, it'd be very inconvenient and inefficient to code in the response handler itself. Using a filter for that purpose is a perfect solution. Similarly, in certain cases, using an input filter to transform the incoming request data is the most wise solution. Think of the same example of having the incoming data coming compressed.

Just like with real life filters, you can pipe several filters to modify each other's output. You can also customize a selection of different filters at run time.

Without much further ado, let's write a simple but useful obfuscation filter for our HTML documents.

We are going to use a very simple obfuscation -- turn an HTML document into a one liner, which will make it harder to read its source without a special processing. To accomplish that we are going to remove characters \012 (\n) and \015 (\r), which depending on the platform alone or as a combination represent the end of line and a carriage return.

And here is the filter handler code:

```
#file:MyApache2/FilterObfuscate.pm
#------------------------------
package MyApache2::FilterObfuscate;

use strict;
use warnings;

use Apache2::Filter ();
use Apache2::RequestRec ();
use APR::Table ();

use Apache2::Const -compile => qw(OK);

use constant BUFF_LEN => 1024;

sub handler {
    my $f = shift;

    unless ($f->ctx) {
        $f->r->headers_out->unset('Content-Length');
        $f->ctx(1);
    }

    while ($f->read(my $buffer, BUFF_LEN)) {
        $buffer =~ s/[\r\n]//g;
        $f->print($buffer);
    }

    return Apache2::Const::OK;
}
1;
```

The directives below configure Apache to apply the `MyApache2::FilterObfuscate` filter to all requests that get mapped to files with an *".html"* extension:

```
<Files ~ "\.html">
    PerlOutputFilterHandler MyApache2::FilterObfuscate
</Files>
```

Filters are expected to return `Apache2::Const::OK` or `Apache2::Const::DECLINED`. But instead of receiving `$r` (the request object) as the first argument, they receive `$f` (the filter object). The filter object is described later in this chapter.

The filter starts by unsetting the `Content-Length` response header, because it modifies the length of the response body (shrinks it). If the response handler sets the `Content-Length` header and the filter doesn't unset it, the client may have problems receiving the response since it will expect more data than it was sent. *Setting the Content-Length Header* below describes how to set the Content-Length header if you need to.

The core of this filter is a read-modify-print expression in a while loop. The logic is very simple: read at most `BUFF_LEN` characters of data into `$buffer`, apply the regex to remove any occurences of `\n` and `\r` in it, and print the resulting data out. The input data may come from a response handler, or from an upstream filter. The output data goes to the next filter in the output chain. Even though in this example we haven't configured any more filters, internally Apache itself uses several core filters to manipulate the data and send it out to the client.

As we are going to explain in detail in the following sections, the same filter may be called many times during a single request, every time receiving a subsequent chunk of data. For example if the POSTed request data is 64k long, an input filter could be invoked 8 times, each time receiving 8k of data. The same may happen during the response phase, where an upstream filter may split 64k of output in 8, 8k chunks. The while loop that we just saw is going to read each of these 8k in 8 calls, since it requests 1k on every `read()` call.

Since it's enough to unset the `Content-Length` header when the filter is called the first time, we need to have some flag telling us whether we have done the job. The method `ctx()` provides this functionality:

```
unless ($f->ctx) {
    $f->r->headers_out->unset('Content-Length');
    $f->ctx(1);
}
```

The `unset()` call will be made only on the first filter call for each request. You can store any kind of a Perl data structure in `$f->ctx` and retrieve it later in subsequent filter invocations of the same request. There are several examples using this method in the following sections.

To be truly useful, the `MyApache2::FilterObfuscate` filter logic should take into account situations where removing new line characters will make the document render incorrectly in the browser. As we mentioned above, this is the case if there are multi-line `<pre>...</pre>` entries. Since this increases the complexity of the filter, we will disregard this requirement for now.

A positive side-effect of this obfuscation algorithm is that it reduces the amount of the data sent to the client. The `Apache::Clean` module, available from the CPAN, provides a production-ready implementation of this technique which takes into account the HTML markup specifics.

mod_perl I/O filtering follows the Perl principle of making simple things easy and difficult things possible. You have seen that it's trivial to write simple filters. As you read through this chapter you will see that much more difficult things are possible, and that the code is more elaborate.

15.3 I/O Filtering Concepts

Before introducing the APIs, mod_perl provides for Apache Filtering, there are several important concepts to understand.

15.3.1 Two Methods for Manipulating Data

mod_perl provides two interfaces to filtering: a direct bucket brigades manipulation interface and a simpler, stream-oriented interface. Apache 2.0 considers all incoming and outgoing data as chunks of information, disregarding their kind and source or storage methods. These data chunks are stored in *buckets*, which form bucket brigades. Input and output filters massage the data in these *bucket brigades*. Response and protocol handlers also receive and send data using bucket brigades, though in most cases this is hidden behind wrappers, such as `read()` and `print()`.

mod_perl 2.0 filters can directly manipulate the bucket brigades or use the simplified streaming interface where the filter object acts similar to a filehandle, which can be read from and printed to.

Even though you don't use bucket brigades directly when you use the streaming filter interface (which works on bucket brigades behind the scenes), it's still important to understand bucket brigades. For example you need to know that an output filter will be invoked as many times as the number of bucket brigades sent from an upstream filter or a content handler. Or you need to know that the end of stream indicator (EOS) is sometimes sent in a separate bucket brigade, so it shouldn't be a surprise that the filter was invoked even though no real data went through. As we delve into the filter details you will see that understanding bucket brigades, will help to understand how filters work.

Moreover you will need to understand bucket brigades if you plan to implement protocol modules.

15.3.2 HTTP Request Versus Connection Filters

HTTP request filters are applied when Apache serves an HTTP request. HTTP request input filters get invoked on the body of the HTTP request only if the body is consumed by the content handler. HTTP request headers are not passed through the HTTP request input filters. HTTP response output filters get invoked on the body of the HTTP response if the content handler has generated one. HTTP response headers are not passed through the HTTP response output filters.

It is also possible to apply filters at the connection level. A connection may be configured to serve one or more HTTP requests, or handle other protocols. Connection filters see all the incoming and outgoing data. If an HTTP request is served, connection filters can modify the HTTP headers and the body of request and

response. If a different protocol is served over the connection (e.g., IMAP), the data could have a completely different pattern than the HTTP protocol (headers + body). Thus, the only difference between connection filters and request filters is that connection filters see everything from the request, i.e., the headers and the body, whereas request filters see only the body.

mod_perl 2.0 may support several other Apache filter types in the future.

15.3.3 Multiple Invocations of Filter Handlers

Unlike other Apache handlers, filter handlers may get invoked more than once during the same request. Filters get invoked as many times as the number of bucket brigades sent from an upstream filter or a content provider.

For example, a content generation handler may send a string, then force a flush, and then send more data:

```
# assuming buffered STDOUT ($|==0)
$r->print("foo");
$r->rflush;
$r->print("bar");
```

In this case, Apache will generate one bucket brigade with two buckets. There are several types of buckets which contain data; in this example, the data type is *transient*:

```
bucket type        data
---------------------
1st     transient  foo
2nd     flush
```

Apache sends this bucket brigade to the filter chain. Then, assuming no more data is sent after `print("bar")`, it will create a last bucket brigade, with one bucket, containing data:

```
bucket type        data
---------------------
1st     transient  bar
```

and send it to the filter chain. Finally it will send yet another bucket brigade with the EOS bucket indicating that there will be no more data sent:

```
bucket type        data
---------------------
1st     eos
```

EOS buckets are valid for request filters. For connection filters, you will get one only in the response filters and only at the end of the connection. You can see a sample workaround for this situation in the module `Apache2::Filter::HTTPHeadersFixup` available on the CPAN.

Note that the EOS bucket may come attached to the last bucket brigade with data, instead of coming in its own bucket brigade. The location depends on the other Apache modules manipulating the buckets and can vary. Filters should never assume that the EOS bucket is arriving alone in a bucket brigade. Therefore the first output filter will be invoked two or three times (three times if EOS is coming in its own brigade), depending on the number of bucket brigades sent by the response handler.

An upstream filter can modify the bucket brigades, by inserting extra bucket brigades or even by collecting the data from multiple bucket brigades and sending it along in just one brigade. Therefore, when coding a filter, never assume that the filter is always going to be invoked once, or any fixed number of times. Neither can you assume how the data is going to come in. To accommodate these situations, a typical filter handler may need to split its logic in three parts.

To illustrate, below is some pseudo-code that represents all three parts, i.e., initialization, processing, and finalization. This is a typical stream-oriented filter handler.

```
sub handler {
    my $f = shift;

    # runs on first invocation
    unless ($f->ctx) {
        init($f);
        $f->ctx(1);
    }

    # runs on all invocations
    process($f);

    # runs on the last invocation
    if ($f->seen_eos) {
        finalize($f);
    }

    return Apache2::Const::OK;
}
sub init     { ... }
sub process  { ... }
sub finalize { ... }
```

The following diagram depicts all three parts:

filter flow logic

Let's explain each part using this pseudo-filter.

1. **Initialization**

   During initialization, the filter runs code that you want executed only once, even if there are multiple invocations of the filter (this is during a single request). The filter context ($f->ctx) is used as a flag to accomplish this task. For each new request the filter context is created before the filter is called for the first time, and it is destroyed at the end of the request.

   ```
   unless ($f->ctx) {
       init($f);
       $f->ctx(1);
   }
   ```

When the filter is invoked for the first time `$f->ctx` returns `undef` and the custom function init() is called. This function could, for example, retrieve some configuration data set in *httpd.conf* or initialize some data structure to a default value.

To make sure that init() won't be called on the following invocations, we must set the filter context before the first invocation is completed:

```
$f->ctx(1);
```

In practice, the context is not just used as a flag, but to store real data. You can use it to hold any data structure and pass it between successive filter invocations. For example, the following filter handler counts the number of times it was invoked during a single request:

```
sub handler {
    my $f = shift;

    my $ctx = $f->ctx;
    $ctx->{invoked}++;
    $f->ctx($ctx);
    warn "filter was invoked $ctx->{invoked} times\n";

    return Apache2::Const::DECLINED;
}
```

Since this filter handler doesn't consume the data from the upstream filter, it's important that this handler return `Apache2::Const::DECLINED`, in which case mod_perl passes the current bucket brigade to the next filter. If this handler returns `Apache2::Const::OK`, the data will be lost, and if that data included a special EOS token, this may cause problems.

Unsetting the `Content-Length` header for filters that modify the response body length is a good example of code to run in the initialization phase:

```
unless ($f->ctx) {
    $f->r->headers_out->unset('Content-Length');
    $f->ctx(1);
}
```

We will see more initialization examples later in this chapter.

2. **Processing**

The next part:

```
process($f);
```

is unconditionally invoked on every filter invocation. That's where the incoming data is read, modified and sent out to the next filter in the filter chain. Here is an example that lowers the case of the characters passing through:

```
use constant READ_SIZE  => 1024;
sub process {
    my $f = shift;
    while ($f->read(my $data, READ_SIZE)) {
        $f->print(lc $data);
    }
}
```

Here the filter operates only on a single bucket brigade. Since it manipulates every character sepa-rately the logic is simple.

In more complicated situations, a filter may need to buffer data before the transformation can be applied. For example, if the filter operates on HTML tokens (e.g., '<img src="me.jpg">'), it's possi-ble that one brigade will include the beginning of the token ('<img ') and the remainder of the token ('src="me.jpg">') will come in the next bucket brigade (on the next filter invocation). To operate on the token as a whole, you would need to capture each piece over several invocations. To do so, you can store the unprocessed data in the filter context and then access it again on the next invocation.

Another good example of the need to buffer data is a filter that performs data compression, because compression is usually effective only when applied to relatively big chunks of data. If a single bucket brigade doesn't contain enough data, the filter may need to buffer the data in the filter context until it collects enough to compress it.

3. **Finalization**

Finally, some filters need to know when they are invoked for the last time, in order to perform various cleanups and/or flush any remaining data. As mentioned earlier, Apache indicates this event by a special end of stream "token", represented by a bucket of type `EOS`. If the filter is using the streaming interface, rather than manipulating the bucket brigades directly, and it was calling `read()` in a while loop, it can check for the EOS token using the `$f->seen_eos` method:

```
if ($f->seen_eos) {
    finalize($f);
}
```

This check should be done at the end of the filter handler because the EOS token can come attached to the tail of some data or all alone such that the last invocation gets only the EOS token. If this test is performed at the beginning of the handler and the EOS bucket was sent in together with the data, the EOS event may be missed and the filter won't function properly.

Filters that directly manipulate bucket brigades must manually look for a bucket whose type is `EOS`. There are examples of this method later in the chapter.

While not all filters need to perform all of these steps, this is a good model to keep in mind while working on your filter handlers. Since filters are called multiple times per request, you will likely use these steps, with initialization, processing, and finishing, on all but the simplest filters.

15.3.4 Blocking Calls

All filters (excluding the core filter that reads from the network and the core filter that writes to it) block at least once when invoked. Depending on whether this is an input or an output filter, the blocking happens when the bucket brigade is requested from the upstream filter or when the bucket brigade is passed to the downstream filter.

Input and output filters differ in the ways they acquire the bucket brigades, and thus in how blocking is handled. Each type is described separately below. Although you can't see the difference when using the streaming API, it's important to understand how things work underneath. Therefore the examples below are transparent filters, passing data through them unmodified. Instead of reading the data in and printing it out, the bucket brigades are passed as is. This makes it easier to observe the blocking behavior.

The first example is a transparent input filter:

```
#file:MyApache2/FilterTransparent.pm (first part)
#---------------------------------------------
package MyApache2::FilterTransparent;

use Apache2::Filter ();

use Apache2::Const -compile => qw(OK);
use APR::Const     -compile => ':common';

sub in {
    my ($f, $bb, $mode, $block, $readbytes) = @_;

    my $rv = $f->next->get_brigade($bb, $mode, $block, $readbytes);
    return $rv unless $rv == APR::Const::SUCCESS;

    return Apache2::Const::OK;
}
```

When the input filter *in()* is invoked, it first asks the upstream filter for the next bucket brigade (using the `get_brigade()` call). That upstream filter is in turn going to ask for the bucket brigade from the next upstream filter and so on up the chain, until the last filter (called `core_in`), the one that reads from the network, is reached. The `core_in` filter reads, using a socket, a portion of the incoming data from the network, processes it, and sends it to its downstream filter. That filter processes the data and send it to its downstream filter, etc., until it reaches the first filter that requested the data. (In reality some other handler triggers the request for the bucket brigade, such as an HTTP response handler or a protocol module, but for this discussion it's sufficient to assume that it's the first filter that issues the `get_brigade()` call.)

The following diagram depicts a typical input filter data flow in addition to the program control flow.

input filter data flow

The black- and white-headed arrows show when the control is passed from one filter to another. In addition, the black-headed arrows show the actual data flow. The diagram includes some pseudo-code, in Perl for the mod_perl filters and in C for the internal Apache filters. You don't have to understand C to understand this diagram. What's important to understand is that when input filters are invoked, they first call each other via the `get_brigade()` call and then block (notice the brick wall on the diagram), waiting

for the call to return. When this call returns, all upstream filters have already completed their filtering task on the bucket brigade.

As mentioned earlier, the streaming interface hides the details, but the first `$f->read()` call will block as the layer under it performs the `get_brigade()` call.

The diagram shows only part of the actual input filter chain for an HTTP request. The `...` indicates that there are more filters in between the mod_perl filter and `http_in`.

Now let's look at what happens in the output filters chain. Here the first filter acquires the bucket brigades containing the response data from the content handler (or another protocol handler if we aren't talking HTTP). It may then make some modification and pass the data to the next filter (using the `pass_brigade()` call), which in turn applies its modifications and sends the bucket brigade to the next filter, etc. This continues all the way down to the last filter (called `core`) which writes the data to the network via the socket the client is listening to.

Even though the output filters don't have to wait to acquire the bucket brigade (since the upstream filter passes it to them as an argument), they still block in a similar fashion to input filters, since they have to wait for the `pass_brigade()` call to return. In this case, they are waiting to pass the data along rather than waiting to receive it.

Here is an example of a transparent output filter:

```
#file:MyApache2/FilterTransparent.pm (continued)
#----------------------------------------------
sub out {
    my ($f, $bb) = @_;

    my $rv = $f->next->pass_brigade($bb);
    return $rv unless $rv == APR::Const::SUCCESS;

    return Apache2::Const::OK;
}
1;
```

The *out()* filter passes `$bb` to the downstream filter unmodified. If you add print statements before and after the `pass_brigade()` call and configure the same filter twice, the print will show the blocking call.

The following diagram depicts a typical output filter data flow in addition to the program control flow:

output filter data flow

Similar to the input filters chain diagram, the arrows show the program control flow and in addition the black-headed arrows show the data flow. Again, it uses Perl pseudo-code for the mod_perl filter and C pseudo-code for the Apache filters and the brick walls represent the waiting. The diagram shows only part of the real HTTP response filters chain, where `...` stands for the omitted filters.

15.4 mod_perl Filters Declaration and Configuration

Now that we have laid out some basic concepts involved in filter use, we can look at how mod_perl filters are declared and configured.

15.4.1 Filter Priority Types

When Apache filters are configured they are inserted into the filters chain according to their priority type. In most cases when using one or two filters things will just work, however if you find that the order of filter invocation is wrong, you should consider the filter priority type. Unfortunately this information is available only in the Apache source code, unless it's documented in the module man pages. Numerical definitions of priority types, such as `AP_FTYPE_CONTENT_SET` and `AP_FTYPE_RESOURCE`, can be found in the Apache source distribution in *include/util_filter.h*.

As of this writing Apache comes with two core filters: `DEFLATE` and `INCLUDES`. Regardless of your configuration directives, e.g.,:

```
  SetOutputFilter DEFLATE
  SetOutputFilter INCLUDES
```

the `INCLUDES` filter will be inserted in the filters chain before the `DEFLATE` filter, even though it was configured after it. This is because the `DEFLATE` filter is of type `AP_FTYPE_CONTENT_SET` (20), whereas the `INCLUDES` filter is of type `AP_FTYPE_RESOURCE` (10).

As of this writing mod_perl provides two kind of filters with fixed priority type (the type is defined by the filter handler's attribute):

```
  Handler's Attribute        Priority            Value
  -------------------------------------------------
  FilterRequestHandler    AP_FTYPE_RESOURCE    10
  FilterConnectionHandler AP_FTYPE_PROTOCOL    30
```

Therefore `FilterRequestHandler` filters (10) will always be invoked before the `DEFLATE` filter (20), whereas `FilterConnectionHandler` filters (30) will be invoked after it. When two filters have the same priority (e.g., the `INCLUDES` filter (10) has the same priority as `FilterRequestHandler` filters (10)), they are run in the order they are configured. Therefore filters are inserted according to the configuration order when `PerlSetOutputFilter` or `PerlSetInputFilter` are used.

15.4.2 `PerlInputFilterHandler`

The `PerlInputFilterHandler` directive registers a filter, and inserts it into the relevant input filters chain.

This handler is of type `VOID`.

The handler's configuration scope is `DIR`.

Arguments

`PerlInputFilterHander` handlers are passed two arguments: an `Apache2::Filter` object; and an `APR::Brigade` object;

See the examples that follow in this chapter for further explanation.

Return

Filters are expected to return `Apache2::Const::OK` or `Apache2::Const::DECLINED`. See the examples that follow in this chapter for further explanation.

Examples

`PerlInputFilterHandler` handlers are automatically `AutoLoaded`, since they need to be compiled before the filter attributes can be accessed. Therefore if the filter handler subroutine is not called `handler`, you must preload the module containing the filter subroutine at server startup. A filter handler can be configured not to be `AutoLoaded`, using the `-` prefix. For example:

```
PerlInputFilterHandler -MyApache2::FilterTest::lc
```

The following sections include several examples that use the `PerlInputFilterHandler` handler.

15.4.3 *PerlOutputFilterHandler*

The `PerlOutputFilterHandler` directive registers a filter, and inserts it into the relevant output filters chain.

This handler is of type `VOID`.

The handler's configuration scope is `DIR`.

Arguments

`PerlOutputFilterHander` handlers are passed five arguments: an `Apache2::Filter` object; an `APR::Brigade` object; an `Apache2::Const :input_mode` constant; an `APR::Const :read_type` constant; and the number of bytes to read.

See the examples that follow in this chapter for further explanation.

Return

Examples

The following sections include several examples that use the `PerlOutputFilterHandler` handler.

Similar to `PerlInputFilterHandler`, `PerlOutputFilterHandler` handlers are automatically `AutoLoaded`.

15.4.4 `PerlSetInputFilter`

The `SetInputFilter` directive, documented at
http://httpd.apache.org/docs-2.0/mod/core.html#setinputfilter, sets the filter or filters which will process
client requests and POST input when they are received by the server (in addition to any filters configured
earlier).

To mix mod_perl and non-mod_perl input filters of the same priority nothing special should be done. For
example if we have an imaginary Apache filter `FILTER_FOO` and mod_perl filter
`MyApache2::FilterInputFoo`, this configuration:

```
SetInputFilter FILTER_FOO
PerlInputFilterHandler MyApache2::FilterInputFoo
```

will add both filters. However the order of their invocation might not be as you expect. To make the invo-
cation order the same as the insertion order, replace `SetInputFilter` with `PerlSetInputFilter`,
like so:

```
PerlSetInputFilter FILTER_FOO
PerlInputFilterHandler MyApache2::FilterInputFoo
```

Now the `FILTER_FOO` filter will always be executed before the `MyApache2::FilterInputFoo`
filter, since it was configured before `MyApache2::FilterInputFoo` (i.e., it'll apply its transforma-
tions on the incoming data last). The diagram below shows the input filters chain and the data flow from
the network to the response handler for the presented configuration:

```
       response handler
             /\
             ||
        FILTER_FOO
             /\
             ||
  MyApache2::FilterInputFoo
             /\
             ||
    core input filters
             /\
             ||
          network
```
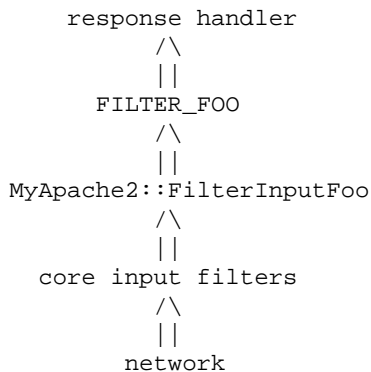
As explained in the section Filter Priority Types this directive won't affect filters of different priority. For
example assuming that `MyApache2::FilterInputFoo` is a `FilterRequestHandler` filter, the
configurations:

```
PerlInputFilterHandler MyApache2::FilterInputFoo
PerlSetInputFilter DEFLATE
```

and

```
  PerlSetInputFilter DEFLATE
  PerlInputFilterHandler MyApache2::FilterInputFoo
```

are equivalent, because mod_deflate's `DEFLATE` filter has a higher priority than `MyApache2::FilterInputFoo`. Thefore, it will always be inserted into the filter chain after `MyApache2::FilterInputFoo`, (i.e. the `DEFLATE` filter will apply its transformations on the incoming data first). The diagram below shows the input filters chain and the data flow from the network to the response handler for the presented configuration:

```
      response handler
             /\
             ||
  MyApache2::FilterInputFoo
             /\
             ||
          DEFLATE
             /\
             ||
      core input filters
             /\
             ||
          network
```
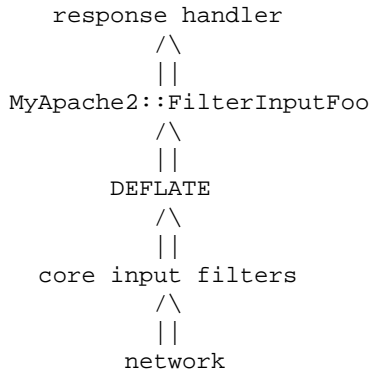
`SetInputFilter`'s `;` semantics are supported as well. For example, in the following configuration:

```
  PerlInputFilterHandler MyApache2::FilterInputFoo
  PerlSetInputFilter FILTER_FOO;FILTER_BAR
```

`MyApache2::FilterOutputFoo` will be executed first, followed by `FILTER_FOO` and finally by `FILTER_BAR` (again, assuming that all three filters have the same priority).

15.4.5 *PerlSetOutputFilter*

The `SetOutputFilter` directive, documented at *http://httpd.apache.org/docs-2.0/mod/core.html#setoutputfilter* sets the filters which will process responses from the server before they are sent to the client (in addition to any filters configured earlier).

To mix mod_perl and non-mod_perl output filters of the same priority nothing special should be done. This configuration:

```
  SetOutputFilter INCLUDES
  PerlOutputFilterHandler MyApache2::FilterOutputFoo
```

As with input filters, to preserve the insertion order replace `SetOutputFilter` with `PerlSetOutputFilter`, like so:

```
  PerlSetOutputFilter INCLUDES
  PerlOutputFilterHandler MyApache2::FilterOutputFoo
```

Now mod_include's `INCLUDES` filter will always be executed before the `MyApache2::FilterOut-putFoo` filter. The diagram below shows the output filters chain and the data flow from the response handler to the network for the presented configuration:

```
    response handler
          ||
          \/
       INCLUDES
          ||
          \/
 MyApache2::FilterOutputFoo
          ||
          \/
   core output filters
          ||
          \/
        network
```
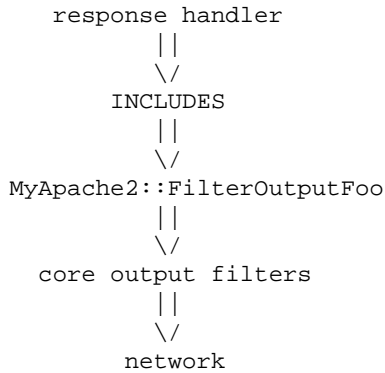
`SetOutputFilter`'s `;` semantics are supported as well. For example, in the following configuration:

```
  PerlOutputFilterHandler MyApache2::FilterOutputFoo
  PerlSetOutputFilter INCLUDES;FILTER_FOO
```

`MyApache2::FilterOutputFoo` will be executed first, followed by `INCLUDES` and finally by `FILTER_FOO` (again, assuming that all three filters have the same priority).

As explained in the `PerlSetInputFilter` section, if filters have different priorities, the insertion order might be different. For example in the following configuration:

```
  PerlSetOutputFilter DEFLATE
  PerlSetOutputFilter INCLUDES
  PerlOutputFilterHandler MyApache2::FilterOutputFoo
```

mod_include's `INCLUDES` filter will be always executed before the `MyApache2::FilterOutput-Foo` filter. The latter will be followed by mod_deflate's `DEFLATE` filter, even though it was configured before the other two filters. This is because it has a higher priority. And the corresponding diagram looks like so:

```
    response handler
          ||
          \/
       INCLUDES
          ||
          \/
 MyApache2::FilterOutputFoo
          ||
          \/
       DEFLATE
          ||
          \/
   core output filters
          ||
          \/
        network
```
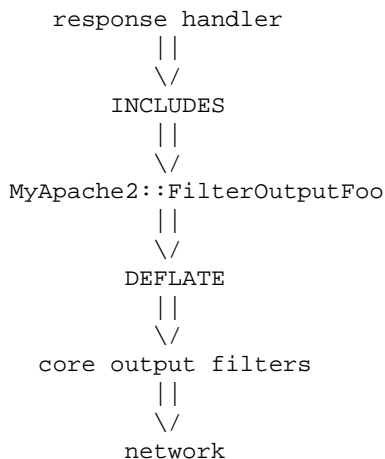
15.4.6 Adding OutputFilters Dynamically

If you have the need to add output filters dymically during the request, mod_perl 2.0 offers you the possibility to push filter callbacks at request time. For example here is how to add an output filter during the Fixup phase:

```
<Files *\.html >
  PerlFixupHandler MyApache2::AddFilterDyn
</Files>
```

And the corresponding module is:

```
#file:MyApache2/AddFilterDyn.pm
#----------------------------
package MyApache2::AddFilterDyn;

use Apache2::Filter;
use MyApache2::FilterObfuscate;

use Apache2::Const -compile => qw(OK);

sub handler {
    my $r = shift;

    $r->add_output_filter(\&MyApache2::FilterObfuscate::handler);

    return Apache2::Const::OK;
}

1;
```

You can also add connection filters dynamically. For more information refer to the `Apache2::Filter` manpages `add_input_filter` and `add_output_filter`.

15.4.7 HTTP Request vs. Connection Filters

mod_perl 2.0 supports connection and HTTP request filtering. mod_perl filter handlers specify the type of the filter using the method attributes.

HTTP request filter handlers are declared using the `FilterRequestHandler` attribute. Consider the following request input and output filters skeleton:

```
package MyApache2::FilterRequestFoo;
use base qw(Apache2::Filter);

sub input  : FilterRequestHandler {
    my ($f, $bb, $mode, $block, $readbytes) = @_;
    #...
}

sub output : FilterRequestHandler {
    my ($f, $bb) = @_;
```

```
      #...
  }

  1;
```

If the attribute is not specified, the default `FilterRequestHandler` attribute is assumed. Filters specifying subroutine attributes must subclass `Apache2::Filter`, others only need to:

```
  use Apache2::Filter ();
```

Request filters are usually configured in the `<Location>` or equivalent sections:

```
  PerlModule MyApache2::FilterRequestFoo
  PerlModule MyApache2::NiceResponse
  <Location /filter_foo>
      SetHandler modperl
      PerlResponseHandler    MyApache2::NiceResponse
      PerlInputFilterHandler  MyApache2::FilterRequestFoo::input
      PerlOutputFilterHandler MyApache2::FilterRequestFoo::output
  </Location>
```

Now we have the request input and output filters configured.

The connection filter handler uses the `FilterConnectionHandler` attribute. Here is a similar example for the connection input and output filters.

```
  package MyApache2::FilterConnectionBar;
  use base qw(Apache2::Filter);

  sub input  : FilterConnectionHandler {
      my ($f, $bb, $mode, $block, $readbytes) = @_;
      #...
  }

  sub output : FilterConnectionHandler {
      my ($f, $bb) = @_;
      #...
  }

  1;
```

With connection filters, unlike the request flters, the configuration must be done outside the `<Location>` or equivalent sections, usually within the `<VirtualHost>` or the global server configuration:

```
  Listen 8005
  <VirtualHost _default_:8005>
      PerlModule MyApache2::FilterConnectionBar
      PerlModule MyApache2::NiceResponse

      PerlInputFilterHandler  MyApache2::FilterConnectionBar::input
      PerlOutputFilterHandler MyApache2::FilterConnectionBar::output
      <Location />
          SetHandler modperl
```

```
            PerlResponseHandler MyApache2::NiceResponse
        </Location>

    </VirtualHost>
```

This configures the connection input and output filters.

As can be seen from the above examples, the only difference between connection filters and request filters is that that connection filters see everything from the request, i.e., the headers and the body, whereas request filters see only the body.

15.4.8 Filter Initialization Phase

There is one more callback in the filter framework. And that's `FilterInitHandler`. This *init* callback runs immediately after the filter handler is inserted into the filter chain, before it is invoked for the first time. Here is a skeleton of an init handler:

```
sub init : FilterInitHandler {
    my $f = shift;
    #...
    return Apache2::Const::OK;
}
```

The attribute `FilterInitHandler` marks the Perl function as suitable to be used as a filter initialization callback.

For example you may decide to dynamically remove a filter before it had a chance to run, if some condition is true:

```
sub init : FilterInitHandler {
    my $f = shift;
    $f->remove() if should_remove_filter();
    return Apache2::Const::OK;
}
```

Not all `Apache2::Filter` methods can be used in the init handler, because it's not a filter. Hence you can use methods that operate on the filter itself, such as `remove()` and `ctx()` or retrieve request information, such as `r()` and `c()`. You cannot use methods that operate on data, such as `read()` and `print()`.

In order to hook an init filter handler, the real filter has to assign this callback using the `Filter-HasInitHandler` function which accepts a reference to the callback function, similar to `push_handlers()`. The callback function referred to must have the `FilterInitHandler` attribute. For example:

```
package MyApache2::FilterBar;
use base qw(Apache2::Filter);
sub init   : FilterInitHandler { ... }
sub filter : FilterRequestHandler FilterHasInitHandler(\&init) {
    my ($f, $bb) = @_;
    # ...
    return Apache2::Const::OK;
}
```

While attributes are parsed during compilation (it's really a sort of source filter), the argument to the
`FilterHasInitHandler()` attribute is compiled at a later stage once the module is compiled.

The argument to `FilterHasInitHandler()` can be any Perl code which when `eval()`'ed returns a
code reference. For example:

```
package MyApache2::OtherFilter;
use base qw(Apache2::Filter);
sub init  : FilterInitHandler { ... }

package MyApache2::FilterBar;
use MyApache2::OtherFilter;
use base qw(Apache2::Filter);
sub get_pre_handler { \&MyApache2::OtherFilter::init }
sub filter : FilterHasInitHandler(get_pre_handler()) { ... }
```

Here   the   `MyApache2::FilterBar::filter`   handler   is   configured   to   run   the
`MyApache2::OtherFilter::init` init handler.

Notice that the argument to `FilterHasInitHandler()` is always `eval()`'ed in the package of the
real filter handler (not the init handler). So the above code leads to the following evaluation:

```
$init_sub = eval "package MyApache2::FilterBar; get_pre_handler()";
```

This part is actually done in C, using the `eval_pv()` C call.

Currently only one initialization callback can be registered per filter handler.

15.5 All-in-One Filter

Before we delve into the details of how to write filters that do something with the data, lets first write a
simple filter that does nothing but snooping on the data that goes through it. We are going to develop the
`MyApache2::FilterSnoop` handler which can snoop on request and connection filters, in input and
output modes.

First we create a simple response handler that dumps the request's *args* and *content* as strings:

```
#file:MyApache2/Dump.pm
#--------------------
package MyApache2::Dump;

use strict;
use warnings;
```

```
use Apache2::RequestRec ();
use Apache2::RequestIO ();
use Apache2::Filter ();
use APR::Brigade ();
use APR::Bucket ();

use Apache2::Const -compile => qw(OK M_POST);

sub handler {
    my $r = shift;
    $r->content_type('text/plain');

    $r->print("args:\n", $r->args, "\n");

    if ($r->method_number == Apache2::Const::M_POST) {
        my $data = content($r);
        $r->print("content:\n$data\n");
    }

    return Apache2::Const::OK;
}

use Apache2::Connection ();

use Apache2::Const -compile => qw(MODE_READBYTES);
use APR::Const    -compile => qw(SUCCESS BLOCK_READ);

use constant IOBUFSIZE => 8192;

sub content {
    my $r = shift;

    my $bb = APR::Brigade->new($r->pool, $r->connection->bucket_alloc);


    my $data = '';
    my $seen_eos = 0;
    do {
        $r->input_filters->get_brigade($bb, Apache2::Const::MODE_READBYTES,
                                       APR::Const::BLOCK_READ, IOBUFSIZE);

        for (my $b = $bb->first; $b; $b = $bb->next($b)) {
            if ($b->is_eos) {
                $seen_eos++;
                last;
            }

            if ($b->read(my $buf)) {
                $data .= $buf;
            }

            $b->remove; # optimization to reuse memory
        }
    } while (!$seen_eos);

    $bb->destroy;
```

```
        return $data;
    }

 1;
```

which is configured as:

```
  PerlModule MyApache2::Dump
  <Location /dump>
      SetHandler modperl
      PerlResponseHandler MyApache2::Dump
  </Location>
```

If we issue the following request:

```
  % echo "mod_perl rules" | POST 'http://localhost:8002/dump?foo=1&bar=2'
```

the response will be:

```
  args:
  foo=1&bar=2
  content:
  mod_perl rules
```

As you can see it simply dumped the query string and the posted data.

Now let's write the snooping filter:

```
  #file:MyApache2/FilterSnoop.pm
  #---------------------------
  package MyApache2::FilterSnoop;

  use strict;
  use warnings;

  use base qw(Apache2::Filter);
  use Apache2::FilterRec ();
  use APR::Brigade ();
  use APR::Bucket ();
  use APR::BucketType ();

  use Apache2::Const -compile => qw(OK DECLINED);
  use APR::Const     -compile => ':common';

  sub connection : FilterConnectionHandler { snoop("connection", @_) }
  sub request    : FilterRequestHandler    { snoop("request",    @_) }

  sub snoop {
      my $type = shift;
      my ($f, $bb, $mode, $block, $readbytes) = @_; # filter args

      # $mode, $block, $readbytes are passed only for input filters
      my $stream = defined $mode ? "input" : "output";

      # read the data and pass-through the bucket brigades unchanged
```

```perl
    if (defined $mode) {
        # input filter
        my $rv = $f->next->get_brigade($bb, $mode, $block, $readbytes);
        return $rv unless $rv == APR::Const::SUCCESS;
        bb_dump($type, $stream, $bb);
    }
    else {
        # output filter
        bb_dump($type, $stream, $bb);
        my $rv = $f->next->pass_brigade($bb);
        return $rv unless $rv == APR::Const::SUCCESS;
    }

    return Apache2::Const::OK;
}

sub bb_dump {
    my ($type, $stream, $bb) = @_;

    my @data;
    for (my $b = $bb->first; $b; $b = $bb->next($b)) {
        $b->read(my $bdata);
        push @data, $b->type->name, $bdata;
    }

    # send the sniffed info to STDERR so not to interfere with normal
    # output
    my $direction = $stream eq 'output' ? ">>>" : "<<<";
    print STDERR "\n$direction $type $stream filter\n";

    my $c = 1;
    while (my ($btype, $data) = splice @data, 0, 2) {
        print STDERR "    o bucket $c: $btype\n";
        print STDERR "[$data]\n";
        $c++;
    }
}
}
1;
```

Recall that there are two types of two filter handlers, one for connection and another for request filtering:

```perl
sub connection : FilterConnectionHandler { snoop("connection", @_) }
sub request    : FilterRequestHandler    { snoop("request",    @_) }
```

Both handlers forward their arguments to the `snoop()` function, which does the real work. These two subroutines are added in order to assign the two different attributes. In addition, the functions pass the filter type to `snoop()` as the first argument, which gets shifted off @_. The rest of @_ are the arguments that were originally passed to the filter handler.

It's easy to know whether a filter handler is running in the input or the output mode. Although the arguments `$f` and `$bb` are always passed, the arguments `$mode`, `$block`, and `$readbytes` are passed only to input filter handlers.

If we are in input mode, in the same call we retrieve the bucket brigade from the previous filter on the input filters stack and immediately link it to the `$bb` variable which makes the bucket brigade available to the next input filter when the filter handler returns. If we forget to perform this linking our filter will become a black hole into which data simply disappears. Next we call `bb_dump()` which dumps the type of the filter and the contents of the bucket brigade to STDERR, without influencing the normal data flow.

If we are in output mode, the `$bb` variable already points to the current bucket brigade. Therefore we can read the contents of the brigade right away, and then we pass the brigade to the next filter.

Let's snoop on connection and request filter levels in both directions by applying the following configuration:

```
Listen 8008
<VirtualHost _default_:8008>
    PerlModule MyApache2::FilterSnoop
    PerlModule MyApache2::Dump

    # Connection filters
    PerlInputFilterHandler  MyApache2::FilterSnoop::connection
    PerlOutputFilterHandler MyApache2::FilterSnoop::connection

    <Location /dump>
        SetHandler modperl
        PerlResponseHandler MyApache2::Dump
        # Request filters
        PerlInputFilterHandler  MyApache2::FilterSnoop::request
        PerlOutputFilterHandler MyApache2::FilterSnoop::request
    </Location>

</VirtualHost>
```

Notice that we use a virtual host because we want to install connection filters.

If we issue the following request:

```
% echo "mod_perl rules" | POST 'http://localhost:8008/dump?foo=1&bar=2'
```

we get the same response as before we installed `MyApache2::FilterSnoop` because our snooping filter didn't change anything. The output didn't change, but there was some new information printed to the *error_log*. We present it all here, in order to understand how filters work.

First we can see the connection input filter at work, as it processes the HTTP headers. We can see that for this request each header is put into a separate brigade with a single bucket. The data is conveniently enclosed by `[ ]` so you can see the new line characters as well.

```
<<< connection input filter
    o bucket 1: HEAP
[POST /dump?foo=1&bar=2 HTTP/1.1
]

<<< connection input filter
    o bucket 1: HEAP
[TE: deflate,gzip;q=0.3
```

```
    ]

  <<< connection input filter
      o bucket 1: HEAP
[Connection: TE, close
  ]

  <<< connection input filter
      o bucket 1: HEAP
[Host: localhost:8008
  ]

  <<< connection input filter
      o bucket 1: HEAP
[User-Agent: lwp-request/2.01
  ]

  <<< connection input filter
      o bucket 1: HEAP
[Content-Length: 14
  ]

  <<< connection input filter
      o bucket 1: HEAP
[Content-Type: application/x-www-form-urlencoded
  ]

  <<< connection input filter
      o bucket 1: HEAP
  [
  ]
```

Here the HTTP header has been terminated by a double new line. So far all the buckets were of the *HEAP*
type, meaning that they were allocated from the heap memory. Notice that the HTTP request input filters
will never see the bucket brigades with HTTP headers because they are consumed by the last core connec-
tion filter.

The following two entries are generated when `MyApache2::Dump::handler` reads the POSTed
content:

```
  <<< connection input filter
      o bucket 1: HEAP
[mod_perl rules]

  <<< request input filter
      o bucket 1: HEAP
[mod_perl rules]
      o bucket 2: EOS
  []
```

As shown earlier, the connection input filter is run before the request input filter. Since our connection
input filter was passing the data through unmodified and no other custom connection input filter was
configured, the request input filter sees the same data. The last bucket in the brigade received by the
request input filter is of type *EOS*, meaning that all the input data from the current request has been

received.

Next we can see that `MyApache2::Dump::handler` has generated its response. However we can see that only the request output filter gets run at this point:

```
>>> request output filter
    o bucket 1: TRANSIENT
[args:
foo=1&bar=2
content:
mod_perl rules
 ]
```

This happens because Apache hasn't yet sent the response HTTP headers to the client. The request filter sees a bucket brigade with a single bucket of type *TRANSIENT* which is allocated from the stack memory.

The moment the first bucket brigade of the response body has entered the connection output filters, Apache injects a bucket brigade with the HTTP headers. Therefore we can see that the connection output filter is filtering the brigade with HTTP headers (notice that the request output filters don't see it):

```
>>> connection output filter
    o bucket 1: HEAP
[HTTP/1.1 200 OK
Date: Tue, 07 Mar 2006 10:59:08 GMT
Server: Apache/2.0.55 (Unix) mod_perl/2.000002
Perl/v5.8.4 mod_ssl/2.0.55 OpenSSL/0.9.7c DAV/2
Connection: close
Transfer-Encoding: chunked
Content-Type: text/plain; charset=ISO-8859-1

 ]
```

This is followed by the first response body's brigade:

```
>>> connection output filter
    o bucket 1: TRANSIENT
[2b
 ]
    o bucket 2: TRANSIENT
[args:
foo=1&bar=2
content:
mod_perl rules

 ]
    o bucket 3: IMMORTAL
[
 ]
```

If the response is large, the request and connection filters will filter chunks of the response one by one. These chunks are typically 8k in size, but this size can vary.

Finally, Apache sends a series of bucket brigades to finish off the response, including the end of stream meta-bucket to tell filters that they shouldn't expect any more data, and flush buckets to flush the data, to make sure that any buffered output is sent to the client:

```
>>> connection output filter
    o bucket 1: IMMORTAL
[0

]
    o bucket 2: EOS
[]

>>> connection output filter
    o bucket 1: FLUSH
[]

>>> connection output filter
    o bucket 1: FLUSH
[]
```

This module helps to illustrate that each filter handler can be called many times during each request and connection. It is called for each bucket brigade. Also it is important to mention that HTTP request input filters are invoked only if there is some POSTed data to read and it's consumed by a content handler.

15.6 Input Filters

mod_perl supports Connection and HTTP Request input filters. In the following sections we will look at each of these in turn.

15.6.1 Connection Input Filters

Let's say that we want to test how our handlers behave when they are requested as HEAD requests, rather than GET requests. We can alter the request headers at the incoming connection level with the alteration transparent to all handlers.

This example's filter handler looks for data like:

```
GET /perl/test.pl HTTP/1.1
```

and turns it into:

```
HEAD /perl/test.pl HTTP/1.1
```

The following input filter handler does that by directly manipulating the bucket brigades:

```
#file:MyApache2/InputFilterGET2HEAD.pm
#---------------------------------
package MyApache2::InputFilterGET2HEAD;

use strict;
use warnings;
```

```
use base qw(Apache2::Filter);

use APR::Brigade ();
use APR::Bucket ();

use Apache2::Const -compile => qw(OK DECLINED);
use APR::Const     -compile => ':common';

sub handler : FilterConnectionHandler {
    my ($f, $bb, $mode, $block, $readbytes) = @_;

    return Apache2::Const::DECLINED if $f->ctx;

    my $rv = $f->next->get_brigade($bb, $mode, $block, $readbytes);
    return $rv unless $rv == APR::Const::SUCCESS;

    for (my $b = $bb->first; $b; $b = $bb->next($b)) {
        $b->read(my $data);
        warn("data: $data\n");

        if ($data and $data =~ s|^GET|HEAD|) {
            my $nb = APR::Bucket->new($bb->bucket_alloc, $data);
            $b->insert_after($nb);
            $b->remove; # no longer needed
            $f->ctx(1); # flag that that we have done the job
            last;
        }
    }

    return Apache2::Const::OK;
}
1;
```

The filter handler is called for each bucket brigade, which then includes buckets with data. The gist of any input filter handler is to request the bucket brigade from the upstream filter, and return it to the downstream filter using the second argument $bb. It's important to remember that you can call methods on this argument, but you shouldn't assign to this argument, or the chain will be broken.

There are two techniques to choose from to retrieve-modify-return bucket brigades:

1. Create a new empty bucket brigade $ctx_bb, pass it to the upstream filter via get_brigade() and wait for this call to return. When it returns, $ctx_bb will be populated with buckets. Now the filter should move the bucket from $ctx_bb to $bb, on the way modifying the buckets if needed. Once the buckets are moved, and the filter returns, the downstream filter will receive the populated bucket brigade.

2. Pass $bb to the upstream filter using get_brigade() so it will be populated with buckets. Once get_brigade() returns, the filter can go through the buckets and modify them in place, or it can do nothing and just return (in which case, the downstream filter will receive the bucket brigade unmodified).

Both techniques allow addition and removal of buckets. Though the second technique is more efficient since it doesn't have the overhead of create the new brigade and moving the bucket from one brigade to another. In this example we have chosen to use the second technique, in the next example we will see the first technique.

Our filter has to perform the substitution of only one HTTP header (which normally resides in one bucket), so we have to make sure that no other data gets mangled (e.g. there could be POSTED data and it may match /^GET/ in one of the buckets). We use $f->ctx as a flag here. When it's undefined the filter knows that it hasn't done the required substitution, though once it completes the job it sets the context to 1.

Using the information stored in the context, the filter can immediately return Apache2::Const::DECLINED when it's invoked after the substitution job has been done:

```
return Apache2::Const::DECLINED if $f->ctx;
```

In that case mod_perl will call get_brigade() internally which will pass the bucket brigade to the downstream filter. Alternatively the filter could do:

```
my $rv = $f->next->get_brigade($bb, $mode, $block, $readbytes);
return $rv unless $rv == APR::Const::SUCCESS;
return Apache2::Const::OK if $f->ctx;
```

but this is a bit less efficient.

[META: the most efficient thing to do is to remove the filter itself once the job is done, so it won't be even invoked after the job has been done.

```
if ($f->ctx) {
    $f->remove;
    return Apache2::Const::DECLINED;
}
```

However, this can't be used with Apache 2.0.49 and lower, since it has a bug when trying to remove the edge connection filter (it doesn't remove it). Most likely that problem will be not fixed in the 2.0 series due to design flows. I don't know if it's going to be fixed in 2.1 series.]

If the job wasn't done yet, the filter calls get_brigade, which populates the $bb bucket brigade. Next, the filter steps through the buckets looking for the bucket that matches the regex: /^GET/. If a match is found, a new bucket is created with the modified data (s/^GET/HEAD/. Now it has to be inserted in place of the old bucket. In our example we insert the new bucket after the bucket that we have just modified and immediately remove that bucket that we don't need anymore:

```
$b->insert_after($nb);
$b->remove; # no longer needed
```

Finally we set the context to 1, so we know not to apply the substitution on the following data, and break from the *for* loop.

The handler returns `Apache2::Const::OK` indicating that everything was fine. The downstream filter will receive the bucket brigade with one bucket modified.

Now let's check that the handler works properly. For example, consider the following response handler:

```
#file:MyApache2/RequestType.pm
#--------------------------
package MyApache2::RequestType;

use strict;
use warnings;

use Apache2::RequestIO ();
use Apache2::RequestRec ();
use Apache2::Response ();

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

sub handler {
    my $r = shift;

    $r->content_type('text/plain');
    my $response = "the request type was " . $r->method;
    $r->set_content_length(length $response);
    $r->print($response);

    return Apache2::Const::OK;
}

1;
```

This handler returns to the client the request type it has issued. For a `HEAD` request Apache will discard the response body, but it will still set the correct `Content-Length` header, which will be 24 for a `GET` request and 25 for a `HEAD` request. Therefore, if this response handler is configured as:

```
Listen 8005
<VirtualHost _default_:8005>
    <Location />
        SetHandler modperl
        PerlResponseHandler +MyApache2::RequestType
    </Location>
</VirtualHost>
```

and a `GET` request is issued to */*:

```
panic% perl -MLWP::UserAgent -le \
'$r = LWP::UserAgent->new()->get("http://localhost:8005/"); \
print $r->headers->content_length . ": ". $r->content'
24: the request type was GET
```

the response body will be:

```
the request type was GET
```

and the `Content-Length` header will be set to 24. This is what we would expect since the request was
processed normally. However, if we enable the `MyApache2::InputFilterGET2HEAD` input connec-
tion filter:

```
Listen 8005
<VirtualHost _default_:8005>
    PerlInputFilterHandler +MyApache2::InputFilterGET2HEAD

    <Location />
        SetHandler modperl
        PerlResponseHandler +MyApache2::RequestType
    </Location>
</VirtualHost>
```

and issue the same `GET` request, we get only:

```
25:
```

This means the body was discarded by Apache, because our filter turned the `GET` request into a `HEAD`
request. If Apache wasn't discarding the body on `HEAD`, the response would be:

```
the request type was HEAD
```

That's why the content length is reported as 25 and not 24 as in the real `GET` request. So the content
length of 25 and lack of body text in the response confirm that our filter is acting as we expected.

15.6.2 HTTP Request Input Filters

Request filters are really non-different from connection filters, other than that they are working on request
and response bodies and have an access to a request object.

15.6.3 Bucket Brigade-based Input Filters

As we have seen, filters can be either bucket brigade based, or stream oriented. Here we look at a request
input filter that lowercases the request's body by directly manipulating the bucket brigade:
`MyApache2::InputRequestFilterLC`.

```
#file:MyApache2/InputRequestFilterLC.pm
#----------------------------------
package MyApache2::InputRequestFilterLC;

use strict;
use warnings;

use base qw(Apache2::Filter);

use Apache2::Connection ();
use APR::Brigade ();
use APR::Bucket ();
```

```
  use Apache2::Const -compile => 'OK';
  use APR::Const      -compile => ':common';

  sub handler : FilterRequestHandler {
      my ($f, $bb, $mode, $block, $readbytes) = @_;

      my $c = $f->c;
      my $bb_ctx = APR::Brigade->new($c->pool, $c->bucket_alloc);
      my $rv = $f->next->get_brigade($bb_ctx, $mode, $block, $readbytes);
      return $rv unless $rv == APR::Const::SUCCESS;

      while (!$bb_ctx->is_empty) {
          my $b = $bb_ctx->first;

          if ($b->is_eos) {
              $bb->insert_tail($b);
              last;
          }

          my $len = $b->read(my $data);
          $b = APR::Bucket->new($bb->bucket_alloc, lc $data) if $len;

          $b->remove;
          $bb->insert_tail($b);
      }

      return Apache2::Const::OK;

  }

  1;
```

As promised, in this filter handler we have used the first technique of bucket brigade modification. The handler creates a temporary bucket brigade (`ctx_bb`), populates it with data using `get_brigade()`, and then moves buckets from it to the bucket brigade `$bb`. This bucket brigade is then retrieved by the downstream filter when our handler returns.

This filter doesn't need to know whether it was invoked for the first time or whether it has already done something. It's a stateless handler, since it has to lower case everything that passes through it. Notice that this filter can't be used as the connection filter for HTTP requests, since it will invalidate the incoming request headers. For example the first header line:

```
  GET /perl/TEST.pl HTTP/1.1
```

becomes:

```
  get /perl/test.pl http/1.1
```

which invalidates the request method, the URL and the protocol.

To test, we can use the `MyApache2::Dump` response handler, presented earlier, which dumps the query string and the content body as a response. Configure the server as follows:

```
<Location /lc_input>
    SetHandler modperl
    PerlResponseHandler    +MyApache2::Dump
    PerlInputFilterHandler +MyApache2::InputRequestFilterLC
</Location>
```

Now when issuing a POST request:

```
% echo "mOd_pErl RuLeS" | POST 'http://localhost:8002/lc_input?FoO=1&BAR=2'
```

we get a response:

```
args:
FoO=1&BAR=2
content:
mod_perl rules
```

We can see that our filter has lowercased the POSTed body before the content handler received it. And you can see that the query string wasn't changed.

We have devoted so much attention to bucket brigade filters, even though they are simple to manipulate, because it is important to understand how the filters work underneath. This understanding is essential when you need to debug filters or to optimize them. There are cases when a bucket brigade filter may be more efficient than the stream-oriented version. For example if the filter applies a transformation to selected buckets, certain buckets may contain open filehandles or pipes, rather than real data. When you call read(), as shown above, the buckets will be forced to read that data in. But if you didn't want to modify these buckets you could pass them as they are and let Apache perform faster techniques for sending data from the file handles or pipes.

The call to $b->read(), or any other operation that internally forces the bucket to read the information into the memory (like the length() op), makes the data handling less efficient because it creates more work. Therefore care should be taken so not to read the data in unless it's really necessary, and sometimes you can gain this efficiency only by working with the bucket brigades.

15.6.4 Stream-oriented Input Filters

Let's now look at the same filter implemented using the stream-oriented API.

```
#file:MyApache2/InputRequestFilterLC2.pm
#------------------------------------
package MyApache2::InputRequestFilterLC2;

use strict;
use warnings;

use base qw(Apache2::Filter);

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

use constant BUFF_LEN => 1024;

sub handler : FilterRequestHandler {
```

```
    my $f = shift;

    while ($f->read(my $buffer, BUFF_LEN)) {
        $f->print(lc $buffer);
    }

    Apache2::Const::OK;
}
1;
```

The logic is very simple here. The filter reads in a loop and prints the modified data, which at some point will be sent to the next filter. The data transmission is triggered every time the internal mod_perl buffer is filled or when the filter returns.

read() populates $buffer to a maximum of BUFF_LEN characters (1024 in our example). Assuming that the current bucket brigade contains 2050 chars, read() will get the first 1024 characters, then 1024 characters more and finally the remaining 2 characters. Note that even though the response handler may have sent more than 2050 characters, every filter invocation operates on a single bucket brigade so you have to wait for the next invocation to get more input. Earlier we showed that you can force the generation of several bucket brigades in the content handler by using rflush(). For example:

```
$r->print("string");
$r->rflush();
$r->print("another string");
```

It's only possible to get more than one bucket brigade from the same filter handler invocation if the filter is not using the streaming interface. In that case you can call get_brigade() as many times as needed or until EOS is received.

The configuration section is nearly identical for the two types of filters:

```
<Location /lc_input2>
    SetHandler modperl
    PerlResponseHandler    +MyApache2::Dump
    PerlInputFilterHandler +MyApache2::InputRequestFilterLC2
</Location>
```

When issuing a POST request:

```
% echo "mOd_pErl RuLeS" | POST 'http://localhost:8002/lc_input2?FoO=1&BAR=2'
```

we get the response:

```
args:
FoO=1&BAR=2
content:
mod_perl rules
```

As before, we see that our filter has lowercased the POSTed body before the content handler received it and the query string wasn't changed.

15.7 Output Filters

As discussed above in the section HTTP Request vs. Connection Filters, mod_perl supports Connection and HTTP Request output filters. In the following sections we will look at each of these in turn.

mod_perl supports Connection and HTTP Request output filters. The differences between connection filters and HTTP request filters are described above in the section HTTP Request vs. Connection Filters.

15.7.1 Connection Output Filters

Connection filters filter **all** the data that is going through the server. Therefore if the connection is of the HTTP request type, connection output filters see the headers and the body of the response, whereas request output filters see only the response body.

META: for now see the request output filter explanations and examples, connection output filter examples will be added soon. Interesting ideas for such filters are welcome (possible ideas: mangling output headers for HTTP requests, pretty much anything for protocol modules).

15.7.2 HTTP Request Output Filters

As mentioned earlier, output filters can be written using the bucket brigades manipulation or the simplified stream-oriented interface. This section will show examples of both.

In order to generate output that can be manipulated by the two types of output filters, we will first develop a response handler that sends two lines of output: numerals 1234567890 and the English alphabet in a single string:

```
#file:MyApache2/SendAlphaNum.pm
#-----------------------------
package MyApache2::SendAlphaNum;

use strict;
use warnings;

use Apache2::RequestRec ();
use Apache2::RequestIO ();

use Apache2::Const -compile => qw(OK);

sub handler {
    my $r = shift;

    $r->content_type('text/plain');

    $r->print(1..9, "0\n");
    $r->print('a'..'z', "\n");

    return Apache2::Const::OK;
}
1;
```

In the examples below, we'll create a filter handler to reverse every line of the response body, preserving the new line characters in their places. Since we want to reverse characters only in the response body, without breaking the HTTP headers, we will use the HTTP request output filter rather than a connection output filter.

15.7.2.1 Stream-oriented Output Filters

The first filter implementation uses the stream-oriented filtering API:

```
#file:MyApache2/FilterReverse1.pm
#---------------------------
package MyApache2::FilterReverse1;

use strict;
use warnings;

use base qw(Apache2::Filter);

use Apache2::Const -compile => qw(OK);

use constant BUFF_LEN => 1024;

sub handler : FilterRequestHandler {
    my $f = shift;

    while ($f->read(my $buffer, BUFF_LEN)) {
        for (split "\n", $buffer) {
            $f->print(scalar reverse $_);
            $f->print("\n");
        }
    }

    return Apache2::Const::OK;
}
1;
```

Next, we add the following configuration to *httpd.conf*:

```
PerlModule MyApache2::FilterReverse1
PerlModule MyApache2::SendAlphaNum
<Location /reverse1>
    SetHandler modperl
    PerlResponseHandler     MyApache2::SendAlphaNum
    PerlOutputFilterHandler MyApache2::FilterReverse1
</Location>
```

Now when a request to *reverse1* is sent, the response handler `MyApache2::SendAlphaNum::handler()` sends:

```
1234567890
abcdefghijklmnopqrstuvwxyz
```

as a response. The output filter handler `MyApache2::FilterReverse1::handler` reverses the lines, so the client gets:

```
0987654321
zyxwvutsrqponmlkjihgfedcba
```

The `Apache2::Filter` module loads the `read()` and `print()` methods which encapsulate the stream-oriented filtering interface.

The reversing filter is quite simple: in the loop it reads the data in the *readline()* mode in chunks up to the buffer length (1024 in our example), and then prints each line reversed while preserving the new line control characters at the end of each line. Behind the scenes `$f->read()` retrieves the incoming brigade and gets the data from it, and `$f->print()` appends to the new brigade which is then sent to the next filter in the stack. `read()` breaks the *while* loop when the brigade is emptied or the end of stream is received.

While this code is simple and easy to explain, there are cases it won't handle correctly. For example, it will have problems if the input lines are longer than 1,024 characters. It also doesn't account for the different line terminators on different platforms (e.g., "\n", "\r", or "\r\n"). Moreover a single line may be split across two or even more bucket brigades, so we have to store the unprocessed string in the filter context so it can be used on the following invocations. Below is an example of a more complete handler, which takes care of these issues:

```
sub handler {
    my $f = shift;

    my $leftover = $f->ctx;
    while ($f->read(my $buffer, BUFF_LEN)) {
        $buffer = $leftover . $buffer if defined $leftover;
        $leftover = undef;
        while ($buffer =~ /([^\r\n]*)([\r\n]*)/g) {
            $leftover = $1, last unless $2;
            $f->print(scalar(reverse $1), $2);
        }
    }

    if ($f->seen_eos) {
        $f->print(scalar reverse $leftover) if defined $leftover;
    }
    else {
        $f->ctx($leftover) if defined $leftover;
    }

    return Apache2::Const::OK;
}
```

The handler uses the `$leftover` variable to store unprocessed data as long as it fails to assemble a complete line or there is an incomplete line following the new line token. On the next handler invocation this data is then prepended to the next chunk that is read. When the filter is invoked for the last time, signaled by the `$f->seen_eos` method, it unconditionally reverses and sends the data down the stream, which is then flushed down to the client.

15.7.2.2 Bucket Brigade-based Output Filters

The following filter implementation uses the bucket brigades API to accomplish exactly the same task as the first filter.

```
#file:MyApache2/FilterReverse2.pm
#------------------------------
package MyApache2::FilterReverse2;

use strict;
use warnings;

use base qw(Apache2::Filter);

use APR::Brigade ();
use APR::Bucket ();

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

sub handler : FilterRequestHandler {
    my ($f, $bb) = @_;

    my $bb_ctx = APR::Brigade->new($f->c->pool, $f->c->bucket_alloc);

    while (!$bb->is_empty) {
        my $b = $bb->first;

        $b->remove;

        if ($b->is_eos) {
            $bb_ctx->insert_tail($b);
            last;
        }

        if ($b->read(my $data)) {
            $data = join "",
                map {scalar(reverse $_), "\n"} split "\n", $data;
            $b = APR::Bucket->new($bb->bucket_alloc, $data);
        }

        $bb_ctx->insert_tail($b);
    }

    my $rv = $f->next->pass_brigade($bb_ctx);

    return $rv unless $rv == APR::Const::SUCCESS;

    return Apache2::Const::OK;
}
1;
```

Below is the corresponding configuration for *httpd.conf*:

```
PerlModule MyApache2::FilterReverse2
PerlModule MyApache2::SendAlphaNum
<Location /reverse2>
    SetHandler modperl
    PerlResponseHandler     MyApache2::SendAlphaNum
    PerlOutputFilterHandler MyApache2::FilterReverse2
</Location>
```

Now when a request to */reverse2* is made, the client gets:

```
0987654321
zyxwvutsrqponmlkjihgfedcba
```

as expected.

The bucket brigades output filter version is just a bit more complicated than the stream-oriented one. The handler receives the incoming bucket brigade $bb as its second argument. When the handler finishes it must pass a brigade to the next filter in the stack, so we create a new bucket brigade into which we put the modified buckets and which eventually we pass to the next filter.

The core of the handler removes buckets from the head of the bucket brigade $bb, while buckets are available, reads the data from the buckets, then reverses and puts the data into a newly created bucket. The new bucket is then inserted on the end of the new bucket brigade using the insert_tail() method. If we see a bucket which designates the end of stream, we insert that bucket on the tail of the new bucket brigade and break the loop. Finally we pass the created brigade with modified data to the next filter and return.

Similar to the original version of MyApache2::FilterReverse1::handler, this filter is not smart enough to handle incomplete lines. The filter could be made more foolproof by building a better matching rule, and using the $leftover buffer as demonstrated in the previous section. This left as an exercise for the reader.

15.8 Filter Applications

The following sections provide various filter applications and their implementation.

15.8.1 Handling Data Underruns

Sometimes filters need to read at least N bytes before they can apply their transformation. It's quite possible that reading one bucket brigade is not enough, but that two or more are needed. This situation is sometimes referred to as an *underrun*.

Let's take an input filter as an example. When the filter realizes that it doesn't have enough data in the current bucket brigade, it can store the read data in the filter context, and wait for the next invocation of itself, which may or may not satisfy its needs. While it is gathering the data from the bucket brigades, it must return an empty bucket brigade to the upstream input filter. However, this is not the most efficient technique to resolve underruns.

Instead of returning an empty bucket brigade, the input filter can request extra bucket brigades until the underrun condition gets resolved. Note that this solution is transparent to any filters before or after the current filter.

Consider this HTTP request:

```
% perl -MLWP::UserAgent -le ' \
  $r = LWP::UserAgent->new()->post("http://localhost:8011/", \
       [content => "x" x (40 * 1024 + 7)]); \
  print $r->is_success ? $r->content : "failed: " . $r->code'
read 40975 chars
```

This client POSTs just a little bit more than 40kb of data to the server. Normally Apache splits incoming POSTed data into 8kb chunks, putting each chunk into a separate bucket brigade. Therefore we expect to get 5 brigades of 8kb, and one brigade with just a few bytes (a total of 6 bucket brigades).

Now let's assume our example filter needs to have 1024*16 + 5 bytes to have a complete token before it can start its processing. The extra 5 bytes are just so we don't perfectly fit into 8kb bucket brigades, making the example closer to real situations. Having 40,975 bytes of input and a token size of 16,389 bytes, we will have 2 full tokens and a remainder of 8,197 bytes.

Before showing any code, let's look at the filter debug output to better explain what we expect to happen:

```
filter called
asking for a bb
asking for a bb
asking for a bb
storing the remainder: 7611 bytes

filter called
asking for a bb
asking for a bb
storing the remainder: 7222 bytes

filter called
asking for a bb
seen eos, flushing the remaining: 8197 bytes
```

We can see that the filter was invoked three times. The first time it has consumed three bucket brigades, collecting one full token of 16,389 bytes with a remainder of 7,611 bytes to be processed on the next invocation. The second time it needed only two more bucket brigades and this time, after completing the second token, 7,222 bytes remained. Finally on the third invocation it consumed the last bucket brigade for a total of six, just as we expected. However, it didn't have enough for the third token and since EOS has been seen (no more data expected), it has flushed the remaining 8,197 bytes as we calculated earlier.

It is clear from the debugging output that the filter was invoked only three times, instead of six times (there were six bucket brigades). Notice that the upstream input filter, if there is one, isn't aware that there were six bucket brigades, since it saw only three. Our example filter didn't do much with those tokens, so it has only repackaged data from 8kb per bucket brigade, to 16,389 bytes per bucket brigade. But of course in a real implementation some transformation would be applied on these tokens.

Now let's look at the implementation details. First let's look at the `response()` handler, which is the first part of the module:

```
#file:MyApache2/Underrun.pm
#------------------------
package MyApache2::Underrun;

use strict;
use warnings;

use constant IOBUFSIZE => 8192;

use Apache2::Const -compile => qw(MODE_READBYTES OK M_POST);
use APR::Const    -compile => qw(SUCCESS BLOCK_READ);

sub response {
    my $r = shift;

    $r->content_type('text/plain');

    if ($r->method_number == Apache2::Const::M_POST) {
        my $data = read_post($r);
        #warn "HANDLER READ: $data\n";
        my $length = length $data;
        $r->print("read $length chars");
    }

    return Apache2::Const::OK;
}

sub read_post {
    my $r = shift;

    my $bb = APR::Brigade->new($r->pool, $r->connection->bucket_alloc);

    my $data = '';
    my $seen_eos = 0;
    do {
        $r->input_filters->get_brigade($bb, Apache2::Const::MODE_READBYTES,
                                       APR::Const::BLOCK_READ, IOBUFSIZE);

        for (my $b = $bb->first; $b; $b = $bb->next($b)) {
            if ($b->is_eos) {

                $seen_eos++;
                last;
            }

            if ($b->read(my $buf)) {
                $data .= $buf;
            }

            $b->remove; # optimization to reuse memory
        }
    } while (!$seen_eos);
```

```
    $bb->destroy;

    return $data;
}
```

The `response()` handler is trivial -- it reads the POSTed data and prints how many bytes it has read. `read_post()` sucks in all POSTed data without parsing it.

Now comes the filter (which lives in the same package):

```
#file:MyApache2/Underrun.pm (continued)
#---------------------------------
use Apache2::Filter ();

use Apache2::Const -compile => qw(OK M_POST);

use constant TOKEN_SIZE => 1024*16 + 5; # ~16k

sub filter {
    my ($f, $bb, $mode, $block, $readbytes) = @_;
    my $ba = $f->r->connection->bucket_alloc;
    my $ctx = $f->ctx;
    my $buffer = defined $ctx ? $ctx : '';
    $ctx = '';  # reset
    my $seen_eos = 0;
    my $data;
    warn "\nfilter called\n";

    # fetch and consume bucket brigades until we have at least TOKEN_SIZE
    # bytes to work with
    do {
        my $tbb = APR::Brigade->new($f->r->pool, $ba);
        my $rv = $f->next->get_brigade($tbb, $mode, $block, $readbytes);
        warn "asking for a bb\n";
        ($data, $seen_eos) = flatten_bb($tbb);
        $tbb->destroy;
        $buffer .= $data;
    } while (!$seen_eos && length($buffer) < TOKEN_SIZE);

    # now create a bucket per chunk of TOKEN_SIZE size and put the remainder
    # in ctx
    for (split_buffer($buffer)) {
        if (length($_) == TOKEN_SIZE) {
            $bb->insert_tail(APR::Bucket->new($ba, $_));
        }
        else {
            $ctx .= $_;
        }
    }

    my $len = length($ctx);
    if ($seen_eos) {
        # flush the remainder
        $bb->insert_tail(APR::Bucket->new($ba, $ctx));
        $bb->insert_tail(APR::Bucket::eos_create($ba));
        warn "seen eos, flushing the remaining: $len bytes\n";
```

```
    }
    else {
        # will re-use the remainder on the next invocation
        $f->ctx($ctx);
        warn "storing the remainder: $len bytes\n";
    }

    return Apache2::Const::OK;
}

# split a string into tokens of TOKEN_SIZE bytes and a remainder
sub split_buffer {
    my $buffer = shift;
    if ($] < 5.007) {
        my @tokens = $buffer =~ /(.{@{[TOKEN_SIZE]}}|.+)/g;
        return @tokens;
    }
    else {
        # available only since 5.7.x+
        return unpack "(A" . TOKEN_SIZE . ")*", $buffer;
    }
}

sub flatten_bb {
    my ($bb) = shift;

    my $seen_eos = 0;

    my @data;
    for (my $b = $bb->first; $b; $b = $bb->next($b)) {
        $seen_eos++, last if $b->is_eos;
        $b->read(my $bdata);
        push @data, $bdata;
    }

    return (join('', @data), $seen_eos);
}

1;
```

The filter calls `get_brigade()` in a do-while loop until it reads enough data or sees EOS. Notice that it may get underruns several times, and then suddenly receive a lot of data at once, which will be enough for more than one minimal size token, so we have to take this into an account. Once the underrun condition is satisfied (we have at least one complete token) the tokens are put into a bucket brigade and returned to the upstream filter for processing, keeping any remainders in the filter context for the next invocations or flushing all the remaining data if EOS is seen.

Note that this example cannot be implemented with streaming filters because each invocation gives the filter exactly one bucket brigade to work with. The streaming interface does not currently provide a facility to fetch extra brigades.

Here is the Apache configuration for this example:

```
PerlModule MyApache2::Underrun
<Location />
  PerlInputFilterHandler MyApache2::Underrun::filter
  SetHandler modperl
  PerlResponseHandler MyApache2::Underrun::response
</Location>
```

15.8.2 Setting the Content-Length Header in Request Output Filters

Earlier we have stated that a filter that modifies the content's length must unset the Content-Length HTTP header. However sometimes it's desirable to have this header set, for example when dealing with proxies.

Since the headers are sent before the data, all the data must first be buffered and processed. You cannot accomplish this task with the streaming filter API since it passes FLUSH buckets through. As soon as the FLUSH bucket is received by the core filter that sends the headers, it generates the headers and sends those out. Therefore the bucket brigade API must be used here to have a complete control over what's going through. Here is a possible implementation:

```
#file:MyApache2/FilterChangeLength.pm
#----------------------------------
package MyApache2::FilterChangeLength;

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

use Apache2::RequestRec ();

use APR::Table ();
use APR::Bucket ();
use APR::Brigade ();

use base qw(Apache2::Filter);

use Apache2::Const -compile => qw(OK);
use APR::Const     -compile => ':common';

sub handler {
    my ($filter, $bb) = @_;

    my $ctx = $filter->ctx;

    # no need to unset the C-L header, since this filter makes sure to
    # correct it before any headers go out.
    #unless ($ctx) {
    #    $filter->r->headers_out->unset('Content-Length');
    #}

    my $data = exists $ctx->{data} ? $ctx->{data} : '';
    $ctx->{invoked}++;
    my ($bdata, $seen_eos) = flatten_bb($bb);
    $bdata =~ s/-//g;
    $data .= $bdata if $bdata;

    if ($seen_eos) {
```

```
        my $len = length $data;
        $filter->r->headers_out->set('Content-Length', $len);
        $filter->print($data) if $data;
    }

    else {
        # store context for all but the last invocation
        $ctx->{data} = $data;
        $filter->ctx($ctx);
    }

    return Apache2::Const::OK;
}

sub flatten_bb {
    my ($bb) = shift;

    my $seen_eos = 0;

    my @data;
    for (my $b = $bb->first; $b; $b = $bb->next($b)) {
        $seen_eos++, last if $b->is_eos;
        $b->read(my $bdata);
        push @data, $bdata;
    }
    return (join('', @data), $seen_eos);
}

1;
```

In this module we use flatten_bb() to read the data from the buckets and signal when the EOS is received. The filter simply collects the data, storing it in the filter context. When it receives EOS it sets the Content-Length header and sends the data out.

The configuration is straightforward:

```
PerlOutputFilterHandler MyApache2::FilterChangeLength
```

15.9 Filter Tips and Tricks

Various tips to use in filters.

15.9.1 Altering the Content-Type Response Header

Let's say that you want to modify the Content-Type header in the request output filter:

```
sub handler : FilterRequestHandler {
    my $f = shift;
    ...
    $f->r->content_type("text/html; charset=$charset");
    ...
```

Request filters have an access to the request object, so we simply modify it.

15.10 Writing Well-Behaving Filters

Filter writers must follow the following rules:

15.10.1 Connection Filters over KeepAlive Connections

Whenever a new HTTP request is processed, request filters get their context (`$f->ctx`) reset. This is also true for the connection filter context, as long as the connection is not a `keepalive`) connection. When the connection is kept alive, there could be many requests processed during a single connection and the same filter context will persist through all of them, until the maximum number of KeepAlive requests over the same connection is reached or until the client breaks the connection.

Sometimes it's desirable to reset the whole context or parts of it before a HTTP request is processed. For example `Apache2::Filter::HTTPHeadersFixup` needs to know when it should start and stop processing HTTP headers. It keeps the state in the filter's context. The problem is that whenever a new HTTP request is coming in, it needs to be able to reset the state machine. If it doesn't, it will process the HTTP headers of the first request and miss the rest of the requests.

So let's say we have a hypothetical module `MyApache2::Filter::StateMachine` which implements an input connection filter and it processes incoming data as long as the *state* flag is down. Once that flag goes up, the filter switches to the pass-through-unmodified mode. Here is a skeleton of the module:

```
#file:MyApache2/Filter/StateMachine.pm
#----------------------------------
package MyApache2::Filter::StateMachine;

use base qw(Apache2::Filter);
use Apache2::Connection ();

use Apache2::Const -compile => qw(OK DECLINED CONN_KEEPALIVE);

sub handler : FilterConnectionHandler {
    my ($f, $bb, $mode, $block, $readbytes) = @_;

    my $ctx = context($f);

    # pass through unmodified
    return Apache2::Const::DECLINED if $ctx->{state};

    # get data, do some processing, send it out
    process(); # your code comes here

    # change the state if some condition is reached
    $ctx->{state}++ if $done_condition;

    return Apache2::Const::OK;
}

sub context {
```

```
    my ($f) = shift;

    my $ctx = $f->ctx;
    unless ($ctx) {
        $ctx = {
            state => 0,
        };

        $f->ctx($ctx);
    }

    return $ctx;
}

1;
```

To make this module work properly over KeepAlive connections, we want to reset the *state* flag at the very beginning of the new request. To accomplish this, all we need to do is to change the `context` wrapper to be:

```
sub context {
    my ($f) = shift;

    my $ctx = $f->ctx;
    unless ($ctx) {
        $ctx = {
            state       => 0,
            keepalives  => $f->c->keepalives,
        };

        $f->ctx($ctx);
        return $ctx;
    }

    my $c = $f->c;
    if ($c->keepalive == Apache2::Const::CONN_KEEPALIVE &&
        $ctx->{state} && $c->keepalives > $ctx->{keepalives}) {

        $ctx->{state}      = 0;
        $ctx->{keepalives} = $c->keepalives;
    }

    return $ctx;
}
```

The only difference from the previous implementation is that we maintain one more state, which stores the number of requests served over the current connection. When Apache reports more served requests than we have in the context that means that we have a new request coming in. So we reset the *state* flag and store the new value of the served connections.

For a more complete real-world implementation, see: http://search.cpan.org/dist/Apache-Filter-HTTP-HeadersFixup/

15.10.2 Adjusting HTTP Headers

The following information is relevant for HTTP filters

- **Unsetting the Content-Length header**

  HTTP response filters modifying the length of the body they process must unset the `Content-Length` header. For example, a compression filter modifies the body length, whereas a lowercasing filter doesn't; therefore the former has to unset the header, and the latter doesn't have to.

  The header must be unset before any output is sent from the filter. If this rule is not followed, an HTTP response header with incorrect `Content-Length` value might be sent.

  Since you want to run this code once during the multiple filter invocations, use the `ctx()` method to set the flag:

  ```
  unless ($f->ctx) {
      $f->r->headers_out->unset('Content-Length');
      $f->ctx(1);
  }
  ```

- META: Same goes for last-modified/etags, which may need to be unset, "vary" might need to be added if you want caching to work properly (depending on what your filter does.

15.10.3 Other issues

META: to be written. Meanwhile collecting important inputs from various sources.

[

If a filter desires to store the incoming buckets for post processing. It must check whether the bucket type is transient. If it is -- the data must be copied away. If not -- the buckets may contain corrupted data when used later. The right thing is accomplished transparently by apr_bucket_setaside, for which we need to provide a perl glue.

]

[

This one will be expanded by Geoff at some point:

HTTP output filter developers are ought to handle conditional GETs properly... (mostly for the reason of efficiency?)

]

[

talk about issues like not losing meta-buckets. e.g. if the filter runs a switch statement and propagates buckets types that were known at the time of writing, it may drop buckets of new types which may be added later, so it's important to ensure that there is a default cause where the bucket is passed as is.

of course mention the fact where things like EOS buckets must be passed, or the whole chain will be broken. Or if some filter decides to inject an EOS bucket by itself, it should probably consume and destroy the rest of the incoming bb. need to check on this issue.

]

[

Need to document somewhere (concepts?) that the buckets should never be modified directly, because the filter can't know ho else could be referencing it at the same time. (shared mem/cache/memory mapped files are examples on where you don't want to modify the data). Instead the data should be moved into a new bucket.

Also it looks like we need to $b->destroy (need to add the API) in addition to $b->remove. Which can be done in one stroke using $b->delete (need to add the API).

]

[

Mention mod_bucketeer as filter debugging tool (in addition to FilterSnoop)

]

15.11 Writing Efficient Filters

As of this writing, the Apache network input filter reads in 8000B chunks (not 8192B) and makes each bucket 8000B in size. Based on this, the most efficient reading technique is:

```
use constant BUFF_LEN => 8000;
while ($f->read(my $buffer, BUFF_LEN)) {
    # manip $buffer
    $f->print($buffer);
}
```

however if there is some filter in between, it may change the size of the buckets. Also this number may change in the future.

Hmm, I've also seen it read in 7819 chunks. I suppose this is not very reliable. But it's probably a good idea to ask at least 8k, so if a bucket brigade has < 8k, nothing will need to be stored in the internal buffer. i.e. `read()` will return less than asked for.

]

[

Bucket Brigades are used to make the data flow between filters and handlers more efficient. e.g. a file handle can be put in a bucket and the read from the file can be postponed to the very moment when the data is sent to the client, thus saving a lot of memory and CPU cycles. though filters writers should be aware that if they call $b->read(), or any other operation that internally forces the bucket to read the information into the memory (like the length() op) and thus making the data handling inefficient. therefore a care should be taken so not to read the data in, unless it's really necessary.

]

15.12 CPAN Modules

Several modules are available on the CPAN that implement mod_perl 2.0 filters. As with all code on the CPAN, the source code is fully available, so you can download these modules and see exactly how they work.

- **`Apache::Clean` - Interface into HTML::Clean for mod_perl 2.0**

  http://search.cpan.org/dist/Apache-Clean/

- **`Apache::Filter::HTTPHeadersFixup` - Manipulate Apache 2 HTTP Headers**

  http://search.cpan.org/dist/Apache-Filter-HTTPHeadersFixup/

15.13 Maintainers

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

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

15.14 Authors

-

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

16   General Handlers Issues

16.1 Description

This chapter discusses issues relevant too any kind of handlers.

16.2 Handlers Communication

Apache handlers can communicate between themselves by writing and reading notes. It doesn't matter in what language the handlers were implemented as long as they can access the notes table.

For example inside a request handler we can say:

```
my $r = shift;
my $c = $r->connection;
$c->notes->set(mod_perl => 'rules');
```

and then later in a mod_perl filter handler this note can be retrieved with:

```
my $f = shift;
my $c = $f->c;
my $is = $c->notes->get("mod_perl");
$f->print("mod_perl $is");
```

16.3 Maintainers

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

• Stas Bekman [http://stason.org/]

16.4 Authors

•

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

17   Preventive Measures for Performance Enhancement

17.1 Description

This chapter explains what should or should not be done in order to keep the performance high

17.2 Memory Leakage

Memory leakage in 1.0 docs.

17.2.1 Proper Memory Pools Usage

Several mod_perl 2.0 APIs are using Apache memory pools for memory management. Mainly because the underlying C API requires that. So every time Apache needs to allocate memory it allocates it using the pool object that is passed as an argument. Apache doesn't frees allocated memory, this happens automatically when a pool ends its life.

Different pools have different life lengths. Request pools (`$r->pool`) are destroyed at the end of each request. Connection pools (`$c->pool`) are destroyed when the connection is closed. Server pools `$s->pool`) and the global pools (accessible in the server startup phases, like `PerlOpenLogsHandler` handlers) are destroyed only when the server exits.

Therefore always use the pool of the shortest possible life if you can. Never use server pools during request, when you can use a request pool. For example inside an HTTP handler, don't call:

```
my $dir = Apache2::ServerUtil::server_root_relative($s->process->pool, 'conf');
```

when you should call:

```
my $dir = Apache2::ServerUtil::server_root_relative($r->pool, 'conf');
```

Of course on special occasions, you may want to have something allocated off the server pool if you want the allocated memory to survive through several subsequent requests or connections. But this is normally doesn't apply to the core mod_perl 2.0, but rather for 3rd party extensions.

17.3 Maintainers

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

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

17.4 Authors

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

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

18   Performance Considerations Under Different MPMs

18.1 Description

This chapter discusses how to choose the right MPM to use (on platforms that have such a choice), and how to get the best performance out of it.

Certain kind of applications may show a better performance when running under one mpm, but not the other. Results also may vary from platform to platform.

CPAN module developers have to strive making their modules function correctly regardless the mpm they are being deployed under. However they may choose to indentify what MPM the code is running under and do better decisions better on this information, as long as it doesn't break the functionality for other platforms. For examples if a developer provides thread-unsafe code, the module will work correctly under the prefork mpm, but may malfunction under threaded mpms.

18.2 Memory Requirements

Since the very beginning mod_perl users have enjoyed the tremendous speed boost mod_perl was providing, but there is no free lunch -- mod_perl has quite big memory requirements, since it has to store the compiled code in the memory to avoid the code loading and recompilation overhead for each request.

18.2.1 Memory Requirements in Prefork MPM

For those familiar with mod_perl 1.0, mod_perl 2.0 has not much new to offer. We still rely on shared memory, try to preload as many things as possible at the server startup and limit the amount of used memory using specially designed for that purpose tools.

The new thing is that the core API has been spread across multiply modules, which can be loaded only when needed (this of course works only when mod_perl is built as DSO). This allows us to save some memory. However the savings are not big, since all these modules are writen in C, making them into the text segments of the memory, which is perfectly shared. The savings are more significant at the startup speed, since the startup time, when DSO modules are loaded, is growing almost quadratically as the number of loaded DSO modules grows (because of symbol relocations).

18.2.2 Memory Requirements in Threaded MPM

The threaded MPM is a totally new beast for mod_perl users. If you run several processes, the same memory sharing techniques apply, but usually you want to run as few processes as possible and to have as many threads as possible. Remember that mod_perl 2.0 allows you to have just a few Perl interpreters in the process which otherwise runs multiple threads. So using more threads doesn't mean using significantly more memory, if the maximum number of available Perl interpreters is limited.

Even though memory sharing is not applicable inside the same process, mod_perl gets a significant memory saving, because Perl interpreters have a shared opcode tree. Similar to the preforked model, all the code that was loaded at the server startup, before Perl interpreters are cloned, will be shared. But there is a significant difference between the two. In the prefork case, the normal memory sharing applies: if a single byte of the memory page gets unshared, the whole page is unshared, meaning that with time less

and less memory is shared. In the threaded mpm case, the opcode tree is shared and this doesn't change as the code runs.

Moreover, since Perl Interpreter pools are used, and the FIFO model is used, if the pool contains three Perl interpreters, but only one is used at any given time, only that interpreter will be ever used, making the other two interpreters consuming very little memory. So if with prefork MPM, you'd think twice before loading mod_perl if all you need is trans handler, with threaded mpm you can do that without paying the price of the significantly increased memory demands. You can have 256 light Apache threads serving static requests, and let's say three Perl interpreters running quick trans handlers, or even heavy but infrequest dynamic requests, when needed.

It's not clear yet, how one will be able to control the amount of running Perl interepreters, based on the memory consumption, because it's not possible to get the memory usage information per thread. However we are thinking about running a garbage collection thread which will cleanup Perl interpreters and occasionaly kill off the unused ones to free up used memory.

18.3 Work with DataBases

18.3.1 Work with DataBases under Prefork MPM

`Apache::DBI` works as with mod_perl 1.0, to share database connections.

18.3.2 Work with DataBases under Threaded MPM

The current `Apache::DBI` should be usable under threaded mpm, though it doesn't share connections across threads. Each Perl interpreter has its own cache, just like in the prefork mpm.

`DBI::Pool` is a work in progress, which should bring the sharing of database connections across threads of the same process. Watch the mod_perl and dbi-dev lists for updates on this work. Once `DBI::Pool` is completed it'll either replace `Apache::DBI` or will be used by it.

18.4 Maintainers

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

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

18.5 Authors

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

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

19   Troubleshooting mod_perl problems

19.1 Description

Frequently encountered problems (warnings and fatal errors) and their troubleshooting.

19.2 Building and Installation

19.2.1 Cannot find -lgdbm / libgdbm.so.3: open failed: No such file or directory

Please see: Missing or Misconfigured libgdbm.so.

Also it seems that on Solaris this exact issue doesn't show up at compile time, but at run time, so you may see the errors like:

```
.../mod_perl-1.99_17/blib/arch/auto/APR/APR.so' for module APR:
ld.so.1: /usr/local/ActivePerl-5.8/bin/perl: fatal:
libgdbm.so.3: open failed: No such file or directory at
...5.8.3/sun4-solaris-thread-multi/DynaLoader.pm line 229.
```

the solution is the same, make sure that you have the libgdbm shared library and it's properly symlinked.

19.3 Configuration and Startup

19.3.1 Can't locate TestFilter/in_str_consume.pm in @INC...

Sometimes you get a problem of perl not being able to locate a certain Perl module. This can happen in the mod_perl test suite or in the normal mod_perl setup. One of the possible reasons is a low limit on the number of files that can be opened by a single process. To check whether this is the problem run the process under strace(1) or an equivalent utility.

For example on OpenBSD 3.5 the default setting for a maximum number of files opened by a single process seems to be 64, so when you try to run the mod_perl test suite, which opens a few hundreds of files, you will have a problem. e.g. the test suite may fail as:

```
[Wed Aug 25 09:49:40 2004] [info] 26 Apache2:: modules loaded
[Wed Aug 25 09:49:40 2004] [info] 7 APR:: modules loaded
[Wed Aug 25 09:49:40 2004] [info] base server + 20 vhosts ready
to run tests
[Wed Aug 25 09:49:40 2004] [error] Can't locate
TestFilter/in_str_consume.pm in @INC (@INC contains: ...
```

Running the system calls tracing program (ktrace(1) on OpenBSD, strace(1) on Linux):

```
% sudo ktrace -d /usr/local/apache/bin/httpd -d /tmp/mod_perl-2.0/t \
  -f /tmp/mod_perl-2.0/t/conf/httpd.conf -DAPACHE2 -X
```

looking at the ktrace dump reveals:

```
16641 httpd    NAMI "/tmp/mod_perl-2.0/t/lib/TestFilter/in_str_consume.pmc"
16641 httpd    RET   stat -1 errno 2 No such file or directory
16641 httpd    CALL  open(0x3cdae100,0,0)
16641 httpd    RET   open -1 errno 24 Too many open files
```

It's clear that Perl can't load *TestFilter/in_str_consume.pm* because it can't open the file.

This problem can be resolved by increasing the open file limit to 128 (or higher):

```
$ ulimit -n 128
```

19.3.2 "mod_perl.c" is not compatible with this version of Apache (found 20020628, need 20020903)

That error message means that mod_perl was built against Apache released on or post-20020628, but you are trying to load it against one released on or post-20020903. You will see the same error message for any other Apache module -- this is an error coming from Apache, not mod_perl.

Apache bumps up a special magic number every time it does a binary incompatible change, and then it makes sure that all modules that it loads were compiled against the same compatibility generation (which may include only one or quite a few Apache releases).

You may encounter this situation when you upgrade to a newer Apache, without rebuilding mod_perl. Or when you have several versions of Apache installed on the same system. Or when you install prepackaged binary versions which aren't coming from the source and aren't made against the same Apache version.

The solution is to have mod_perl built against the same Apache installed on your system. So either build from source or contact your binary version supplier and get a proper package(s) from them.

19.3.3 Server Hanging at the Startup

First you need to figure out where it hangs. strace(1) or an equivalent utility can be used to discover which call the server hangs on. You need to start the process in the single server mode so you will have only one process to monitor.

For example if the server hangs during 'make test', you should run:

```
% cd modperl-2.0
% strace /path/to/httpd -d t -f t/conf/httpd.conf \
  -DAPACHE2 -DONE_PROCESS -DNO_DETATCH
```

(and may be -DPERL_USEITHREADS if it was in the original output of make test.)

If the trace ends with:

```
open("/dev/random", O_RDONLY)          = 3
read(3, <unfinished ...>
```

then you have a problem with your OS, as */dev/random* doesn't have enough entropy to give the required random data, and therefore it hangs. This may happen in `apr_uuid_get()` C call or Perl `APR::UUID->new`.

The solution in this case is either to fix the problem with your OS, so that

```
% perl -le 'open I, "/dev/random"; read I, $d, 10; print $d'
```

will print some random data and not block. Or you can use an even simpler test:

```
% cat /dev/random
```

which should print some random data and not block.

If you can't fix the OS problem, you can rebuild Apache 2.0 with `--with-devran-dom=/dev/urandom` - however, that is not secure for certain needs. Alternatively setup EGD and rebuild Apache 2.0 with `--with-egd`. Apache 2.1/apr-1.1 will have a self-contained PRNG generator built-in, which won't rely on */dev/random*.

19.3.4 (28)No space left on device

httpd-2.0 is not very helpful at telling which device has run out of precious space. Most of the time when you get an error like:

```
(28)No space left on device:
mod_rewrite: could not create rewrite_log_lock
```

it means that your system have run out of semaphore arrays. Sometimes it's full with legitimate semaphores at other times it's because some application has leaked semaphores and haven't cleaned them up during the shutdown (which is usually the case when an application segfaults).

Use the relevant application to list the ipc facilities usage. On most Unix platforms this is usually an `ipcs(1)` utility. For example linux to list the semaphore arrays you should execute:

```
% ipcs -s
------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x00000000 2686976    stas       600        1
0x00000000 2719745    stas       600        1
0x00000000 2752514    stas       600        1
```

Next you have to figure out what are the dead ones and remove them. For example to remove the semid 2719745 execute:

```
% ipcrm -s 2719745
```

Instead of manually removing each (and sometimes there can be many of them), and if you know that none of listed the semaphores is really used (all leaked), you can try to remove them all:

```
  % ipcs -s | perl -ane ''ipcrm -s $F[1]''
```

httpd-2.0 seems to use the key `0x00000000` for its semaphores on Linux, so to remove only those that match that key you can use:

```
  % ipcs -s | perl -ane '/^0x00000000/ && `ipcrm -s $F[1]`'
```

Notice that on other platforms the output of `ipcs -s` might be different, so you may need to apply a different Perl one-liner.

19.3.5 Segmentation Fault when Using DBI

Update DBI to at least version 1.31.

19.3.6 <Perl> directive missing closing '>'

See the Apache2::PerlSections manpage.

19.3.7 'Invalid per-unknown PerlOption: ParseHeaders' on HP-UX 11 for PA-RISC

When building mod_perl 2.0 on HP-UX 11 for PA-RISC architecture, using the HP ANSI C compiler, please make sure you have installed patches PHSS_29484 and PHSS_29485. Once installed the issue should go away.

19.4 Shutdown and Restart

Issues happening during server shutdown and restart, or during specific interpreter shutdown at runtime with threaded mpm.

19.4.1 Subroutines in <perl> sections under threaded mpm

If you have defined a subroutine inside a <perl> section, under threaded mpm (or under perl with enabled ithreads which spawn its own ithreads), like so:

```
  <Perl>
    sub foo {}
  </Perl>
```

At the server shutdown, or when any interpreter quits you will see the following error in the *error_log*:

```
Attempt to free temp prematurely: SV 0x91b8e74,
Perl interpreter: 0x8547698 during global destruction.
Scalars leaked: 1
```

This is a bug in Perl and as of Perl 5.8.4 it's not resolved. For more information see:

http://rt.perl.org:80/rt3/Ticket/Display.html?id=29018

19.4.2 Modules using `Scalar::Util::weaken` under threaded mpm

Modules using `Scalar::Util::weaken` under threaded mpm may get:

```
Attempt to free unreferenced scalar SV 0x8154f74.
```

when each interprter exits.

This is a bug in Perl and as of Perl 5.8.4 it's not resolved. For more information see:

http://rt.perl.org:80/rt3/Ticket/Display.html?id=24660

19.5 Code Parsing and Compilation

19.5.1 Segfault with __read_nocancel Backtrace

If your application segfaults and you get a similar to the following backtrace:

```
(gdb) bt
#0  0x4030d4d1 in __read_nocancel () from /lib/tls/libpthread.so.0
#1  0x00000000 in ?? ()
```

that usually means that you've build your non-mod_perl modules with ithreads enabled perl. Then you have built a new perl **without** ithreads. But you didn't nuke/rebuild the old non-mod_perl modules. Now when you try to run those, you get the above segfault. To solve the problem recompile all the modules. The easiest way to accomplish that is to either remove all the modules completely, build the new perl and then install the new modules. You could also try to create a bundle of the existing modules using `CPAN.pm` prior to deleting the old modules, so you can easily reinstall all the modules you previously had.

19.5.2 Registry scripts fail to load with: Unrecognized character \xEF at ...

Certain editors (in particular on win32) may add a UTF-8 Byte Order Marker (BOM: http://www.unicode.org/faq/utf_bom.html#BOM) at the beginning of the file. Since `ModPerl::RegistryCooker` adds extra code in front of the original script, before compiling it, it creates a situation where BOM appears past the beginning of the file, which is why the error:

```
Unrecognized character \xEF at ...
```

is thrown by Perl.

The simplest solution is to configure your editor to not add BOMs (or switch to another editor which allows you to do that).

You could also subclass `ModPerl::RegistryCooker` or its existing subclasses to try to remove BOM in ModPerl::RegistryCooker::read_script():

```
# remove BOM
${$self->{CODE}} =~ s/^(?:
    \xef\xbb\xbf      |
    \xfe\xff          |
    \xff\xfe          |
    \x00\x00\xfe\xff  |
    \xff\xfe\x00\x00
)//x;
```

but do you really want to add an overhead of this operation multiple times, when you could just change the source file once? Probably not. It was also reported that on win32 the above s/// doesn't work.

19.6 Runtime

19.6.1 error_log is Full of Escaped \n, \t, etc.

It's an Apache "feature", see -DAP_UNSAFE_ERROR_LOG_UNESCAPED.

19.6.2 Problems with Catching Signals

See Using Signal Handlers.

19.6.3 APR::Socket::recv: (11) Resource temporarily unavailable at ...

You need to make sure that the socket is set to blocking IO mode before using it.

19.6.4 `APR::UUID->new` Hanging

See Server Hanging at the Startup.

19.6.5 Memory Leaks

- **s/// in perls 5.8.1 and 5.8.2**

  p5-porters report: http://www.xray.mpe.mpg.de/mailing-lists/perl5-porters/2003-12/msg00634.html

Fixed in 5.8.3. There is no workaround but to upgrade to 5.8.3 or higher.

19.6.6 C Libraries Don't See `%ENV` Entries Set by Perl Code

For example some people have reported problems with `DBD::Oracle` (whose guts are implemented in C), which doesn't see environment variables (like `ORACLE_HOME`, `ORACLE_SID`, etc.) set in the perl script and therefore fails to connect.

The issue is that the C array `environ[]` is not thread-safe. Therefore mod_perl 2.0 unties `%ENV` from the underlying `environ[]` array under the *perl-script* handler.

The `DBD::Oracle` driver or client library uses `getenv()` (which fetches from the `environ[]` array). When `%ENV` is untied from `environ[]`, Perl code will see `%ENV` changes, but C code will not.

The *modperl* handler does not untie `%ENV` from `environ[]`. Still one should avoid setting `%ENV` values whenever possible. And if it is required, should be done at startup time.

In the particular case of the `DBD::` drivers, you can set the variables that don't change (`$ENV{ORACLE_HOME}` and `$ENV{NLS_LANG}` in the startup file, and those that change pass via the `connect()` method, e.g.:

```
my $sid      = 'ynt0';
my $dsn      = 'dbi:Oracle:';
my $user     = 'username/password';
my $password = '';
$dbh = DBI->connect("$dsn$sid", $user, $password)
    or die "Cannot connect: " . $DBI::errstr;
```

Also remember that `DBD::Oracle` requires that *ORACLE_HOME* (and any other stuff like *NSL_LANG* stuff) be in `%ENV` when `DBD::Oracle` is loaded (which might happen indirectly via the `DBI` module. Therefore you need to make sure that wherever that load happens `%ENV` is properly set by that time.

Another solution that works **only with prefork mpm**, is to use `Env::C` (http://search.cpan.org/dist/Env-C/). This module sets the process level environ, bypassing Perl's `%ENV`. This module is not thread-safe, due to the nature of environ process struct, so don't even try using it in a threaded environment.

19.6.7 Error about not finding Apache.pm with CGI.pm

You need to install at least version 3.11 of CGI.pm to work under mod_perl 2.0, as earlier CGI.pm versions aren't mod_perl 2.0 aware.

19.6.8 20014:Error string not specified yet

This error is reported when some undefined Apache error happens. The known cases are:

- **when using mod_deflate**

    A bug in mod_deflate was triggering this error, when a response handler would flush the data that was flushed earlier: http://nagoya.apache.org/bugzilla/show_bug.cgi?id=22259 It has been fixed in httpd-2.0.48.

19.6.9 (22)Invalid argument: core_output_filter: writing data to the network

Apache uses the sendfile syscall on platforms where it is available in order to speed sending of responses. Unfortunately, on some systems, Apache will detect the presence of sendfile at compile-time, even when it does not work properly. This happens most frequently when using network or other non-standard file-system.

The whole story and the solutions are documented at:
http://httpd.apache.org/docs-2.0/faq/error.html#error.sendfile

19.6.10 undefined symbol: apr_table_compress

After a successful mod_perl build, sometimes during the startup or a runtime you'd get an "undefined symbol: foo" error. The following is one possible scenario to encounter this problem and possible ways to resolve it.

Let's say you ran mod_perl's test suite:

```
% make test
```

and got errors, and you looked in the *error_log* file (*t/logs/error_log*) and saw one or more "undefined symbol" errors, e.g.

```
% undefined symbol: apr_table_compress
```

- **Step 1**

    From the source directory (same place you ran "make test") run:

    ```
    % ldd blib/arch/auto/APR/APR.so | grep apr-
    ```

    ldd is not available on all platforms, e.g. not on Darwin/OS X. Instead on Darwin/OS X, you can use their otool.

    You you should get a full path, for example:

    ```
    libapr-0.so.0 => /usr/local/apache2/lib/libapr-0.so.0 (0x40003000)
    ```

    or

```
libapr-0.so.0 => /some/path/to/apache/lib/libapr-0.so.0 (0x40003000)
```

or something like that. It's that full path to libapr-0.so.0 that you want.

- **Step 2**

  Do:

  ```
  % nm /path/to/your/libapr-0.so.0 | grep table_compress
  ```

  for example:

  ```
  % nm /usr/local/apache2/lib/libapr-0.so.0 | grep table_compress
  ```

  You should get something like this:

  ```
  0000d010 T apr_table_compress
  ```

  If you get the message:

  ```
   nm: /usr/local/apache2/lib/libapr-0.so.0: no symbols
  ```

  that means that the library was stripped. You probably want to obtain Apache 2.x or libapr source, matching your binary and check it instead. Or rebuild it with debugging enabled, which will not strip the symbols.

  Note that the "grep table_compress" is only an example, the exact string you are looking for is the name of the "undefined symbol" from the *error_log* file. So, if you get:

  ```
  undefined symbol apr_holy_grail
  ```

  then you would do:

  ```
  % nm /usr/local/apache2/lib/libapr-0.so.0 | grep holy_grail
  ```

- **Step 3**

  Now, let's see what shared libraries your apache binary has. So, if in step 1 you got */usr/local/apache2/lib/libapr-0.so.0* then you will do:

  ```
  % ldd /usr/local/apache2/bin/httpd
  ```

  if in step 1 you got */foo/bar/apache/lib/libapr-0.so.0* then you do:

  ```
  % ldd /foo/bar/apache/bin/httpd
  ```

  The output should look something like this:

  ```
  libssl.so.2 => /lib/libssl.so.2 (0x40023000)
  libcrypto.so.2 => /lib/libcrypto.so.2 (0x40054000)
  libaprutil-0.so.0 => /usr/local/apache2/lib/libaprutil-0.so.0 (0x40128000)
  libgdbm.so.2 => /usr/lib/libgdbm.so.2 (0x4013c000)
  libdb-4.0.so => /lib/libdb-4.0.so (0x40143000)
  ```

```
libexpat.so.0 => /usr/lib/libexpat.so.0 (0x401eb000)
libapr-0.so.0 => /usr/local/apache2/lib/libapr-0.so.0 (0x4020b000)
librt.so.1 => /lib/librt.so.1 (0x40228000)
libm.so.6 => /lib/i686/libm.so.6 (0x4023a000)
libcrypt.so.1 => /lib/libcrypt.so.1 (0x4025c000)
libnsl.so.1 => /lib/libnsl.so.1 (0x40289000)
libdl.so.2 => /lib/libdl.so.2 (0x4029f000)
libpthread.so.0 => /lib/i686/libpthread.so.0 (0x402a2000)
libc.so.6 => /lib/i686/libc.so.6 (0x42000000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

Those are name => value pairs showing the shared libraries used by the `httpd` binary.

Take note of the value for *libapr-0.so.0* and compare it to what you got in step 1. They should be the same, if not, then mod_perl was compiled pointing to the wrong Apache installation. You should run "make clean" and then

```
% perl Makefile.pl MP_APACHE_CONFIG=/path/to/apache/bin/apr-config
```

using the correct path for the Apache installation.

● **Step 4**

You should also search for extra copies of *libapr-0.so.0*. If you find one in */usr/lib* or */usr/local/lib* that will explain the problem. Most likely you have an old pre-installed apr package which gets loaded before the copy you found in step 1.

On most Linux (and Mac OS X) machines you can do a fast search with:

```
% locate libapr-0.so.0
```

which searches a database of files on your machine. The "locate" database isn't always up-to-date so a slower, more comprehensive search can be run (as root if possible):

```
% find / -name "libapr-0.so.0*"
```

or

```
% find /usr/local -name "libapr-0.so.0*"
```

You might get output like this:

```
/usr/local/apache2.0.47/lib/libapr-0.so.0.9.4
/usr/local/apache2.0.47/lib/libapr-0.so.0
/usr/local/apache2.0.45/lib/libapr-0.so.0.9.3
/usr/local/apache2.0.45/lib/libapr-0.so.0
```

in which case you would want to make sure that you are configuring and compiling mod_perl with the latest version of apache, for example using the above output, you would do:

```
% perl Makefile.PL MP_AP_CONFIG=/usr/local/apache2.0.47
% make
% make test
```

There could be other causes, but this example shows you how to act when you encounter this problem.

19.6.11 Variable $x will not stay shared at

This warning is normally as a result of variables that your script is sharing with subroutines globally, rather than passing by value or reference. As the cause and solution of this is virtually identical to another commonly encountered problem (Sometimes it works, sometimes it doesn't), the text is not repeated here but is instead included in that section which follows this one.

You may have read somewhere *out there* that this warning can be ignored, but if you read on you will see that you should *never* ignore the warning. The other thing that might confuse you is that this warning is normally encountered when defining subroutines within subroutines. So why would you experience it in your script where that is not the case? The reason is because mod_perl wraps your script in its own subroutine (see the Perl Reference documentation for more details).

19.6.12 Sometimes it Works, Sometimes it Doesn't

When you start running your scripts under mod_perl, you might find yourself in a situation where a script seems to work, but sometimes it screws up. And the more it runs without a restart, the more it screws up. Often the problem is easily detectable and solvable. You have to test your script under a server running in single process mode (`httpd -X`).

Generally the problem is the result of using global variables (normally accompanied by a Variable $x will not stay shared at warning). Because global variables don't change from one script invocation to another unless you change them, you can find your scripts do strange things.

Let's look at three real world examples:

19.6.12.1 An Easy Break-in

The first example is amazing: Web Services. Imagine that you enter some site where you have an account, perhaps a free email account. Having read your own mail you decide to take a look at someone else's.

You type in the username you want to peek at and a dummy password and try to enter the account. On some services this will work!!!

You say, why in the world does this happen? The answer is simple: **Global Variables**. You have entered the account of someone who happened to be served by the same server child as you. Because of sloppy programming, a global variable was not reset at the beginning of the program and voila, you can easily peek into someone else's email! Here is an example of sloppy code:

```
use vars ($authenticated);
my $q = new CGI;
my $username = $q->param('username');
my $passwd   = $q->param('passwd');
```

```
authenticate($username,$passwd);
  # failed, break out
unless ($authenticated){
  print "Wrong passwd";
  exit;
}
  # user is OK, fetch user's data
show_user($username);

sub authenticate{
  my ($username,$passwd) = @_;
  # some checking
  $authenticated = 1 if SOME_USER_PASSWD_CHECK_IS_OK;
}
```

Do you see the catch? With the code above, I can type in any valid username and any dummy password and enter that user's account, provided she has successfully entered her account before me using the same child process! Since `$authenticated` is global--if it becomes 1 once, it'll stay 1 for the remainder of the child's life!!! The solution is trivial--reset `$authenticated` to 0 at the beginning of the program.

A cleaner solution of course is not to rely on global variables, but rely on the return value from the function.

```
my $q = CGI->new;
my $username = $q->param('username');
my $passwd   = $q->param('passwd');
my $authenticated = authenticate($username,$passwd);
  # failed, break out
unless ($authenticated){
  print "Wrong passwd";
  exit;
}
  # user is OK, fetch user's data
show_user($username);

sub authenticate{
  my ($username,$passwd) = @_;
  # some checking
  return (SOME_USER_PASSWD_CHECK_IS_OK) ? 1 : 0;
}
```

Of course this example is trivial--but believe me it happens!

19.6.12.2 Thinking mod_cgi

Just another little one liner that can spoil your day, assuming you forgot to reset the `$allowed` variable. It works perfectly OK in plain mod_cgi:

```
$allowed = 1 if $username eq 'admin';
```

But using mod_perl, and if your system administrator with superuser access rights has previously used the system, anybody who is lucky enough to be served later by the same child which served your administrator will happen to gain the same rights.

The obvious fix is:

```
$allowed = $username eq 'admin' ? 1 : 0;
```

19.6.12.3 Regular Expression Memory

Another good example is usage of the `/o` regular expression modifier, which compiles a regular expression once, on its first execution, and never compiles it again. This problem can be difficult to detect, as after restarting the server each request you make will be served by a different child process, and thus the regex pattern for that child will be compiled afresh. Only when you make a request that happens to be served by a child which has already cached the regex will you see the problem. Generally you miss that. When you press reload, you see that it works (with a new, fresh child). Eventually it doesn't, because you get a child that has already cached the regex and won't recompile because of the `/o` modifier.

An example of such a case would be:

```
my $pat = $q->param("keyword");
foreach( @list ) {
  print if /$pat/o;
}
```

To make sure you don't miss these bugs always test your CGI in single process mode.

To solve this particular `/o` modifier problem refer to Compiled Regular Expressions.

For more details and further examples please see the Perl Reference documentation.

19.7 Issues with APR Used Outside of mod_perl

It doesn't strictly belong to this document, since it's talking about APR usages outside of mod_perl, so this may move to its own dedicated page, some time later.

Whenever using an `APR::` package outside of mod_perl, you need to:

```
use APR;
```

in order to load the XS subroutines. For example:

```
% perl -MAPR -MAPR::UUID -le 'print APR::UUID->new->format'
```

19.8 Maintainers

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

- **Stas Bekman**

19.9 Authors

- **Stas Bekman**

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

20   User Help

20.1 Description

This chapter is for those needing help using mod_perl and related software.

There is a parallel Getting Help document written mainly for mod_perl core developers, but may be found useful to non-core problems as well.

20.2 Reporting Problems

Whenever you want to report a bug or a problem remember that in order to help you, you need to provide us the information about the software that you are using and other relevant details. Please follow the instructions in the following sections when reporting problems.

The most important thing to understand is that you should try hard to provide **all** the information that may assist to understand and reproduce the problem. When you prepare a bug report, put yourself in the position of a person who is going to try to help you, realizing that a guess-work on behalf of that helpful person, more often doesn't work than it does. Unfortunately most people don't realize that, and it takes several emails to squeeze the needed details from the person reporting the bug, a process which may drag for days.

20.2.1 Wrong Apache/mod_perl combination

First of all:

```
Apache 2.0 doesn't work with mod_perl 1.0.
Apache 1.0 doesn't work with mod_perl 2.0.
```

So if you aren't using Apache 2.x with mod_perl 2.0 please do not send any bug reports.

META: mod_perl-1.99_xx is mod_perl 2.0 to-be.

20.2.2 Using the Wrong Compiler

To build mod_perl, you **must** also use the same compiler that Perl was built with. You can find that out by running `perl -V` and looking at the `Compiler:` section. If you have used a different compiler and have encountered problems (which most likely will be the case) recompile Perl with the same compiler and then recompile mod_perl again.

20.2.3 Before Posting a Report

Before you post the report, make sure that you've checked the *error_log* file (t/logs/error_log in case of the failing test suite). Usually the errors are self-descriptive and if you remember to always check this file whenever you have a problem, chances are that you won't need to ask for help.

20.2.4 Test with the Latest mod_perl 2.0 Version

If you are using an older version than the most recently released one, chances are that a bug that you are about to report has already been fixed. If possible, save us and yourself time and try first to upgrade to the latest version, and only if the bug persists report it.

Reviewing the Changes file may help as well. Here is the Changes file of the most recenly released version: http://apache.org/dist/perl/mod_perl-2.0-current/Changes .

If the problem persists with the latest version, you may also want to try to reproduce the problem with the latest development version. It's possible that the problem was resolved since the last release has been made. Of course if this version solves the problem, don't rush to put it in production unless you know what you are doing. Instead ask the developers when the new version will be released.

20.2.5 Use a Proper Subject

Make sure to include a good subject like explaining the problem in a few words. Also please mention that this a problem with mod_perl 2.0 and not mod_perl 1.0. Here is an example of a good subject:

```
Subject: [mp2] protocol module doesn't work with filters
```

This is especially important now that we support mod_perl versions 1.0 and 2.0 on the same list.

20.2.6 Send the Report Inlined

When sending the bug report, please inline it and don't attach it to the email. It's hard following up on the attachments.

20.2.7 Important Information

Whenever you send a bug report make sure to include the information about your system.

- If you haven't yet installed mod_perl and/or you are having problems with the test suite -- you should do:

  ```
  % cd modperl-2.0
  % t/REPORT > mybugreport
  ```

  where `modperl-2.0` is the source directory where mod_perl was built. The `t/REPORT` utility is autogenerated when `perl Makefile.PL` is run, so you should have it already after building mod_perl.

- If you have already installed mod_perl and are having problems with things unrelated to the the test suite -- you should do:

```
% mp2bug > mybugreport
```

`mp2bug` should have been installed at the same time mod_perl 2.0 was installed. If for some reason you can't find it, you can alternatively run the following command, which does the same:

```
% perl -MApache2 -MApache::TestReportPerl \
-le 'Apache::TestReportPerl->new->run' > mybugreport
```

Please post the report (*mybugreport*) **inlined** in the text of your message, and not as an attachment!

Now add the problem description to the report and send it to the mod_perl users mailing list.

20.2.8 *Problem Description*

If the problem is with the mod_perl distribution test suite, refer to the 'make test' Failures section.

If the problem incurs with your own code, please try to reduce the code to the very minimum and include it in the bug report. Remember that if you include a long code, chances that somebody will look at it are low. If the problem is with some CPAN module, just provide its name.

Also remember to include the relevant part of *httpd.conf* and of *startup.pl* if applicable. Don't include whole files, only the parts that should aid to understand and reproduce the problem.

Finally don't forget to copy-n-paste (not type!) the **relevant** part of the *error_log* file (not the whole file!).

To further increase the chances that bugs your code exposes will be investigated, try using `Apache-Test` to create a self-contained test that core developers can easily run. To get you started, an `Apache-Test` bug skeleton has been created:

http://perl.apache.org/~geoff/bug-reporting-skeleton-mp2.tar.gz

Detailed instructions are contained within the `README` file in that distribution.

Finally, if you get a segfault with or without a core dump, refer to the Resolving Segmentation Faults section.

20.2.9 '`make test`' *Failures*

If when running `make test` some of the tests fail, please re-run them in the verbose mode and post the output of that run and the contents of the *t/logs/error_log* file to the list. Please **do not post** the *t/logs/error_log* file from `make test` that runs a complete test suite, as it contains a lot of irrelevant information.

For example if '`make test`' reports:

```
Failed Test           Stat Wstat Total Fail  Failed  List of Failed
-------------------------------------------------------------
compat/apache_util.t            15    1   6.67%  13
modperl/pnotes.t                 5    1    20%   2
```

Do the following:

```
% cd modperl-2.0.xx
% make test TEST_VERBOSE=1 \
  TEST_FILES="compat/apache_util.t modperl/pnotes.t"
```

or use an alternative way:

```
% cd modperl-2.0.xx
% t/TEST -clean
% t/TEST -verbose compat/apache_util.t modperl/pnotes.t
```

In the latter approach, `t/TEST -clean` cleans things up before starting a new test. Make sure that you don't forget to run it, before running the individual tests.

Now post to the mailing list the output of the individual tests running and the contents of *t/logs/error_log*.

Also please notice that there is more than one `make test` being run. The first one is running at the top directory, the second in a sub-directory *ModPerl-Registry/*. The first logs errors to *t/logs/error_log*, the second too, but relative to *ModPerl-Registry/*. Therefore if you get failures in the second run, make sure to `chdir()` to that directory before you look at the *t/logs/error_log* file and re-run tests in the verbose mode. For example:

```
% cd modperl-2.0.xx/ModPerl-Registry
% t/TEST -clean
% t/TEST -verbose closure.t
```

At the moment the second test suite is not run if the first one fails.

20.2.10 Resolving Segmentation Faults

If during `make test` or the use of mod_perl you get a segmentation fault you should send to the list a stack backtrace. This section explains how to get the core file and extract this backtrace. Once a proper stack backtrace is obtained append it to the bug report as explained in the previous section.

20.2.11 Please Ask Only Questions Related to mod_perl

If you have general Apache questions, please refer to: http://httpd.apache.org/lists.html.

If you have general Perl questions, please refer to: http://lists.perl.org/.

For other remotely related to mod_perl questions see the references to other documentation.

Finally, if you are not familiar with the modperl list etiquette, please refer to the mod_perl mailing lists' Guidelines before posting.

20.3 Help on Related Topics

When developing with mod_perl, you often find yourself having questions regarding other projects and topics like Apache, Perl, SQL, etc. This document will help you find the right resource where you can find the answers to your questions.

20.4 Maintainers

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

- **Stas Bekman**

20.5 Authors

- **Stas Bekman**

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

Table of Contents:

Table of Contents:

Table of Contents: