

CHAPTER 24

mod_perl 2.0: Installation and Configuration

Since Doug MacEachern introduced mod_perl 1.0* in 1996, he has had to tweak it with every change in Apache and Perl, while maintaining compatibility with the older versions. These rewrites have led to very complex source code, with hundreds of #ifdefs and workarounds for various incompatibilities in older Perl and Apache versions.

Apache 2.0, however, is based on a new threads design, requiring that mod_perl be based on a thread-safe Perl interpreter. Perl 5.6.0 was the first Perl version to support internal thread-safety across multiple interpreters. Since Perl 5.6.0 and Apache 2.0 are the very minimum requirements for the newest version of mod_perl, backward compatibility was no longer a concern, so this seemed like a good time to start from scratch. mod_perl 2.0 was the result: a leaner, more efficient mod_perl that's streamlined for Apache 2.0.

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

Many other interesting changes 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 chapter, and some you will discover on your own while reading mod_perl documentation.

At the time of this writing, mod_perl 2.0 is considered beta when used with the *pre-fork* Multi-Processing Model module (MPM) and alpha when used with a threaded

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

MPM. It is likely that Perl 5.8.0+ will be required for mod_perl 2.0 to move past alpha with threaded MPMs. Also, the Apache 2.0 API hasn't yet been finalized, so it's possible that certain examples in this chapter may require modifications once production versions of Apache 2.0 and mod_perl 2.0 are released.

In this chapter, we'll first discuss the new features in Apache 2.0, Perl 5.6 and later, and mod_perl 2.0 (in that order). Then we'll cover the installation and configuration of mod_perl 2.0. Details on the new functionality implemented in mod_perl 2.0 are provided in Chapter 25.

What's New in Apache 2.0

Whereas Apache 1.2 and 1.3 were based on the NCSA *httpd* code base, Apache 2.0 rewrote big chunks of the 1.3 code base, mainly to support numerous new features and enhancements. Here are the most important new features:

Apache Portable Runtime (APR)

The APR presents a standard API for writing portable client and 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 efficient way. One important effect is that it significantly simplifies the code that uses the APR, making it much easier to review and understand the Apache code, and 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 memory leaks (which always haunt C programmers).

I/O filtering

Apache 2.0 allows multiple modules to filter both the request and the response. Now one module can pipe its output to another module as if it were being sent directly from the TCP stream. The same mechanism works with the generated response.

With I/O filtering in place, simple filters (e.g., data compression and decompression) can easily be implemented, and complex filters (e.g., SSL) can now be implemented without needing to modify the the server code (unlike with Apache 1.3).

To make the filtering mechanism efficient and avoid unnecessary copying, the *bucket brigades* model was used, as follows.

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: files, data blocks, end-of-stream indicators, pools, etc. You don't need to know anything about the internal representation of a bucket in order to manipulate it.

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.

Figure 24-1 depicts an imaginary bucket brigade. The figure shows that after the presented bucket brigade has passed through several filters, some buckets were removed, some were modified, and some were added. Of course, the handler that gets the brigade doesn't know the history of the brigade; it can only see the existing buckets in the brigade. We will see bucket brigades in use when discussing protocol handlers and filters.

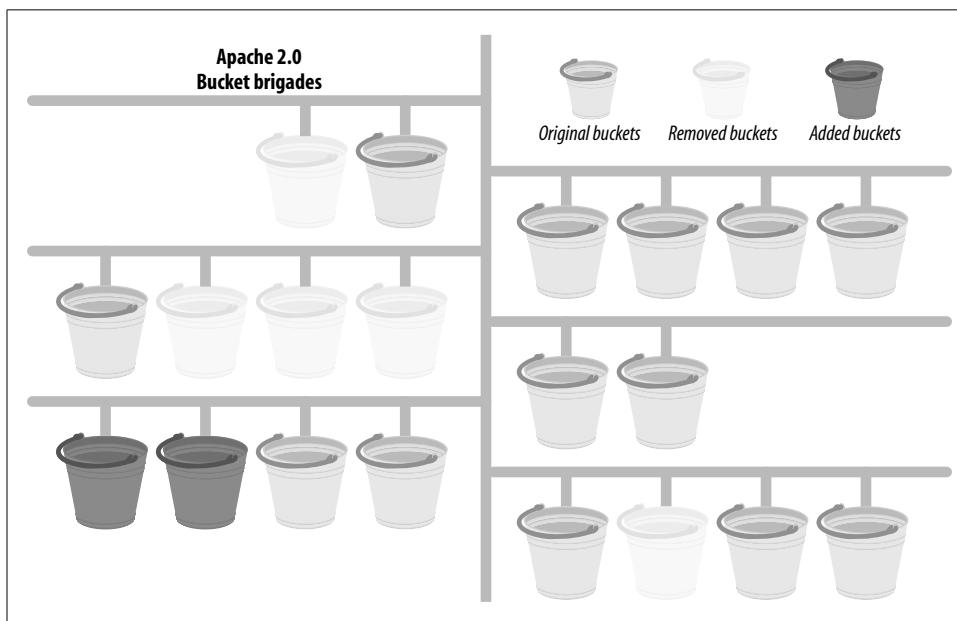


Figure 24-1. Imaginary bucket brigade

Multi-Processing Model modules (MPMs)

In the previous Apache generation, the same code base was trying to manage incoming requests for different platforms, which led to scalability problems on certain (mostly non-Unix) platforms. This also led to an undesired complexity of the code.

Apache 2.0 introduces the concept of MPMs, 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, Apache 2.0 on Windows is much more efficient and maintainable now, since it uses *mpm_winnt*, which deploys native Windows features.

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

prefork

The *prefork* MPM implements Apache 1.3's preforking model, in which 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 *pthread*s standard.

mpmt_os2, netware, winnt, and beos

These MPMs also implement the hybrid multi-process/multi-threaded model, like *worker*, but unlike *worker*, each is based on the native OS thread implementations, while *worker* uses the *pthread* library available on Unix.

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

New hook scheme

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

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, NNTP, POP3, RPC flavors, and the like. The main advantage is that protocol plug-ins can take advantage of Apache's portability, process/thread management, configuration mechanism, and plug-in API.

GNU Autoconf-based configuration

Apache 2.0 uses the ubiquitous GNU *Autoconf* for its configuration process, to make the configuration process more portable.

Parsed configuration tree

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

All these new features boost Apache's 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 already greatly optimized API.

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

And, of course, an important impact of these features is the simplification and added flexibility for the core and third-party Apache module developers.

What's New in Perl 5.6.0–5.8.0

As mentioned earlier, Perl 5.6.0 is the minimum requirement for `mod_perl 2.0`. However, certain new features work only with Perl 5.8.0 and higher.

The following are the important changes in the recent Perl versions that had an impact on `mod_perl`. For a complete list of changes, see the appropriate *perldelta* manpage. The 5.6 generation of Perl 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 (with the `:method` attribute). See the *attributes* manpage.
- The warnings pragma, which allows programmers 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 fine control over what warnings should be reported. See the *perllexwarn* manpage.
- Certain `CORE::` functions can now be overridden via the `CORE::GLOBAL::` namespace. For example, `mod_perl` now can override `exit()` globally by defining `CORE::GLOBAL::exit`. So when `exit()` is called, `CORE::GLOBAL::exit()` gets invoked. Note that you can still use `CORE::exit()` to get the original behavior. See the *perlsub* manpage.
- The *XSLoader* extension as a simpler alternative to *DynaLoader*. See the *XSLoader* manpage.
- Large-file support. If you have filesystems that support files larger than 2 GB), you may now also be able to create and access them from Perl. See the *perl561delta* manpage.

- Multiple performance enhancements. 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.
- Perl is now available on new platforms: GNU/Hurd, Rhapsody/Darwin, and EPOC.

Overall, multiple bugs and problems were fixed in 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, which will then incorporate the bug fixes from Perl 5.8.0.

Perl 5.8.0 has introduced the following features:

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

For example, this allows `mod_perl` to interoperate with the APR I/O layer and even use the APR I/O layer in Perl code. See the *APR::PerlIO* manpage.

Another example of using this 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 keywords are now overridable via `CORE::GLOBAL::`. 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 an inconsistent state. Now Perl delays signal handling until it is safe.
- `File::Temp` was added to allow 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 meant only as a temporary debugging aid while securing the code of old legacy applications. It 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.
- Thread implementation is much improved since 5.6.0. The Perl interpreter should now be completely thread-safe, and 5.8.0 marks the arrival of the threads module, which allows Perl programs to work with threads (creating them, sharing variables, etc.).
- Much better support for Unicode has been added.

- Numerous bugs and memory leaks have been fixed. For example, now you can localize the tied Apache: :DBI database handles without leaking memory.
- Perl is now available on new platforms: AtheOS, Mac OS Classic, MinGW, NCR MP-RAS, NonStop-UX, NetWare, and UTS. Also, the following platforms are again supported: BeOS, DYNIX/ptx, POSIX-BC, VM/ESA, and z/OS (OS/390).


What's New in mod_perl 2.0

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

Thread 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 is enabled at compile time and ensures that each Perl interpreter instance is reentrant—that is, multiple Perl interpreters can be used concurrently within the same process without locking, as each instance has its own copy of any mutable data (symbol tables, stacks, etc.). This of course requires that each Perl interpreter instance is accessed by only 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 in mod_perl 1.0, where the main interpreter created at startup time compiles any preloaded Perl code. A *clone* is created from the parent using the Perl API `perl_clone()` function. At request time, parent interpreters are used only for making more clones, as the clones are the interpreters that 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 creating a `PerlInterpreter` for each 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 just mod_perl requests, having a smaller number of `PerlInterpreters` than the number of threads will clearly cut down on memory usage. Finally, perhaps the biggest win is memory reuse: 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 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 it's put back at the head of the same list when it's done. This means that if, for example, you have ten interpreters configured to be cloned at startup time, but no more than five are ever used concurrently, those five continue to reuse Perl's allocations, while the other five 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, currently used to manage a pool of `PerlInterpreter` objects, can be used to manage any data structure in which you wish to have a smaller number of items than the number of configured threads.

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 functions and extensions that call into third-party C/C++ libraries are thread-safe. 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 Perl 5.8.0's *perlthrtut* manpage.

Note that while Perl data is thread-private unless explicitly shared and threads themselves are separate execution threads, the threads can affect process-scope state, affecting all the threads. For example, if one thread does `chdir("/tmp")`, the current working directory of all threads is now `/tmp`. While each thread can correct its current working directory by storing the original value, there are functions whose process-scope changes cannot be undone. For example, `chroot()` changes the root directory of all threads, and this change is not reversible. Refer to the *perlthrtut* manpage for more information.

Perl Interface to the APR and Apache APIs

As we mentioned earlier, Apache 2.0 uses two APIs:

- The Apache Portable Runtime (APR) API, which implements a portable and efficient API to 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 APIs, almost without a need for any extra code (just a few thin wrappers to make the API more Perl-ish).

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

Other New Features

In addition to the already mentioned new features in `mod_perl 2.0`, the following are of 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 simple-to-use interface to the Apache filtering API; this is of great interest because in `mod_perl 1.0` the `Apache::Filter` and `Apache::OutputChain` modules, used for filtering, had to go to great lengths to implement filtering and couldn't be used for filtering output generated by non-Perl modules. Moreover, incoming-stream filtering has now become possible. We will discuss filtering and see a few examples later on.
- A feature-full and flexible `Apache::Test` framework was developed especially for `mod_perl` testing. While intended to test the core `mod_perl` features, it is also used by third-party module writers to easily test their modules. Moreover, `Apache::Test` was adopted by Apache and is currently used to test the Apache 1.3, 2.0, and other ASF projects. Anything that runs on top of Apache can be tested with `Apache::Test`, whether the target is written in Perl, C, PHP, etc.
- The support of the new MPMs makes `mod_perl 2.0` able to scale better on a wider range of platforms. For example, if you've happened to try `mod_perl 1.0` on Win32 you probably know that parallel 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, `mod_perl 2.0` can now efficiently process parallel requests on Win32 platforms (using its native `win32` MPM).

Improved and More Flexible Configuration

`mod_perl 2.0` provides new configuration directives for the newly added features and improves upon existing ones. For example, the `PerlOptions` directive provides fine-grained configuration for what were compile-time only options in the first `mod_perl` generation. The `Perl*FilterHandler` directives provide a much simpler Apache filtering API, hiding most of the details underneath. We will talk in detail about these and other options in the section “Configuring `mod_perl 2.0`.”

The new `Apache::Directive` module provides a Perl interface to the Apache configuration tree, which is another new feature in Apache 2.0.

Optimizations

The rewrite of `mod_perl` gives us a chance to build a smarter, stronger, and faster implementation based on lessons learned over the years since `mod_perl` was introduced. There are some optimizations that can be made in the `mod_perl` source code, some that can be made in the Perl space by optimizing its syntax tree, and some that are a combination of both.

Installing `mod_perl` 2.0

Since as of this writing `mod_perl` 2.0 hasn't yet been released, the installation instructions may change a bit, but the basics should be the same. Always refer to the `mod_perl` documentation for the correct information.

Installing from Source

First download the latest stable sources of Apache 2.0, `mod_perl` 2.0, and Perl 5.8.0.* Remember that `mod_perl` 1.0 works only with Apache 1.3, and `mod_perl` 2.0 requires Apache 2.0. You can get the sources from:

- `mod_perl` 2.0—<http://perl.apache.org/dist/>
- Apache 2.0—<http://httpd.apache.org/dist/>
- Perl 5.8.0—<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 the directory 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
```

* Perl 5.6.1 can be used with *prefork*, but if you build from source why not go for the best?

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 or you want to run one of the threaded MPMs, which require Perl 5.8.0, 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 the installation of *mod_perl* 2.0, it's advisable to install at least the LWP package into your newly installed Perl distribution so that you can fully test *mod_perl* 2.0 later. 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, you can proceed with the *mod_perl* 2.0 installation:

1. Extract the source:

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

2. Remember the nightmare number of options for *mod_perl* 1.0? You need only two options 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/user/>. Configure:

```
panic% cd mod_perl-2.0.x
panic% perl Makefile.PL MP_AP_PREFIX=/home/stas/httpd/prefork \
MP_INST_APACHE2=1
```

The *MP_AP_PREFIX* option specifies the base directory of the installed Apache 2.0, under which the *include/* directory with Apache C header files can be found. For example, if you have installed Apache 2.0 in the directory *\Apache2* on Win32, you should use:

```
MP_AP_PREFIX=\Apache2
```

The *MP_INST_APACHE2* option is needed only if you have *mod_perl* 1.0 installed under the same Perl tree. You can remove this option if you don't have or don't plan to install *mod_perl* 1.0.

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:\modperl-2.0> nmake
C:\modperl-2.0> nmake test
C:\modperl-2.0> nmake install
```

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 binary versions for their platforms.

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

There are two ways of obtaining a binary `mod_perl 2.0` package for Win32:

PPM

The first, for ActivePerl users, is through PPM, which assumes you already have ActivePerl (build 6xx or later), available from <http://www.activestate.com/>, and a Win32 Apache 2.0 binary, available from <http://www.apache.org/dist/httpd/binaries/win32/>. 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 2.0, you can then install `mod_perl 2.0` via the PPM utility. ActiveState does not maintain `mod_perl` in its PPM repository, so you must get it from somewhere else. One way is simply to do:

```
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 follows (the lines are broken here 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 2.0` 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 1.0`.

See the next section for instructions on how 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 that will offer to copy this file to your *Apache2* modules directory (e.g., *C:\Apache2\modules*) should be run. If this is not done, you can get the file *mod_perl-2.tar.gz* from <http://theoryx5.uwinnipeg.ca/ppmpackages/x86/>. This file, 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 a particularly 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 called *Apache2.tar.gz* containing a binary version of Apache 2.0 with mod_perl 2.0. 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 2.0 and mod_perl 2.0. 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 2.0/mod_perl 2.0.

Apache/mod_perl/Perl 5.8 binary distribution

Because mod_perl 2.0 works best with Perl 5.8 in threaded environments such as Apache 2.0 with the *win32* MPM, there is a package including Perl 5.8, Apache 2.0, and mod_perl 2.0. To get this, look for the *perl-5.8-win32-bin.tar.gz* package at <ftp://theoryx5.uwinnipeg.ca/pub/other/>, and extract it to *C:*, which will give you an *Apache2* directory containing the Apache 2.0 installation along with mod_perl 2.0, and a *Perl* directory containing the Perl installation (you should add this *Perl* directory to your path).

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 the 1.0 settings, but some directives were renamed and new directives were added.

Enabling mod_perl

To enable mod_perl as a DSO, add this 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, so you should put it somewhere after `ServerRoot` is specified.

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

```
LoadFile "/Path/to/your/Perl/bin/perl5x.dll"
```

to `httpd.conf` to load your Perl DLL, before loading `mod_perl.so`.

Accessing the `mod_perl 2.0` Modules

To prevent you 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`: `@INC` is the same as the core `@INC`, but with `Apache2/` prepended. This module has to be loaded just after `mod_perl` has been enabled. This can be accomplished with:

```
use Apache2 ();
```

in the startup file. If you don't use a startup file, you can add:

```
PerlModule Apache2
```

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

Startup File

Next, a startup file with Perl code usually is loaded:

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

It's used to adjust Perl module search paths in `@INC`, preload commonly used modules, precompile constants, etc. A typical `startup.pl` file for `mod_perl 2.0` is shown in Example 24-1.

Example 24-1. 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
```

Example 24-1. startup.pl (continued)

```
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 module is loaded, so the 2.0 modules will be found. Afterwards, @INC is adjusted to include nonstandard directories with Perl modules:

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

If you need to use the backward-compatibility layer, to get 1.0 modules that haven't yet been ported to work with mod_perl 2.0, load Apache::compat:

```
use Apache::compat ();
```

Next, preload the commonly used mod_perl 2.0 modules and precompile the common constants. You can preload all mod_perl 2.0 modules by uncommenting the following two lines:

```
use ModPerl::MethodLookup;
ModPerl::MethodLookup::preload_all_modules();
```

Finally, the *startup.pl* file must be terminated with 1;

Perl's Command-Line Switches

Now you can pass Perl's command-line switches in *httpd.conf* by using the PerlSwitches directive, instead of using complicated workarounds.

For example, to enable warnings and taint checking, add:

```
PerlSwitches -wT
```

The *-I* command-line switch can be used to adjust @INC values:

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

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

mod_perl 2.0 Core Handlers

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

modperl

`modperl` is configured as:

```
SetHandler modperl
```

This is 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`.

perl-script

`perl-script` is configured as:

```
SetHandler perl-script
```

Most `mod_perl` handlers use the `perl-script` handler. Here are a few things to note:

- `PerlOptions +GlobalRequest` is in effect unless:

```
PerlOptions -GlobalRequest
```

is specified.
- `PerlOptions +SetupEnv` is in effect unless:

```
PerlOptions -SetupEnv
```

is specified.
- `STDOUT` and `STDOUT` get tied to the request object `$r`, which makes it possible to read from `STDIN` and print directly to `STDOUT` via `print()`, instead of having to use implicit calls like `$r->print()`.

- Several special global Perl variables are saved before the handler is called and restored afterward (as in `mod_perl 1.0`). These include `%ENV`, `@INC`, `$/`, and `STDOUT`'s `$|` and `END` blocks.

A simple response handler example

Let's demonstrate the differences between the `modperl` and `perl-script` core handlers. Example 24-2 represents a simple `mod_perl` response handler that prints out the environment variables as seen by it.

Example 24-2. Apache/PrintEnv1.pm

```
package Apache::PrintEnv1;

use strict;
use warnings;

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

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

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 for the `perl-script` handler:

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

Now issue a request to `http://localhost/print_env1`, and you should see all the environment variables printed out.

The same response handler, adjusted to work with the `modperl` core handler, is shown in Example 24-3.

Example 24-3. Apache/PrintEnv2.pm

```
package Apache::PrintEnv2;

use strict;
use warnings;

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

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

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 like this:

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

Apache::PrintEnv2 cannot use `print()`, so it 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`.

PerlOptions Directive

The `PerlOptions` directive 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 are discussed in the following sections.

Enable

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

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

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. Tuning each host to have its own pool means that the hosts will continue to reuse the Perl allocations in their specific modules.

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

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

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 that the namespace was shared by all code within the process. Consider two developers using the same server, each of whom 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 path in `@INC`:

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

<VirtualHost ...>
  ServerName dev2
```

```
PerlOptions +Parent
PerlSwitches -Mblib=/home/dev2/lib/perl
</VirtualHost>
```

Perl*Handler

Disable specific 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 it doesn't want users to configure anything but the response handler:

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

AutoLoad

Resolve Perl*Handlers at startup time; loads the modules from disk if they're not already loaded.

In mod_perl 1.0, configured Perl*Handlers that are not 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
```

GlobalRequest

Set up the global Apache::RequestRec object 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>
```

ParseHeaders

Scan output for HTTP headers. This option provides the same functionality as `mod_perl 1.0`'s `PerlSendHeaders` option, but it's more robust. It usually must be enabled for registry scripts that send the HTTP header with:

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

MergeHandlers

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

```
PerlFixupHandler Apache::FixupA

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

a request for `/inside` runs only `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 the `Apache::FixupA` and `Apache::FixupB` handlers.

SetupEnv

Set up environment variables for each request, à la `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}`.

Those who have moved to the `mod_perl` API no longer need this extra `%ENV` population and can gain by disabling it, since `%ENV` population is expensive. Code using the `CGI.pm` module requires `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 affect the whole server:

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

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
<Location /perl>
  PerlSetEnv TEST hi
  SetHandler perl-script
  PerlHandler ModPerl::Registry
  Options +ExecCGI
</Location>
```

and you issue a request for `setupenvvoff.pl` from Example 24-4.

Example 24-4. setupenvvoff.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 got the value of the environment variable TEST.

Thread-Mode–Specific Directives

The following directives are enabled only in a threaded MPM `mod_perl`:

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 until an interpreter becomes available.

PerlInterpMinSpare

The minimum number of available interpreters this parameter will clone 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 and replaced with a fresh clone when this number is reached.

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 available only 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 put back immediately afterwards, before Apache moves on to the authentication 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 (e.g., `mod_ftp`) might hook into `mod_perl` and provide a `request_rec` record. In this case, the default scope is that of the request (the download of one file). 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
```

Retrieving Server Startup Options

The `httpd` server startup options can be retrieved using `Apache::exists_config_define()`. For example, to check if the server was started in single-process mode:

```
panic% httpd -DONE_PROCESS
```

use the following code:

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

Resources

For up-to-date documentation on `mod_perl 2.0`, see:

<http://perl.apache.org/docs/2.0/>