

# 1 Introduction and Incentives

## 1.1 Description

An introduction to what `mod_perl` is all about, its different features, and some explanations of the C API, `Apache::Registry`, `Apache::PerlRun`, and the Apache/Perl API.

## 1.2 What is `mod_perl`?

The Apache/Perl integration project brings together the full power of the Perl programming language and the Apache HTTP server. With `mod_perl`, it is possible to write Apache modules entirely in Perl, letting you easily do things (such as running sub-requests) that are more difficult or impossible in regular CGI programs. In addition, the persistent Perl interpreter embedded in the server saves the overhead of starting an external interpreter, i.e. the penalty of Perl start-up time. And not the least important feature is code caching, where modules and scripts are loaded and compiled only once, and for the rest of the server's life they are served from the cache. Thus the server spends its time only running already loaded and compiled code, which is very fast.

The primary advantages of `mod_perl` are power and speed. You have full access to the inner workings of the web server and can intervene at any stage of request processing. This allows for customized processing of (to name just a few of the phases) URI->filename translation, authentication, response generation, and logging. There is very little run-time overhead. In particular, it is not necessary to start a separate process, as is often done with web-server extensions. The most wide-spread such extension, the Common Gateway Interface (CGI), can be replaced entirely with Perl code that handles the response generation phase of request processing. `mod_perl` includes two general purpose modules for this purpose: `Apache::Registry`, which can transparently run existing perl CGI scripts and `Apache::PerlRun`, which does a similar job but allows you to run "dirtier" (to some extent) scripts.

You can configure your httpd server and handlers in Perl (using `PerlSetVar`, and `<Perl>` sections). You can even define your own configuration directives.

For examples on how you use `mod_perl`, see our [What is `mod\_perl`?](#) section.

Many people ask "How much of a performance improvement does `mod_perl` give?" Well, it all depends on what you are doing with `mod_perl` and possibly who you ask. Developers report speed boosts from 200% to 2000%. The best way to measure is to try it and see for yourself! (See [Technologie Extraordinaire](#) for the facts.)

### 1.2.1 *mod\_cgi*

When you run your CGI scripts by using a configuration like this:

```
ScriptAlias /cgi-bin/ /home/httpd/cgi-bin/
```

you are running them under a `mod_cgi` handler, although you never define it explicitly. Apache does all the configuration work behind the scenes, when you use a `ScriptAlias`.

By the way, don't confuse `ScriptAlias` with the `ExecCGI` configuration option, which we enable so that the script will be executed rather than returned as a plain text file. For example for `mod_perl` and `Apache::Registry` you would use a configuration such as:

```
<Location /perl>
  SetHandler perl-script
  PerlHandler Apache::Registry
  Options +ExecCGI
  PerlSendHeader On
</Location>
```

## 1.2.2 C API

The Apache C API has been present for a long time, and has been the usual way to program extensions (such as `mod_perl`) to Apache. When you write C extension modules, you write C code that is not independent, but will be linked into the Apache `httpd` executable either at build time (if the module is statically linked), or at runtime (if it is compiled as a Dynamic Shared Object, or DSO). Either way, you do as with any C library: you write functions that receive a certain number of arguments and make use of external API functions, provided by Apache or by other libraries.

The difference is that with Apache extension modules, these functions are *registered* inside a module record: you tell Apache already at compile-time for which phases you wish to run any functions. Of course, you probably won't be handling all the phases. Here is an example of a module handling only the content generation phase:

```
/* Dispatch list of content handlers */
static const handler_rec hello_handlers[] = {
    { "hello", hello_handler },
    { NULL, NULL }
};

/* Dispatch list for API hooks */
module MODULE_VAR_EXPORT hello_module = {
    STANDARD_MODULE_STUFF,
    NULL, /* module initializer */
    NULL, /* create per-dir config structures */
    NULL, /* merge per-dir config structures */
    NULL, /* create per-server config structures */
    NULL, /* merge per-server config structures */
    NULL, /* table of config file commands */
    hello_handlers, /* [#8] MIME-typed-dispatched handlers */
    NULL, /* [#1] URI to filename translation */
    NULL, /* [#4] validate user id from request */
    NULL, /* [#5] check if the user is ok_here_ */
    NULL, /* [#3] check access by host address */
    NULL, /* [#6] determine MIME type */
    NULL, /* [#7] pre-run fixups */
    NULL, /* [#9] log a transaction */
    NULL, /* [#2] header parser */
    NULL, /* child_init */
    NULL, /* child_exit */
    NULL /* [#0] post read-request */
};
```

Using this configuration (and a correctly built `hello_handler()` function), you'd then be able to use the following configuration to allow your module to handle the requests for the `/hello` URI.

```
<Location /hello>
    SetHandler hello
</Location>
```

When Apache sees a request for the `/hello` URI, it will then figure out what the "hello" handler corresponds to by looking it up in the handler record, and match that to the `hello_handler` function pointer, which will execute the `hello_handler` function of your module with a `request_rec *r` as an argument. From that point, your handler is free to do whatever it wants, returning content, declining the request, or doing other bizarre things based on user input.

It is not the object of this guide to explain how to program C handlers. However, this example lets you in on some of the secrets of the Apache core, which you will probably understand anyway by using `mod_perl`. If you want more information on writing C modules, you should read the Apache API documentation at <http://httpd.apache.org/docs/misc/API.html> and more importantly *Writing Apache Modules with Perl and C*, which will teach you about *both* `mod_perl` and C modules!

## 1.2.3 Perl API

After a while, C modules were found hard to write and difficult to maintain, mostly because code had to be recompiled or just because of the low-level nature of the C language, and because these modules were so intricately linked with Apache that a small bug could put at risk your whole server environment. In comes `mod_perl`. Programmed in C and using all the techniques described above and more, it allows Perl modules, written in good Perl style, to access the (almost) complete API provided to the conventional C extensions.

However, the structure used for Perl Apache modules is a little different. If you've programmed normal Perl modules (like those found on CPAN) before, you'll be happy to know that programming for `mod_perl` using the Apache API doesn't involve anything else than writing a Perl module that defines a `handler` subroutine (that is the convention--we'll see that that doesn't necessarily have to be the name). This subroutine accepts an argument, `$r`, which is the Perl API equivalent of the C API `request_rec *r`.

`$r` is your entry point to the whole Perl Apache API. Through it you access methods in good object-oriented fashion, which makes it slightly easier than with C, and looks a lot more familiar to Perl programmers.

Furthermore, Perl Apache modules do not define handler records like C modules. You only need to create your handler subroutine(s), and then control which requests they should handle solely with `mod_perl` configuration directives inside your Apache configuration.

Let's look at a sample handler that returns a greeting and the current local time.

```
file:My/Greeting.pm
-----
package My::Greeting;
use strict;
```

```

use Apache::Constants qw(OK);

sub handler {
    my $r = shift;
    my $now = scalar localtime;
    my $server_name = $r->server->server_hostname;

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

    print <<EOT;
Thanks for visiting $server_name.
The local time is $now.
EOT

    return OK;
}
1; # modules must return true

```

As you can see, we're mixing Perl standard functions (like `localtime()`) with Apache functions (`$r->send_http_header()`). To return the above greeting when accessing the `/hello` URI, you would configure Apache like this:

```

<Location /hello>
    SetHandler perl-script
    PerlHandler My::Greeting
</Location>

```

When it sees this configuration, `mod_perl` loads the `My::Greeting` module, finds the `handler()` subroutine, and calls it to allow it to return the appropriate content. There are equivalent `Perl*Handler` directives for the different phases we saw were available to C handlers.

The Perl API gives you an incredible number of possibilities, which you can then use to be more productive or creative. `mod_perl` is an enabling technology; it won't make you smarter or more creative, but it will do its best to make you lose less time because of "*accidental difficulties*" of programming, and let you concentrate more on the important parts.

## 1.2.4 Apache::Registry

From the viewpoint of the Perl API, `Apache::Registry` is simply another handler that's not conceptually different from any other handler. `Apache::Registry` reads in the script file, compiles, executes it and stores into the cache. Since the perl interpreter keeps running from child process' creation to its death, any code compiled by the interpreter is kept in memory until the child dies.

To prevent script name collisions, `Apache::Registry` creates a unique key for each cached script by prepending `Apache::ROOT::` to the mangled path of the script's URI. This key is actually the package name that the script resides in. So if you have requested a script `/perl/project/test.pl`, the scripts would be wrapped in code which starts with a package declaration of:

```

package Apache::ROOT::perl::project::test_e2pl;

```

Apache::Registry also stores the script's last modification time. Everytime the script changes, the cached code is discarded and recompiled using the modified source. However, it doesn't check the modification times of any of the perl libraries the script might use.

Apache::Registry overrides CORE::exit() with Apache::exit(), so CGI scripts that use exit() will run correctly. We will talk about all these details in depth later.

From the viewpoint of the programmer, there is almost no difference between running a script as a plain CGI script under mod\_cgi and running it under mod\_perl. There is however a great speed improvement, but at the expense of much heavier memory usage (there is no free lunch :).

When they run under mod\_cgi, your CGI scripts are loaded each time they are called and then they exit. Under mod\_perl they are loaded once and cached. This gives a big performance boost. But because the code is cached and doesn't exit, it won't cleanup memory as it would under mod\_cgi. This can have unexpected effects.

Your scripts will be recompiled and reloaded by mod\_perl when it detects that you have changed them, but remember that any libraries that your scripts might require() or use() will not be recompiled when they are changed. You will have to take action yourself to ensure that they are recompiled.

Of course the guide will answer all these issues in depth.

Let's see what happens to your script when it's being executed under Apache::Registry. If we take the simplest code of (URI /perl/project/test.pl)

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

Apache::Registry will convert it into the following:

```
package Apache::ROOT::perl::project::test_e2pl;
use Apache qw(exit);
sub handler {
    print "Content-type: text/html\n\n";
    print "It works\n";
}
```

The first line provides a unique namespace for the code to use, and a unique key by which the code can be referenced from the cache.

The second line imports Apache::exit which over-rides perl's built-in exit.

The sub handler subroutine is wrapped around your code. By default (i.e. if you do not specify an alternative), when you use mod\_perl and your code's URI is called, mod\_perl will seek to execute the URI's associated handler subroutine.

Apache::Registry is usually configured in this way:

```
Alias /perl/ /usr/local/apache/bin/  
<Location /perl>  
    SetHandler perl-script  
    PerlHandler Apache::Registry  
</Location>
```

In short, we see that `Apache::Registry` is just another `mod_perl` handler, which is executed when requests are made for the `/perl` directory, and then does some special handling of the Perl scripts in that directory to turn *them* into Apache handlers.

### 1.2.5 *Apache::PerlRun*

`Apache::PerlRun` is very similar to `Apache::Registry`. It uses the same basic concepts, i.e. it runs CGI scripts under `mod_perl` for additional speed. However, unlike `Apache::Registry`, `Apache::PerlRun` will not cache scripts. The reason for this is that it's designed for use with CGI scripts that may have been "dirty", which might cause problems when run persistently under `mod_perl`. Apart from that, the configuration is the same. We discuss `Apache::PerlRun` in `Apache::PerlRun`, a closer look.

## 1.3 What you will learn

This document was written in an effort to help you start using Apache's `mod_perl` extension as quickly and easily as possible. It includes information about the installation and configuration of both Perl and the Apache web server and delves deeply into the issues of writing and porting existing Perl scripts to run under `mod_perl`. Note that it does not attempt to enter the big world of using the Perl API or C API. You will find pointers to coverage of these topics in the Offsite resources section of this site. This guide tries to cover the most of the `Apache::Registry` and `Apache::PerlRun` modules. Along with `mod_perl` related topics, there are many more issues related to administering Apache servers, debugging scripts, using databases, `mod_perl` related Perl, code snippets and more.

It is assumed that you know at least the basics of building and installing Perl and Apache. (If you do not, just read the `INSTALL` documents which are part of the distribution of each package.) However, in this guide you will find specific Perl and Apache installation and configuration notes, which will help you successfully complete the `mod_perl` installation and get the server running in a short time.

If after reading this guide and the other documentation you feel that your questions remain unanswered, you could try asking the `apache/mod_perl` mailing list to help you. But first try to browse the mailing list archive. Often you will find the answer to your question by searching the mailing list archive, since most questions have been asked and answered already! If you ignore this advice, do not be surprised if your question goes unanswered - it bores people when they're asked to answer the same question repeatedly - especially if the answer can be found in the archive or in the documentation. This does not mean that you should avoid asking questions, just do not abuse the available help and **RTFM** before you call for **HELP**. When asking your question, be sure to have read the email-etiquette and How to report problems

If you find errors in these documents, please contact the maintainer, after having read about how to submit documentation patches.

## 1.4 Maintainers

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

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

## 1.5 Authors

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

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

# Table of Contents:

1	Introduction and Incentives . . . . .	1
1.1	Description . . . . .	2
1.2	What is mod_perl? . . . . .	2
1.2.1	mod_cgi . . . . .	2
1.2.2	C API . . . . .	3
1.2.3	Perl API . . . . .	4
1.2.4	Apache::Registry . . . . .	5
1.2.5	Apache::PerlRun . . . . .	7
1.3	What you will learn . . . . .	7
1.4	Maintainers . . . . .	8
1.5	Authors . . . . .	8