

Presentation Handouts: mod_perl 2.0, the Next Generation

by **Stas Bekman**
<http://stason.org/>
<stas@stason.org>
TicketMaster

ApacheCon US 2002
Monday, Nov 20 2002
Las Vegas, Nevada, USA

This talk is available from: <http://stason.org/talks/>

Last modified Sun Nov 17 06:15:58 2002 GMT

1 The Next Generation: mod_perl 2.0

1.1 About

1. What's new in Apache 2.0
2. What's new in Perl 5.6.0 - 5.8.0
3. What's new in mod_perl 2.0
4. Installing mod_perl 2.0
5. Configuring mod_perl 2.0
6. Working Examples
7. Migrating from 1.x to 2.0

1.2 Thank you!

* TicketMaster rules!!!



1.3 Versioning Convention

Here and in the rest of this document we refer to mod_perl 1.x series as mod_perl 1.0 and, 2.0.x as mod_perl 2.0 to keep things simple. Similarly we call Apache 1.3.x series as Apache 1.3 and 2.0.x as Apache 2.0

1.4 Why mod_perl, the Next Generation

Since Doug MacEachern has introduced mod_perl 1.0 in 1996, he had to adjust source code to the many changes Apache and Perl went through, while staying compatible with the older versions, leading to a very complex source code, with hundreds of `#ifdefs` and workarounds for various incompatibilities in older Perl and Apache versions. When Apache 2.0 development was underway, the new threads design was introduced, which couldn't be supported by the existing Perl version, since it required thread-safe Perl interpreters.

Think of it as a conspiracy or just a lucky coincidence, on March 10, 2002, the first Apache 2.0 alpha version was released. 13 days later, on March 23, 2002, Perl 5.6.0 has been released. And guess what, Perl 5.6.0 was the first Perl version to support the internal thread-safeness across multiple interpreters.

Since Perl 5.6.0 and Apache 2.0 were the very minimum requirements there was no need to support older version and it was a great idea to start mod_perl 2.0 code base from scratch, incorporating the lessons learned during the 5 years of mod_perl's existence.

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.

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 will be covered in this document and some you will discover on your own while reading mod_perl documentation.

1.4.1 The Apache::Test Framework

Another important new 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.

1.5 What's new in Apache 2.0

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

- ***Apache Portable Runtime (APR)***

The APR presents a standard API for server applications, covering file I/O, logging, shared memory, threads, managing child processes and many other functionalities needed for developing the Apache core and third party modules in a portable and effective way. One of the important effects is that it significantly simplifies the code that uses the APR making it much easier to review and understand the Apache code, increasing the number of revealed bugs and contributed patches.

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.

- ***Multi Processing Model modules (MPMs).***

In the previous Apache generation the same code base was trying to handle a management of incoming requests for different platforms, which lead to scalability problems on certain platforms, mainly on those which are different from Unix. This also lead to an undesired complexity of the code.

Apache 2.0 introduces the concept of Multi Processing Model modules, whose main responsibility is to map the incoming requests to either threads, processes or a threads/processes hybrid. 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**

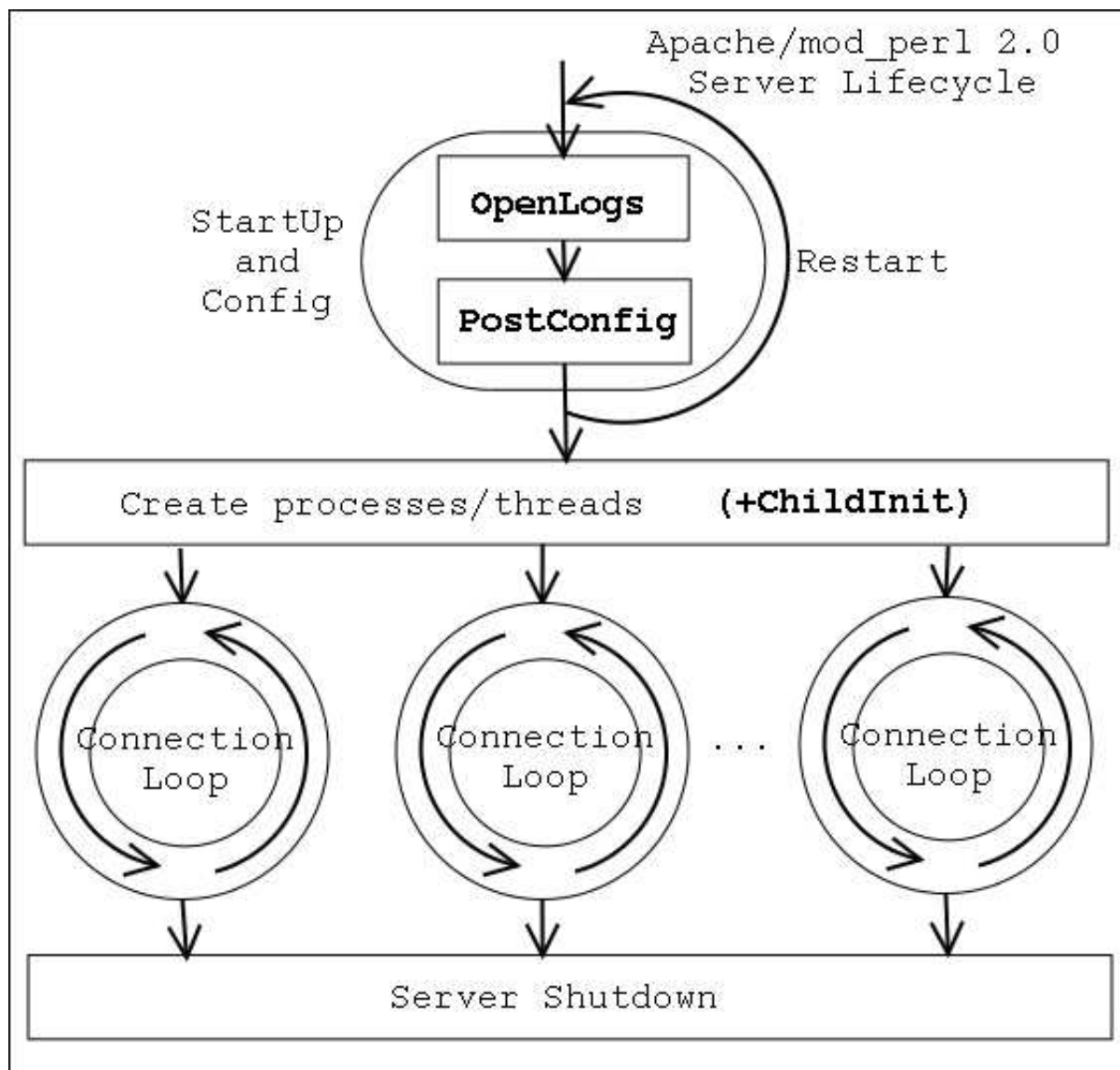
The *worker* MPM implements a hybrid multi-process multi-threaded approach based on the *pthreads* standard. It uses one acceptor thread, multiple worker threads.

- **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.

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.

The following diagram depicts the Apache 2.0 server life cycle and highlights which handlers are available to mod_perl 2.0:

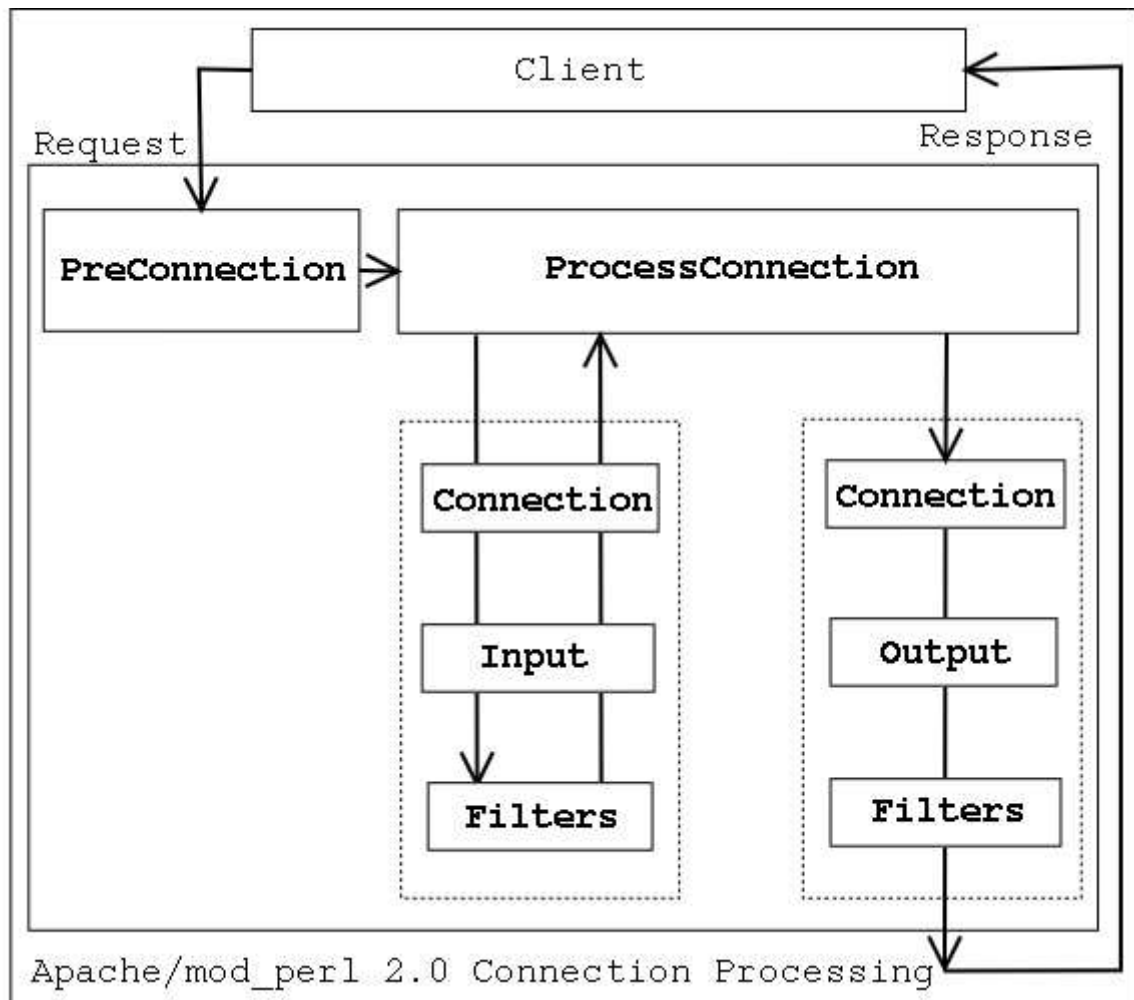


The spawned processes and threads that serve the requests depend on the chosen MPM.

- **Protocol Modules**

The previous Apache generation could speak only the HTTP protocol. Apache 2.0 has introduced a "server framework" architecture making it possible to plug in 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.

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



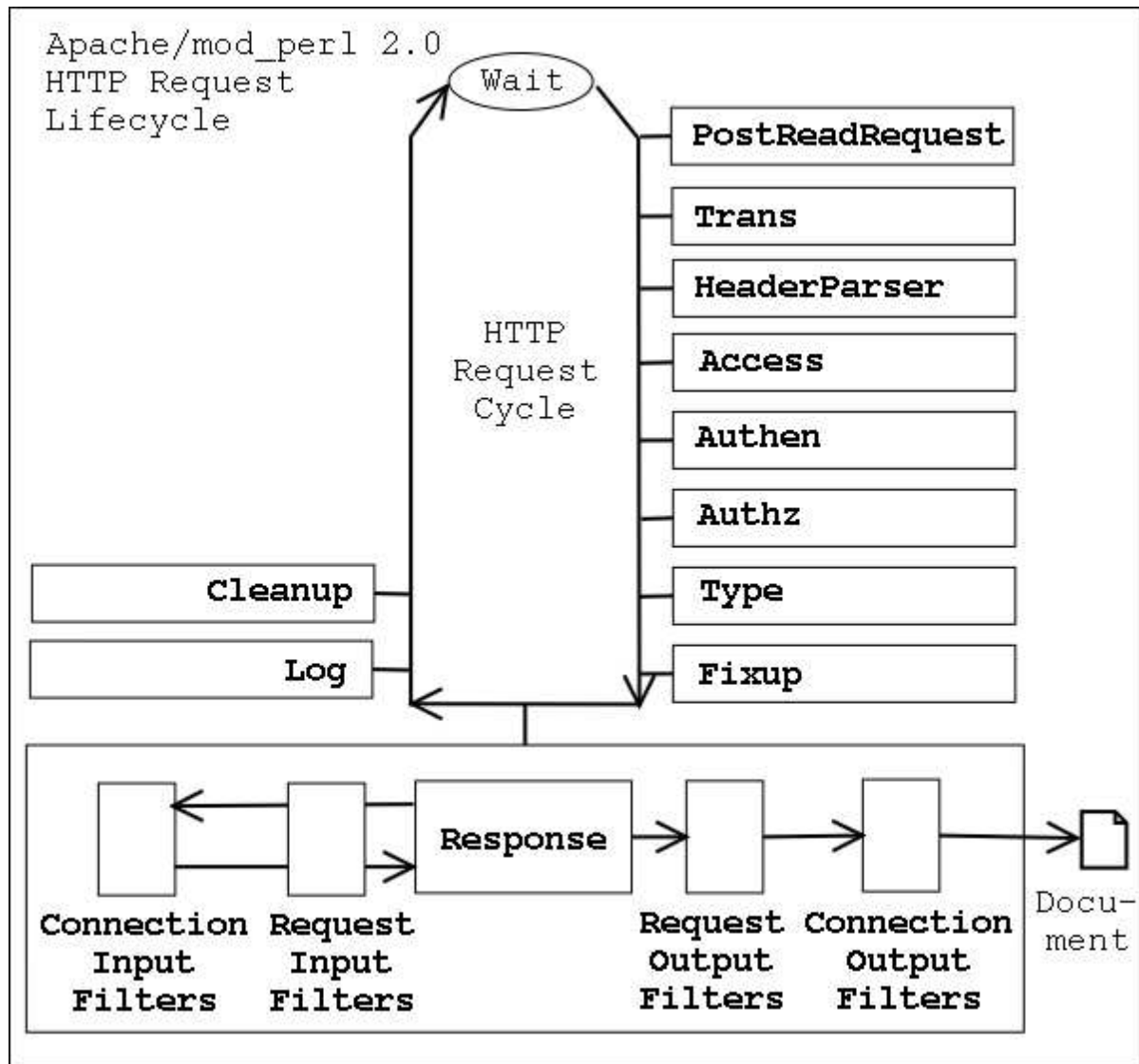
The protocol modules are plugged in the PreConnection and ProcessConnection stages.

- **I/O Filtering**

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, things like SSL, data (de-)compression and other manipulations are done very easily.

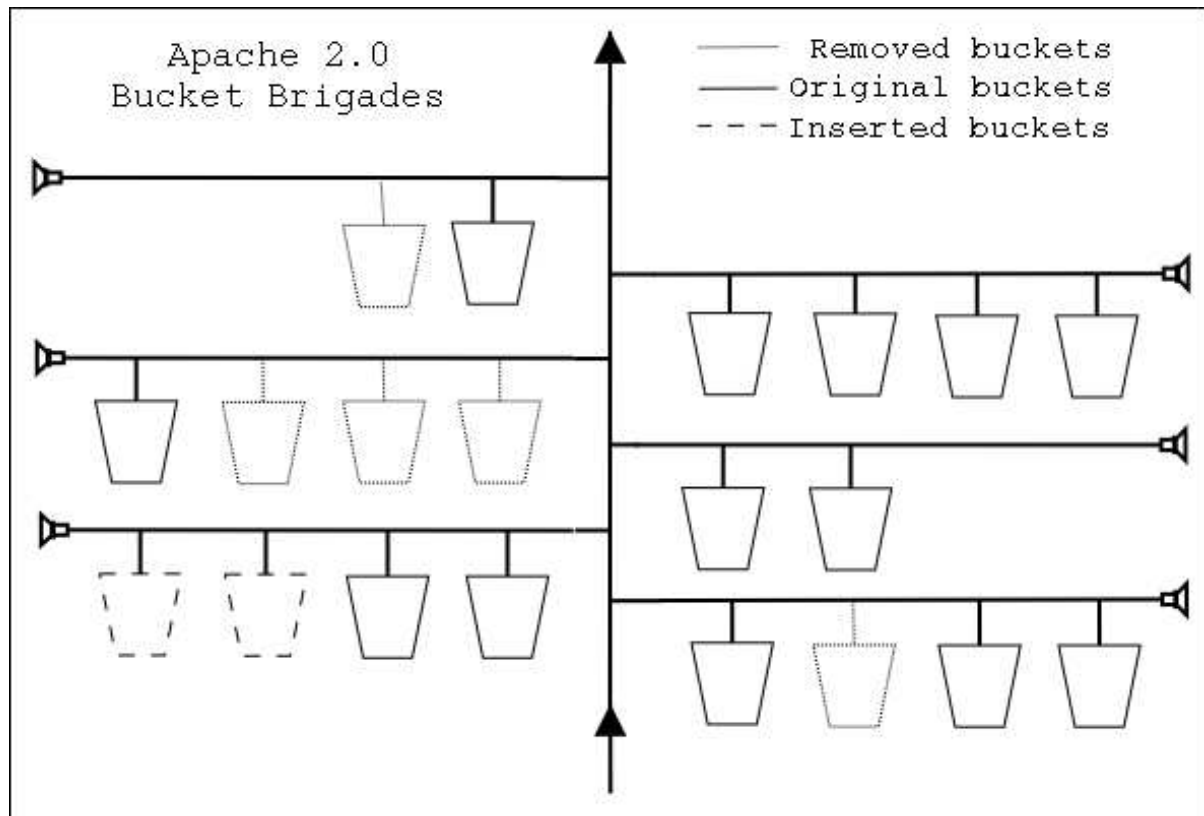
The following diagram depicts the HTTP request life cycle and highlights which handlers are available to mod_perl 2.0:



The I/O filtering is added to the response phase.

The I/O filtering is based on the concept of bucket brigades and implemented in the APR.

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.

- **New Hook Scheme**

In Apache 2.0 it's possible to dynamically register functions for each Apache hook, and allows more than one function to be registered per hook. Moreover when adding new functions, it's possible to specify where the new function should be added, e.g. a function can be pushed between two already registered functions or in front of them.

- **Parsed Configuration Tree**

Apache 2.0 makes the parsed configuration tree available at run time, so modules needing to read the configuration data (e.g., mod_info) don't have to re-parse the configuration file, but can re-use the parsed 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.

1.6 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* manpage.

The 5.6 Perl generation has introduced the following features:

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

- Multiple performance enhancements were made. See the *perl561delta* manpage.
- Numerous memory leaks were fixed. See the *perl561delta* manpage.
- Improved security features: more potentially unsafe operations taint their results for improved security. See the *perlsec* and *perl561delta* 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 book is released 5.6.2 will be out.

The Perl 5.8.0 has introduced the following features:

- The introduced in 5.6.0 experimental PerlIO layer has been stabilized and become the default IO layer in 5.8.0. Now the IO stream can be filtered through multiple layers. See the *perlapio* and *perliol* 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.

- More overridable via `CORE:::GLOBAL:::` keywords. See the *perlsub* 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* manpage.

A new special variable `$_{^TAINT}` was introduced. It indicates whether taint mode is enabled. See the *perlvar* 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).

1.7 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.

1.7.1 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()` 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.

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.

1.7.2 Thread-safety

It's important to notice that the Perl "ithreads" implementation ensures that Perl code is thread safe, at least with respect to the Apache threads in which it is running. However, it does not ensure that extensions which call into third-party C/C++ libraries 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. In the case of non-thread-safe extensions, if it is not possible to fix those routines, care needs to be taken to serialize calls into such functions (either at the XS or Perl level). See the *perlthrtut* manpage.

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 different workarounds.

1.7.3 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, threads, processes, shared memory, etc.
- the Apache API, which handles issues specific to the web server.

mod_perl 2.0 provides its own very flexible special purpose XS code generator, which is capable of doing things none of the existing generators can handle. It's possible that in the future this generator will be generalized and used for other projects of a high complexity.

This generator creates the Perl glue code for the public APR and Apache API, almost without a need for any extra code, but a few thin wrappers to make the API more Perl-ish.

In particular, since APR can be used outside of Apache, the Perl `APR::` modules can be used outside of Apache as well.

1.7.4 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.

1.7.5 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

1.8 Installing mod_perl 2.0

Since as of this writing mod_perl 2.0 wasn't released yet, the installation instructions may change a bit, but the core should be the same.

1.8.1 Installing from Source

First download the latest stable sources of Apache 2.0, mod_perl 2.0 and Perl 5.8.0.

- mod_perl 2.0 from <http://perl.apache.org/dist/>.
- Apache 2.0 from <http://httpd.apache.org/dist/>.
- Perl 5.8.0 from <http://cpan.org/src/>.

You can always find the most up-to-date download information at <http://perl.apache.org/download/>

Next, build Apache 2.0:

1. Extract the source (as usual replace *x* with the correct version number):

```
panic% tar -xzvf httpd-2.0.xx
```

If you don't have GNU tar(1) use the appropriate tools and flags to extract the source.

2. Configure:

```
panic% cd httpd-2.0.xx
panic% ./configure --prefix=/home/httpd/httpd-2.0 --with-mpm=prefork
```

Adjust the `--prefix` option to a directory of your choice, where you want Apache 2.0 to be installed. If you want to use a different MPM, adjust the `--with-mpm` option. The easiest way to find all of the configuration options for Apache 2.0 is to run:

```
panic% ./configure --help
```

3. Finally, build and install:

```
panic% make && make install
```

If you don't have Perl 5.6.0 or higher installed or you need to rebuild it because you want to enable certain compile-time features, build Perl (we will assume that you build Perl 5.8.0):

1. Extract the source:

```
panic% tar -xzvf perl-5.8.0.tar.gz
```

2. Configure:

```
panic% cd perl-5.8.0
panic% ./Configure -des -Dprefix=$HOME/perl/perl-5.8.0 -Dusethreads
```

This configuration accepts all the defaults suggested by the *Configure* script and produces a terse output. The `-Dusethreads` option enables Perl ithreads. The `-Dprefix` option specifies a custom installation directory, which you may want to adjust. For example you may decide to install it in the default location provided by Perl, which is */usr/local* under most systems.

For a complete list of configuration options and for information on installation on non-Unix systems, refer to the `INSTALL` document.

3. Now build, test and install Perl.

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

Before proceeding with installation of `mod_perl 2.0`, it's advisable to install at least the LWP package into newly installed Perl so later you can fully test `mod_perl 2.0`. You can use `CPAN.pm` to accomplish that:

```
panic% $HOME/perl/perl-5.8.0/bin/perl -MCPAN -e 'install("LWP")'
```

Now that you have Perl 5.8.0 and Apache 2.0 installed we can proceed with `mod_perl 2.0` installation:

1. Extract the source:

```
panic% tar -xzvf mod_perl-2.0.x.tar.gz
```

2. Configure:

Remember the nightmare number of options for `mod_perl 1.0`? You only need one option to build `mod_perl 2.0`. If you need more control, read *install.pod* in the source `mod_perl` distribution or online at <http://perl.apache.org/docs/2.0/>.

```
panic% cd mod_perl-2.0.x
panic% perl Makefile.PL MP_AP_PREFIX=/home/stas/src/httpd-2.0.xx
```

The `MP_AP_PREFIX` option specifies the base location of the installed Apache 2.0 or its source directory where the Apache *include/* directory can be found.

3. Now build, test and install `mod_perl 2.0`:

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

On Win32 you have to use `nmake` instead of `make` and the `&&` chaining doesn't work on all Win32 platforms, so instead you should do:

```
C:\> nmake
C:\> nmake test
C:\> nmake install
```

1.8.2 Installing Binaries

Apache 2.0 binaries can be obtained from: <http://httpd.apache.org/dist/binaries/>.

Perl 5.6.1 or 5.8.0 binaries can be obtained from: <http://cpan.org/ports/index.html>.

For `mod_perl 2.0`, as of this writing only the binaries for the Win32 platform are available, kindly prepared and maintained by Randy Kobes.: Once `mod_perl 2.0` is released various OS distributions will provide a binary version for their platforms.

If you are not on Win32 platform you can safely skip to the next section.

There are two ways of obtaining a binary mod_perl-2 package for Win32:

- **PPM**

The first, for ActivePerl users, is through PPM - this assumes you already have ActivePerl (build 6xx) from <http://www.activestate.com/> and a Win32 Apache-2 binary from <http://httpd.apache.org/>. In installing this, you may find it convenient when transcribing any Unix-oriented documentation to choose installation directories that do not have spaces in their names (e.g., `C:\Apache2`).

After installing Perl and Apache-2, you can then install mod_perl 2.0 via the PPM utility. ActiveState does not maintain mod_perl in their ppm repository, so you must get it from a different location other than ActiveState's site. One way is simply as:

```
C:\> ppm install http://theoryx5.uwinnipeg.ca/ppmpackages/mod_perl-2.ppd
```

Another way, which will be useful if you plan on installing additional Apache modules, is to set the repository within the ppm shell utility as (here and afterwards broken over 2 lines for readability):

```
PPM> set repository theoryx5
      http://theoryx5.uwinnipeg.ca/cgi-bin/ppmserver?urn:/PPMServer
```

or, for ppm3:

```
PPM> rep add theoryx5
      http://theoryx5.uwinnipeg.ca/cgi-bin/ppmserver?urn:/PPMServer
```

mod_perl-2 can then be installed as:

```
PPM> install mod_perl-2
```

This will install the necessary modules under an *Apache2/* subdirectory in your Perl tree, so as not to disturb an existing *Apache/* directory from mod_perl-1. See the section below on configuring mod_perl to add this directory to the @INC path for searching for modules.

The mod_perl PPM package also includes the necessary Apache DLL `mod_perl.so`; a post-installation script should be run which will offer to copy this file to your Apache2 modules directory (e.g., `C:\Apache2\modules`). If this is not done, you can get the file `mod_perl-2.tar.gz` from <http://theoryx5.uwinnipeg.ca/ppmpackages/x86/> which, when unpacked, contains `mod_perl.so` in the top-level directory.

Note that the mod_perl package available from this site will always use the latest mod_perl sources compiled against the latest official Apache release; depending on changes made in Apache, you may or may not be able to use an earlier Apache binary. However, in the Apache Win32 world it is particularly a good idea to use the latest version, for bug and security fixes.

- **Apache/mod_perl binary**

At <ftp://theoryx5.uwinnipeg.ca/pub/other/> you can find an archive *Apache2.tar.gz* containing a binary version of Apache-2/mod_perl-2. This archive unpacks into an *Apache2* directory, underneath which is a *blib* subdirectory containing the necessary mod_perl files (enabled with a `PerlSwitches` directive in *httpd.conf*). Some editing of *httpd.conf* will be necessary to reflect the location of the installed directory. See the *Apache2.readme* file for further information.

This package, which is updated periodically, is compiled against recent cvs sources of Apache-2 and mod_perl-2. As such, it may contain features, and bugs, not present in the current official releases. Also for this reason, these may not be binary compatible with other versions of Apache-2/mod_perl-2.

1.9 Configuring mod_perl 2.0

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.

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.

Win32 users need to make sure that the path to the Perl binary (e.g., *C:\Perl\bin*) is in the `PATH` environment variable.

1.9.1 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.

1.9.2 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 ();

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 ;`.

1.9.3 Perl's Command Line Switches

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
```

The `-I` command switch now can be used to adjust `@INC` values:

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

For example you can use that technique to set different different `@INC` values for different virtual hosts as we will see later.

1.9.4 mod_perl 2.0 Core Handlers

`mod_perl 2.0` provides two types of core handlers: `modperl` and `perl-script`.

1.9.4.1 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 unless:

```
PerlOptions -GlobalRequest
```

is specified.

- `PerlOptions +SetupEnv` is in effect unless:

```
PerlOption -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->print()`.
- 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`).

1.9.4.2 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 and GATEWAY_INTERFACE (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 MyApache::VarTest
  PerlSetVar VarTest VarTestValue
</Location>
```

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

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

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

1.9.4.3 A Simple Response Handler Example

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::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`.

1.9.5 PerlOptions Directive

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:

1.9.5.1 Enable

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

```

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

```

1.9.5.2 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 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>
```

1.9.5.3 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`:

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

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

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

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

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

1.9.5.4 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 `Perl-LogHandler` becomes `Log` which can be used to disable `PerlLogHandler`:

```
PerlOptions -Log
```

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

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

Or maybe everything but the response handler:

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

1.9.5.5 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 MyApache::Magick
```

In this case, `MyApache::Magick` is the package name, and the subroutine name will default to *handler*. If the `MyApache::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 MyApache::Magick
```

in our example.

1.9.5.6 GlobalRequest

Setup the global `request_rec` for use with `Apache->request`. This setting is needed for example if you use `CGI.pm` to process the incoming request.

This setting is enabled by default for sections configured as:

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

And can be disabled with:

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

1.9.5.7 ParseHeaders

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

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

1.9.5.8 MergeHandlers

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

```
PerlFixupHandler MyApache::FixupA

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

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

```
PerlFixupHandler MyApache::FixupA

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

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

1.9.5.9 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
    PerlHandler 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*.

1.9.6 Threads Mode Specific Directives

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

1.9.6.1 PerlInterpStart

The number of interpreters to clone at startup time.

1.9.6.2 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()).

1.9.6.3 PerlInterpMinSpare

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

1.9.6.4 PerlInterpMaxSpare

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

1.9.6.5 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.

1.9.6.6 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 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 PerlFix-upHandler 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
```

1.9.7 Retrieving Server Startup Options

The httpd server startup options can be retrieved using `Apache::exists_config_define()`. For example to check whether the server has been started in a single mode:

```
% httpd -DONE_PROCESS
```

use:

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

1.10 New Apache Phases and Corresponding Perl*Handlers

Since the majority of the Apache phases supported by mod_perl haven't changed since mod_perl 1.0, in this section we will discuss only phases and the corresponding handlers that were added or changed in mod_perl 2.0.

1.10.1 Server Configuration (Startup) Phases

`open_logs`, configured with `PerlOpenLogsHandler`, and `post_config`, configured with `PerlPost-ConfigHandler`, are the two new phases available during the server startup.

1.10.1.1 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.

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).

The PerlOpenLogsHandler directive may appear in the main configuration files and within virtual host sections.

Apache will continue executing all registered for this phase handlers until the first handler returns something other than `Apache::OK` or `Apache::DECLINED`.

For example here is the `MyApache::OpenLogs` handler that opens a custom log file:

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

use strict;

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

use File::Spec::Functions;

my $log_file = catfile "logs", "mylog";

sub handler {
    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 my $log, ">>$log_path" or die "can't open $log_path: $!";
    return Apache::OK;
}
1;
```

The *open_logs* phase handlers accept four arguments: the configuration pool, the logging streams pool, the temporary pool and the server object. 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. Of course in the real world handlers the module needs to be extended to provide an accessor that can write to this log file.

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

```
PerlOpenLogsHandler MyApache::OpenLogs
```

1.10.1.2 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.

The *post_config* phase is exactly the same as the *open_logs* phase. The `PerlPostConfigHandler` directive may appear in the main configuration files and within virtual host sections. Apache will run all registered for this phase handlers until the first handler returns something other than `Apache::OK` or `Apache::DECLINED`. This phase's handlers receive the same four arguments as the *open_logs* phase's handlers:

```
sub handler {
    my ($conf_pool, $log_pool, $temp_pool, $s) = @_;
    # ...
    return Apache::OK;
}
```

1.10.2 Connection Phases

Since Apache 2.0 makes it possible to implement other than HTTP protocols, the connection phases *pre_connection*, configured with `PerlPreConnectionHandler`, and *process_connection*, configured with `PerlProcessConnectionHandler`, were added. The standard HTTP handlers normally don't need these phases, since HTTP is already handled by the request phases. Therefore these phases are used mostly for the implementation of non-HTTP protocol implementations.

1.10.2.1 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. The core server uses this phase to setup the connection record based on the type of connection that is being used.

For example this phase could be a good place to automatically reload modified perl modules during the development, similar to `Apache::Reload`.

Apache will continue executing all registered for this phase handlers until the first handler returns something other than `Apache::OK` or `Apache::DECLINED`.

The `PerlPreConnectionHandler` directive may appear in the main configuration files and within virtual host sections.

A *pre_connection* handler accepts connection record and socket objects as its arguments:

```

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

```

1.10.2.2 PerlProcessConnectionHandler

The *process_connection* phase is used to actually process the connection that was received. 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.).

Apache will continue executing all registered for this phase handlers until the first handler returns something other than `Apache::DECLINED`.

The `PerlPreConnectionHandler` directive may appear in the main configuration files and within virtual host sections.

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;
}

```

1.10.2.2.1 MyApache::Eliza Protocol Module

Apache 2.0 ships with an example protocol module, `mod_echo`, which simply reads data from the client and echos it right back. Here we'll take a look at a Perl version of that module, called `MyApache::Eliza`, with a twist--instead of echoing whatever was read back to the client, it sends the read data as an input to `Chatbot::Eliza`, which implements a mockery Rogerian psychotherapist, and forwards the response from the psychotherapist back to the client.

A protocol handler is configured using the `PerlProcessConnectionHandler` directive and we will use the `Listen` and `<VirtualHost>` directives to bind to a non-standard port **8084**:

```

Listen 8084
<VirtualHost _default_:8084>
    PerlModule MyApache::Eliza
    PerlProcessConnectionHandler MyApache::Eliza
</VirtualHost>

```

`MyApache::Eliza` is then enabled when starting Apache:

```
% httpd
```

And we give it a whirl:

```
% telnet localhost 8084
Trying 127.0.0.1...
Connected to localhost (127.0.0.1).
Escape character is '^]'.
Hello Eliza
How do you do. Please state your problem.

How are you?
Oh, I?

Why do I have core dumped?
You say Why do you have core dumped?

I feel like writing some tests today, you?
I'm not sure I understand you fully.

Good bye, Eliza
Does talking about this bother you?

Connection closed by foreign host.
```

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 a 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/print 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, be we need only 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 the "good bye" string. Otherwise if the handler receives some data, it sends this data to the `$eliza` object which represents the psychotherapist, whose returned text is then sent back to the client with the `APR::Socket::send()` method. When the read/print loop is finished the handler returns `Apache::OK`, telling Apache to terminate the connection.

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

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

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

require Chatbot::Eliza;

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

use constant BUFF_LEN => 1024;
```

```

my $eliza = new Chatbot::Eliza;

sub handler {
    my Apache::Connection $c = shift;
    my APR::Socket $socket = $c->client_socket;

    my $buff;
    my $last = 0;
    for (;;) {
        my($rlen, $wlen);
        $rlen = BUFF_LEN;
        $socket->recv($buff, $rlen);
        last if $rlen <= 0;

        # \r is sent instead of \n if the client is talking over telnet
        $buff =~ s/[\r\n]*$//;
        $last++ if $buff =~ /good bye/i;
        $buff = $eliza->transform( $buff ) . "\n\n";
        $socket->send($buff, length $buff);
        last if $last;
    }

    Apache::OK;
}

1;

```

1.10.2.2.2 *MyApache::Eliza2 Protocol Module*

The previous implementation can't work with filters. The following implementation uses bucket brigades and therefore can use I/O filters.

```

file:MyApache/Eliza2.pm
-----
package MyApache::Eliza2;

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

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

require Chatbot::Eliza;

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

my $eliza = new Chatbot::Eliza;

sub handler {
    my Apache::Connection $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 "[eliza] 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) {
            $data =~ s/[\r\n]*$//;
            $last++ if $data =~ /good bye/i;
            $data = $eliza->transform( $data ) . "\n\n";
            $bucket = APR::Bucket->new($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 : FilterConnectionHandler {
    my $filter = shift;

    while ($filter->read(my $buffer, BUFF_LEN)) {
        $filter->print(lc $buffer);
    }
}

```

```

    }

    return Apache::OK;
}

1;

```

And the corresponding configuration:

```

Listen 8085
<VirtualHost _default_:8085>
    PerlModule MyApache::Eliza2
    PerlProcessConnectionHandler MyApache::Eliza2
    PerlOutputFilterHandler MyApache::Eliza2::lowercase
</VirtualHost>

```

`MyApache::Eliza2::lowercase` lowers the case of the response sent by Eliza, so if we run the same session we might see:

```

% telnet localhost 8085
Trying 127.0.0.1...
Connected to localhost (127.0.0.1).
Escape character is '^]'.
Hello Eliza
how do you do. please state your problem.

How are you?
oh, i?

Why do I have core dumped?
you say why do you have core dumped?

I feel like writing some tests today, you?
i'm not sure i understand you fully.

Good bye, Eliza
does talking about this bother you?

Connection closed by foreign host.

```

1.10.3 Request Phases

In the request phase nothing has changed, other than renaming the `PerlHandler` directive to `PerlResponseHandler` to better match the corresponding Apache phase name (*response*).

1.10.4 I/O Filtering Phases

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*. Both input and output filters massage these bucket brigades and modify them if necessary.

As of this writing the mod_perl filtering API hasn't been finalized yet, and it's possible that it will change by the time the production version of mod_perl 2.0 is released. However most concepts presented here won't change, and you should find the discussion and the examples useful for understanding how filters work.

mod_perl provides two interfaces to filtering: a direct mapping to buckets and bucket brigades and a simpler, stream-oriented interface (as of this writing the latter is available only for the output filtering). The following examples will help to understand the difference between the two. The filters can do connection and request filtering. Apache distinguish between more types, and mod_perl will support those in the future. mod_perl handlers specify the type of the filter using the subroutine attributes. For example a request filter handler is declared using the `FilterRequestHandler` attribute:

```
package MyApache::InputRequestFilterFoo;
sub handler : FilterRequestHandler {
    my($filter, $bb, $mode, $block, $readbytes) = @_;
    #...
}
1;
```

and is usually configured in the `<Location>` or equivalent sections:

```
<Location /input_filter>
    SetHandler modperl
    PerlResponseHandler MyApache::NiceResponse
    PerlInputFilterHandler MyApache::InputRequestFilterFoo
</Location>
```

And as you can guess a connection handler uses the `FilterConnectionHandler` attribute (this time we use the output filter as an example):

```
package MyApache::OutputConnectionFilterBar;
sub handler : FilterConnectionHandler {
    my($filter, $bb, $mode, $block, $readbytes) = @_;
    #...
}
1;
```

and configured outside the `<Location>` or equivalent sections, usually within the `<VirtualHost>` or equivalent sections:

```
Listen 8005
<VirtualHost _default_:8005>
    PerlOutputFilterHandler MyApache::OutputConnectionFilterBar
    <Location />
        SetHandler modperl
        PerlResponseHandler MyApache::NiceResponse
    </Location>
</VirtualHost>
```

1.10.4.1 PerlInputFilterHandler

The `PerlInputFilterHandler` handler registers a filter for input filtering.

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. So here is the input filter handler that does that:

```
file:MyApache/InputFilterGET2HEAD.pm
-----
package MyApache::InputFilterGET2HEAD;

use strict;
use warnings;

use base qw(Apache::Filter);

use Apache::RequestRec ();
use Apache::RequestIO ();
use APR::Brigade ();
use APR::Bucket ();

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

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

    my $c = $filter->c;
    my $ctx_bb = APR::Brigade->new($c->pool, $c->bucket_alloc);
    my $rv = $filter->next->get_brigade($ctx_bb, $mode, $block, $readbytes);
    return $rv unless $rv == APR::SUCCESS;

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

        $bucket->remove;

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

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

        if ($data and $data =~ s|^GET|HEAD|) {
```

```

        $bucket = APR::Bucket->new($data);
    }

    $bb->insert_tail($bucket);
}

Apache::OK;
}

1;

```

The filter handler is called for each bucket brigade, which in turn includes buckets with data. The gist of any filter handler is to retrieve the bucket brigade sent from the previous filter, prepare a new empty brigade, and move buckets from the former brigade to the latter optionally modifying the buckets on the way, which may include removing or adding new buckets. Of course if the filter doesn't want to modify any of the buckets it may decide to path through the original brigade without doing any work.

In our example the handler first removes the bucket at the top of the brigade and looks at its type. If it sees an end of stream, that removed bucket is linked to the tail of the bucket brigade that will go to the next filter and it doesn't attempt to read any more buckets. If this event doesn't happen the handler reads the data from that bucket and if it finds that the data is of interest to us, it modifies the data, creates a new bucket using the modified data and links it to the tail of the outgoing brigade, while discarding the original bucket. In our case the interesting data is a such that matches the regex `/^GET/`. If the data is not interesting to the handler, it simply links the unmodified bucket to the outgoing brigade.

The handler looks for data like:

```
GET /perl/test.pl HTTP/1.1
```

and turns it into:

```
HEAD /perl/test.pl HTTP/1.1
```

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

```

the response 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 you can see that if Apache wasn't discarding the body on HEAD, the response would be:

the request type was HEAD

that's why the body length is 25 and not 24 as in the previous case.

1.10.4.2 PerlOutputFilterHandler

The `PerlOutputFilterHandler` handler registers and configures output filters.

Let's look at the example of a stream-oriented output filter and see how it works through this example.

`MyApache::ROT13` implements the simple Caesar-cypher encryption that replaces each English letter with the one 13 places forward or back along the alphabet, so that *"mod_perl 2.0 rules!"* becomes *"zbg_crey 2.0 ehryf!"*. Since the English alphabet consists of 26 letters, the ROT13 encryption is self-inverse, so the same code can be used for encoding and decoding. In our example `MyApache::ROT13` reads portions of the output generated by some previous handler, rotates the characters and sends them downstream.

The first argument to a filter handler is an `Apache::Filter` object, which as of this writing provides two methods *read* and *print*. The *read* method reads a chunk of the output stream into the given buffer, returning the number of characters read. An optional size argument may be given to specify the maximum size to read into the buffer. If omitted, an arbitrary size will fill the buffer, depending on the upstream filter or handler. The *print* method passes data down to the next filter.

```
file:MyApache/ROT13.pm
-----
package MyApache::ROT13;

use strict;

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

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

use constant BUFF_LEN => 1024;

sub handler {
    my $filter = shift;

    while ($filter->read(my $buffer, BUFF_LEN)) {
        $buffer =~ y/A-Za-z/N-ZA-Mn-za-m/;
        $filter->print($buffer);
    }

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

Let's say that we want to encrypt the output of the registry scripts accessed through a */perl-rot13* location using the rot13 algorithm. The following configuration section accomplishes that.

```

PerlModule MyApache::ROT13
Alias /perl-rot13/ /home/httpd/perl/
<Location /perl-rot13>
    SetHandler perl-script
    PerlResponseHandler ModPerl::Registry
    PerlOutputFilterHandler MyApache::ROT13
    Options +ExecCGI
    #PerlOptions +ParseHeaders
</Location>

```

Now that you know how to write input and output filters, you can write a pair of filters that decode ROT13 input before the request processing starts and then encode the generated response back to ROT13 on the way back to the client.

1.11 Migrating from mod_perl 1.0 to mod_perl 2.0

The following sections discuss what should be done in order to migrate services from mod_perl 1.0 to 2.0 and if possible making the new services based on mod_perl 2.0 back compatible with mod_perl 1.0.

Several configuration directives were 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 and load those new packages containing them.

Since as of this writing mod_perl 2.0 wasn't released yet, it's possible that certain things have changed after the document has been published. If something doesn't work as explained here, please refer to the documents in the mod_perl distribution or the online version at <http://perl.apache.org/docs/2.0/> for the updated documentation.

1.11.1 *The Shortest Migration Path*

mod_perl 2.0 provides two backwards-compatibility layers: one for the configuration files and the other for the code. If you are concerned to preserve the backwards compatibility with mod_perl 1.0, or simply want to try your services under mod_perl 2.0, simply enable the code compatibility layer by adding:

```

use Apache2;
use Apache::compat;

```

at the top of your startup file. The configuration backwards-compatibility is enabled by default.

1.11.2 *Migrating Configuration Files*

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. If you don't need the backwards compatibility consider using the directives that have replaced them.

1.11.2.1 PerlHandler

PerlHandler was replaced with PerlResponseHandler.

1.11.2.2 PerlSendHeader

PerlSendHeader was replaced with PerlOptions +/-ParseHeaders directive.

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

1.11.2.3 PerlSetupEnv

PerlSetupEnv was replaced with PerlOptions +/-SetupEnv directive.

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

1.11.2.4 PerlTaintCheck

The tainting mode now can be turned on with:

```
PerlSwitches -T
```

The default is *Off*. You cannot turn it *Off* once it's turned *On*.

1.11.2.5 PerlWarn

Warnings now can be enabled globally with:

```
PerlSwitches -w
```

1.11.2.6 PerlFreshRestart

PerlFreshRestart is a mod_perl 1.0 legacy and doesn't exist in mod_perl 2.0. A full tear-down 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>
```

1.11.3 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. The online `mod_perl` documentation includes a document (<http://perl.apache.org/docs/2.0/user/compat/compat.html>) that 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`.

1.11.4 ModPerl::Registry Family

In mod_perl 2.0, Apache::Registry and friends (Apache::PerlRun, Apache::RegistryNG, etc) have migrated into the ModPerl:: namespace. The new family is based on the idea of Apache::RegistryNG from mod_perl 1.0, where you can customize pretty much all the functionality by providing your own hooks. The functionality of the modules Apache::Registry, Apache::RegistryBB and Apache::PerlRun hasn't changed from the user's perspective. All these modules are derived from the Apache::RegistryCooker class. So if you want to change the functionality of any of the existing sub-classes, or want to "cook" your own registry module it can be done easily. Refer to the Apache::RegistryCooker manpage for more information.

Here is a typical registry section configuration in mod_perl 2.0:

```
Alias /perl/ /home/httpd/perl/
<Location /perl>
    SetHandler perl-script
    PerlResponseHandler ModPerl::Registry
    Options +ExecCGI
    PerlOptions +ParseHeaders
</Location>
```

As we have explained earlier, the ParseHeaders option is needed if the headers are being sent via print() (i.e. without using mod_perl API) and comes as a replacement for the PerlSendHeader option in mod_perl 1.0.

Here is a simple registry script that prints the environment variables.

```
file:print_env.pl
-----
print "Content-type: text/plain\n\n";
for (sort keys %ENV){
    print "$_ => $ENV{$_}\n";
}
```

Save the file in */home/httpd/perl/print_env.pl* and make it executable:

```
panic% chmod 0700 /home/stas/modperl/mod_perl_rules1.pl
```

Now issue a request to *http://localhost/perl/print_env.pl* and you should see all the environment variables printed out.

The only change for registry scripts from mod_perl 1.0 is that Perl doesn't chdir()'s into the script's directory before executing it. This is because chdir() is not a thread-safe function, and as you've learned by now, mod_perl 2.0 should run properly in the threaded environment. To accommodate for this change, the directory of the script is being pushed as the first element in @INC for the duration of the script's execution, so relative to the script's directory require() calls will succeed. This however doesn't solve the problem for other operations like file open() calls, when a relative to the script's directory path is used. In these cases the code needs to be changed to figure out the full path to the file at run time.

1.11.5 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) = @_;
    ...;
}
```

Starting from Perl version 5.6, you can use subroutine attributes, and that's what mod_perl 2.0 does instead of conventional prototypes:

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

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

See the *attributes* manpage.

mod_perl 2.0 doesn't support the (\$\$) prototypes, mainly because several callbacks in 2.0 have more arguments than \$r, so the (\$\$) prototype doesn't make sense anymore. Therefore if you want your code to work with both mod_perl generations, you should use the subroutine attributes.

1.11.6 Apache::StatINC Replacement

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.

1.12 Important Links

- All the updated docs are at <http://perl.apache.org/docs/>
- If you have any questions please ask at the mod_perl mailing list: modperl@perl.apache.org

1.13 A shameless plug

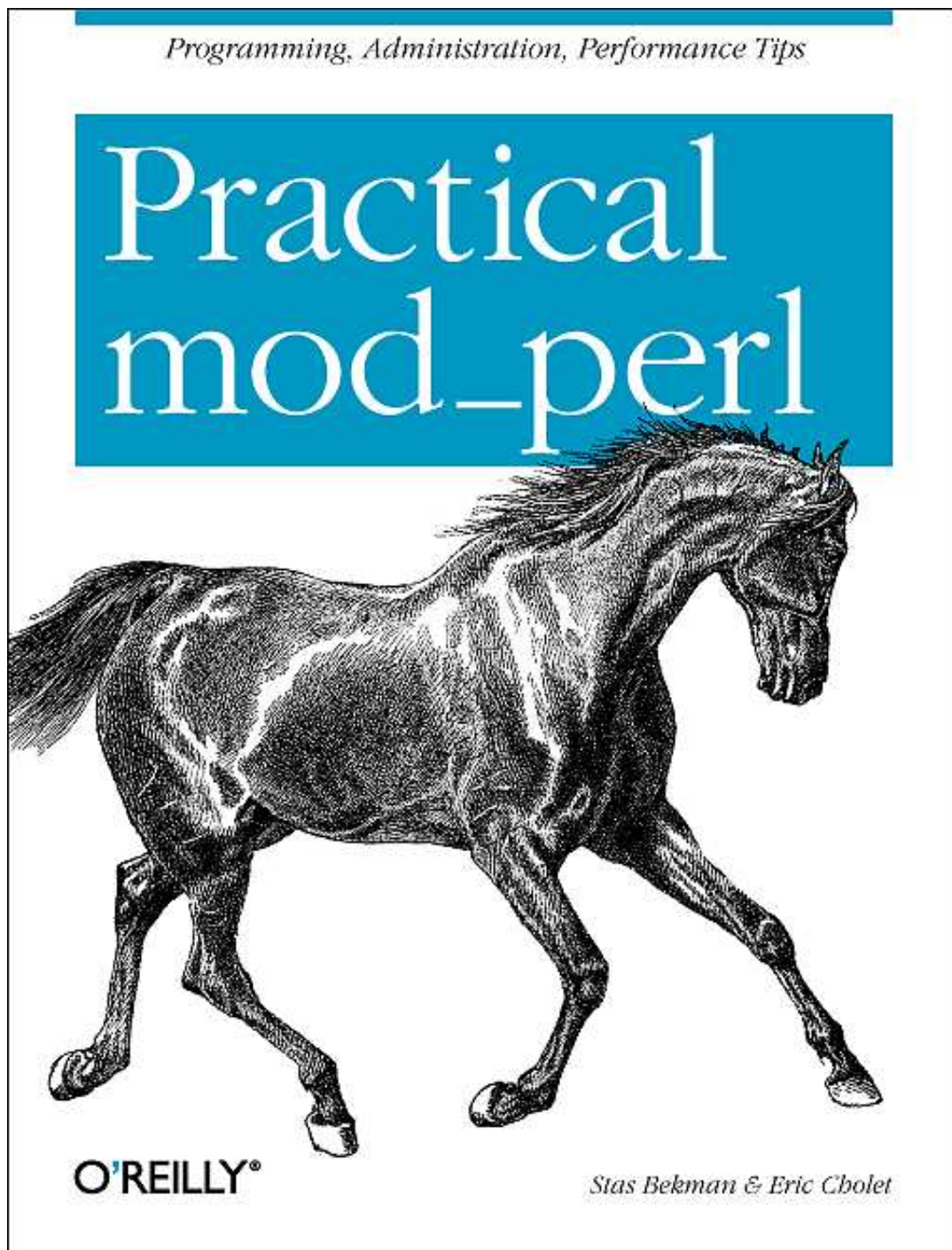


Table of Contents:

Presentation Handouts: mod_perl 2.0, the Next Generation	1
The Next Generation: mod_perl 2.0	2
1 The Next Generation: mod_perl 2.0	2
1.1 About	3
1.2 Thank you!	3
1.3 Versioning Convention	3
1.4 Why mod_perl, the Next Generation	3
1.4.1 The Apache::Test Framework	4
1.5 What's new in Apache 2.0	4
1.6 What's new in Perl 5.6.0 - 5.8.0	10
1.7 What's new in mod_perl 2.0	12
1.7.1 Threads Support	12
1.7.2 Thread-safety	13
1.7.3 Perl interface to the APR and Apache APIs	13
1.7.4 Other New Features	14
1.7.5 Optimizations	14
1.8 Installing mod_perl 2.0	14
1.8.1 Installing from Source	15
1.8.2 Installing Binaries	16
1.9 Configuring mod_perl 2.0	18
1.9.1 Accessing the mod_perl 2.0 Modules	18
1.9.2 Startup File	19
1.9.3 Perl's Command Line Switches	20
1.9.4 mod_perl 2.0 Core Handlers	20
1.9.4.1 perl-script	20
1.9.4.2 modperl	21
1.9.4.3 A Simple Response Handler Example	21
1.9.5 PerlOptions Directive	23
1.9.5.1 Enable	23
1.9.5.2 Clone	23
1.9.5.3 Parent	24
1.9.5.4 Perl*Handler	24
1.9.5.5 AutoLoad	25
1.9.5.6 GlobalRequest	25
1.9.5.7 ParseHeaders	26
1.9.5.8 MergeHandlers	26
1.9.5.9 SetupEnv	26
1.9.6 Threads Mode Specific Directives	28
1.9.6.1 PerlInterpStart	28
1.9.6.2 PerlInterpMax	28
1.9.6.3 PerlInterpMinSpare	28
1.9.6.4 PerlInterpMaxSpare	28
1.9.6.5 PerlInterpMaxRequests	28
1.9.6.6 PerlInterpScope	28

1.9.7 Retrieving Server Startup Options	29
1.10 New Apache Phases and Corresponding Perl*Handlers	29
1.10.1 Server Configuration (Startup) Phases	29
1.10.1.1 PerlOpenLogsHandler	30
1.10.1.2 PerlPostConfigHandler	31
1.10.2 Connection Phases	31
1.10.2.1 PerlPreConnectionHandler	31
1.10.2.2 PerlProcessConnectionHandler	32
1.10.2.2.1 MyApache::Eliza Protocol Module	32
1.10.2.2.2 MyApache::Eliza2 Protocol Module	34
1.10.3 Request Phases	36
1.10.4 I/O Filtering Phases	36
1.10.4.1 PerlInputFilterHandler	38
1.10.4.2 PerlOutputFilterHandler	41
1.11 Migrating from mod_perl 1.0 to mod_perl 2.0	42
1.11.1 The Shortest Migration Path	42
1.11.2 Migrating Configuration Files	42
1.11.2.1 PerlHandler	43
1.11.2.2 PerlSendHeader	43
1.11.2.3 PerlSetupEnv	43
1.11.2.4 PerlTaintCheck	43
1.11.2.5 PerlWarn	43
1.11.2.6 PerlFreshRestart	43
1.11.3 Code Porting	43
1.11.4 ModPerl::Registry Family	45
1.11.5 Method Handlers	46
1.11.6 Apache::StatINC Replacement	46
1.12 Important Links	46
1.13 A shameless plug	47