

User's guide

All you need to know about using mod_perl 2.0

Last modified Sat Dec 6 12:10:17 2003 GMT

Part I: Introduction

- ▶ 1. Getting Your Feet Wet with mod_perl
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.
- ▶ 2. Overview of mod_perl 2.0
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 discussed.
- ▶ 3. Notes on the design and goals of mod_perl-2.0
Notes on the design and goals of mod_perl-2.0.

Part II: Installation

- ▶ 4. Installing mod_perl 2.0
This chapter provides an in-depth mod_perl 2.0 installation coverage.
- ▶ 5. mod_perl 2.0 Server Configuration
This chapter provides an in-depth mod_perl 2.0 configuration details.
- ▶ 6. Apache Server Configuration Customization in Perl
This chapter explains how to create custom Apache configuration directives in Perl.

Part III: Coding

- ▶ 7. Writing mod_perl Handlers and Scripts
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.
- ▶ 8. Cooking Recipes
As the chapter's title implies, here you will find ready-to-go mod_perl 2.0 recipes.

Part IV: Porting

- ▶ 9. Porting Apache:: Perl Modules from mod_perl 1.0 to 2.0
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.
- ▶ 10. A Reference to mod_perl 1.0 to mod_perl 2.0 Migration.
This chapter is a reference for porting code and configuration files from mod_perl 1.0 to mod_perl 2.0.

Part V: mod_perl Handlers

- ▶ 11. Introducing mod_perl Handlers
This chapter provides an introduction into mod_perl handlers.
- ▶ 12. Server Life Cycle Handlers
This chapter discusses server life cycle and the mod_perl handlers participating in it.
- ▶ 13. Protocol Handlers
This chapter explains how to implement Protocol (Connection) Handlers in mod_perl.
- ▶ 14. HTTP Handlers
This chapter explains how to implement the HTTP protocol handlers in mod_perl.
- ▶ 15. Input and Output Filters
This chapter discusses mod_perl's input and output filter handlers.
- ▶ 16. General Handlers Issues
This chapter discusses issues relevant too any kind of handlers.

Part VI: Performance

- ▶ 17. Preventive Measures for Performance Enhancement
This chapter explains what should or should not be done in order to keep the performance high
- ▶ 18. Performance Considerations Under Different MPMs
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.

Part VII: Troubleshooting

- ▶ 19. Troubleshooting mod_perl problems
Frequently encountered problems (warnings and fatal errors) and their troubleshooting.
- ▶ 20. User Help
This chapter is for those needing help using mod_perl and related software.

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 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 MP_INST_APACHE2=1
```

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
```

Next, tell Perl where to find mod_perl2 libraries:

```
PerlModule Apache2
```

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 Apache::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:

```
[Thu May 29 12:22:12 2003] [notice] Apache/2.0.46-dev (Unix)
mod_perl/1.99_10-dev Perl/v5.9.0 mod_ssl/2.0.46-dev OpenSSL/0.9.7
DAV/2 configured -- resuming normal operations
```

1.5 Registry Scripts

To enable registry scripts add to `httpd.conf`:

```
Alias /perl/ /home/httpd/httpd-2.0/perl/
<Location /perl/>
    SetHandler perl-script
    PerlResponseHandler ModPerl::Registry
    PerlOptions +ParseHeaders
    Options +ExecCGI
</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: on 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:MyApache/Rocks.pm
#-----
package MyApache::Rocks;

use strict;
use warnings;

use Apache::RequestRec ();
use Apache::RequestIO ();

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

sub handler {
    my $r = shift;

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

    return Apache::OK;
}
1;
```

Save the code in the file *MyApache/Rocks.pm*, somewhere where mod_perl can find it. For example let's put it under */home/httpd/httpd-2.0/perl/MyApache/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 MyApache::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 persists, 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 <stas (at) stason.org>

1.9 Authors

- Stas Bekman <stas (at) 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 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 an automatic building of the Perl interface to Apache API, which allowed us to easily adjust mod_perl 2.0 to 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 portability 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 `Apache::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 mechanism, 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 *standalone_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.

- **New Hook Scheme**

In Apache 1.3, modules were registered using the *module* structure, normally static to *mod_foo.c*. This structure contains pointers to the command table, configuration creation and merging functions, response handler table and function pointers for all of the other hooks, such as *child_init* and *check_user_id*. In Apache 2.0, this structure has been pruned down to the first three items mentioned and a new function pointer added called *register_hooks*. It is the job of *register_hooks* to register

functions for all other hooks (such as *child_init* and *check_user_id*). Not only is hook registration now dynamic, it is also possible for modules to register more than one function per hook, unlike 1.3. The new hook mechanism also makes it possible to sort registered functions, unlike 1.3 with function pointers hardwired into the module structure, and each module structure into a linked list. Order in 1.3 depended on this list, which was possible to order using compile-time and startup-time configuration, but that was left to the user. Whereas in 2.0, the *add_hook* functions accept an order preference parameter, those commonly used are:

- **FIRST**
- **MIDDLE**
- **LAST**

For mod_perl, dynamic registration provides a cleaner way to bypass the `Perl*Handler` configuration directives. By simply adding this configuration:

```
PerlModule Apache::Foo
```

`Apache::Foo` can register hooks itself at server startup:

```
Apache::Hook->add(PerlAuthenHandler => \&authenticate,
                  Apache::Hook::MIDDLE);
Apache::Hook->add(PerlLogHandler      => \&logger,
                  Apache::Hook::LAST);
```

META: Not implemented yet (API will change?)

However, this means that Perl subroutines registered via this mechanism will be called for **every** request. It will be left to that subroutine to decide if it was to handle or decline the given phase. As there is overhead in entering the Perl runtime, it will most likely be to your advantage to continue using `Perl*Handler` configuration directives to reduce this overhead. If it is the case that your `Perl*Handler` should be invoked for every request, the hook registration mechanism will save some configuration keystrokes.

● 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 `$Apache::Server::SaveConfig` variable can be set to save <Perl> configuration in the `%Apache::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 *pnodes* 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.com/perl5.8.0/pod/perl56delta.html>, <http://perldoc.com/perl5.8.0/pod/perl561delta.html> and <http://perldoc.com/perl5.8.0/pod/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 *perlembded* (<http://perldoc.com/perl5.6.1/pod/perlembded.html>) and *perl561delta* (<http://perldoc.com/perl5.6.1/pod/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.com/perl5.6.1/pod/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.com/perl5.6.1/pod/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.com/perl5.6.1/pod/perl561delta.html>) manpage.
- Multiple performance enhancements were made. See the *perl561delta* (<http://perldoc.com/perl5.6.1/pod/perl561delta.html>) manpage.
- Numerous memory leaks were fixed. See the *perl561delta* (<http://perldoc.com/perl5.6.1/pod/perl561delta.html>) manpage.
- Improved security features: more potentially unsafe operations taint their results for improved security. See the *perlsec* (<http://perldoc.com/perl5.6.1/pod/perlsec.html>) and *perl561delta* (<http://perldoc.com/perl5.6.1/pod/perl561delta.html>) manpages.
- Available on new platforms: GNU/Hurd, Rhapsody/Darwin, EPOC.

Overall multiple bugs and problems were 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 *perlapi* (<http://perldoc.com/perl5.8.0/pod/perlapi.html>) and *perliol* (<http://perldoc.com/perl5.8.0/pod/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.com/perl5.8.0/pod/func/open.html>).

- More overridable via `CORE::GLOBAL::` keywords. See the *perlsub* (<http://perldoc.com/perl5.8.0/pod/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.com/perl5.8.0/pod/perlrun.html>) manpage.

A new special variable `${^TAINT}` was introduced. It indicates whether taint mode is enabled. See the *perlvar* (<http://perldoc.com/perl5.8.0/pod/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://www.perldoc.com/perl5.8.0/pod/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 were needed to make the API more Perl-ish. 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 `Apache::` 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 -MApache2 -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 `Apache::*.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 <stas (at) stason.org>

2.9 Authors

- Doug MacEachern <doug (at) covalent.net>
- Stas Bekman <stas (at) 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 interpreters. 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 interpreters 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**

3.3.1 TIPool

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 subrequests by default, however, it is possible to configure the interpreter 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 interpreters 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 traditional Perl*Handler configuration fashion and via dynamic registration methods (the `ap_hook_*` functions).

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*Handlers 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 Perl-ish. 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 `Apache::FunctionTable` and `Apache::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 Perlsh 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 Perlsh 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 Apache::Server 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 Apache::Build module. The *Makefile.PL* will construct an Apache::Build object which will have all the info it needs to generate scripts and *Makefiles* 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 *Makefiles* 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 switchability 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 *Makefiles* 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 Apache::* 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 Apache::* Library

The majority of the standard `Apache::*` 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 Apache::Request $r = shift;
```

Tells the Perl compiler to expect an object in the `Apache::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*d. 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 pluggable, 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.

- **Perl**

- **prefork MPM**

Requires at least Perl version 5.6.0. But we strongly suggest to use at least version 5.6.1, since 5.6.0 is quite buggy. The only reason we support 5.6.0 is for development reasons (so the build can be tested on systems having only 5.6.0) and those users who want to give it a try, without first having the hassle of updating their perl version.

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.

- **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`.

- **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 3.01**
- **Compress::Zlib 1.09**

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 CVS Sources

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

- **Perl**

4.2.3 Configuring and Installing Prerequisites

```
# (--delete to ensure a clean state)
% rsync -acvz --delete --force \
  rsync://ftp.linux.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")'
```

- **Apache**

To download the cvs version of `httpd-2.0` and bring it to the same state of the distribution package, execute the following commands:

```
% cvs -d :pserver:anoncvs@cvs.apache.org:/home/cvspublic login
```

The password is "anoncvs". Now extract the `APACHE_2_0_BRANCH` branch of `httpd-2.0.xx`. If you don't use this branch you will get `httpd-2.1.xx` which at this moment is not supported. Similarly you need `APR_0_9_BRANCH` and `APU_0_9_BRANCH` cvs branches for `apr` and `apr-util` projects, respectively.

```
% cvs -d :pserver:anoncvs@cvs.apache.org:/home/cvspublic co \
  -r APACHE_2_0_BRANCH -d httpd-2.0 httpd-2.0
% cd httpd-2.0/src/lib
% cvs -d :pserver:anoncvs@cvs.apache.org:/home/cvspublic co \
  -r APR_0_9_BRANCH -d apr apr
% cvs -d :pserver:anoncvs@cvs.apache.org:/home/cvspublic co \
  -r APU_0_9_BRANCH -d apr-util apr-util
% cvs -d :pserver:anoncvs@cvs.apache.org:/home/cvspublic co \
  -r APU_0_9_BRANCH -d apr-iconv apr-iconv
% cd ..
% ./buildconf
```

Once extracted, whenever you want to sync with the latest `httpd-2.0` version and rebuild, run:

```
% cd httpd-2.0
% cvs up -dP
% make distclean && ./buildconf
```

4.2.3 *Configuring and Installing Prerequisites*

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

- **Perl**

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

If you need the threads support, run:

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

- **Apache**

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

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/>

This direct link which symlinks to the latest release should work too:
http://perl.apache.org/dist/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.xx.tar.gz*, where *x.xx* should be replaced with the real version number.

Open the package with:

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

or an equivalent command.

- **CVS Bleeding-Edge Version**

To download the cvs version of modperl-2.0 execute the following commands:

```
% cvs -d :pserver:anoncvs@cvs.apache.org:/home/cvspublic login
```

The password is "anoncvs".

```
% cvs -d :pserver:anoncvs@cvs.apache.org:/home/cvspublic co modperl-2.0
```

You can also try the latest CVS snapshot:

<http://cvs.apache.org/snapshots/modperl-2.0/>

4.4.2 Configuring mod_perl

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

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

```
% cd modperl-1.99_xx
% perl Makefile.PL <options>
```

where *options* is an optional list of (key,value) pairs.

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

If you want to have mod_perl 1.0 and 2.0 installed under the same perl tree you need to enable MP_INST_APACHE2:

```
% perl Makefile.PL MP_INST_APACHE2=1 <other options>
```

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_INST_APACHE2=1 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_INST_APACHE2=1 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_INST_APACHE2=1 MP_AP_PREFIX=$HOME/httpd/prefork
```

These and other options are discussed in the following sections.

4.4.2.1 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.

4.4.2.1.1 *MP_PROMPT_DEFAULT*

Accept default values for all would-be prompts.

4.4.2.1.2 *MP_GENERATE_XS*

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

4.4.2.1.3 *MP_USE_DSO*

Build `mod_perl` as a DSO (*mod_perl.so*). This is the default. It'll be turned off if `MP_USE_STATIC=1` is used.

4.4.2.1.4 *MP_USE_STATIC*

Build static `mod_perl` (*mod_perl.a*). This is the default. It'll be turned off if `MP_USE_DSO=1` is used.

`MP_USE_DSO` and `MP_USE_STATIC` are both enabled by default. So `mod_perl` is built once as *mod_perl.a* and *mod_perl.so*, but afterwards you can choose which of the two to use.

META: The following is not implemented yet.

```
mod_perl and ends up with a src/modules/perl/mod_perl.{so,a} and
src/modules/perl/ldopts. to link modperl static with httpd, we just
need some config.m4 magic to add 'ldopts' and mod_perl.a to the build.
so one could then build httpd like so:
```

4.4.2 Configuring mod_perl

```
ln -s ~/apache/modperl-2.0/src/modules/perl $PWD/src/modules
./configure --with-mpm=prefork --enable-perl=static ...
```

we not be configuring/building httpd for the user as 1.x attempted.

downside is one will need to have configured httpd first, so that headers generated. so it will probably be more like:

```
./configure --with-mpm=prefork ...
(go build modperl)
./config.nice --enable-perl=static && make
```

we could of course provide a wrapper script todo this, but don't want to have this stuff buried and tangled like it is in 1.x

4.4.2.1.5 *MP_STATIC_EXTS*

Build Apache::*.xs as static extensions.

4.4.2.1.6 *MP_USE_GTOP*

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

4.4.2.1.7 *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:

- Environment variable GATEWAY_INTERFACE will be enabled only if PerlOptions +SetupEnv is enabled and its value would be the default:

```
CGI/1.1
```

and not:

```
CGI-Perl/1.1
```

The use of \$ENV{GATEWAY_INTERFACE} is deprecated and the existence of \$ENV{MOD_PERL} should be checked instead.

- Deprecated special variable, \$Apache::__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 Apache::Server ();
use Apache::ServerUtil ();
use Apache::Process ();
my $pool = Apache->server->process->pool;
push @INC, Apache::server_root_relative($pool, "");
push @INC, Apache::server_root_relative($pool, "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.

4.4.2.1.8 MP_DEBUG

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

4.4.2.1.9 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 \
-DAP_DEBUG -DAP_HAVE_DESIGNATED_INITIALIZER
```

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

4.4.2.1.10 MP_TRACE

Enable tracing

4.4.2.1.11 MP_INST_APACHE2

Install all the `*.pm` modules relative to the *Apache2/* directory.

4.4.2.2 Non-Boolean Build Options

set the non-boolean options with `MP_XXX=value`.

4.4.2.2.1 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`.

4.4.2.2.2 MP_AP_PREFIX

Apache installation prefix, under which the `include/` directory with Apache C header files can be found. For example if you've have 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
```

4.4.2.2.3 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.

4.4.2.2.4 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.)

4.4.2.2.5 MP_OPTIONS_FILE

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

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

4.4.2.3 mod_perl-specific Compiler Options

4.4.2.3.1 -DMP_IOBUFSIZE

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

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

4.4.2.4 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.

4.4.3 Re-using Configure Options

Since mod_perl remembers what build options were used to build it in 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.xx
% perl -MApache::Build -e rebuild
```

4.4.4 Compiling mod_perl

Next stage is to build mod_perl:

```
% make
```

4.4.5 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.

4.4.6 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 <stas (at) stason.org>

4.7 Authors

- Stas Bekman <stas (at) stason.org>
- Doug MacEachern <doug (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.

5.4 Accessing the mod_perl 2.0 Modules

In order to prevent from inadvertently loading mod_perl 1.0 modules mod_perl 2.0 Perl modules are installed into dedicated directories under *Apache2/*. The *Apache2* module prepends the locations of the mod_perl 2.0 libraries to `@INC`, which are the same as the core `@INC`, but with *Apache2/* appended. This module has to be loaded just after mod_perl has been enabled. This can be accomplished with:

```
use Apache2 ();
```

in the startup file. Only if you don't use a startup file you can add:

```
PerlModule Apache2
```

to *httpd.conf*, due to the order the `PerlRequire` and `PerlModule` directives are processed.

5.5 Startup File

Next usually a startup file with Perl code is loaded:

```
PerlRequire "/home/httpd/httpd-2.0/perl/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 Apache2 ();

use lib qw(/home/httpd/perl);

# enable if the mod_perl 1.0 compatibility is needed
# use Apache::compat ();

# preload all mp2 modules
# use ModPerl::MethodLookup;
# ModPerl::MethodLookup::preload_all_modules();

use ModPerl::Util (); #for CORE::GLOBAL::exit

use Apache::RequestRec ();
use Apache::RequestIO ();
use Apache::RequestUtil ();

use Apache::Server ();
use Apache::ServerUtil ();
use Apache::Connection ();
use Apache::Log ();

use APR::Table ();

use ModPerl::Registry ();

use Apache::Const -compile => ':common';
use APR::Const -compile => ':common';

1;
```

In this file the Apache2 modules is loaded, so the 2.0 modules will be found. Afterwards @INC is 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 Apache::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 ; .

5.6 Server Configuration Directives

5.6.1 *PerlRequire*

META: to be written

5.6.2 *PerlModule*

META: to be written

5.6.3 *PerlLoadModule*

META: to be written

discussed somewhere in docs::2.0::user::config::custom

5.6.4 *PerlSetVar*

META: to be written

5.6.5 *PerlAddVar*

META: to be written

5.6.6 *PerlSetEnv*

META: to be written

5.6.7 *PerlPassEnv*

META: to be written

5.6.8 *<Perl> Sections*

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

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

META: a dedicated chapter with examples?

5.6.9 *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.

5.6.10 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.

5.6.10.1 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 (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` (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 Apache::VarTest
    PerlSetVar VarTest VarTestValue
</Location>
```

this value can be retrieved inside `Apache::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.

5.6.10.2 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 handler is called and restored afterwards (similar to `mod_perl 1.0`). This includes: `%ENV`, `@INC`, `$/`, `STDOUT's $|` and `END` blocks array (`PL_endav`).

5.6.10.3 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:MyApache/PrintEnv1.pm
-----
package MyApache::PrintEnv1;
use strict;

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

sub handler {
```

```

    my $r = shift;

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

    return Apache::OK;
}

1;

```

This is the required configuration:

```

PerlModule MyApache::PrintEnv1
<Location /print_env1>
    SetHandler perl-script
    PerlResponseHandler MyApache::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:MyApache/PrintEnv2.pm
-----
package MyApache::PrintEnv2;
use strict;

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

use Apache::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 Apache::OK;
}

1;

```

The configuration now will look as:

```

PerlModule MyApache::PrintEnv2
<Location /print_env2>
    SetHandler modperl
    PerlResponseHandler MyApache::PrintEnv2
</Location>

```

MyApache::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`.

5.6.11 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 `PerlInterpreter` is used for a `<VirtualHost>` or location configured with `<Location>`, `<Directory>`, etc.

Options are enabled by prepending `+` and disabled with `-`. The options include:

5.6.11.1 Enable

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

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

5.6.11.2 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.

When cloning a Perl interpreter, to inherit base Perl interpreter's `PerlSwitches` use:

```
<VirtualHost ...>
    ...
    PerlSwitches +inherit
</VirtualHost>
```

5.6.11.3 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
    PerlModule Apache2
</VirtualHost>

<VirtualHost ...>
    ServerName dev2
    PerlOptions +Parent
    PerlSwitches -Mlib=/home/dev2/lib/perl
    PerlModule Apache2
</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.

5.6.11.4 Perl*Handler

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

```
PerlOptions -Log
```

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

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

Or maybe everything but the response handler:

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

5.6.11.5 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.

5.6.11.6 GlobalRequest

Setup the global `request_rec` for use with `Apache->request`.

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

```
<Location ...>
    SetHandler perl-script
    ...
</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 $r = shift;
    Apache->request($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.

5.6.11.7 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";
```

5.6.11.8 MergeHandlers

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

```
PerlFixupHandler Apache::FixupA

<Location /inside>
    PerlFixupHandler Apache::FixupB
</Location>
```

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

```
PerlFixupHandler Apache::FixupA

<Location /inside>
    PerlOptions +MergeHandlers
    PerlFixupHandler Apache::FixupB
</Location>
```

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

5.6.11.9 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 `$ENV{QUERY_STRING}` environment variable is initialized with the contents of `Apache::args()`, and the value returned by `Apache::server_hostname()` is put into `$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 = Apache->request();
$r->send_http_header('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*.

5.7 Server Life Cycle Handlers Directives

See Server life cycle.

5.7.1 *PerlOpenLogsHandler*

See PerlOpenLogsHandler.

5.7.2 PerlPostConfigHandler

See PerlPostConfigHandler.

5.7.3 PerlChildInitHandler

See PerlChildInitHandler.

5.7.4 PerlChildExitHandler

See PerlChildExitHandler.

5.8 Protocol Handlers Directives

See Protocol handlers.

5.8.1 PerlPreConnectionHandler

See PerlPreConnectionHandler.

5.8.2 PerlProcessConnectionHandler

See PerlProcessConnectionHandler.

5.9 Filter Handlers Directives

mod_perl filters are described in the filter handlers tutorial, `Apache::Filter` and `Apache::FilterRec` manpages.

The following filter handler configuration directives are available:

5.9.1 PerlInputFilterHandler

See PerlInputFilterHandler.

5.9.2 PerlOutputFilterHandler

See PerlOutputFilterHandler.

5.9.3 PerlSetInputFilter

See PerlSetInputFilter.

5.9.4 PerlSetOutputFilter

See PerlSetInputFilter.

5.10 HTTP Protocol Handlers Directives

See HTTP protocol handlers.

5.10.1 PerlPostReadRequestHandler

See PerlPostReadRequestHandler.

5.10.2 PerlTransHandler

See PerlTransHandler.

5.10.3 PerlMapToStorageHandler

See PerlMapToStorageHandler.

5.10.4 PerlInitHandler

See PerlInitHandler.

5.10.5 PerlHeaderParserHandler

See PerlHeaderParserHandler.

5.10.6 PerlAccessHandler

See PerlAccessHandler.

5.10.7 PerlAuthenHandler

See PerlAuthenHandler.

5.10.8 PerlAuthzHandler

See PerlAuthzHandler.

5.10.9 PerlTypeHandler

See PerlTypeHandler.

5.10.10 PerlFixupHandler

See PerlFixupHandler.

5.10.11 PerlResponseHandler

See PerlResponseHandler.

5.10.12 PerlLogHandler

See PerlLogHandler.

5.10.13 PerlCleanupHandler

See PerlCleanupHandler.

5.11 Threads Mode Specific Directives

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

5.11.1 PerlInterpStart

The number of interpreters to clone at startup time.

Default value: 3

5.11.2 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

5.11.3 *PerlInterpMinSpare*

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

Default value: 3

5.11.4 *PerlInterpMaxSpare*

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

Default value: 3

5.11.5 *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

5.11.6 *PerlInterpScope*

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

```
PerlInterpScope request
```

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

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

```
PerlInterpScope subrequest
```

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

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

```
PerlInterpScope handler
```

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

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

```
PerlInterpScope connection
```

Default value: `request`

5.12 Debug Directives

5.12.1 *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 Perl runtime interaction
h handlers
i interpreter pool management
m memory allocations
o I/O
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.

5.13 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
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

5.13 mod_perl Directives Argument Types and Allowed Location

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 Single Phase's Multiple Handlers Behavior.

5.14 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 `Apache::exists_config_define`. For example in a startup file you can say:

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

5.14.1 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 `Apache::exists_config_define()`, for example:

```
if (Apache::exists_config_define("MODPERL2")) {
    # some 2.0 specific code
}
```

5.15 Perl Interface to the Apache Configuration Tree

For now refer to the `Apache::Directive` manpage and the test `t/response/TestApache/confree.pm` in the `mod_perl` source distribution.

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

5.16 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
```

5.16.1 PERL5LIB and PERLLIB Environment Variables

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

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

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

5.16.2 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
  PerlModule Apache2
</VirtualHost>

<VirtualHost ...>
  ServerName dev2
  PerlOptions +Parent
  PerlSwitches -I/home/dev2/lib/perl
  PerlModule Apache2
</VirtualHost>
```

5.17 General Issues

5.18 Maintainers

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

- Stas Bekman <stas (at) stason.org>

5.19 Authors

- Doug MacEachern <doug (at) covalent.net>
- Stas Bekman <stas (at) 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.

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

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

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

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

use Apache::CmdParms ();
use Apache::Module ();

our @APACHE_MODULE_COMMANDS = (
    {
        name          => 'MyParameter',
        func           => __PACKAGE__ . '::MyParameter',
        req_override   => Apache::OR_ALL,
        args_how       => Apache::ITERATE,
        errmsg         => 'MyParameter Entry1 [Entry2 ... [EntryN]]',
    },
    {
        name          => 'MyOtherParameter',
    },
);

sub MyParameter {
    my($self, $parms, @args) = @_;
    $self->{MyParameter} = \@args;
}
1;
```

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

```
# first load the module so Apache will recognize the new directives
PerlLoadModule MyApache::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 two groups of elements:

- A global array `@APACHE_MODULE_COMMANDS` for declaring the new directives and their behavior.
- A subroutine per each new directive, which is called when the directive is seen

6.3.1 @APACHE_MODULE_COMMANDS

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

```
our @APACHE_MODULE_COMMANDS = (
  {
    name      => 'MyParameter',
    func      => __PACKAGE__ . '::MyParameter',
    req_override => Apache::OR_ALL,
    args_how  => Apache::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.

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 `Apache::Const` package.

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

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

and then assigned to the `req_override` attribute:

```
req_override => Apache::OR_ALL,
```

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

```
req_override => Apache::RSRC_CONF | Apache::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 `Apache::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 `Apache::Const` package provides special constants which map 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 `Apache::ITERATE` constant:

```
args_how => Apache::ITERATE,
```

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

META: the default may change to use a constant corresponding to the *func* prototype.

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:

```
our @APACHE_MODULE_COMMANDS = (
  {
    name          => '<Location',
    # func defaults to Redirect()
    req_override => Apache::RSRC_CONF,
    args_how      => Apache::RAW_ARGS,
  },
  {
    name          => '<LocationMatch',
    func          => Redirect,
    req_override => Apache::RSRC_CONF,
    args_how      => Apache::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 `Apache::CmdParms` class.

```
use Apache::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 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*.)

6.3.2.1 Apache:::OR_NONE

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

6.3.2.2 Apache:::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.

6.3.2.3 Apache:::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.

6.3.2.4 Apache:::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.

6.3.2.5 Apache:::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.

6.3.2.6 Apache:::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.

6.3.2.7 Apache:::OR_UNSET

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

6.3.2.8 Apache:::ACCESS_CONF

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

6.3.2.9 `Apache::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.

6.3.2.10 `Apache::OR_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 `Apache::OR_EXEC_ON_READ` turned on.

6.3.2.11 `Apache::OR_ALL`

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

6.3.3 *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 (`Apache::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 `Apache::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 `Apache::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.

6.3.4 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 `Apache::Const` package. For example:

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

6.3.4.1 Apache::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}++;
}
```

6.3.4.2 Apache::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;
}
```

6.3.4.3 Apache::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};
}
```

6.3.4.4 Apache::TAKE3

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

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

6.3.4.5 Apache::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'};
}
```

6.3.4.6 Apache::TAKE23

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

6.3.4.7 Apache::TAKE123

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

6.3.4.8 Apache::ITERATE

Apache::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;
}
```

6.3.4.9 Apache::ITERATE2

Apache::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;
}
```

6.3.4.10 Apache::RAW_ARGS

An *args_how* of Apache::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 Apache::TAKE1), the third of which is a string-valued scalar containing the text following the directive.

```
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 Apache::OR_EXEC_ON_READ. e.g.:

```
req_override => Apache::OR_ALL | Apache::OR_EXEC_ON_READ,
```

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

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 `Apache::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      => Apache::NO_ARGS,
    req_override  => Apache::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
```

6.3.4.11 Apache::FLAG

When `Apache::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
}
```

6.3.5 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 MyApache::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.

6.3.6 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.

6.3.6.1 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 `Apache::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 MyApache::MyParameters;
...
use Apache::Module ();
use Apache::CmdParms ();
our @APACHE_MODULE_COMMANDS = (...);
...
sub SERVER_CREATE {
    my($class, $parms) = @_;
    return bless {
        name => __PACKAGE__,
    }, $class;
}
```

To retrieve that value later, you can use:

```
use Apache::Module ();
...
my $srv_cfg = Apache::Module->get_config('MyApache::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 Apache::Module ();
use Apache::Server ();
use Apache::ServerUtil ();
...
if ($s->is_virtual) {
    my $base_srv_cfg = Apache::Module->get_config('MyApache::MyParameters',
                                                Apache->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.

6.3.6.2 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.

6.3.6.3 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 MyApache::MyParameters;
...
use Apache::Module ();
use Apache::CmdParms ();
```

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

To retrieve that value later, you can use:

```
use Apache::Module ();
...
my $dir_cfg = Apache::Module->get_config('MyApache::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 *Apache::Module* to get the directory configuration object.

6.3.6.4 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:MyApache/CustomDirectives.pm
#-----
package MyApache::CustomDirectives;

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

use Apache::CmdParms ();
use Apache::Module ();
use Apache::ServerUtil ();
```

6.4.1 Merging at Work

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

our @APACHE_MODULE_COMMANDS = (
    { name => 'MyPlus' },
    { name => 'MyList' },
    { name => 'MyAppend' },
    { name => 'MyOverride' },
);

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 $srv_cfg = Apache::Module->get_config($self,
                                                    $parms->server);
        $srv_cfg->{$key} = $arg;
    }
}

sub push_val {
    my($key, $self, $parms, $arg) = @_;

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

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

    my %mrg = ();
    for my $key (keys %$base, %$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 `@APACHE_MODULE_COMMANDS` 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, `Apache::TAKE1`, the default `args_how`, is what we need. We also allow the directives to appear anywhere, so `Apache::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 `Apache::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 `Apache::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 `MyApache::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 MyApache::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 MyApache::CustomDirectivesTest
  </Location>
  <Location /custom_directives_test/subdir>
    MyPlus 1
    MyList      "SubDir"
    MyAppend    "SubDir"
    MyOverride  "SubDir"
  </Location>
</VirtualHost>
<Location /custom_directives_test>
  SetHandler modperl
  PerlResponseHandler MyApache::CustomDirectivesTest
</Location>

```

PerlLoadModule loads the Perl module `MyApache::CustomDirectives` and then installs a new Apache module named `MyApache::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 certain things identically, while others differently. So here it the handler:

```

#file:MyApache/CustomDirectivesTest.pm
#-----
package MyApache::CustomDirectivesTest;

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

use Apache::RequestRec ();
use Apache::RequestIO ();
use Apache::Server ();
use Apache::ServerUtil ();
use Apache::Module ();

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

sub get_config {
    Apache::Module->get_config('MyApache::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) {
        $secs{"1: Main Server"} = get_config($s->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 Apache::OK;
}

```

6.4.1 Merging at Work

```
}  
  
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:

6.4.1 Merging at Work

```
$mrg{$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 <stas (at) stason.org>

6.6 Authors

- Stas Bekman <stas (at) 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 Goodies Toolkit

7.4.1 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/1.99_03-dev
```

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 => ($mod_perl::VERSION >= 1.99);
# die "I want mod_perl 2.0!" unless MP2;
```

- `$ENV{GATEWAY_INTERFACE}` - is set to `CGI-Perl/1.1` for compatibility with mod_perl 1.0. This variable is deprecated in mod_perl 2.0. Use `$ENV{MOD_PERL}` instead.

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`.

7.4.2 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 `Apache::MPM->is_threaded` can be used. For example:

```
use Apache::MPM ();
if (Apache::MPM->is_threaded) {
    require APR::OS;
    my $tid = APR::OS::thread_current();
    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.

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

7.5 Code Developing Nuances

7.5.1 Auto-Reloading Modified Modules with `Apache::Reload`

META: need to port `Apache::Reload` notes from the guide here. but the gist is:

```
PerlModule Apache::Reload
PerlInitHandler Apache::Reload
#PerlPreConnectionHandler Apache::Reload
PerlSetVar ReloadAll Off
PerlSetVar ReloadModules "ModPerl:* Apache:*
```

Use:

```
PerlInitHandler Apache::Reload
```

if you need to debug HTTP protocol handlers. Use:

```
PerlPreConnectionHandler Apache::Reload
```

for any handlers.

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

```
package Apache::Whatever;

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

FATAL => 'all' escalates all warnings into fatal errors. So when Apache::Whatever is modified and reloaded by Apache::Reload the request is aborted. Therefore if you follow this very healthy style and want to use Apache::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.

7.6 Integration with Apache Issues

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

7.6.1 Sending HTTP Response Headers

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 send 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 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 Apache::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 orders to flush the data.

META: add a link to the notes on how to write well-behaved filters at handlers/filters

7.6.2 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 Perl Specifics in the mod_perl Environment

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

7.7.1 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.

7.7.2 *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 to 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.

7.8 Threads Coding Issues Under mod_perl

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

7.8.1 *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.com/perl5.8.0/pod/perlthrtut.html>) manpage.

7.8.2 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>.

7.8.3 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.

7.9 Maintainers

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

-

7.10 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 Apache::RequestRec ();
use APR::Table ();

use Apache::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(Apache::REDIRECT);

    return Apache::REDIRECT;
}
1;
```

8.3 Sending Cookies in REDIRECT Response (handlers)

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

use Apache::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 Apache::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 Maintainers

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

- Stas Bekman <stas (at) stason.org>

8.5 Authors

- Stas Bekman <stas (at) 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 `Apache::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 `Apache::compat` with no further changes

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

- Loads all the mod_perl 2.0 Apache:: 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 `Apache::compat` is the performance hit, which can be significant.

Authors of CPAN and other publicly distributed modules should not use `Apache::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 `Apache::porting`

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

9.4 Using the `Apache::compat` Layer

The `Apache::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.

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

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

or add to `httpd.conf`:


```
PerlModule Apache2
PerlModule Apache::compat
```

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

Remember, however, that using `Apache::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 -MApache2 -MModPerl::MethodLookup -e print_method content_type
```

This prints:

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

We do what it says and add this `use()` statement to our code, restart our server (unless we're using `Apache::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 exist 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 Apache::RequestIO ();
    use Apache::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 `Apache::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 `Apache::RequestIO` and `Apache::Filter` expect an object of type `Apache::RequestRec` and type `Apache::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::MethodLookup`, 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 or 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:

9.5.3 Requiring a specific mod_perl version.

```
% perl -MApache2 -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 `Apache::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:

```
$parsed_uri = Apache::URI->parse($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 `Apache::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 Apache2;
use mod_perl 2.0;
```

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

```
use Apache2;
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.

Let's say that you have a module `Apache::Friendly` whose release version compliant with mod_perl 1.0 is 1.57. You keep this version on CPAN and release a new version, 2.01, which is compliant with mod_perl 2.0 and preserves the name of the module. It's possible that a user may need to have both versions of the module on the same machine. Since the two have the same name they obviously cannot live under the same tree.

One attempt to solve this problem is to use *Makefile.PL*'s `MP_INST_APACHE2` option. If the module is configured as:

```
% perl Makefile.PL MP_INST_APACHE2=1
```

it'll be installed relative to the *Apache2/* directory.

META: but of course this won't work in non-core mod_perl, since a generic *Makefile.PL* has no idea what to do about `MP_INST_APACHE2=1`. Need to provide copy-n-paste recipe for this. Or even add to the core a supporting module that will handle this functionality.

The second step is to change the documentation of your 2.0 compliant module to instruct users to use `Apache2 ()`; in their code (or in *startup.pl* or via `PerlModule Apache2` in *httpd.conf*) before the module is required. This will cause `@INC` to be modified to include the *Apache2/* directory first.

The introduction of the *Apache2/* directory is similar to how Perl installs its modules in a version specific directory. For example:

```
lib/5.7.1
lib/5.7.2
```

9.5.5 *Using Apache::compat As a Tutorial*

Even if you have followed the recommendation and eschewed use of the `Apache::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 `Apache::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 `Apache/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-porting 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 `Apache::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 Apache::Reload
PerlInitHandler Apache::Reload
PerlSetVar ReloadAll Off
PerlSetVar ReloadModules "ModPerl:* Apache:*"
PerlSetVar ReloadConstantRedefineWarnings Off
```

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

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

tells `Apache::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 Apache::Reload
PerlInitHandler Apache::Reload
PerlSetVar ReloadAll Off
```

9.5.6 How Apache::MP3 was Ported to mod_perl 2.0

```
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 `Apache::compat` module. Now my *startup.pl* looked like this:

```
#file:startup.pl
#-----
use Apache2 ();
use lib qw(/home/httpd/2.0/perl);
use Apache::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 getting 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 `Apache::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: Apache::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 *Apache::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} , ',');
    }
}
```

9.5.6 How Apache::MP3 was Ported to mod_perl 2.0

```

        (my $description = $format) =~ s{%(.[atfglnrcrdmsqS%])}
@@ -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{%(atfglnrcrdmsqS%)}
        {$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_sockaddr_in()`, and in the scalar context it's calling `pack_sockaddr_in()`. So I have found `sockaddr_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::SocketAddr` 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::SocketAddr;
+    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) {
```

9.5.6 How Apache::MP3 was Ported to mod_perl 2.0

```
@@ -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 Apache::compat Dependency

The final stage is going to get rid of Apache::compat since this is a CPAN module, which must not load Apache::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 Apache::compat in *startup.pl*:

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

9.5.6.4 Ensuring that Apache::compat is not loaded

The second step is to make sure that Apache::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 "Apache::compat is loaded loaded" if $INC{'Apache/compat.pm'};
+}
+
+use strict;
```

and indeed, even though I've commented out the loading of Apache::compat from *startup.pl*, this module was still getting loaded. I knew that because the request to */mp3* were failing with the error message:

Apache::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("Apache::compat is loaded") if $_[0] =~ /compat/;
        CORE::require(@_);
    };
}
```

or you can modify *Apache/compat.pm* and make it print the calls trace when it gets compiled:

```
--- Apache/compat.pm.orig    2003-06-03 16:11:07.000000000 +1000
+++ Apache/compat.pm        2003-06-03 16:11:58.000000000 +1000
@@ -1,5 +1,9 @@
    package Apache::compat;

+BEGIN {
+    use Carp;
+    Carp::cluck("Apache::compat is loaded by");
+}
```

I've used this last technique, since it's the safest one to use. Remember that `Apache::compat` can also be loaded with:

```
do "Apache/compat.pm";
```

in which case, neither `grep(1)`'ping for `Apache::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:

```
Apache::compat is loaded by at .../Apache2/Apache/compat.pm line 6
Apache::compat::BEGIN() called at .../Apache2/Apache/compat.pm line 8
eval {...} called at .../Apache2/Apache/compat.pm line 8
require Apache/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/Apache/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 `Apache::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 `Apache::compat` (which should be never done by CPAN modules). Once I've upgraded `CGI.pm` to version 2.93 and restarted the server, `Apache::compat` wasn't getting loaded any longer.

9.5.6.5 Installing the ModPerl::MethodLookup Helper

Now that `Apache::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 Apache2 ();
use lib qw(/home/httpd/2.0/perl);

{
    package ModPerl::MethodLookupAuto;
    use ModPerl::MethodLookup;

    use Carp;
    sub handler {

        # look inside mod_perl:: Apache:: 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 Apache::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 `Apache::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 `Apache::` or `APR::` namespaces. The `Apache::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 Apache::$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);
```

9.5.6 How Apache::MP3 was Ported to mod_perl 2.0

```

    }

-   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 @@

```

9.5.6 How Apache::MP3 was Ported to mod_perl 2.0

```
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 Apache::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
"Apache::RequestRec" at ../Apache/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"}) {
```

9.5.6 How Apache::MP3 was Ported to mod_perl 2.0

```
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 Apache::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 Apache::Connection ();
+use Apache::SubRequest ();
+use Apache::Access ();
+use Apache::RequestIO ();
+use Apache::RequestUtil ();
+use Apache::RequestRec ();
+use Apache::ServerUtil ();
+use Apache::Log;
    use APR::Table ();

    use Apache::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 Apache::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 Apache::OK if $r->header_only;

# local user
@@ -495,7 +495,7 @@
return Apache::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 Apache::OK if $self->r->header_only;

$self->page_top($dir);

```

also I've noticed that there was this code:

```
return Apache::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 `Apache::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 `Apache::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 Apache::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 variable `$mod_perl::VERSION`:

```
use mod_perl;
use constant MP2 => ($mod_perl::VERSION >= 1.99);
# ...
require Symbol if MP2;
# ...

my $fh = MP2 ? Symbol::gensym : Apache->gensym;
open $fh, $file or die "Can't open $file: $!";
```

Though, make sure that you don't use `$mod_perl::VERSION` string anywhere in the code before you have declared your module's own `$VERSION`, since PAUSE will pick the wrong version when you submit the module on CPAN. It requires that module's `$VERSION` will be declared first. You can verify whether it'll pick the `Foo.pm`'s version correctly, by running this code:

```
% perl -MExtUtils::MakeMaker -le 'print MM->parse_version(shift)' Foo.pm
```

There is more information about this issue here:

http://pause.perl.org/pause/query?ACTION=pause_04about#conventions

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}
? eval { require mod_perl; $mod_perl::VERSION >= 1.99 ? 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 `Apache::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 Apache::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 `MyApache::Method` which has a single method that works for both `mod_perl` generations:

The configuration:

```
PerlModule MyApache::Method
<Location /method>
    SetHandler perl-script
    PerlHandler MyApache::Method->handler
</Location>
```

The code:

```
#file:MyApache/Method.pm
package MyApache::Method;

# PerlModule MyApache::Method
# <Location /method>
#     SetHandler perl-script
#     PerlHandler MyApache::Method->handler
# </Location>

use strict;
use warnings;

use mod_perl;
use constant MP2 => $mod_perl::VERSION < 1.99 ? 0 : 1;

BEGIN {
    if (MP2) {
        require Apache::RequestRec;
        require Apache::RequestIO;
        require Apache::Const;
        Apache::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 ? Apache::OK : Apache::Constants::OK;
}
```

Here are two complete examples. The second example implements `MyApache::Method2`, which is very similar to `MyApache::Method`, but uses separate methods for `mod_perl` 1.0 and 2.0 servers.

The configuration is the same:

```
PerlModule MyApache::Method2
<Location /method2>
    SetHandler perl-script
    PerlHandler MyApache::Method2->handler
</Location>
```

The code:

```
#file:MyApache/Method2.pm
package MyApache::Method2;

# PerlModule MyApache::Method
# <Location /method>
#     SetHandler perl-script
#     PerlHandler MyApache::Method->handler
# </Location>

use strict;
use warnings;

use mod_perl;
use constant MP2 => $mod_perl::VERSION < 1.99 ? 0 : 1;

BEGIN {
    warn "running $mod_perl::VERSION!\n";
    if (MP2) {
        require Apache::RequestRec;
        require Apache::RequestIO;
        require Apache::Const;
        Apache::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 Apache::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  
MyApache::Method was called  
  
% lynx --source http://localhost:8001/method2  
mp1: MyApache::Method2 was called  
  
% lynx --source http://localhost:8002/method  
MyApache::Method was called  
  
% lynx --source http://localhost:8002/method2  
mp2: MyApache::Method2 was called
```

9.7 Maintainers

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

Stas Bekman <stas (at) stason.org>

9.8 Authors

- Nick Tonkin <nick (at) tonkinresolutions.com>
- Stas Bekman <stas (at) 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 will be explained in details later loading `Apache::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 *PerlSendHeader*

`PerlSendHeader` was replaced with `PerlOptions +/ParseHeaders` directive.

```
PerlSendHeader On  => PerlOptions +ParseHeaders
PerlSendHeader Off => PerlOptions -ParseHeaders
```

10.2.3 *PerlSetupEnv*

`PerlSetupEnv` was replaced with `PerlOptions +/SetupEnv` directive.

```
PerlSetupEnv On  => PerlOptions +SetupEnv
PerlSetupEnv Off => PerlOptions -SetupEnv
```

10.2.4 *PerlTaintCheck*

The taint mode now can be turned on with:

```
PerlSwitches -T
```

As with standard Perl, by default the taint mode is disabled and once enabled cannot be turned off inside the code.

10.2.5 *PerlWarn*

Warnings now can be enabled globally with:

```
PerlSwitches -w
```

10.2.6 *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>
```

10.2.7 *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.

10.2.8 *@INC Manipulation*

- **Directories Added Automatically to @INC**

Only if mod_perl was built with MP_COMPAT_1X=1, two directories: *\$ServerRoot* and *\$ServerRoot/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 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 compatibility with 1.0 while running under 2.0, you should load the compatibility module as early as possible:

```
use Apache::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.

If you have mod_perl 1.0 and 2.0 installed on the same system and the two use the same perl libraries directory (e.g. */usr/lib/perl5*), to use mod_perl 2.0 make sure to load first the Apache2 module which will perform the necessary adjustments to @INC.

```
use Apache2; # if you have 1.0 and 2.0 installed
use Apache::compat;
```

So if before loading *Apache2.pm* the @INC array consisted of:

```
/home/stas/perl/ithread/lib/5.8.0/i686-linux-thread-multi
/home/stas/perl/ithread/lib/5.8.0
/home/stas/perl/ithread/lib/site_perl/5.8.0/i686-linux-thread-multi
/home/stas/perl/ithread/lib/site_perl/5.8.0
/home/stas/perl/ithread/lib/site_perl
.
```

It will now look as:

```
/home/stas/perl/ithread/lib/site_perl/5.8.0/i686-linux-thread-multi/Apache2
/home/stas/perl/ithread/lib/5.8.0/i686-linux-thread-multi
/home/stas/perl/ithread/lib/5.8.0
/home/stas/perl/ithread/lib/site_perl/5.8.0/i686-linux-thread-multi
/home/stas/perl/ithread/lib/site_perl/5.8.0
/home/stas/perl/ithread/lib/site_perl
.
```

Notice that a new directory was prepended to the search path, so if for example the code attempts to load *Apache::RequestRec* and there are two versions of this module under */home/stas/perl/ithread/lib/site_perl/*:

```
5.8.0/i686-linux-thread-multi/Apache/RequestRec.pm
5.8.0/i686-linux-thread-multi/Apache2/Apache/RequestRec.pm
```

The mod_perl 2.0 version will be loaded first, because the directory *5.8.0/i686-linux-thread-multi/Apache2* is coming before the directory *5.8.0/i686-linux-thread-multi* in @INC.

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 preferably avoided in the production environment if you want to save memory.

10.4 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 `Apache::` which are used for apr and apache features, respectively.

At this moment `ModPerl::Registry` (and others) doesn't `chdir()` into the script's dir like `Apache::Registry` does, because `chdir()` affects the whole process under threads. This should be resolved by the time mod_perl 2.0 is released. Arthur Bergman works on the solution in form of: `ex::threads::cwd`. See: <http://www.perl.com/pub/a/2002/06/11/threads.html?page=2> Someone should pick up and complete this module to make it really useful.

Meanwhile if you are using a prefork MPM and you have to rely on mod_perl performing `chdir` to the script's directory, you can use the following subclass of `ModPerl::Registry`:

```
#file:ModPerl/RegistryPrefork.pm
#-----
package ModPerl::RegistryPrefork;

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

our $VERSION = '0.01';

use base qw(ModPerl::Registry);

use File::Basename ();

sub handler : method {
    my $class = (@_ >= 2) ? shift : __PACKAGE__;
    my $r = shift;
    return $class->new($r)->default_handler();
}

sub chdir_file {
    my $file = @_ == 2 ? $_[1] : $_[0]->{FILENAME};
    my $dir = File::Basename::dirname($file);
    chdir $dir or die "Can't chdir to $dir: $!";
}
```

```

}

1;
__END__

```

Adjust your *httpd.conf* to have:

```

Alias /perl /path/to/perl/scripts
<Location /perl>
    SetHandler perl-script
    PerlResponseHandler ModPerl::RegistryPrefork
    Options +ExecCGI
    PerlOptions +ParseHeaders
</Location>

```

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`.

10.4.1 *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.

10.5 Apache::Constants

`Apache::Constants` has been replaced by three classes:

- **Apache::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 `Apache::compat` doesn't provide a complete back compatibility layer.

10.5.1 *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 MyApache::Foo;

use strict;
use warnings;

use mod_perl;
use constant MP2 => $mod_perl::VERSION >= 1.99;

BEGIN {
    if (MP2) {
        require Apache::Const;
        Apache::Const->import(-compile => qw(OK DECLINED));
    }
    else {
        require Apache::Constants;
        Apache::Constants->import(qw(OK DECLINED));
    }
}

sub handler {
    # ...
    return MP2 ? Apache::OK : Apache::Constants::OK;
}
1;
```

Notice that if you don't use the idiom:

```
return MP2 ? Apache::OK : Apache::Constants::OK;
```

but something like the following:

```
sub handler1 {
    ...
    return Apache::Constants::OK();
}
sub handler2 {
    ...
    return Apache::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.

10.5.2 *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 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.

10.5.3 *SERVER_VERSION()*

Apache::Constants::SERVER_VERSION() has been replaced with:

```
Apache::get_server_version();
```

10.5.4 *export()*

Apache::Constants::export() has no replacement in 2.0 as it's not needed.

10.6 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.)

10.7 Special Environment Variables

10.7.1 *`$ENV{GATEWAY_INTERFACE}`*

The environment variable `$ENV{GATEWAY_INTERFACE}` is deprecated in mod_perl 2.0 (See: `MP_COMPAT_1X=0`). Instead use `$ENV{MOD_PERL}` (available in both mod_perl generations), which is set to something like this:

```
mod_perl/1.99_03-dev
```

However to check the version it's better to use `$mod_perl::VERSION`:

```
use mod_perl;
use constant MP2 => ($mod_perl::VERSION >= 1.99);
```

10.8 `Apache::Methods`

10.8.1 *`Apache->request`*

`Apache->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 `Apache->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 Apache::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 Apache::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 Apache->request being available at run time. For any location that uses Apache->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 Apache->request available **only** during the response phase (PerlResponseHandler). Other phases can make Apache->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 Apache->request available in the calls made from it:

```
package MyApache::Auth;

# PerlAuthenHandler MyApache::Auth

use Apache::RequestUtil ();
#...
sub handler {
    my $r = shift;
```

10.8.2 Apache->define

```
Apache->request($r);  
# do some calls that rely on Apache->request being available  
#...  
}
```

10.8.2 Apache->define

Apache->define has been replaced with `Apache::exists_config_define()` residing inside `Apache::ServerUtil`.

See the `Apache::ServerUtil` manpage.

10.8.3 Apache->can_stack_handlers

Apache->can_stack_handlers is no longer needed, as `mod_perl 2.0` can always stack handlers.

10.8.4 Apache->untaint

Apache->untaint has moved to `Apache::ServerUtil` 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.

`Apache::compat` provides the backward compatible with `mod_perl 1.0` implementation.

10.8.5 Apache->get_handlers

To get handlers for the server level, `mod_perl 2.0` code should use:

```
$s->get_handlers(...);
```

or:

```
Apache->server->get_handlers(...);
```

Apache->get_handlers is available via `Apache::compat`.

10.8.6 Apache->push_handlers

To push handlers at the server level, `mod_perl 2.0` code should use:

```
$s->push_handlers(...);
```

or:

```
Apache->server->push_handlers(...);
```

Apache->push_handlers is available via Apache::compat.

10.8.7 Apache->set_handlers

To set handlers at the server level, mod_perl 2.0 code should use:

```
$s->set_handlers(...);
```

or:

```
Apache->server->set_handlers(...);
```

Apache->set_handlers is available via Apache::compat.

10.8.8 Apache->httpd_conf

Apache->httpd_conf is now \$s->add_config or \$r->add_config. e.g.:

```
require Apache::ServerUtil;  
Apache->server->httpd_conf(['require valid-user']);
```

See the Apache::ServerUtil manpage.

Apache->httpd_conf is available via Apache::compat.

10.8.9 Apache::exit()

Apache::exit() has been replaced with ModPerl::Util::exit(), which is a function (not a method) and accepts a single optional argument: status, whose default is 0 (== do nothing).

See the ModPerl::Util manpage.

10.8.10 Apache::gensym()

Since Perl 5.6.1 filehandlers are autovified 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.

10.8.11 Apache::module()

Apache::module() has been replaced with the function Apache::Module::loaded(), which now accepts a single argument: the module name.

10.8.12 Apache::log_error()

`Apache::log_error()` is not available in mod_perl 2.0 API. You can use:

```
Apache->server->log_error
```

instead. See the `Apache::Log` manpage.

10.9 Apache::Variables

10.9.1 \$Apache::__T

`$Apache::__T` is deprecated in mod_perl 2.0. Use `${^TAINT}` instead.

10.10 Apache::Server::Methods and Variables

10.10.1 \$Apache::Server::CWD

`$Apache::Server::CWD` is deprecated and exists only in `Apache::compat`.

10.10.2 \$Apache::Server::AddPerlVersion

`$Apache::Server::AddPerlVersion` is deprecated and exists only in `Apache::compat`.

10.11 Server Object Methods

10.11.1 \$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. e.g.:

```
sub cleanup_callback { my $data = shift; ... }
$s->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.

10.12 Request Object Methods

10.12.1 *\$r->cgi_env*

See the next item

10.12.2 *\$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.

10.12.3 *\$r->current_callback*

`$r->current_callback` is now simply a `Apache::current_callback` and can be called for any of the phases, including those where `$r` simply doesn't exist.

`Apache::compat` implements `$r->current_callback` for backwards compatibility.

10.12.4 *\$r->get_remote_host*

`get_remote_host()` is now invoked on the *connection object*:

```
use Apache::Connection;
$r->connection->get_remote_host();
```

`$r->get_remote_host` is available through `Apache::compat`.

10.12.5 *\$r->cleanup_for_exec*

`$r->cleanup_for_exec` doesn't exist in the Apache 2.0 API, it is now being internally called by the Apache process spawning functions. For more information see `Apache::SubProcess` manpage.

There is `$pool->cleanup_for_exec`, but it's not the same as `$r->cleanup_for_exec` in the mod_perl 1.0 API.

10.12.6 *\$r->content*

See the next item.

10.12.7 *\$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 `$r->headers_in->{'content-length'}` into a single buffer (bad for performance, memory bloat, possible dos attack, etc.)
- in general duplicates functionality (and does so poorly) that is done better in `Apache::Request`.
- if one wishes to simply read POST data, there is the more modern `{setup,should,get}_client_block` API, and even more modern filter API, along with continued support for `read(STDIN, ...)` and `$r->read($buf, $r->headers_in->{'content-length'})`

For now you can use `CGI.pm` or the code in `Apache::compat` (it's slower).

META: when `Apache::Request` will be ported to `mod_perl 2.0`, you will have the fast C implementation of these functions.

10.12.8 `$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`.

10.12.9 `$r->is_main`

`$r->is_main` is not part of the `mod_perl 2.0` API. Use `!$r->main` instead.

Refer to the `Apache::RequestRec` manpage.

10.12.10 `$r->finfo`

Probably won't be implemented, because `Apache 2.0`'s `finfo` datastructure can't be mapped into the Perl `finfo` datastructure.

`Apache::compat` handles that for now with:

```
sub finfo {
    my $r = shift;
    stat $r->filename;
    \*_;
```

10.12.11 \$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.

If `Apache::compat` is loaded the old API:

```
$r->notes($key => $val);  
$val = $r->notes($key);
```

is supported as well.

See the `Apache::RequestRec` manpage.

10.12.12 \$r->header_in

See the next item.

10.12.13 \$r->header_out

See the next item.

10.12.14 \$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 `Apache::RequestRec` manpage.

10.12.15 \$r->log_reason

`$r->log_reason` is not available in mod_perl 2.0 API. Use the other standard logging functions provided by the `Apache::Log` module. For example:

```
$r->log_error("it works!");
```

See the `Apache::Log` manpage.

10.12.16 `$r->register_cleanup`

10.12.16 `$r->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.

10.12.17 `$r->post_connection`

`$r->post_connection` has been replaced with:

```
$r->connection->pool->cleanup_register();
```

See the *APR::Pool* manpage.

10.12.18 `$r->request`

Use `Apache->request`.

10.12.19 `$r->send_fd`

Apache 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:

```
$r->sendfile($fh);
```

There is also a compatibility implementation in pure perl in `Apache::compat`.

10.12.20 `$r->send_fd_length`

currently available only in the 1.0 compatibility layer. The problem is that Apache has changed the API and its functionality. See the implementation in `Apache::compat`.

XXX: needs a better resolution

10.12.21 *\$r->send_http_header*

This method is not needed in 2.0, though available in `Apache::compat`. 2.0 handlers only need to set the *Content-type* via `$r->content_type($type)`.

10.12.22 *\$r->server_root_relative*

`Apache::server_root_relative` is a function in 2.0 and its first argument is the *pool* object. For example:

```
# during request
my $conf_dir = Apache::server_root_relative($r->pool, 'conf');
# during startup
my $conf_dir = Apache::server_root_relative($s->pool, 'conf');
```

The old way:

```
my $conf_dir = Apache::server_root_relative('Apache', 'conf');
```

will work as well, but you shouldn't use it, because it'll internally use a global pool, which is a bit slower, but the worst drawback is that every time you use this way the allocated from this pool memory won't be cleaned up, till the server quits so there will be a memory leak. When you are inside a request and use `$r->pool`, this pool is freed at the end of each request.

See the `Apache::ServerUtil` manpage.

10.12.23 *\$r->hard_timeout*

See the next item.

10.12.24 *\$r->reset_timeout*

See the next item.

10.12.25 *\$r->soft_timeout*

See the next item.

10.12.26 *\$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.

10.12.27 *\$r->set_byterange*

See the next item.

10.12.28 *\$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*.

10.13 Apache::Connection

10.13.1 *\$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*.

Apache::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.

10.13.2 *\$connection->user*

This method is deprecated in mod_perl 1.0 and *\$r->user* should be used instead for both versions of mod_perl. *\$r->user()* method is available since mod_perl version 1.24_01.

10.13.3 *\$connection->local_addr*

See the next item.

10.13.4 *\$connection->remote_addr*

\$connection->local_addr and *\$connection->remote_addr* return an *APR::SocketAddr* 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 ($serverport, $serverip) = sockaddr_in($r->connection->local_addr);
my ($remoteport, $remoteip) = sockaddr_in($r->connection->remote_addr);
```

now it'll be written as:

```
require APR::SockAddr;
my $serverport = $c->local_addr->port_get;
my $serverip   = $c->local_addr->ip_get;
my $remoteport = $c->remote_addr->port_get;
my $remoteip   = $c->remote_addr->ip_get;
```

META: it's not simple to fix this in the compat layer, since it'll break the API for normal Apache 2.0 modules. Stay tuned for solutions.

10.14 Apache::File

The methods from mod_perl 1.0's module `Apache::File` have been either moved to other packages or removed.

10.14.1 *open() and close()*

The methods `open()` and `close()` were removed. See the back compatibility implementation in the module `Apache::compat`.

10.14.2 *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 `Apache::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.

10.15 Apache::Util

A few `Apache::Util` functions have changed their interface.

10.15.1 *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. See the `APR::String` manpage.

10.15.2 Apache::Util::escape_uri()

Apache::Util::escape_uri() has been replaced with Apache::Util::escape_path() and requires a pool object as a second argument. For example:

```
$escaped_path = Apache::Util::escape_path($path, $r->pool);
```

10.15.3 Apache::Util::unescape_uri()

Apache::Util::unescape_uri() has been replaced with Apache::unescape_url().

10.15.4 Apache::Util::escape_html()

Apache::Util::escape_html currently is available only via Apache::compat until *ap_escape_html* is reworked to not require a pool.

10.15.5 Apache::Util::parsedate()

Apache::Util::parsedate() has been replaced with APR::Date::parse_http().

10.15.6 Apache::Util::ht_time()

Apache::Util::ht_time() has been replaced (temporary?) with Apache::Util::format_time(), which requires a pool object as a forth argument. All four arguments are now required.

For example:

```
use Apache::Util ();
$fmt = '%a, %d %b %Y %H:%M:%S %Z';
$gmt = 1;
$fmt_time = Apache::Util::format_time(time(), $fmt, $gmt, $r->pool);
```

See the Apache::Util manpage.

10.15.7 Apache::Util::validate_password()

Apache::Util::validate_password() has been replaced with APR::password_validate(). For example:

```
my $ok = Apache::Util::password_validate("stas", "ZeO.RAc3iYvpA");
```

10.16 Apache::URI

10.16.1 Apache::URI->parse(\$r, [\$uri])

`parse()` and its associate 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.

10.16.2 unparse()

Other than moving to the `APR::URI` package, `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
```

Notice that if `Apache::compat` is loaded, `unparse()` will transparently set `scheme` to `http` to preserve the backwards compatibility with `mod_perl 1.0`.

See the `APR::URI` manpage for more information.

10.17 Miscellaneous

10.17.1 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.

10.17.2 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 Single Phase's Multiple Handlers Behavior for a complete description of each hook and its run-type.

10.18 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 Apache2;
use mod_perl 1.99;
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

10.19 Apache::Table

`Apache::Table` has been renamed to `APR::Table`.

10.20 Apache::SIG

`Apache::SIG` currently exists only `Apache::compat` and it does nothing.

10.21 Apache::StatINC

`Apache::StatINC` has been replaced by `Apache::Reload`, which works for both mod_perl generations. To migrate to `Apache::Reload` simply replace:

```
PerlInitHandler Apache::StatINC
```

with:

```
PerlInitHandler Apache::Reload
```

However `Apache::Reload` provides an extra functionality, covered in the module's manpage.

10.22 Maintainers

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

- Stas Bekman <stas (at) stason.org>

10.23 Authors

- Stas Bekman <stas (at) 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: `PerlFooHandler`, 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:MyApache/CurrentTime.pm
-----
package MyApache::CurrentTime;

use strict;
use warnings;

use Apache::RequestRec ();
use Apache::RequestIO ();

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

sub handler {
    my $r = shift;

    $r->content_type('text/plain');
    $r->print("Now is: " . scalar(localtime) . "\n");

    return Apache::OK;
}
1;
```

This handler simply returns the current date and time as a response.

Since this is a response handler, we configure it as such in *httpd.conf*:

```
PerlResponseHandler MyApache::CurrentTime
```

Since the response handler should be configured for a specific location, let's write a complete configuration section:

```
PerlModule MyApache::CurrentTime
<Location /time>
    SetHandler modperl
    PerlResponseHandler MyApache::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. The only value that can be returned by all handlers is `Apache::OK`, which tells Apache that the handler has successfully finished its execution.

`Apache::DECLINED` is another return value that indicates success, but it's only relevant for phases of type `RUN_FIRST`.

HTTP handlers may also return `Apache::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 `Apache::DONE` will cause an abort of the request cycle, by also will be interpreted as an error. Therefore you don't want to return `Apache::HTTP_OK` from your HTTP response handler, but `Apache::OK` and Apache will send the 200 OK status by itself.

Filter handlers return `Apache::OK` to indicate that the filter has successfully finished. If the return value is `Apache::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 `Apache::OK` or `Apache::DONE`, will prevent from `PerlConnectionHandler` to be run. `PerlPreConnectionHandler` handlers should always return `Apache::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 Single Phase's Multiple Handlers Behavior

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 than 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 `Apache::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 `Apache::DECLINED`. If the return value is `Apache::DECLINED`, the next handler in the chain will be run. If the return value is `Apache::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 `Apache::OK` or `Apache::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::HOOK_REALLY_FIRST`**
run this hook first, before ANYTHING.
- **`APR::HOOK_FIRST`**
run this hook first.
- **`APR::HOOK_MIDDLE`**

run this hook somewhere.

- **APR: :HOOK_LAST**

run this hook after every other hook which is defined.

- **APR: :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.

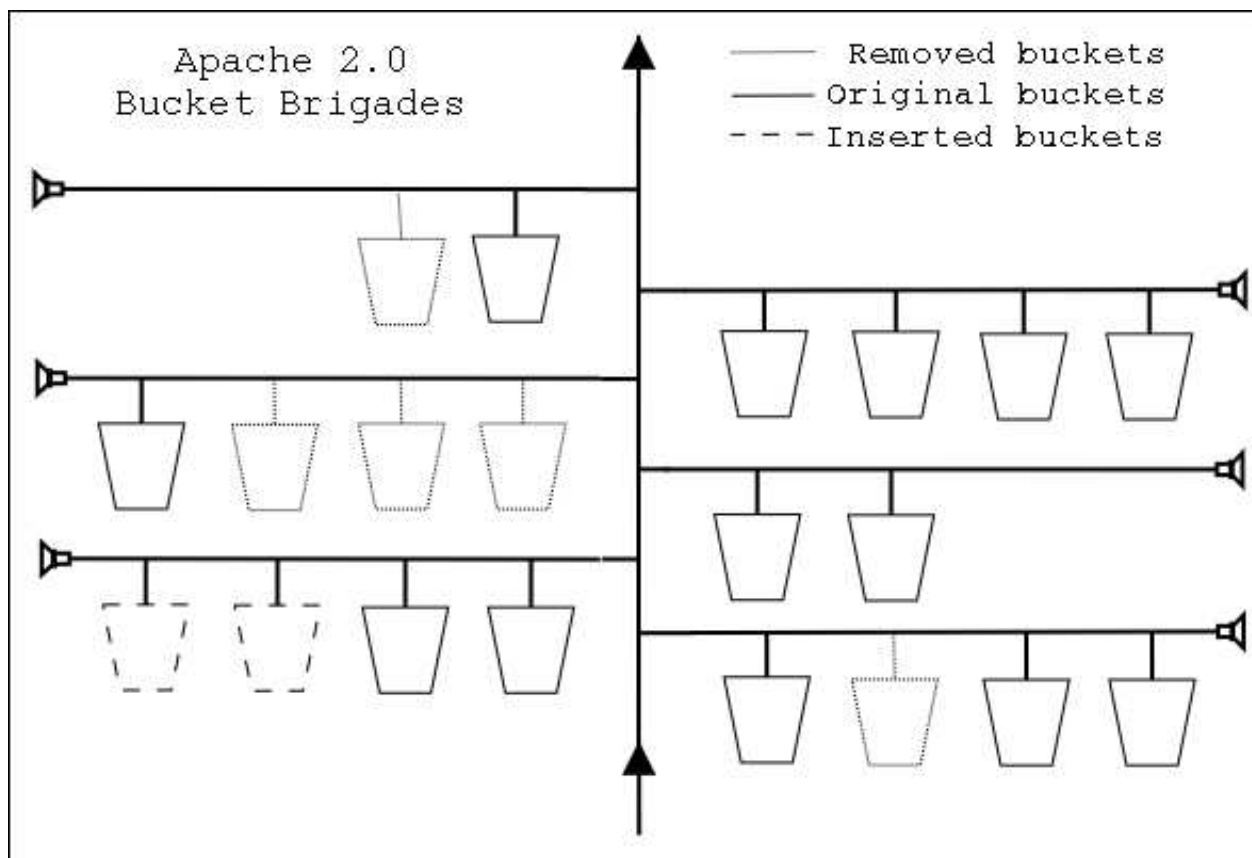
With I/O filtering in place, simple filters, like data compression and decompression, can be easily implemented and complex filters, like SSL, are now possible without needing to modify the the server code which was the case with Apache 1.3.

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:



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 <stas (at) stason.org>

11.9 Authors

-

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

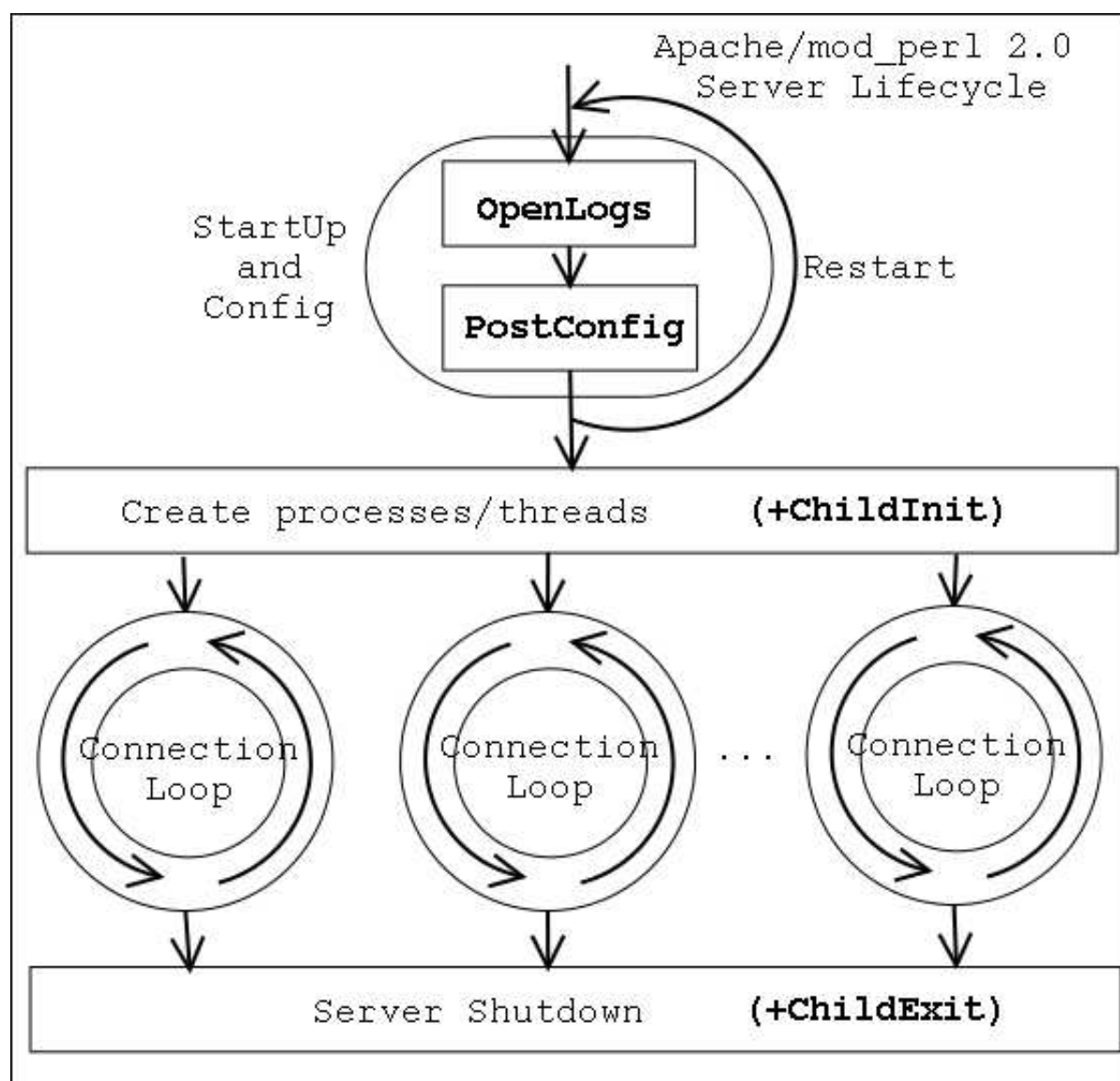
12 Server Life Cycle Handlers

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:



Apache 2.0 starts by parsing the configuration file. After the configuration file is parsed, the `PerlOpenLogsHandler` 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 and 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 `PerlChildInit` 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:MyApache/StartupLog.pm
-----
package MyApache::StartupLog;

use strict;
use warnings;

use Apache::Log ();
use Apache::ServerUtil ();

use File::Spec::Functions;

use Apache::Const -compile => 'OK';

my $log_file = catfile "logs", "startup_log";
my $log_fh;

sub open_logs {
    my($conf_pool, $log_pool, $temp_pool, $s) = @_;
    my $log_path = Apache::server_root_relative($conf_pool, $log_file);

    $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 Apache::OK;
}

sub post_config {
    my($conf_pool, $log_pool, $temp_pool, $s) = @_;
    say("configuration is completed");
    return Apache::OK;
}

sub child_init {
    my($child_pool, $s) = @_;
    say("process $$ is born to serve");
    return Apache::OK;
}
```

```

sub child_exit {
    my($child_pool, $s) = @_;
    say("process $$ now exits");
    return Apache::OK;
}

sub say {
    my($caller) = (caller(1))[3] =~ /^(:+)$/;
    if (defined $log_fh) {
        printf $log_fh "[%s] - %11s: %s\n",
            scalar(localtime), $caller, $_[0];
    }
    else {
        # when the log file is not open
        warn __PACKAGE__ . " says: $_[0]\n";
    }
}

END {
    say("process $$ is shutdown\n");
}

1;

```

And the *httpd.conf* configuration section:

```

<IfModule prefork.c>
    StartServers          4
    MinSpareServers       4
    MaxSpareServers       4
    MaxClients            10
    MaxRequestsPerChild   0
</IfModule>

PerlModule                MyApache::StartupLog
PerlOpenLogsHandler       MyApache::StartupLog::open_logs
PerlPostConfigHandler     MyApache::StartupLog::post_config
PerlChildInitHandler      MyApache::StartupLog::child_init
PerlChildExitHandler      MyApache::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 it its log information. So when we perform:

```
% bin/apachectl start && bin/apachectl stop
```

the following is getting logged to *logs/startup_log*:

```

[Thu May 29 13:11:08 2003] - open_logs   : process 21823 is born to reproduce
[Thu May 29 13:11:08 2003] - post_config: configuration is completed
[Thu May 29 13:11:09 2003] - END        : process 21823 is shutdown

[Thu May 29 13:11:10 2003] - open_logs   : process 21825 is born to reproduce
[Thu May 29 13:11:10 2003] - post_config: configuration is completed
[Thu May 29 13:11:11 2003] - child_init  : process 21830 is born to serve

```

```
[Thu May 29 13:11:11 2003] - child_init : process 21831 is born to serve
[Thu May 29 13:11:11 2003] - child_init : process 21832 is born to serve
[Thu May 29 13:11:11 2003] - child_init : process 21833 is born to serve
[Thu May 29 13:11:12 2003] - child_exit : process 21833 now exits
[Thu May 29 13:11:12 2003] - child_exit : process 21832 now exits
[Thu May 29 13:11:12 2003] - child_exit : process 21831 now exits
[Thu May 29 13:11:12 2003] - child_exit : process 21830 now exits
[Thu May 29 13:11:12 2003] - END      : process 21825 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 only.

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 `MyApache::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`.

As we have seen in the `MyApache::StartupLog::open_logs` handler, the *open_logs* phase handlers accept four arguments: the configuration pool, the logging stream pool, the temporary pool and the server object:

```

sub open_logs {
    my($conf_pool, $log_pool, $temp_pool, $s) = @_;
    my $log_path = Apache::server_root_relative($conf_pool, $log_file);

    $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 Apache::OK;
}

```

In our example the handler uses the function `Apache::server_root_relative()` to set the full path to the log file, which is then opened for appending and set to unbuffered mode. Finally it 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 *httpd.conf*:

```
PerlOpenLogsHandler MyApache::StartupLog::open_logs
```

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 `Apache::Directive`).

This phase is of type `RUN_ALL`.

The handler's configuration scope is `SRV`.

In our `MyApache::StartupLog` example we used the *post_config()* handler:

```

sub post_config {
    my($conf_pool, $log_pool, $temp_pool, $s) = @_;
    say("configuration is completed");
    return Apache::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 MyApache::StartupLog::post_config
```

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 `Apache::Resource` sets the process' resources limits.

This phase is of type VOID.

The handler's configuration scope is SRV.

In our `MyApache::StartupLog` example we used the *child_init()* handler:

```
sub child_init {
    my($child_pool, $s) = @_;
    say("process $$ is born to serve");
    return Apache::OK;
}
```

The *child_init()* handler accepts two arguments: the child process pool and the server object. 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 MyApache::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.

In our `MyApache::StartupLog` example we used the *child_exit()* handler:

```
sub child_exit {
    my($child_pool, $s) = @_;
    say("process $$ now exits");
    return Apache::OK;
}
```

The *child_exit()* handler accepts two arguments: the child process pool and the server object. 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    MyApache::StartupLog::child_exit
```

12.3 Maintainers

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

- Stas Bekman <stas (at) stason.org>

12.4 Authors

-

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

13 Protocol Handlers

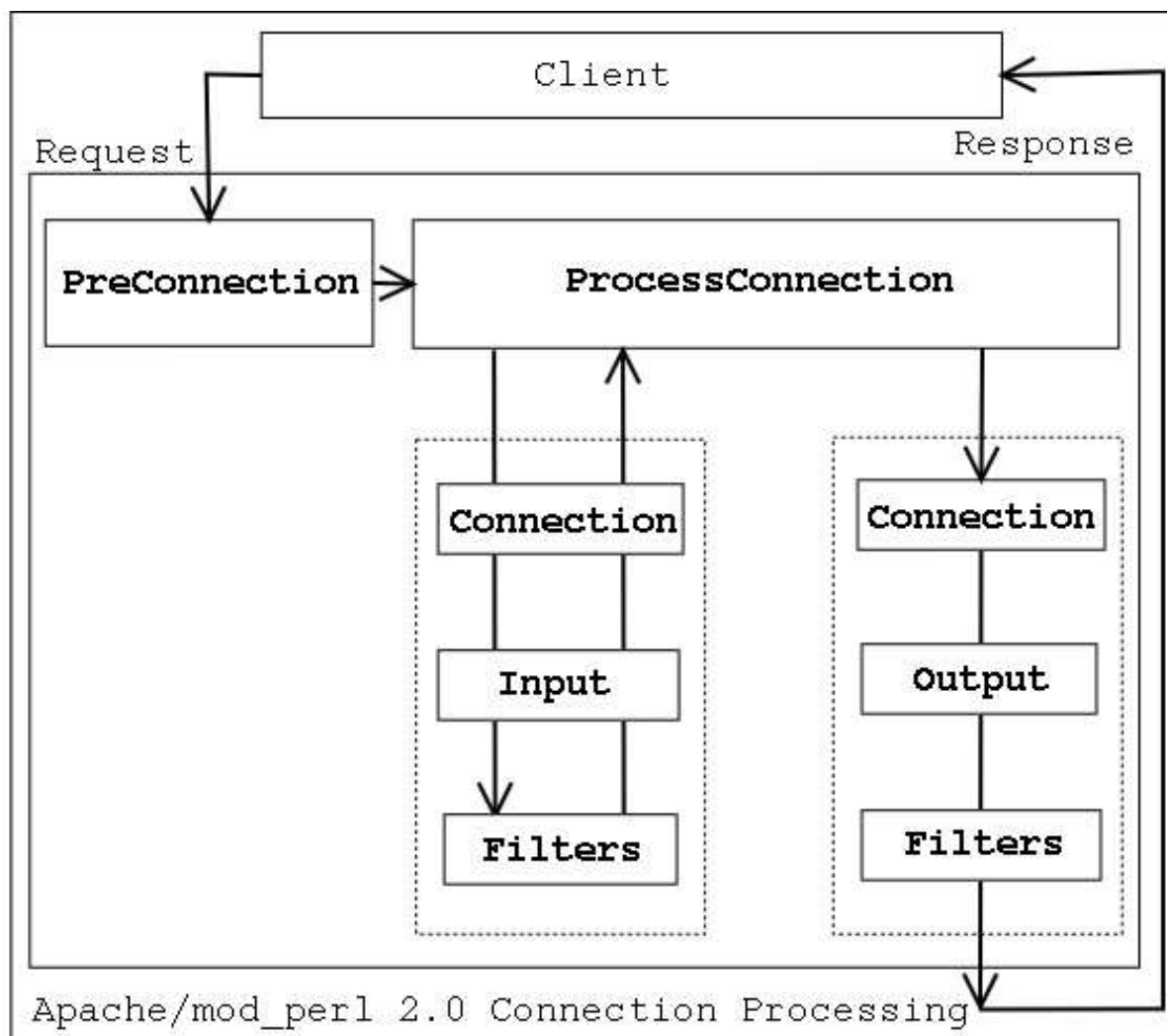
13.1 Description

This chapter explains how to implement Protocol (Connection) Handlers in mod_perl.

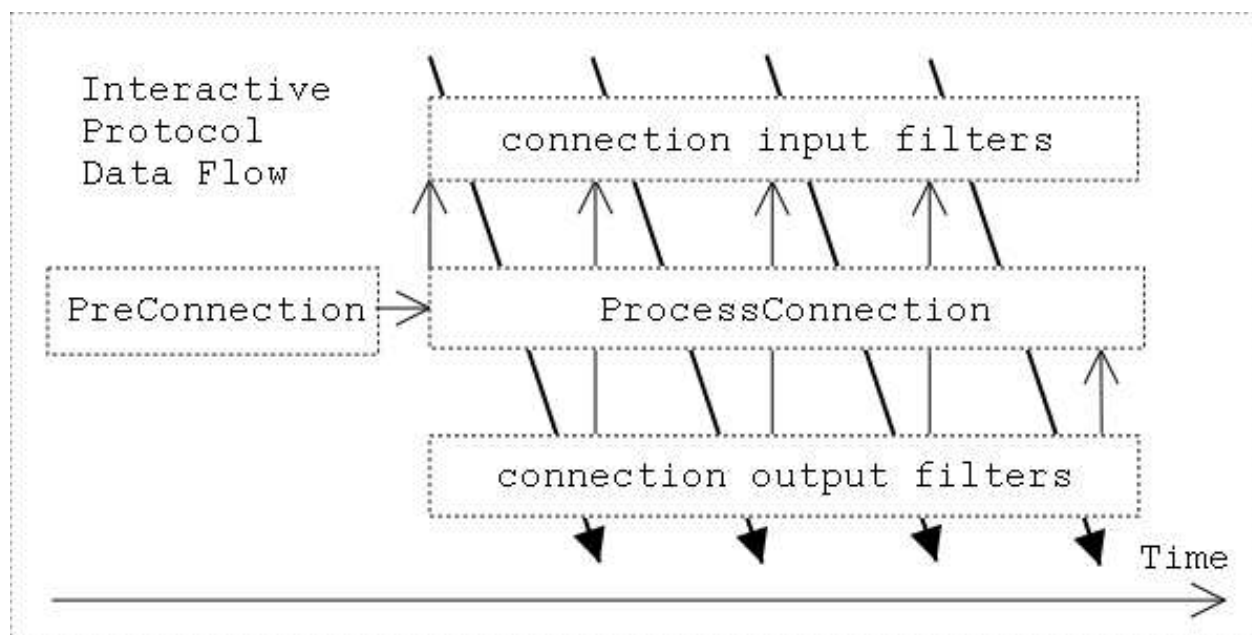
13.2 Connection Cycle Phases

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:



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 through connection output filters. Filters are usually used for modifying the data flowing through 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:



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 `PerlProcessConnectionHandler` 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 `Apache::Reload` handler during the *pre_connection* phase. However this development-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 connection 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.

A *pre_connection* handler accepts a connection record at its argument:

```
sub handler {
    my $c = shift;
    # ...
    return Apache::OK;
}
```

[META: There is another argument passed (the actual client socket), but it currently an undef]

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.

`MyApache::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:MyApache/BlockIP2.pm
#-----
package MyApache::BlockIP2;

use strict;
use warnings;

use Apache::Connection ();

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

my %bad_ips = map {$_ => 1} qw(127.0.0.1 10.0.0.4);

sub handler {
    my Apache::Connection $c = shift;

    my $ip = $c->remote_ip;
    if (exists $bad_ips{$ip}) {
        warn "IP $ip is blocked\n";
        return Apache::FORBIDDEN;
    }
}
```

```

    }

    return Apache::OK;
}

1;
```

This all happens during the *pre_connection* phase:

```
PerlPreConnectionHandler MyApache::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.

A *process_connection* handler accepts a connection record object as its only argument, a socket object can be retrieved from the connection record object.

```

sub handler {
    my ($c) = @_;
    my $socket = $c->client_socket;
    # ...
    return Apache::OK;
}
```

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 `MyApache::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                               MyApache::EchoSocket
    PerlProcessConnectionHandler MyApache::EchoSocket
</VirtualHost>

```

`MyApache::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:MyApache/EchoSocket.pm
-----
package MyApache::EchoSocket;

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

use Apache::Connection ();
use APR::Socket ();

use Apache::Const -compile => 'OK';

use constant BUFF_LEN => 1024;

sub handler {
    my $c = shift;
    my $socket = $c->client_socket;

    my $buff;
    while (1) {
        my $rlen = BUFF_LEN;
        $socket->recv($buff, $rlen);

        last if $rlen <= 0 or $buff =~ /\r\n+$/;

        my $wlen = $rlen;
        $socket->send($buff, $wlen);

        last if $wlen != $rlen;
    }
}

```

```

        Apache::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 `Apache::Connection` class. We have direct access to the client socket via `Apache::Connection's client_socket` method. This returns an object blessed into the `APR::Socket` class.

Inside the read/send loop, the handler attempts to read `BUFF_LEN` bytes from the client socket into the `$buff` buffer. The `$rlen` parameter will be set to the number of bytes actually read. The `APR::Socket::recv()` method returns an APR status value, but we need only to check the read length to break out of the loop if it is less than or equal to 0 bytes. The handler also breaks the loop after processing an input including nothing but new lines characters, which is how we 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 `Apache::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 lower-case equivalents.

The following configuration defines a virtual host listening on port 8011 and which enables the `MyApache::EchoBB` connection handler, which will run its output through `MyApache::EchoBB::lowercase_filter` filter:

```

Listen 8011
<VirtualHost _default_:8011>
    PerlModule                MyApache::EchoBB
    PerlProcessConnectionHandler MyApache::EchoBB
    PerlOutputFilterHandler    MyApache::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 now was all in lower case, because of the output filter.

And here is the implementation of the connection and the filter handlers.

```
file:MyApache/EchoBB.pm
-----
package MyApache::EchoBB;

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

use Apache::Connection ();
use APR::Bucket ();
use APR::Brigade ();
use APR::Util ();

use APR::Const -compile => qw(SUCCESS EOF);
use Apache::Const -compile => qw(OK MODE_GETLINE);

sub handler {
    my $c = shift;

    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 $rv = $c->input_filters->get_brigade($bb_in, Apache::MODE_GETLINE);

        if ($rv != APR::SUCCESS or $bb_in->empty) {
            my $error = APR::strerror($rv);
            unless ($rv == APR::EOF) {
                warn "get_brigade: $error\n";
            }
            $bb_in->destroy;
            last;
        }

        while (!$bb_in->empty) {
            my $bucket = $bb_in->first;

            $bucket->remove;

            if ($bucket->is_eos) {
```

```

        $bb_out->insert_tail($bucket);
        last;
    }

    my $data;
    my $status = $bucket->read($data);
    return $status unless $status == APR::SUCCESS;

    if ($data) {
        $last++ if $data =~ /^[\r\n]+$/;
        # could do something with the data here
        $bucket = APR::Bucket->new($data);
    }

    $bb_out->insert_tail($bucket);
}

my $b = APR::Bucket::flush_create($c->bucket_alloc);
$bb_out->insert_tail($b);
$c->output_filters->pass_brigade($bb_out);
last if $last;
}

Apache::OK;
}

use base qw(Apache::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 Apache::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 ($bb_in = get_brigade()) {
    while ($bucket_in = $bb_in->get_bucket()) {
        my $data = $bucket_in->read();
        # do something with data
        $bucket_out = new_bucket($data);

        $bb_out->insert_tail($bucket_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.

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 bucket has been seen (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.

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 sections. 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.

13.3 Maintainers

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

- Stas Bekman <stas (at) stason.org>

13.4 Authors

-

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

14 HTTP Handlers

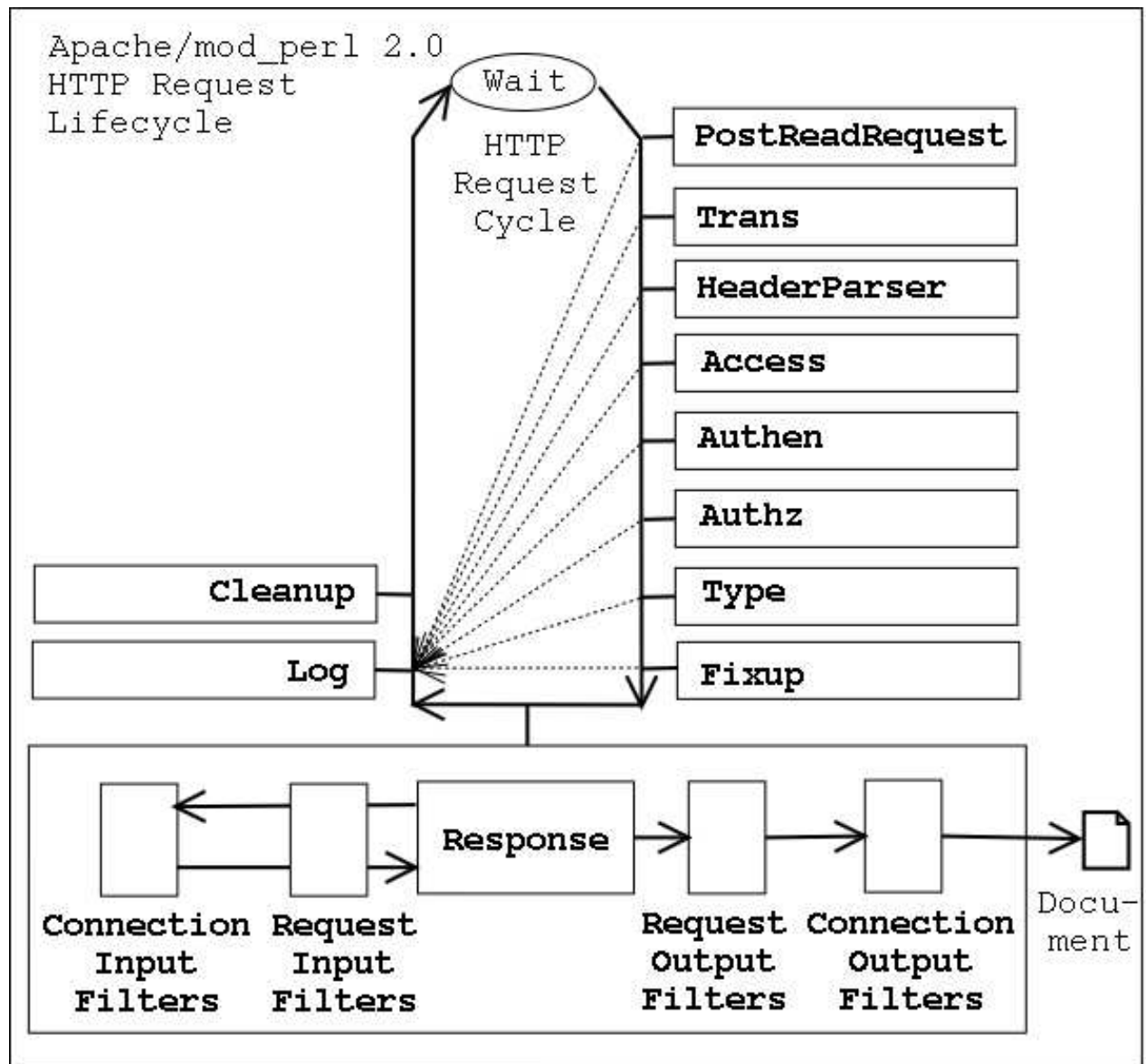
14.1 Description

This chapter explains how to implement the HTTP protocol handlers in `mod_perl`.

14.2 HTTP Request 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 only difference is in the *response* phase which now includes filtering. Also the `PerlHandler` directive has been renamed to `PerlResponseHandler` to better match the corresponding Apache phase name (*response*).

The following diagram depicts the HTTP request life cycle and highlights which handlers are available to `mod_perl` 2.0:



From the diagram it can be seen that an HTTP request is processed by 11 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 this chapter.

Before discussing each handler in detail remember that if you use stacked handlers feature (META: add link to where it's discussed [go read 1.0 docs for now, as it works the same]) all handlers in the chain will be run as long as they return `Apache::OK` or `Apache::DECLINED`. Because stacked handlers is a special case. So don't be surprised if you've returned `Apache::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.

14.2.1 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 `Apache::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.

Now, let's look at an example. Consider the following registry script:

```
touch.pl
-----
use strict;
use warnings;

use Apache::ServerUtil ();
use File::Spec::Functions qw(catfile);

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

```
my $conf_file = catfile Apache::server_root_relative($r->pool, 'conf'),
    "httpd.conf";

printf "$conf_file is %0.2f minutes old", 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 `PerlPostReadRequestHandler` handler, which will be executed as the very first thing of each requests, comes handy here:

```
file:MyApache/TimeReset.pm
-----
package MyApache::TimeReset;

use strict;
use warnings;

use Apache::RequestRec ();

use Apache::Const -compile => 'OK';

sub handler {
    my $r = shift;
    ^T = $r->request_time;
    return Apache::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 MyApache::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.

14.2.2 *PerlTransHandler*

The *translate* phase is used to perform the translation of a request's URI into an corresponding filename. If no custom handler is provided, the server's standard translation rules (e.g., `Alias` directives, `mod_rewrite`, etc.) will continue to be used. A `PerlTransHandler` handler can alter the default translation mechanism or completely override it.

In addition to doing the translation, this stage can be used to modify the URI itself and the request method. This is also a good place to register new handlers for the following phases based on the URI.

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.

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:MyApache/RewriteURI.pm
-----
package MyApache::RewriteURI;

use strict;
use warnings;

use Apache::RequestRec ();

use Apache::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 Apache::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 `Apache::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 +MyApache::RewriteURI
```

14.2.3 PerlMapToStorageHandler META: add something here

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.

14.2.4 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`.

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 MyApache::SendEmail
</Location>
```

and here is the `MyApache::SendEmail` handler:

```
file:MyApache/SendEmail.pm
-----
package MyApache::SendEmail;

use strict;
use warnings;

use Apache::RequestRec ();
use Apache::RequestIO ();
use Apache::RequestUtil ();

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

use constant METHOD          => 'EMAIL';
use constant SMTP_HOSTNAME => "localhost";

sub handler {
    my $r = shift;

    return Apache::DECLINED unless $r->method eq METHOD;

    Apache::method_register($r->pool, METHOD);
    $r->handler("perl-script");
    $r->push_handlers(PerlResponseHandler => \&send_email_handler);

    return Apache::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 Apache::OK;
}

sub content {
    my $r = shift;

    $r->setup_client_block;
    return '' unless $r->should_client_block;
    my $len = $r->headers_in->get('content-length');
    my $buf;
    $r->get_client_block($buf, $len);
}
```

```

        return $buf;
    }

    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;
    }

    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 Apache::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. Finally it pushes the function `send_email_handler()` to the `PerlResponseHandler` list of handlers:

```

Apache::method_register($r->pool, METHOD);
$r->handler("perl-script");
$r->push_handlers(PerlResponseHandler => \&send_email_handler);

```

The function terminates the `header_parser` phase by:

```
return Apache::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 `Apache::OK`:

```
$r->content_type('text/plain');
$r->print($status ? "ACK" : "NACK");
return Apache::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.

14.2.5 *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 `PerlPostReadRequestHandler` described earlier.

It is the first handler to be invoked when serving a request.

This phase is of type `RUN_ALL`.

The best example here would be to use `Apache::Reload` which takes the benefit of this directive. Usually `Apache::Reload` is configured as:

```
PerlInitHandler Apache::Reload
PerlSetVar ReloadAll Off
PerlSetVar ReloadModules "MyApache::"
```

which during the current HTTP request will monitor and reload all `MyApache::*` modules that have been modified since the last HTTP request. However if we move the global configuration into a `<Location>` container:

```
<Location /devel>
    PerlInitHandler Apache::Reload
    PerlSetVar ReloadAll Off
    PerlSetVar ReloadModules "MyApache::"
    SetHandler perl-script
    PerlResponseHandler ModPerl::Registry
    Options +ExecCGI
</Location>
```

`Apache::Reload` will reload the modified modules, only when a request to the `/devel` namespace is issued, because `PerlInitHandler` plays the role of `PerlHeaderParserHandler` here.

14.2.6 *PerlAccessHandler*

The *access_checker* phase is the first of three handlers that are involved in what's known as AAA: Authentication and 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`.

The concept behind access checker handler is very simple, return `Apache::FORBIDDEN` if the access is not allowed, otherwise return `Apache::OK`.

The following example handler denies requests made from IPs on the blacklist.

```
file:MyApache/BlockByIP.pm
-----
package MyApache::BlockByIP;

use strict;
use warnings;

use Apache::RequestRec ();
use Apache::Connection ();

use Apache::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}
        ? Apache::FORBIDDEN
        : Apache::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 MyApache::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 MyApache::BlockByIP
</Location>
```

14.2.7 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 `Apache::OK`. Otherwise the handler returns `Apache::HTTP_UNAUTHORIZED` to indicate that the user has not authenticated successfully. 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`.

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:MyApache/SecretLengthAuth.pm
-----
package MyApache::SecretLengthAuth;

use strict;
use warnings;

use Apache::Access ();
use Apache::RequestUtil ();

use Apache::Const -compile => qw(OK DECLINED HTTP_UNAUTHORIZED);

use Apache::Access();

use constant SECRET_LENGTH => 14;

sub handler {
    my $r = shift;

    my ($status, $password) = $r->get_basic_auth_pw;
    return $status unless $status == Apache::OK;

    return Apache::OK
        if SECRET_LENGTH == length join " ", $r->user, $password;

    $r->note_basic_auth_failure;
    return Apache::HTTP_UNAUTHORIZED;
}

1;
```

First the handler retrieves the status of the authentication and the password in plain text. The status will be set to `Apache::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, it 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 `Apache::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 MyApache::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 MyApache::SecretLengthAuth
    AuthType Basic
    AuthName "The Gate"
    Require valid-user
</Location>
```

14.2.8 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 `Apache::DECLINED` to defer the decision, `Apache::OK` to indicate its acceptance of the user's authorization, or `Apache::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`.

Here is the `MyApache::SecretResourceAuthz` handler which grants access to certain resources only to certain users who have already properly authenticated:

```
file:MyApache/SecretResourceAuthz.pm
-----
package MyApache::SecretResourceAuthz;

use strict;
use warnings;

use Apache::Access ();
use Apache::RequestUtil ();

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

use Apache::Access ();

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 Apache::OK if grep { $_ eq $user } @$users;
    }
    else {
        return Apache::OK;
    }
}

$r->note_basic_auth_failure;
return Apache::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 Apache::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 MyApache::SecretLengthAuth
    PerlAuthzHandler MyApache::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 MyApache::SecretLengthAuth
    PerlAuthzHandler MyApache::SecretResourceAuthz
    AuthType Basic
    AuthName "The Secret Gate"
    Require valid-user
</Location>

```

14.2.9 *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`.

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 `Apache::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.

14.2.10 *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`.

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:MyApache/FileExtDispatch.pm
-----
package MyApache::FileExtDispatch;

use strict;
use warnings;

use Apache::RequestIO ();
use Apache::RequestRec ();

use Apache::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       ],
);

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 Apache::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 Apache::OK;
}

1;

```

In the example we have used the following mapping.

```
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:

```
$r->handler($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 MyApache::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.

14.2.11 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 MyApache::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.

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:MyApache/Deparse.pm
-----
package MyApache::Deparse;

use strict;
use warnings;

use Apache::RequestRec ();
use Apache::RequestIO ();
use B::Deparse ();

use Apache::Const -compile => 'OK';

sub handler {
    my $r = shift;

    $r->content_type('text/plain');
    $r->print('sub handler ', B::Deparse->new->coderef2text(\&handler));

    return Apache::OK;
}
1;
```

To enable this handler add to *httpd.conf*:

```
<Location /deparse>
    SetHandler modperl
    PerlResponseHandler MyApache::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 MyApache::Deparse;
    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!

14.2.12 *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.

Imagine a situation where you have to log requests into individual files, one per user. Assuming that all requests start with */users/username/*, so it's easy to categorize requests by the second URI path component. Here is the log handler that does that:

```
file:MyApache/LogPerUser.pm
-----
package MyApache::LogPerUser;

use strict;
use warnings;

use Apache::RequestRec ();
use Apache::Connection ();
use Fcntl qw(:flock);

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

sub handler {
    my $r = shift;

    my($username) = $r->uri =~ m|^/users/([^/]+)|;
    return Apache::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 $log_path = Apache::server_root_relative($r->pool,
        "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 Apache::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 `Apache::DECLINED`, letting other log handlers to do the logging. Though it could return `Apache::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, inside the wanted section, which was */users/* in our example:

```
<Location /users/>
    SetHandler perl-script
    PerlResponseHandler ModPerl::Registry
    PerlLogHandler MyApache::LogPerUser
    Options +ExecCGI
</Location>
```

After restarting the server and issuing requests to the following URIs:

```
http://localhost/users/stas/test.pl
http://localhost/users/eric/test.pl
http://localhost/users/stas/date.pl
```

The `MyApache::LogPerUser` handler will append to *logs/stas.log*:

```
127.0.0.1 [Sat Aug 31 01:50:38 2002] "/users/stas/test.pl" 200 8
127.0.0.1 [Sat Aug 31 01:50:40 2002] "/users/stas/date.pl" 200 44
```

and to *logs/eric.log*:

```
127.0.0.1 [Sat Aug 31 01:50:39 2002] "/users/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 MyApache::LogPerUser
</Location>
```

Since the `PerlLogHandler` phase is of type `RUN_ALL`, all other logging handlers will be called as well.

14.2.13 *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`.

There are two ways to register and run cleanup handlers:

1. Using the `PerlCleanupHandler` phase

```
PerlCleanupHandler MyApache::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 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.

```
#file:MyApache/Cleanup1.pm
#-----
package MyApache::Cleanup1;

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



```

use File::Spec::Functions qw(catfile);

use Apache::RequestRec ();
use Apache::RequestIO ();
use Apache::RequestUtil ();

use Apache::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::SUCCESS;

    $r->push_handlers(PerlCleanupHandler => \&cleanup);

    return Apache::OK;
}

sub cleanup {
    my $r = shift;

    die "Can't find file: $file" unless -e $file;
    unlink $file or die "failed to unlink $file";

    return Apache::OK;
}
1;

```

Next we add the following configuration:

```

<Location /cleanup1>
    SetHandler modperl
    PerlResponseHandler MyApache::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 Apache::MPM;
    require APR::OS;
    return Apache::MPM->is_threaded
        ? "$$. " . ${ APR::OS::thread_current() }
        : $$;
}

```

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 use 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:MyApache/Cleanup2.pm
#-----
package MyApache::Cleanup2;

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

use File::Spec::Functions qw(catfile);

use Apache::RequestRec ();
use Apache::RequestIO ();
use Apache::RequestUtil ();
use APR::UUID ();
use APR::Pool ();

use Apache::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::SUCCESS;

    $r->pool->cleanup_register(&cleanup, $file);

    return Apache::OK;
}

sub cleanup {
    my $file = shift;

    die "Can't find file: $file" unless -e $file;
    unlink $file or die "failed to unlink $file";

    return Apache::OK;
}
1;

```

Similarly to the first handler, we add the configuration:

```

<Location /cleanup2>
    SetHandler modperl
    PerlResponseHandler MyApache::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 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 OK if $r->header_only;

```

This logic is no longer needed in mod_perl 2.0, because Apache 2.0 automatically discards the response body for HEAD requests. (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 Extending HTTP Protocol

Extending HTTP under mod_perl is a trivial task. Look at the example of adding a new method EMAIL for details.

14.5 Maintainers

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

- Stas Bekman <stas (at) stason.org>

14.6 Authors

-

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 `Apache::Filter` and `Apache::FilterRec` manpages.

15.2 Your First Filter

You certainly already know how filters work. That's 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:



If you are smoker, chances are that you smoke cigarettes with filters:



If you are a coffee gourmand, you have certainly tried a filter coffee:



The shower that you use, may have a water filter:



When the sun is too bright, you protect your eyes by wearing sun goggles with UV filter:



If are a photographer you can't go a step without using filter lenses:



If you love music, you might be unaware of it, but your super-modern audio system is literally loaded with various electronic filters:



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:MyApache/FilterObfuscate.pm
#-----
package MyApache::FilterObfuscate;

use strict;
use warnings;
```



```

use Apache::Filter ();
use Apache::RequestRec ();
use APR::Table ();

use Apache::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 Apache::OK;
}
1;

```

Next we configure Apache to apply the `MyApache::FilterObfuscate` filter to all requests that get mapped to files with an `".html"` extension:

```

<Files ~ "\.html">
    PerlOutputFilterHandler MyApache::FilterObfuscate
</Files>

```

Filter handlers are similar to HTTP handlers, they are expected to return `Apache::OK` or `Apache::DECLINED`, but instead of receiving `$r` (the request object) as the first argument, they receive `$f` (the filter object).

The filter starts by unsetting of the `Content-Length` response header, because it modifies the length of the response body (shrinks it). If the response handler had set the `Content-Length` header and the filter hasn't unset it, the client may have problems receiving the response since it'd expect more data than it was sent.

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 occurrences 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 by itself uses several core filters to manipulate the data and send it out to the client.

As we are going to explain in great detail in the next sections, the same filter may be called many times during a single request, every time receiving a 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 response phase, where an upstream filter may split 64k 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. Of course 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. We will show plenty of examples using this method in the following sections.

Of course the `MyApache::FilterObfuscate` filter logic should take into account situations where removing new line characters will break the correct rendering, as is the case if there are multi-line `<pre>...</pre>` entries, but since it escalates the complexity of the filter, we will disregard this requirement for now.

A positive side effect of this obfuscation algorithm is in shortening the amount of the data sent to the client. If you want to look at the production ready implementation, which takes into account the HTML markup specifics, the `Apache::Clean` module, available from CPAN, does just that.

`mod_perl` I/O filtering follows the Perl's 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 tutorial you will see that much more difficult things are possible, even though a more elaborated code will be needed.

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*

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 *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.

Connection level filters are applied 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 connection (e.g. IMAP), the data could have a completely different pattern, than the HTTP protocol (headers + body).

Apache supports several other filter types, which mod_perl 2.0 may support 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 if a content generation handler sends a string, and then forces a flush, following by more data:

```
# assuming buffered STDOUT ($|=0)
$r->print("foo");
$r->rflush;
$r->print("bar");
```

Apache will generate one bucket brigade with two buckets (there are several types of buckets which contain data, one of them is *transient*):

bucket	type	data
1st	transient	foo
2nd	flush	

and send it to the filter chain. Then assuming that no more data was sent after `print("bar")`, it will create a last bucket brigade containing data:

```
bucket type      data
-----
1st    transient  bar
```

and send it to the filter chain. Finally it'll send yet another bucket brigade with the EOS bucket indicating that there will be no more data sent:

```
bucket type      data
-----
1st      eos
```

Notice that the EOS bucket may come attached to the last bucket brigade with data, instead of coming in its own bucket brigade. Filters should never make an assumption 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.

A user may install an upstream filter, and that filter may decide to insert extra bucket brigades or collect all the data in all bucket brigades passing through it and send it all down in one brigade. What's important to remember is when coding a filter, one should never assume that the filter is always going to be invoked once, or a fixed number of times. Neither one can make assumptions on the way the data is going to come in. Therefore a typical filter handler may need to split its logic in three parts.

Jumping ahead we will show some pseudo-code that represents all three parts. This is how a typical stream-oriented filter handler looks like:

```
sub handler {
  my $f = shift;

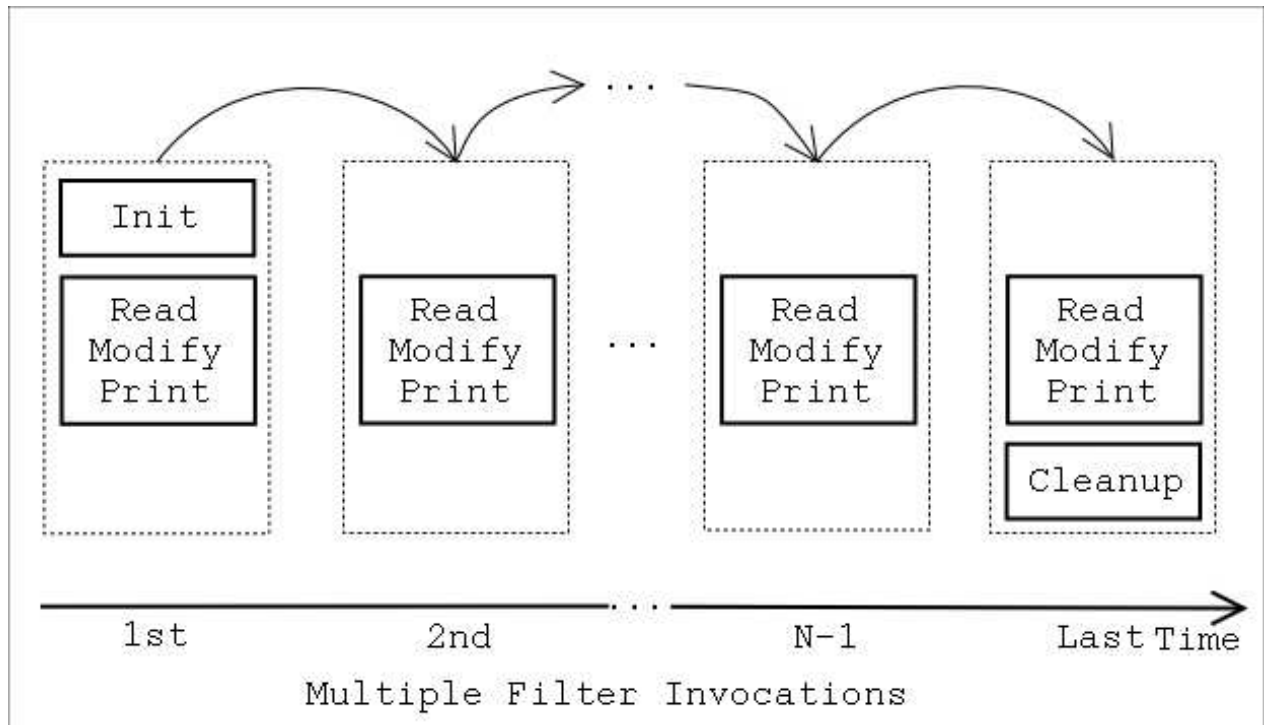
  # runs on first invocation
  unless ($f->ctx) {
    init($f);
    $f->ctx(1);
  }

  # runs on all invocations
  process($f);

  # runs on the last invocation
  if ($f->seen_eos) {
    finalize($f);
  }

  return Apache::OK;
}
sub init      { ... }
sub process   { ... }
sub finalize  { ... }
```

The following diagram depicts all three parts:



Let's explain each part using this pseudo-filter.

1. Initialization

During the initialization, the filter runs all the code that should be performed only once across multiple invocations of the filter (this is during a single request). The filter context is used to accomplish that task. For each new request the filter context is created before the filter is called for the first time and its 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 datastructure to its 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 served as a flag, but used to store real data. 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 Apache::DECLINED;
}
```

Since this filter handler doesn't consume the data from the upstream filter, it's important that this handler returns `Apache::DECLINED`, in which case `mod_perl` passes the current bucket brigade to the next filter. If this handler returns `Apache::OK`, the data will be simply lost. And if that data included a special EOS token, this may wreck havoc.

Unsetting the `Content-Length` header for filters that modify the response body length is a good example of the code to be used in the initialization phase:

```
unless ($f->ctx) {
    $f->r->headers_out->unset('Content-Length');
    $f->ctx(1);
}
```

We will see more of 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 separately the logic is really simple.

In more complicated filters the filters may need to buffer data first before the transformation can be applied. For example if the filter operates on html tokens (e.g., ''), it's possible that one brigade will include the beginning of the token ('') will come in the next bucket brigade (on the next filter invocation). In certain cases it may involve more than two bucket brigades to get the whole token. In such a case the filter will have to store the remainder of unprocessed data in the filter context and then reuse it on the next invocation. Another good example is a filter that performs data compression (compression is usually effective only when applied to relatively big chunks of data), so if a single bucket brigade doesn't contain enough data, the filter may need to buffer the data in the filter context till it collects enough of it.

We will see the implementation examples in this chapter.

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 whether this is the last time it's invoked, 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 sometimes the EOS "token" comes attached to the tail of data (the last invocation gets both the data and EOS) and sometimes it comes all alone (the last invocation gets only EOS). So 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 filter won't function properly.

Jumping ahead, filters, directly manipulating bucket brigades, have to look for a bucket whose type is EOS to accomplish this. We will see examples later in the chapter.

Some filters may need to deploy all three parts of the described logic, others will need to do only initialization and processing, or processing and finalization, while the simplest filters might perform only the normal processing (as we saw in the example of the filter handler that lowers the case of the characters going through it).

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.

First of all, the input and output filters differ in the ways they acquire the bucket brigades (which includes the data that they filter). Even though when a streaming API is used the difference can't be seen, it's important to understand how things work underneath. Therefore we are going to show examples of transparent filters, which pass data through them unmodified. Instead of reading the data in and printing it out the bucket brigades are now passed as is.

Here is a code for a transparent input filter:

```
#file:MyApache/FilterTransparent.pm (first part)
#-----
package MyApache::FilterTransparent;

use Apache::Const -compile => qw(OK);
use APR::Const -compile => ':common';

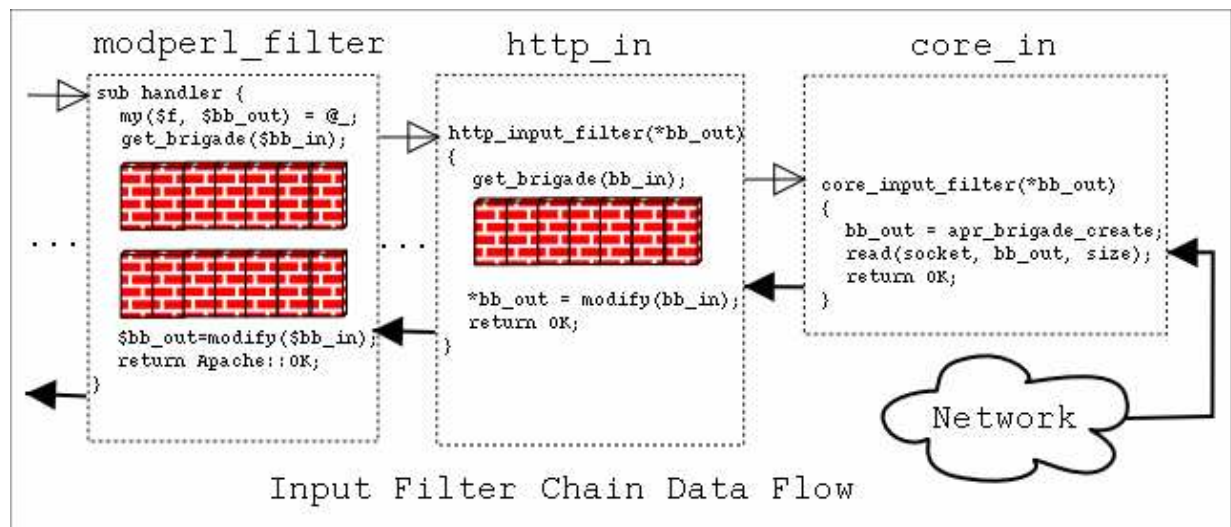
sub in {
    my ($f, $bb, $mode, $block, $readbytes) = @_;

    my $rv = $f->next->get_brigade($bb, $mode, $block, $readbytes);
    return $rv unless $rv == APR::SUCCESS;

    return Apache::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 in chain, etc., till the last filter (called *core_in*), 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, which will process the data and send it to its downstream filter, etc., till it reaches the very first filter who has asked for the data. (In reality some other handler triggers the request for the bucket brigade, e.g., an HTTP response handler, or a protocol module, but for our discussion it's good enough to assume that it's the first filter that issues the *get_brigade()* call.)

The following diagram depicts a typical input filters chain data flow in addition to the program control flow.



The black- and white-headed arrows show when the control is switched from one filter to another. In addition the black-headed arrows show the actual data flow. The diagram includes some pseudo-code, both for 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 finishing their filtering task.

As mentioned earlier, the streaming interface hides these details, however the first `$f->read()` call will block, as underneath it performs the `get_brigade()` call.

The diagram shows a part of the actual input filter chain for an HTTP request, the `...` shows 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 then may apply 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., 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.

Here is an example of a transparent output filter:

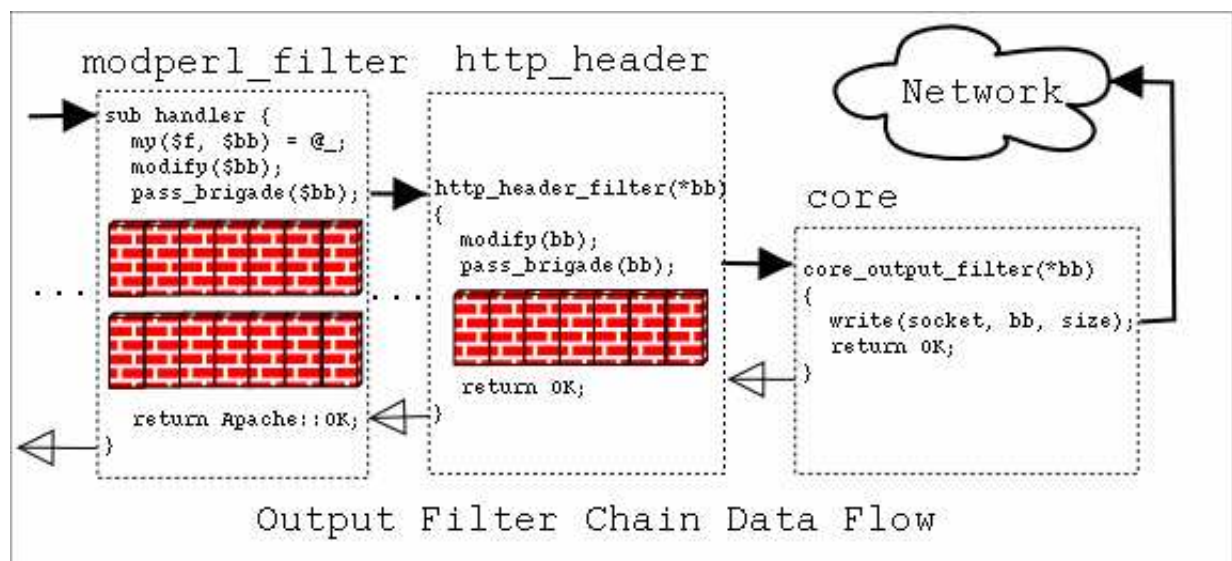
```
#file:MyApache/FilterTransparent.pm (continued)
#-----
sub out {
    my ($f, $bb) = @_;

    my $rv = $f->next->pass_brigade($bb);
    return $rv unless $rv == APR::SUCCESS;

    return Apache::OK;
}
1;
```

The *out()* filter passes *\$bb* to the downstream filter unmodified and if you add debug prints before and after the *pass_brigade()* call and configure the same filter twice, the debug print will show the blocking call.

The following diagram depicts a typical output filters chain data flow in addition to the program control 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 a Perl pseudo-code for the mod_perl filter and C pseudo-code for the Apache filters, similarly the brick walls represent the waiting. And again, the diagram shows a part of the real HTTP response filters chain, where `...` stands for the omitted filters.

15.4 mod_perl Filters Declaration and Configuration

Now let's see 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, the filter priority type should be consulted. Unfortunately this information is available only by consulting the source code, unless it's documented in the module man pages. Numerical definitions of priority types, such as `AP_FTYPE_CONTENT_SET`, `AP_FTYPE_RESOURCE`, can be found in `include/util_filter.h`.

As of this writing Apache comes with two core filters: `DEFLATE` and `INCLUDES`. For example in the following configuration:

```
SetOutputFilter DEFLATE
SetOutputFilter INCLUDES
```

the `DEFLATE` filter will be inserted in the filters chain after the `INCLUDES` filter, even though it was configured before 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:

Handler	Priority	Value
-----	-----	-----
<code>FilterRequestHandler</code>	<code>AP_FTYPE_RESOURCE</code>	10
<code>FilterConnectionHandler</code>	<code>AP_FTYPE_PROTOCOL</code>	30

Therefore `FilterRequestHandler` filters (10) will be always invoked before the `DEFLATE` filter (20), whereas `FilterConnectionHandler` filters (30) after it. The `INCLUDES` filter (10) has the same priority as `FilterRequestHandler` filters (10), and therefore it'll be inserted according to the configuration order, when `PerlSetOutputFilter` or `PerlSetInputFilter` is 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`.

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.

The following sections include several examples that use the `PerlOutputFilterHandler` handler.

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).

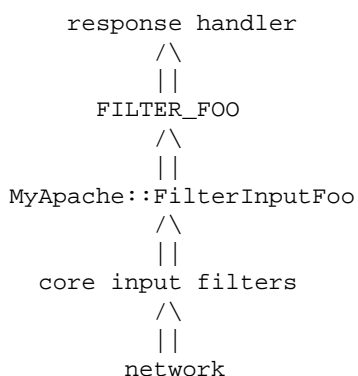
To mix `mod_perl` and non-`mod_perl` input filters of the same priority nothing special should be done. For example if we have an imaginary Apache filter `FILTER_FOO` and `mod_perl` filter `MyApache::FilterInputFoo`, this configuration:

```
SetInputFilter FILTER_FOO
PerlInputFilterHandler MyApache::FilterInputFoo
```

will add both filters, however the order of their invocation might be not the one that you've expected. To make the invocation order the same as the insertion order replace `SetInputFilter` with `PerlSetInputFilter`, like so:

```
PerlSetInputFilter FILTER_FOO
PerlInputFilterHandler MyApache::FilterInputFoo
```

now `FILTER_FOO` filter will be always executed before the `MyApache::FilterInputFoo` filter, since it was configured before `MyApache::FilterInputFoo` (i.e., it'll apply its transformations on the incoming data last). Here is a diagram input filters chain and the data flow from the network to the response handler for the presented configuration:



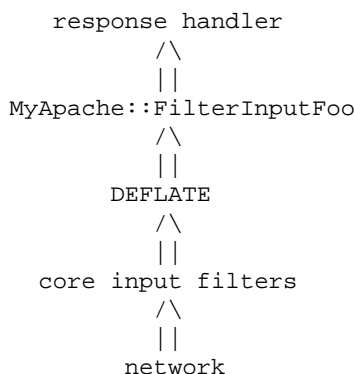
As explained in the section *Filter Priority Types* this directive won't affect filters of different priority. For example assuming that `MyApache::FilterInputFoo` is a `FilterRequestHandler` filter, the configurations:

```
PerlInputFilterHandler MyApache::FilterInputFoo
PerlSetInputFilter DEFLATE
```

and

```
PerlSetInputFilter DEFLATE
PerlInputFilterHandler MyApache::FilterInputFoo
```

are equivalent, because `mod_deflate`'s `DEFLATE` filter has a higher priority than `MyApache::FilterInputFoo`, therefore it'll always be inserted into the filter chain after `MyApache::FilterInputFoo`, (i.e. the `DEFLATE` filter will apply its transformations on the incoming data first). Here is a diagram input filters chain and the data flow from the network to the response handler for the presented configuration:



`SetInputFilter`'s ; semantics are supported as well. For example, in the following configuration:

```
PerlInputFilterHandler MyApache::FilterInputFoo
PerlSetInputFilter FILTER_FOO;FILTER_BAR
```

`MyApache::FilterOutputFoo` will be executed first, followed by `FILTER_FOO` and finally by `FILTER_BAR` (again, assuming that all three filters have the same priority).

The `PerlSetInputFilter` directives's configuration scope is `DIR`.

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:

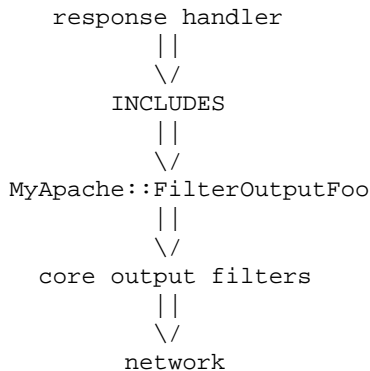
```
SetOutputFilter INCLUDES
PerlOutputFilterHandler MyApache::FilterOutputFoo
```

will add all two filters to the filter chain, however the order of their invocation might be not the one that you've expected. To preserve the insertion order replace `SetOutputFilter` with `PerlSetOutputFilter`, like so:

15.4.5 PerlSetOutputFilter

```
PerlSetOutputFilter INCLUDES
PerlOutputFilterHandler MyApache::FilterOutputFoo
```

now `mod_include`'s `INCLUDES` filter will be always executed before the `MyApache::FilterOutputFoo` filter. Here is a diagram input filters chain and the data flow from the response handler to the network for the presented configuration:



`SetOutputFilter`'s `;` semantics are supported as well. For example, in the following configuration:

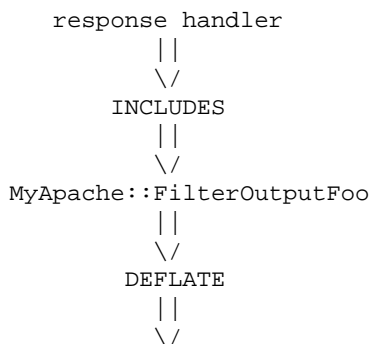
```
PerlOutputFilterHandler MyApache::FilterOutputFoo
PerlSetOutputFilter INCLUDES;FILTER_FOO
```

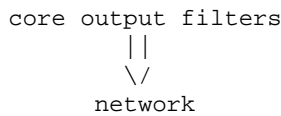
`MyApache::FilterOutputFoo` will be executed first, followed by `INCLUDES` and finally by `FILTER_FOO` (again, assuming that all three filters have the same priority).

Just 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 MyApache::FilterOutputFoo
```

`mod_include`'s `INCLUDES` filter will be always executed before the `MyApache::FilterOutputFoo` 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:





The `PerlSetOutputFilter` directives's configuration scope is `DIR`.

15.4.6 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 MyApache::FilterRequestFoo;
use base qw(Apache::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 `Apache::Filter`, others only need to:

```
use Apache::Filter ();
```

The request filters are usually configured in the `<Location>` or equivalent sections:

```

PerlModule MyApache::FilterRequestFoo
PerlModule MyApache::NiceResponse
<Location /filter_foo>
    SetHandler modperl
    PerlResponseHandler      MyApache::NiceResponse
    PerlInputFilterHandler   MyApache::FilterRequestFoo::input
    PerlOutputFilterHandler  MyApache::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 MyApache::FilterConnectionBar;
use base qw(Apache::Filter);

sub input : FilterConnectionHandler {
    my($f, $bb, $mode, $block, $readbytes) = @_;
    #...
}

sub output : FilterConnectionHandler {
    my($f, $bb) = @_;
    #...
}

1;

```

This time 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 MyApache::FilterConnectionBar
    PerlModule MyApache::NiceResponse

    PerlInputFilterHandler MyApache::FilterConnectionBar::input
    PerlOutputFilterHandler MyApache::FilterConnectionBar::output
    <Location />
        SetHandler modperl
        PerlResponseHandler MyApache::NiceResponse
    </Location>
</VirtualHost>

```

This accomplishes the configuration of the connection input and output filters.

Notice that for HTTP requests the only difference between connection filters and request filters is that the former see everything: the headers and the body, whereas the latter see only the body.

`mod_perl` provides two interfaces to filtering: a direct bucket brigades manipulation interface and a simpler, stream-oriented interface. The examples in the following sections will help you to understand the difference between the two interfaces.

15.4.7 Filter Initialization Phase

Like in any cool application, there is a hidden door, that let's you do cool things. `mod_perl` is not an exception.

where you can plug yet another callback. This *init* callback runs immediately after the filter handler is inserted into the filter chain, before it was invoked for the first time. Here is a skeleton of an init handler:


```
sub init : FilterInitHandler {
    my $f = shift;
    #...
    return Apache::OK;
}
```

The attribute `FilterInitHandler` marks the Perl function suitable to be used as a filter initialization callback, which is called immediately after a filter is inserted to the filter chain and before it's actually called.

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 Apache::OK;
}
```

Not all `Apache::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()`. But not 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 `FilterHasInitHandler` which accepts a reference to the callback function, similar to `push_handlers()`. The used callback function has to have the `FilterInitHandler` attribute. For example:

```
package MyApache::FilterBar;
use base qw(Apache::Filter);
sub init : FilterInitHandler { ... }
sub filter : FilterRequestHandler FilterHasInitHandler(\&init) {
    my ($f, $bb) = @_;
    # ...
    return Apache::OK;
}
```

While attributes are parsed during the code 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 MyApache::OtherFilter;
use base qw(Apache::Filter);
sub init : FilterInitHandler { ... }

package MyApache::FilterBar;
use MyApache::OtherFilter;
use base qw(Apache::Filter);
sub get_pre_handler { \&MyApache::OtherFilter::init }
sub filter : FilterHasInitHandler(get_pre_handler()) { ... }
```

Here the `MyApache::FilterBar::filter` handler is configured to run the `MyApache::OtherFilter::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 MyApache::FilterBar; get_pre_handler()";
```

though, this is done in C, using the `eval_pv()` C call.

META: currently only one initialization callback can be registered per filter handler. If the need to register more than one arises it should be very easy to extend the functionality.

15.5 All-in-One Filter

Before we delve into the details of how to write filters that do something with the data, let's first write a simple filter that does nothing but snooping on the data that goes through it. We are going to develop the `MyApache::FilterSnoop` handler which can snoop on request and connection filters, in input and output modes.

But first let's develop a simple response handler that simply dumps the request's *args* and *content* as strings:

```
file:MyApache/Dump.pm
-----
package MyApache::Dump;

use strict;
use warnings;

use Apache::RequestRec ();
use Apache::RequestIO ();
use APR::Table ();

use Apache::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 == Apache::M_POST) {
        my $data = content($r);
        $r->print("content:\n$data\n");
    }

    return Apache::OK;
}

sub content {
    my $r = shift;
```

```

    $r->setup_client_block;

    return '' unless $r->should_client_block;

    my $len = $r->headers_in->get('content-length');
    my $buf;
    $r->get_client_block($buf, $len);

    return $buf;
}

1;

```

which is configured as:

```

PerlModule MyApache::Dump
<Location /dump>
    SetHandler modperl
    PerlResponseHandler MyApache::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:MyApache/FilterSnoop.pm
-----
package MyApache::FilterSnoop;

use strict;
use warnings;

use base qw(Apache::Filter);
use Apache::FilterRec ();
use APR::Brigade ();
use APR::Bucket ();

use Apache::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
if (defined $mode) {
    # input filter
    my $rv = $f->next->get_brigade($bb, $mode, $block, $readbytes);
    return $rv unless $rv == APR::SUCCESS;
    bb_dump($type, $stream, $bb);
}
else {
    # output filter
    bb_dump($type, $stream, $bb);
    my $rv = $f->next->pass_brigade($bb);
    return $rv unless $rv == APR::SUCCESS;
}

return Apache::OK;
}

sub bb_dump {
    my($type, $stream, $bb) = @_;

    my @data;
    for (my $b = $bb->first; $b; $b = $bb->next($b)) {
        $b->read(my $bdata);
        $bdata = '' unless defined $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;

```

This package provides two filter handlers, one for connection and another for request filtering:

```

sub connection : FilterConnectionHandler { snoop("connection", @_) }
sub request    : FilterRequestHandler   { snoop("request",    @_) }

```

Both handlers forward their arguments to the `snoop()` function that does the real job. We needed to add these two subroutines in order to assign the two different attributes. Plus the functions pass the filter type to `snoop()` as the first argument, which gets shifted off `@_` and the rest of the `@_` 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. The arguments `$f` and `$bb` are always passed, whereas the arguments `$mode`, `$block`, and `$readbytes` are passed only to input filter handlers.

If we are in the 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 in 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 the output mode, the `$bb` variable already points to the current bucket brigade. Therefore we can read the contents of the brigade right away. After that 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 MyApache::FilterSnoop
    PerlModule MyApache::Dump

    # Connection filters
    PerlInputFilterHandler  MyApache::FilterSnoop::connection
    PerlOutputFilterHandler MyApache::FilterSnoop::connection

    <Location /dump>
        SetHandler modperl
        PerlResponseHandler MyApache::Dump
        # Request filters
        PerlInputFilterHandler  MyApache::FilterSnoop::request
        PerlOutputFilterHandler MyApache::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, when using `MyApache::FilterSnoop`, because our snooping filter didn't change anything. Though there was a lot of output printed to *error_log*. We present it all here, since it helps a lot 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.

15.5 All-in-One Filter

```
<<< 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, as it has been consumed by the last core connection filter.

The following two entries are generated when `MyApache::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 we saw earlier on the diagram, 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 `MyApache::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 sent yet 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, 19 Nov 2002 15:59:32 GMT
Server: Apache/2.0.44-dev (Unix) mod_perl/1.99_08-dev
Perl/v5.8.0 mod_ssl/2.0.44-dev OpenSSL/0.9.6d DAV/2
Connection: close
Transfer-Encoding: chunked
Content-Type: text/plain; charset=ISO-8859-1

]

```

and 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.

META: what's the size of the chunks? 8k?

Finally, Apache sends a series of the 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 understand that each filter handler can be called many time during each request and connection. It's called for each bucket brigade.

Also it's 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:

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. We can alter the request headers at the incoming connection level transparently 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:MyApache/InputFilterGET2HEAD.pm
-----
package MyApache::InputFilterGET2HEAD;

use strict;
use warnings;

use base qw(Apache::Filter);

use APR::Brigade ();
use APR::Bucket ();

use Apache::Const -compile => 'OK';
use APR::Const    -compile => ':common';

sub handler : FilterConnectionHandler {
    my($f, $bb, $mode, $block, $readbytes) = @_;

    return Apache::DECLINED if $f->ctx;

    my $rv = $f->next->get_brigade($bb, $mode, $block, $readbytes);
    return $rv unless $rv == APR::SUCCESS;

    for (my $b = $bb->first; $b; $b = $bb->next($b)) {
        my $data;
        my $status = $b->read($data);
        return $status unless $status == APR::SUCCESS;
        warn("data: $data\n");

        if ($data and $data =~ s|^GET|HEAD|) {
            my $bn = APR::Bucket->new($data);
            $b->insert_after($bn);
            $b->remove; # no longer needed
            $f->ctx(1); # flag that that we have done the job
                        last;
        }
    }

    Apache::OK;
}

1;
```

The filter handler is called for each bucket brigade, which in turn includes buckets with data. The gist of any input filter handler is to request the bucket brigade from the upstream filter, and return it 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. You have 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` is 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 `get_brigade()` to the upstream filter, 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.

To optimize the speed, the filter immediately returns `Apache::DECLINED` when it's invoked after the substitution job has been done:

```
return Apache::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::SUCCESS;
return Apache::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 Apache::DECLINED;
}
```

However, this can't be used with Apache 2.0.46 and lower, since it has a bug when trying to remove the edge connection filter (it doesn't remove it). Don't know if it's going to be fixed in 2.0.47]

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 that happens, 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:

```
$b->insert_after($bn);
$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 `Apache::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:MyApache/RequestType.pm
-----
package MyApache::RequestType;

use strict;
use warnings;

use Apache::RequestIO ();
use Apache::RequestRec ();
use Apache::Response ();

use Apache::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);

    Apache::OK;
}

1;
```

which returns to the client the request type it has issued. In the case of the HEAD request Apache will discard the response body, but it'll still set the correct `Content-Length` header, which will be 24 in case of the GET request and 25 for HEAD. Therefore if this response handler is configured as:

```
Listen 8005
<VirtualHost _default_:8005>
    <Location />
        SetHandler modperl
        PerlResponseHandler +MyApache::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
```

where the response's body is:

```
the request type was GET
```

And the Content-Length header is set to 24.

However if we enable the `MyApache::InputFilterGET2HEAD` input connection filter:

```
Listen 8005
<VirtualHost _default_:8005>
    PerlInputFilterHandler +MyApache::InputFilterGET2HEAD

    <Location />
        SetHandler modperl
        PerlResponseHandler +MyApache::RequestType
    </Location>
</VirtualHost>
```

And issue the same GET request, we get only:

```
25:
```

which means that the body was discarded by Apache, because our filter turned the GET request into a HEAD request and 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.

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

Let's look at the request input filter that lowers the case of the request's body: `MyApache::InputRequestFilterLC`:

```
file:MyApache/InputRequestFilterLC.pm
-----
package MyApache::InputRequestFilterLC;

use strict;
use warnings;
```

```

use base qw(Apache::Filter);

use Apache::Connection ();
use APR::Brigade ();
use APR::Bucket ();

use Apache::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::SUCCESS;

    while (!$bb_ctx->empty) {
        my $b = $bb_ctx->first;

        $b->remove;

        if ($b->is_eos) {
            $bb->insert_tail($b);
            last;
        }

        my $data;
        my $status = $b->read($data);
        return $status unless $status == APR::SUCCESS;

        $b = APR::Bucket->new(1c $data) if $data;

        $bb->insert_tail($b);
    }

    Apache::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`, which 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 state-less 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
```

will become:

```
get /perl/test.pl http/1.1
```

which messes up the request method, the URL and the protocol.

Now if we use the `MyApache::Dump` response handler, we have developed before in this chapter, which dumps the query string and the content body as a response, and configure the server as follows:

```
<Location /lc_input>
    SetHandler modperl
    PerlResponseHandler +MyApache::Dump
    PerlInputFilterHandler +MyApache::InputRequestFilterLC
</Location>
```

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
```

indeed we can see that our filter has lowercased the POSTed body, before the content handler received it. You can see that the query string wasn't changed.

15.6.4 *Stream-oriented Input Filters*

Let's now look at the same filter implemented using the stream-oriented API.

```
file:MyApache/InputRequestFilterLC2.pm
-----
package MyApache::InputRequestFilterLC2;

use strict;
use warnings;

use base qw(Apache::Filter);

use Apache::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);
    }
}
```

```

        Apache::OK;
    }
    1;

```

Now you probably ask yourself why did we have to go through the bucket brigades filters when this all can be done so much simpler. The reason is that we wanted you to understand how the filters work underneath, which will assist a lot when you will need to debug filters or optimize their speed. In certain cases a bucket brigade filter may be more efficient than the stream-oriented. For example if the filter applies transformation to selected buckets, certain buckets may contain open filehandles or pipes, rather than real data. And when you call `read()` 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 do faster techniques for sending data from the file handles or pipes.

The logic is very simple here, the filter reads in loop, and prints the modified data, which at some point will be sent to the next filter. This point happens every time the internal `mod_perl` buffer is full 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. Notice 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. In one of the earlier examples we have shown 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 and by simply calling `get_brigade()` as many times as needed or till EOS is received.

The configuration section is pretty much identical:

```

<Location /lc_input2>
    SetHandler modperl
    PerlResponseHandler    +MyApache::Dump
    PerlInputFilterHandler +MyApache::InputRequestFilterLC2
</Location>

```

When issuing a POST request:

```
% echo "mOd_pErL RuLeS" | POST 'http://localhost:8002/lc_input2?FoO=1&BAR=2'
```

we get a response:

```
args:
FoO=1&BAR=2
content:
mod_perl rules
```

indeed we can see that our filter has lowercased the POSTed body, before the content handler received it. You can see that the query string wasn't changed.

15.7 Output Filters

mod_perl supports Connection and HTTP Request output 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 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.

First let's develop a response handler that sends two lines of output: numerals 1234567890 and the English alphabet in a single string:

```
file:MyApache/SendAlphaNum.pm
-----
package MyApache::SendAlphaNum;

use strict;
use warnings;

use Apache::RequestRec ();
use Apache::RequestIO ();

use Apache::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");
}
```



```

    Apache::OK;
}
1;

```

The purpose of our filter handler is 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.

15.7.2.1 Stream-oriented Output Filters

The first filter implementation is using the stream-oriented filtering API:

```

file:MyApache/FilterReverse1.pm
-----
package MyApache::FilterReverse1;

use strict;
use warnings;

use base qw(Apache::Filter);

use Apache::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");
        }
    }

    Apache::OK;
}
1;

```

Next, we add the following configuration to *httpd.conf*:

```

PerlModule MyApache::FilterReverse1
PerlModule MyApache::SendAlphaNum
<Location /reverse1>
    SetHandler modperl
    PerlResponseHandler      MyApache::SendAlphaNum
    PerlOutputFilterHandler MyApache::FilterReverse1
</Location>

```

Now when a request to */reverse1* is made, the response handler `MyApache::SendAlphaNum::handler()` sends:

```
1234567890
abcdefghijklmnopqrstuvwxyz
```

as a response and the output filter handler `MyApache::FilterReverse1::handler` reverses the lines, so the client gets:

```
0987654321
zyxwvutsrqponmlkjihgfedcba
```

The `Apache::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.

In order not to distract the reader from the purpose of the example the used code is oversimplified and won't handle correctly input lines which are longer than 1024 characters and possibly using a different line termination token (could be `"\n"`, `"\r"` or `"\r\n"` depending on a platform). Moreover a single line may be split between 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. So here is an example of a more complete handler, which does 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 Apache::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 on the last time, it

unconditionally reverses and flushes any remaining data.

15.7.2.2 Bucket Brigade-based Output Filters

The following filter implementation is using the bucket brigades API to accomplish exactly the same task as the first filter.

```
file:MyApache/FilterReverse2.pm
-----
package MyApache::FilterReverse2;

use strict;
use warnings;

use base qw(Apache::Filter);

use APR::Brigade ();
use APR::Bucket ();

use Apache::Const -compile => 'OK';
use APR::Const -compile => ':common';

sub handler : FilterRequestHandler {
    my($f, $bb) = @_;

    my $c = $f->c;
    my $bb_ctx = APR::Brigade->new($c->pool, $c->bucket_alloc);

    while (!$bb->empty) {
        my $bucket = $bb->first;

        $bucket->remove;

        if ($bucket->is_eos) {
            $bb_ctx->insert_tail($bucket);
            last;
        }

        my $data;
        my $status = $bucket->read($data);
        return $status unless $status == APR::SUCCESS;

        if ($data) {
            $data = join "",
                map { scalar(reverse $_), "\n" } split "\n", $data;
            $bucket = APR::Bucket->new($data);
        }
    }
}
```

15.8 Filter Applications

```
        $bb_ctx->insert_tail($bucket);
    }

    my $rv = $f->next->pass_brigade($bb_ctx);
    return $rv unless $rv == APR::SUCCESS;

    Apache::OK;
}
1;
```

and the corresponding configuration:

```
PerlModule MyApache::FilterReverse2
PerlModule MyApache::SendAlphaNum
<Location /reverse2>
    SetHandler modperl
    PerlResponseHandler      MyApache::SendAlphaNum
    PerlOutputFilterHandler  MyApache::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. Since when the handler is completed it must pass a brigade to the next filter in the stack, we create a new bucket brigade into which we are going to put the modified buckets and which eventually we pass to the next filter.

The core of the handler is in removing buckets from the head of the bucket brigade \$bb while there are some, reading the data from the buckets, reversing and putting it into a newly created bucket which is inserted to the end of the new bucket brigade. If we see a bucket which designates the end of stream, we insert that bucket to 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.

Similarly to the original version of `MyApache::FilterReverse1::handler`, this filter is not smart enough to handle incomplete lines. However the exercise of making the filter foolproof should be trivial by porting a better matching rule and using the `$leftover` buffer from the previous section is trivial and left as an exercise to 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 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. Meanwhile it must return an empty bb to the upstream input filter. This is not the most efficient technique to resolve underruns.

Instead of returning an empty bb, the input filter can initiate the retrieval of extra bucket brigades, until the underrun condition gets resolved. Notice that this solution is absolutely 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 say that the filter needs to have $1024 * 16 + 5$ bytes to have a complete token and then 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 40975 bytes of input and a token size of 16389 bytes, we will have 2 full tokens and 8197 remainder.

Jumping ahead let's look at the filter debug output:

```
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
```

So we can see that the filter was invoked three times. The first time it has consumed three bucket brigades, collecting one full token of 16389 bytes and has a remainder of 7611 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, 7222 bytes have remained. Finally on the third invocation it has consumed the last bucket brigade (total of six, just as we have 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 8197 bytes as we have 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 any) wasn'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 16389 bytes per bucket brigade. But of course in real world some transformation is applied on these tokens.

Now you understand what did we want from the filter, it's time for the implementation details. First let's look at the `response()` handler (the first part of the module):

```
#file:MyApache/Underrun.pm
#-----
package MyApache::Underrun;

use strict;
use warnings;

use constant IOBUFSIZE => 8192;

use Apache::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 == Apache::M_POST) {
        my $data = read_post($r);
        #warn "HANDLER READ: $data\n";
        my $length = length $data;
        $r->print("read $length chars");
    }

    return Apache::OK;
}

sub read_post {
    my $r = shift;
    my $debug = shift || 0;

    my @data = ();
    my $seen_eos = 0;
    my $filters = $r->input_filters();
    my $ba = $r->connection->bucket_alloc;
    my $bb = APR::Brigade->new($r->pool, $ba);
```

```

do {
    my $rv = $filters->get_brigade($bb,
        Apache::MODE_READBYTES, APR::BLOCK_READ, IOBUFSIZE);

    if ($rv != APR::SUCCESS) {
        return $rv;
    }

    while (!$bb->empty) {
        my $buf;
        my $b = $bb->first;

        $b->remove;

        if ($b->is_eos) {
            warn "EOS bucket:\n" if $debug;
            $seen_eos++;
            last;
        }

        my $status = $b->read($buf);
        warn "DATA bucket: [$buf]\n" if $debug;
        if ($status != APR::SUCCESS) {
            return $status;
        }
        push @data, $buf;
    }

    $bb->destroy;

} while (!$seen_eos);

return join '', @data;
}

```

The `response()` handler is trivial -- it reads the POSTed data and prints how many bytes it has read. `read_post()` sucks all POSTed data without parsing it.

Now comes the filter (which lives in the same package):

```

#file:MyApache/Underrun.pm (continued)
#-----
use Apache::Filter ();

use Apache::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 untill 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) {
        $bb->insert_tail(APR::Bucket->new($_));
    }
    else {
        $ctx .= $_;
    }
}

my $len = length($ctx);
if ($seen_eos) {
    # flush the remainder
    $bb->insert_tail(APR::Bucket->new($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 Apache::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);
    $bdata = '' unless defined $bdata;
    push @data, $bdata;
}
return (join('', @data), $seen_eos);
}

1;

```

The filter calls `get_brigade()` in a do-while loop till it reads enough data or sees EOS. Notice that it may get underruns for 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 care 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 has been seen.

Notice that this won't be possible with streaming filters where every invocation gives the filter exactly one bucket brigade to work with and provides not facilities to fetch extra brigades. (META: however this can be fixed, by providing a method which will fetch the next bucket brigade, so the read in a while loop can be repeated)

And here is the configuration for this setup:

```

PerlModule MyApache::Underrun
<Location />
    PerlInputFilterHandler MyApache::Underrun::filter
    SetHandler modperl
    PerlResponseHandler MyApache::Underrun::response
</Location>

```

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 *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).

15.10.2 *Other issues*

META: to be written. Meanwhile collecting important inputs from various sources.

[

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

META: to be written

[

As of this writing the network input filter reads in 8000B chunks (not 8192B!), and making each bucket 8000B in size, so it seems that 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 `$bucket->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 Maintainers

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

- Stas Bekman <stas (at) stason.org>

15.13 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 <stas (at) 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 free 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 `PerlOpenLogHandler` 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 $conf_dir = Apache::server_root_relative($s->pool, 'conf');
```

when you can call:

```
my $conf_dir = Apache::server_root_relative($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 <stas (at) stason.org>

17.4 Authors

- Stas Bekman <stas (at) 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 to save some memory. However the savings are not big, since all these modules are written 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 infrequent dynamic requests, when needed.

It's not clear yet, how one will be able to control the amount of running Perl interpreters, 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 occasionally 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 <stas (at) stason.org>

18.5 Authors

- Stas Bekman <stas (at) 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

19.3 Configuration and Startup

19.3.1 (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.

19.3.2 Segmentation Fault when Using DBI

Update DBI to at least version 1.31.

19.3.3 <Perl> directive missing closing '>'

See the `Apache::PerlSections` manpage.

19.4 Shutdown and Restart

19.5 Code Parsing and Compilation

19.6 Runtime

19.6.1 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.

19.6.2 Error about not finding Apache.pm with CGI.pm

You need to install at least version 2.87 of CGI.pm to work under mod_perl 2.0, as earlier CGI.pm versions aren't mod_perl 2.0 aware.

19.6.3 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.

19.6.4 (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>

19.6.5 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-
```

META: ldd is not available on all platforms, e.g. not on Darwin/OS X

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
```

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. So, if you got "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.

19.7 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 -MApache2 -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 *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.

20.2.3 *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 recently released version: http://perl.apache.org/dist/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.

20.2.4 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.

20.2.5 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.

20.2.6 Important Information

Whenever you send a bug report make sure to include the information about your system by doing the following:

```
% 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 auto-generated when perl Makefile.PL is run, so you should have it already after building mod_perl.

META: soon we will have mp2bug report script which will be installed system-wide. For now, if you don't have the source, you can create the report by running the following:

```
% perl -MApache2 -MApache::TestReportPerl \
-le 'Apache::TestReportPerl->new->run'
```

Now add the problem description to the report and send it to the list.

20.2.7 Problem Description

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!).

20.2.8 '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 the run and the contents of the *error_log* file to the list.

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-1.99_xx
% make test TEST_VERBOSE=1 \
  TEST_FILES="compat/apache_util.t modperl/pnotes.t"
```

or use an alternative way:

```
% cd modperl-1.99_xx
% rm t/logs/error_log
% t/TEST -verbose compat/apache_util.t modperl/pnotes.t
```

If you are using the latter, remember to remove the *error_log* file before running tests, so you won't have clutter from the previous run. `make test` always removes the old *error_log* file for you.

Also please notice that there is more than one `make test` run. The first one is running at the top directory, the second inside a sub-directory *ModPerl-Registry/*. The first logs errors to *t/logs/error_log*, the second to *ModPerl-Registry/t/logs/error_log*. 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-1.99_xx/ModPerl-Registry
% rm t/logs/error_log
% t/TEST -verbose closure.t
```

20.2.9 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 extract this backtrace.

Of course to generate a useful backtrace you need to have `mod_perl` with debugging symbols in it (and probably `perl` and/or `httpd` too) and also to be able to see the arguments in the calls trace. To accomplish that do:

- **mod_perl**

rebuild mod_perl with MP_DEBUG=1.

```
% perl Makefile.PL MP_DEBUG=1 ...
% make && make test && make install
```

- **httpd**

If the segfault happens inside *ap_* or *apr_* calls, rebuild httpd with `--enable-maintainer-mode`:

```
% ./configure --enable-maintainer-mode ...
% make && make install
```

- **perl**

If the segfault happens inside *Perl_* calls, rebuild perl with `-Doptimize='-g'`:

```
% ./Configure -Doptimize='-g' ...
% make && make test && make install
```

- **3rd party perl modules**

if the trace happens in one of the 3rd party perl modules, make sure to rebuild them, now that you've perl re-built with debugging flags. They will automatically pick the right compile flags from perl.

Once a proper stack backtrace is obtained append it to the bug report as explained in the previous section.

20.2.10 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:

User's guide	1
Getting Your Feet Wet with mod_perl	4
1 Getting Your Feet Wet with mod_perl	4
1.1 Description	5
1.2 Installation	5
1.3 Configuration	5
1.4 Server Launch and Shutdown	6
1.5 Registry Scripts	6
1.6 Handler Modules	7
1.7 Troubleshooting	8
1.8 Maintainers	8
1.9 Authors	8
Overview of mod_perl 2.0	9
2 Overview of mod_perl 2.0	9
2.1 Description	10
2.2 Version Naming Conventions	10
2.3 Why mod_perl, The Next Generation	10
2.4 What's new in Apache 2.0	11
2.5 What's new in Perl 5.6.0 - 5.8.0	14
2.6 What's new in mod_perl 2.0	16
2.6.1 Threads Support	16
2.6.2 Thread-environment Issues	17
2.6.3 Perl Interface to the APR and Apache APIs	18
2.7 Integration with 2.0 Filtering	19
2.7.1 Other New Features	19
2.7.2 Optimizations	19
2.8 Maintainers	20
2.9 Authors	20
Notes on the design and goals of mod_perl-2.0	21
3 Notes on the design and goals of mod_perl-2.0	21
3.1 Description	22
3.2 Introduction	22
3.3 Interpreter Management	22
3.3.1 TIPool	24
3.3.2 Virtual Hosts	24
3.3.3 Further Enhancements	25
3.4 Hook Code and Callbacks	25
3.5 Perl interface to the Apache API and Data Structures	25
3.5.1 Advantages to generating XS code	27
3.5.2 Lvalue methods	28
3.6 Filter Hooks	28
3.7 Directive Handlers	28
3.8 <Perl> Configuration Sections	28
3.9 Protocol Module Support	29

Table of Contents:

3.10	mod_perl MPM	29
3.11	Build System	29
3.12	Test Framework	29
3.13	CGI Emulation	29
3.14	Apache::* Library	30
3.15	Perl Enhancements	30
3.15.1	GvSHARED	30
3.15.2	Shared SvPVX	31
3.15.3	Compile-time method lookups	31
3.15.4	Memory management hooks	31
3.15.5	Opcodes hooks	31
3.16	Maintainers	32
3.17	Authors	32
	Installing mod_perl 2.0	33
4	Installing mod_perl 2.0	33
4.1	Description	34
4.2	Prerequisites	34
4.2.1	Downloading Stable Release Sources	35
4.2.2	Getting Bleeding Edge CVS Sources	35
4.2.3	Configuring and Installing Prerequisites	36
4.3	Installing mod_perl from Binary Packages	37
4.4	Installing mod_perl from Source	37
4.4.1	Downloading the mod_perl Source	37
4.4.2	Configuring mod_perl	38
4.4.2.1	Boolean Build Options	39
4.4.2.1.1	MP_PROMPT_DEFAULT	39
4.4.2.1.2	MP_GENERATE_XS	39
4.4.2.1.3	MP_USE_DSO	39
4.4.2.1.4	MP_USE_STATIC	39
4.4.2.1.5	MP_STATIC_EXTS	40
4.4.2.1.6	MP_USE_GTOP	40
4.4.2.1.7	MP_COMPAT_1X	40
4.4.2.1.8	MP_DEBUG	41
4.4.2.1.9	MP_MAINTAINER	41
4.4.2.1.10	MP_TRACE	41
4.4.2.1.11	MP_INST_APACHE2	41
4.4.2.2	Non-Boolean Build Options	41
4.4.2.2.1	MP_APXS	42
4.4.2.2.2	MP_AP_PREFIX	42
4.4.2.2.3	MP_APR_CONFIG	42
4.4.2.2.4	MP_CCOPTS	42
4.4.2.2.5	MP_OPTIONS_FILE	42
4.4.2.3	mod_perl-specific Compiler Options	43
4.4.2.3.1	-DMP_IOBUFSIZE	43
4.4.2.4	mod_perl Options File	43
4.4.3	Re-using Configure Options	43
4.4.4	Compiling mod_perl	43

4.4.5 Testing mod_perl	44
4.4.6 Installing mod_perl	44
4.5 If Something Goes Wrong	44
4.6 Maintainers	44
4.7 Authors	44
mod_perl 2.0 Server Configuration	45
5 mod_perl 2.0 Server Configuration	45
5.1 Description	46
5.2 mod_perl configuration directives	46
5.3 Enabling mod_perl	46
5.4 Accessing the mod_perl 2.0 Modules	46
5.5 Startup File	46
5.6 Server Configuration Directives	48
5.6.1 PerlRequire	48
5.6.2 PerlModule	48
5.6.3 PerlLoadModule	48
5.6.4 PerlSetVar	48
5.6.5 PerlAddVar	48
5.6.6 PerlSetEnv	48
5.6.7 PerlPassEnv	48
5.6.8 <Perl> Sections	48
5.6.9 PerlSwitches	48
5.6.10 SetHandler	49
5.6.10.1 modperl	49
5.6.10.2 perl-script	50
5.6.10.3 Examples	50
5.6.11 PerlOptions	52
5.6.11.1 Enable	52
5.6.11.2 Clone	52
5.6.11.3 Parent	53
5.6.11.4 Perl*Handler	53
5.6.11.5 AutoLoad	54
5.6.11.6 GlobalRequest	54
5.6.11.7 ParseHeaders	55
5.6.11.8 MergeHandlers	56
5.6.11.9 SetupEnv	56
5.7 Server Life Cycle Handlers Directives	57
5.7.1 PerlOpenLogsHandler	57
5.7.2 PerlPostConfigHandler	58
5.7.3 PerlChildInitHandler	58
5.7.4 PerlChildExitHandler	58
5.8 Protocol Handlers Directives	58
5.8.1 PerlPreConnectionHandler	58
5.8.2 PerlProcessConnectionHandler	58
5.9 Filter Handlers Directives	58
5.9.1 PerlInputFilterHandler	58
5.9.2 PerlOutputFilterHandler	58

Table of Contents:

5.9.3 PerlSetInputFilter	59
5.9.4 PerlSetOutputFilter	59
5.10 HTTP Protocol Handlers Directives	59
5.10.1 PerlPostReadRequestHandler	59
5.10.2 PerlTransHandler	59
5.10.3 PerlMapToStorageHandler	59
5.10.4 PerlInitHandler	59
5.10.5 PerlHeaderParserHandler	59
5.10.6 PerlAccessHandler	59
5.10.7 PerlAuthenHandler	59
5.10.8 PerlAuthzHandler	60
5.10.9 PerlTypeHandler	60
5.10.10 PerlFixupHandler	60
5.10.11 PerlResponseHandler	60
5.10.12 PerlLogHandler	60
5.10.13 PerlCleanupHandler	60
5.11 Threads Mode Specific Directives	60
5.11.1 PerlInterpStart	60
5.11.2 PerlInterpMax	60
5.11.3 PerlInterpMinSpare	61
5.11.4 PerlInterpMaxSpare	61
5.11.5 PerlInterpMaxRequests	61
5.11.6 PerlInterpScope	61
5.12 Debug Directives	62
5.12.1 PerlTrace	62
5.13 mod_perl Directives Argument Types and Allowed Location	63
5.14 Server Startup Options Retrieval	65
5.14.1 MODPERL2 Define Option	66
5.15 Perl Interface to the Apache Configuration Tree	66
5.16 Adjusting @INC	66
5.16.1 PERL5LIB and PERLLIB Environment Variables	67
5.16.2 Modifying @INC on a Per-VirtualHost	67
5.17 General Issues	68
5.18 Maintainers	68
5.19 Authors	68
Apache Server Configuration Customization in Perl	69
6 Apache Server Configuration Customization in Perl	69
6.1 Description	70
6.2 Incentives	70
6.3 Creating and Using Custom Configuration Directives	70
6.3.1 @APACHE_MODULE_COMMANDS	72
6.3.1.1 name	72
6.3.1.2 func	72
6.3.1.3 req_override	73
6.3.1.4 args_how	73
6.3.1.5 errmsg	73
6.3.1.6 cmd_data	74

6.3.2 Directive Scope Definition Constants	75
6.3.2.1 Apache::OR_NONE	75
6.3.2.2 Apache::OR_LIMIT	75
6.3.2.3 Apache::OR_OPTIONS	75
6.3.2.4 Apache::OR_FILEINFO	75
6.3.2.5 Apache::OR_AUTHCFG	75
6.3.2.6 Apache::OR_INDEXES	75
6.3.2.7 Apache::OR_UNSET	75
6.3.2.8 Apache::ACCESS_CONF	75
6.3.2.9 Apache::RSRC_CONF	76
6.3.2.10 Apache::OR_EXEC_ON_READ	76
6.3.2.11 Apache::OR_ALL	76
6.3.3 Directive Callback Subroutine	76
6.3.4 Directive Syntax Definition Constants	77
6.3.4.1 Apache::NO_ARGS	77
6.3.4.2 Apache::TAKE1	77
6.3.4.3 Apache::TAKE2	78
6.3.4.4 Apache::TAKE3	78
6.3.4.5 Apache::TAKE12	78
6.3.4.6 Apache::TAKE23	78
6.3.4.7 Apache::TAKE123	78
6.3.4.8 Apache::ITERATE	78
6.3.4.9 Apache::ITERATE2	79
6.3.4.10 Apache::RAW_ARGS	79
6.3.4.11 Apache::FLAG	80
6.3.5 Enabling the New Configuration Directives	80
6.3.6 Creating and Merging Configuration Objects	81
6.3.6.1 SERVER_CREATE	81
6.3.6.2 SERVER_MERGE	82
6.3.6.3 DIR_CREATE	82
6.3.6.4 DIR_MERGE	83
6.4 Examples	83
6.4.1 Merging at Work	83
6.4.1.1 Merging Entries Whose Values Are References	89
6.4.1.2 Merging Order Consequences	90
6.5 Maintainers	91
6.6 Authors	91
Writing mod_perl Handlers and Scripts	92
7 Writing mod_perl Handlers and Scripts	92
7.1 Description	93
7.2 Prerequisites	93
7.3 Where the Methods Live	93
7.4 Goodies Toolkit	93
7.4.1 Environment Variables	93
7.4.2 Threaded MPM or not?	94
7.4.3 Writing MPM-specific Code	94
7.5 Code Developing Nuances	94

Table of Contents:

7.5.1 Auto-Reloading Modified Modules with Apache::Reload	94
7.6 Integration with Apache Issues	95
7.6.1 Sending HTTP Response Headers	96
7.6.2 Sending HTTP Response Body	96
7.7 Perl Specifics in the mod_perl Environment	96
7.7.1 Request-localized Globals	96
7.7.2 exit()	97
7.8 Threads Coding Issues Under mod_perl	97
7.8.1 Thread-environment Issues	97
7.8.2 Deploying Threads	98
7.8.3 Shared Variables	98
7.9 Maintainers	98
7.10 Authors	98
Cooking Recipes	99
8 Cooking Recipes	99
8.1 Description	100
8.2 Sending Cookies in REDIRECT Response (ModPerl::Registry)	100
8.3 Sending Cookies in REDIRECT Response (handlers)	100
8.4 Maintainers	101
8.5 Authors	101
Porting Apache:: Perl Modules from mod_perl 1.0 to 2.0	102
9 Porting Apache:: Perl Modules from mod_perl 1.0 to 2.0	102
9.1 Description	103
9.2 Introduction	103
9.3 Using Apache::porting	104
9.4 Using the Apache::compat Layer	104
9.5 Porting a Perl Module to Run under mod_perl 2.0	105
9.5.1 Using ModPerl::MethodLookup to Discover Which mod_perl 2.0 Modules Need to Be Loaded	105
9.5.1.1 Handling Methods Existing In More Than One Package	106
9.5.1.2 Using ModPerl::MethodLookup Programmatically	106
9.5.1.3 Pre-loading All mod_perl 2.0 Modules	107
9.5.2 Handling Missing and Modified mod_perl 1.0 Methods and Functions	107
9.5.2.1 Methods that No Longer Exist	107
9.5.2.2 Methods Whose Usage Has Been Modified	108
9.5.3 Requiring a specific mod_perl version.	108
9.5.4 Should the Module Name Be Changed?	109
9.5.5 Using Apache::compat As a Tutorial	109
9.5.6 How Apache::MP3 was Ported to mod_perl 2.0	110
9.5.6.1 Preparations	110
9.5.6.1.1 <i>httpd.conf</i>	111
9.5.6.1.2 <i>startup.pl</i>	112
9.5.6.1.3 <i>Apache/MP3.pm</i>	112
9.5.6.2 Porting with Apache::compat	113
9.5.6.3 Getting Rid of the Apache::compat Dependency	120
9.5.6.4 Ensuring that Apache::compat is not loaded	120
9.5.6.5 Installing the ModPerl::MethodLookup Helper	122

9.5.6.6 Adjusting the code to run under mod_perl 2	122
9.6 Porting a Module to Run under both mod_perl 2.0 and mod_perl 1.0	131
9.6.1 Making Code Conditional on Running mod_perl Version	131
9.6.2 Method Handlers	132
9.7 Maintainers	135
9.8 Authors	135
A Reference to mod_perl 1.0 to mod_perl 2.0 Migration.	136
10 A Reference to mod_perl 1.0 to mod_perl 2.0 Migration.	136
10.1 Description	137
10.2 Configuration Files Porting	137
10.2.1 PerlHandler	137
10.2.2 PerlSendHeader	137
10.2.3 PerlSetupEnv	137
10.2.4 PerlTaintCheck	138
10.2.5 PerlWarn	138
10.2.6 PerlFreshRestart	138
10.2.7 Apache Configuration Customization	138
10.2.8 @INC Manipulation	138
10.3 Code Porting	139
10.4 Apache::Registry, Apache::PerlRun and Friends	140
10.4.1 ModPerl::RegistryLoader	141
10.5 Apache::Constants	141
10.5.1 mod_perl 1.0 and 2.0 Constants Coexistence	142
10.5.2 Deprecated Constants	143
10.5.3 SERVER_VERSION()	143
10.5.4 export()	143
10.6 Issues with Environment Variables	143
10.7 Special Environment Variables	144
10.7.1 \$ENV{GATEWAY_INTERFACE}	144
10.8 Apache::Methods	144
10.8.1 Apache->request	144
10.8.2 Apache->define	146
10.8.3 Apache->can_stack_handlers	146
10.8.4 Apache->untaint	146
10.8.5 Apache->get_handlers	146
10.8.6 Apache->push_handlers	146
10.8.7 Apache->set_handlers	147
10.8.8 Apache->httpd_conf	147
10.8.9 Apache::exit()	147
10.8.10 Apache::gensym()	147
10.8.11 Apache::module()	147
10.8.12 Apache::log_error()	148
10.9 Apache::Variables	148
10.9.1 \$Apache::__T	148
10.10 Apache::Server::Methods and Variables	148
10.10.1 \$Apache::Server::CWD	148
10.10.2 \$Apache::Server::AddPerlVersion	148

Table of Contents:

10.11 Server Object Methods	148
10.11.1 \$s->register_cleanup	148
10.12 Request Object Methods	148
10.12.1 \$r->cgi_env	149
10.12.2 \$r->cgi_var	149
10.12.3 \$r->current_callback	149
10.12.4 \$r->get_remote_host	149
10.12.5 \$r->cleanup_for_exec	149
10.12.6 \$r->content	149
10.12.7 \$r->args in an Array Context	149
10.12.8 \$r->chdir_file	150
10.12.9 \$r->is_main	150
10.12.10 \$r->finfo	150
10.12.11 \$r->notes	151
10.12.12 \$r->header_in	151
10.12.13 \$r->header_out	151
10.12.14 \$r->err_header_out	151
10.12.15 \$r->log_reason	151
10.12.16 \$r->register_cleanup	152
10.12.17 \$r->post_connection	152
10.12.18 \$r->request	152
10.12.19 \$r->send_fd	152
10.12.20 \$r->send_fd_length	152
10.12.21 \$r->send_http_header	153
10.12.22 \$r->server_root_relative	153
10.12.23 \$r->hard_timeout	153
10.12.24 \$r->reset_timeout	153
10.12.25 \$r->soft_timeout	153
10.12.26 \$r->kill_timeout	153
10.12.27 \$r->set_byterange	154
10.12.28 \$r->each_byterange	154
10.13 Apache::Connection	154
10.13.1 \$connection->auth_type	154
10.13.2 \$connection->user	154
10.13.3 \$connection->local_addr	154
10.13.4 \$connection->remote_addr	154
10.14 Apache::File	155
10.14.1 open() and close()	155
10.14.2 tmpfile()	155
10.15 Apache::Util	155
10.15.1 Apache::Util::size_string()	155
10.15.2 Apache::Util::escape_uri()	156
10.15.3 Apache::Util::unescape_uri()	156
10.15.4 Apache::Util::escape_html()	156
10.15.5 Apache::Util::parsedate()	156
10.15.6 Apache::Util::ht_time()	156
10.15.7 Apache::Util::validate_password()	156

10.16	Apache::URI	157
10.16.1	Apache::URI->parse(\$r, [\$uri])	157
10.16.2	unparse()	157
10.17	Miscellaneous	157
10.17.1	Method Handlers	158
10.17.2	Stacked Handlers	158
10.18	Apache::src	159
10.19	Apache::Table	159
10.20	Apache::SIG	159
10.21	Apache::StatINC	159
10.22	Maintainers	160
10.23	Authors	160
	Introducing mod_perl Handlers	161
11	Introducing mod_perl Handlers	161
11.1	Description	162
11.2	What are Handlers?	162
11.3	Handler Return Values	163
11.4	mod_perl Handlers Categories	163
11.5	Single Phase's Multiple Handlers Behavior	164
11.5.1	VOID	165
11.5.2	RUN_FIRST	165
11.5.3	RUN_ALL	165
11.6	Hook Ordering (Position)	165
11.7	Bucket Brigades	166
11.8	Maintainers	167
11.9	Authors	167
	Server Life Cycle Handlers	168
12	Server Life Cycle Handlers	168
12.1	Description	169
12.2	Server Life Cycle	169
12.2.1	Startup Phases Demonstration Module	170
12.2.2	PerlOpenLogsHandler	172
12.2.3	PerlPostConfigHandler	173
12.2.4	PerlChildInitHandler	174
12.2.5	PerlChildExitHandler	174
12.3	Maintainers	175
12.4	Authors	175
	Protocol Handlers	176
13	Protocol Handlers	176
13.1	Description	177
13.2	Connection Cycle Phases	177
13.2.1	PerlPreConnectionHandler	178
13.2.2	PerlProcessConnectionHandler	180
13.2.2.1	Socket-based Protocol Module	180
13.2.2.2	Bucket Brigades-based Protocol Module	182
13.3	Maintainers	185
13.4	Authors	185

HTTP Handlers	186
14 HTTP Handlers	186
14.1 Description	187
14.2 HTTP Request Cycle Phases	187
14.2.1 PerlPostReadRequestHandler	189
14.2.2 PerlTransHandler	191
14.2.3 PerlMapToStorageHandler META: add something here	192
14.2.4 PerlHeaderParserHandler	192
14.2.5 PerlInitHandler	196
14.2.6 PerlAccessHandler	196
14.2.7 PerlAuthenHandler	197
14.2.8 PerlAuthzHandler	200
14.2.9 PerlTypeHandler	202
14.2.10 PerlFixupHandler	202
14.2.11 PerlResponseHandler	204
14.2.12 PerlLogHandler	206
14.2.13 PerlCleanupHandler	208
14.3 Handling HEAD Requests	211
14.4 Extending HTTP Protocol	211
14.5 Maintainers	211
14.6 Authors	212
Input and Output Filters	213
15 Input and Output Filters	213
15.1 Description	214
15.2 Your First Filter	214
15.3 I/O Filtering Concepts	218
15.3.1 Two Methods for Manipulating Data	218
15.3.2 HTTP Request Versus Connection Filters	219
15.3.3 Multiple Invocations of Filter Handlers	219
15.3.4 Blocking Calls	223
15.4 mod_perl Filters Declaration and Configuration	226
15.4.1 Filter Priority Types	227
15.4.2 PerlInputFilterHandler	227
15.4.3 PerlOutputFilterHandler	227
15.4.4 PerlSetInputFilter	228
15.4.5 PerlSetOutputFilter	229
15.4.6 HTTP Request vs. Connection Filters	231
15.4.7 Filter Initialization Phase	232
15.5 All-in-One Filter	234
15.6 Input Filters	240
15.6.1 Connection Input Filters	240
15.6.2 HTTP Request Input Filters	244
15.6.3 Bucket Brigade-based Input Filters	244
15.6.4 Stream-oriented Input Filters	246
15.7 Output Filters	248
15.7.1 Connection Output Filters	248
15.7.2 HTTP Request Output Filters	248

xi

Table of Contents:

19.6.1	C Libraries Don't See %ENV Entries Set by Perl Code	270
19.6.2	Error about not finding <i>Apache.pm</i> with <i>CGI.pm</i>	271
19.6.3	20014:Error string not specified yet	271
19.6.4	(22)Invalid argument: core_output_filter: writing data to the network	271
19.6.5	undefined symbol: apr_table_compress	271
19.7	Issues with APR Used Outside of mod_perl	274
19.8	Maintainers	274
19.9	Authors	274
User Help		275
20	User Help	275
20.1	Description	276
20.2	Reporting Problems	276
20.2.1	Wrong Apache/mod_perl combination	276
20.2.2	Before Posting a Report	276
20.2.3	Test with the Latest mod_perl 2.0 Version	276
20.2.4	Use a Proper Subject	277
20.2.5	Send the Report Inlined	277
20.2.6	Important Information	277
20.2.7	Problem Description	277
20.2.8	'make test' Failures	278
20.2.9	Resolving Segmentation Faults	278
20.2.10	Please Ask Only Questions Related to mod_perl	279
20.3	Help on Related Topics	279
20.4	Maintainers	280
20.5	Authors	280