

1 Apache::* modules

1.1 Description

Overview of some of the most popular modules for `mod_perl`, both to use directly from your code and as `mod_perl` handlers.

Over the time, `mod_perl` has collected an impressive amount of modules which are distributed in the standard Perl way, over CPAN. Found in the `Apache::` namespace, these implement various functionalities you might need when creating a `mod_perl`-based website. For `mod_perl`, we can actually make a distinction between two types of modules:

- Apache handlers, which handle request phases or whole requests and are standalone (`Apache::GTopLimit` for example).
- Convenience modules, which are like standard Perl modules, implementing some useful aspect of web programming, usually using `mod_perl` API for a greater performance or functionality unavailable in plain Perl. (A good example of this is `Apache::Session`.) These modules exist under the `Apache::` namespace because they can only be used under `mod_perl`.

For a complete list of modules, see the [Apache/Perl Modules](#) .

1.2 `Apache::Session` - Maintain session state across HTTP requests

This module provides the `Apache/mod_perl` user with a mechanism for storing persistent user data in a global hash, which is independent of the underlying storage mechanism. Currently you can choose from these storage mechanisms `Apache::Session::DBI`, `Apache::Session::Win32`, `Apache::Session::File`, `Apache::Session::IPC`. Read the man page of the mechanism you want to use for a complete reference.

`Apache::Session` provides persistence to a data structure. The data structure has an ID number, and you can retrieve it by using the ID number. In the case of `Apache`, you would store the ID number in a cookie or the URL to associate it with one browser, but the method of dealing with the ID is completely up to you. The flow of things is generally:

```
Tie a session to Apache::Session.  
Get the ID number.  
Store the ID number in a cookie.  
End of Request 1.
```

```
(time passes)
```

```
Get the cookie.  
Restore your hash using the ID number in the cookie.  
Use whatever data you put in the hash.  
End of Request 2.
```

Using `Apache::Session` is easy: simply tie a hash to the session object, stick any data structure into the hash, and the data you put in automatically persists until the next invocation. Here is an example which uses cookies to track the user's session.

```
# pull in the required packages
use Apache::Session::DBI;
use Apache;

use strict;

# read in the cookie if this is an old session
my $r = Apache->request;
my $cookie = $r->header_in('Cookie');
$cookie =~ s/SESSION_ID=(\w*)/$1/;

# create a session object based on the cookie we got from the
# browser, or a new session if we got no cookie
my %session;
tie %session, 'Apache::Session::DBI', $cookie,
    {DataSource => 'dbi:mysql:sessions',
     UserName   => $db_user,
     Password   => $db_pass
    };

# might be a new session, so lets give them their cookie back
my $session_cookie = "SESSION_ID=$session{_session_id}";
$r->header_out("Set-Cookie" => $session_cookie);
```

After setting this up, you can stick anything you want into `%session` (except file handles and code references and using `_session_id`), and it will still be there when the user invokes the next page.

It is possible to write an Apache authentication handler using `Apache::Session`. You can put your authentication token into the session. When a user invokes a page, you open their session, check to see if they have a valid token, and authenticate or forbid based on that.

By way of comparison note that IIS's sessions are only valid on the same web server as the one that issued the session. `Apache::Session`'s session objects can be shared amongst a farm of many machines running different operating systems, including even Win32. IIS stores session information in RAM. `Apache::Session` stores sessions in databases, file systems, or RAM. IIS's sessions are only good for storing scalars or arrays. `Apache::Session`'s sessions allow you to store arbitrarily complex objects. IIS sets up the session and automatically tracks it for you. With `Apache::Session`, you setup and track the session yourself. IIS is proprietary. `Apache::Session` is open-source. `Apache::Session::DBI` can issue 400+ session requests per second on light Celeron 300A running Linux. IIS?

An alternative to `Apache::Session` is `Apache::ASP`, which has session tracking abilities. `HTML::Embperl` hooks into `Apache::Session` for you.

1.3 Apache::DBI - Initiate a persistent database connection

See `mod_perl` and relational Databases

1.4 Apache::Watchdog::RunAway - Hanging Processes Monitor and Terminator

This module monitors hanging Apache/`mod_perl` processes. You define the time in seconds after which the process is to be counted as *hanging* or *run away*.

When the process is considered to be *hanging* it will be killed and the event logged in a log file.

Generally you should use the `amprapmon` program that is bundled with this module's distribution package, but you can write your own code using the module as well. See the `amprapmon` manpage for more information about it.

Note that it requires the `Apache::Scoreboard` module to work.

Refer to the `Apache::Watchdog::RunAway` manpage for the configuration details.

1.5 Apache::VMonitor -- Visual System and Apache Server Monitor

`Apache::VMonitor` is the next generation of `mod_status`. It provides all the information `mod_status` provides and much more.

This module emulates the reporting functions of the `top()`, `mount()`, `df()` and `ifconfig()` utilities. There is a special mode for `mod_perl` processes. It has visual alert capabilities and a configurable *automatic refresh* mode. It provides a Web interface, which can be used to show or hide all the sections dynamically.

There are two main modes:

- Multi processes mode -- All system processes and information is shown.
- Single process mode -- In-depth information about a single process is shown.

The main advantage of this module is that it reduces the need to telnet to the machine in order to monitor it. Indeed it provides information about `mod_perl` processes that cannot be acquired from telneting to the machine.

1.5.0.1 Configuration

```
# Configuration in httpd.conf
```

```

<Location /sys-monitor>
    SetHandler perl-script
    PerlHandler Apache::VMonitor
</Location>

# startup file or <Perl> section:
use Apache::VMonitor();
$Apache::VMonitor::Config{BLINKING} = 0; # Blinking is evil
$Apache::VMonitor::Config{REFRESH} = 0;
$Apache::VMonitor::Config{VERBOSE} = 0;
$Apache::VMonitor::Config{SYSTEM} = 1;
$Apache::VMonitor::Config{APACHE} = 1;
$Apache::VMonitor::Config{PROCS} = 1;
$Apache::VMonitor::Config{MOUNT} = 1;
$Apache::VMonitor::Config{FS_USAGE} = 1;
$Apache::VMonitor::Config{NETLOAD} = 1;

@Apache::VMonitor::NETDEVS = qw(lo eth0);
$Apache::VMonitor::PROC_REGEX = join "\\|", qw(httpd mysql squid);

```

More information is available in the module's extensive manpage.

It requires `Apache::Scoreboard` and `GTop` to work. `GTop` in turn requires the `libgtop` library but is not available for all platforms. See the docs in the source at <ftp://ftp.gnome.org/pub/GNOME/stable/sources/gtop/> to check whether your platform/flavor is supported.

1.6 Apache::GTopLimit - Limit Apache httpd processes

This module allows you to kill off Apache processes if they grow too large or if they share too little of their memory. You can choose to set up the process size limiter to check the process size on every request:

The module is thoroughly explained in the section: [Preventing Your Processes from Growing](#)

1.7 Apache::Request (libapreq) - Generic Apache Request Library

This package contains modules for manipulating client request data via the Apache API with Perl and C. Functionality includes:

- **parsing of application/x-www-form-urlencoded data**
- **parsing of multipart/form-data**
- **parsing of HTTP Cookies**

The Perl modules are simply a thin xs layer on top of `libapreq`, making them a lighter and faster alternative to `CGI.pm` and `CGI::Cookie`. See the `Apache::Request` and `Apache::Cookie` documentation for more details and `eg/perl/` for examples.

Apache::Request and libapreq are tied tightly to the Apache API, to which there is no access in a process running under mod_cgi.

(Apache::Request)

1.8 Apache::RequestNotes - Allow Easy, Consistent Access to Cookie and Form Data Across Each Request Phase

Apache::RequestNotes provides a simple interface allowing all phases of the request cycle access to cookie or form input parameters in a consistent manner. Behind the scenes, it uses libapreq Apache::Request) functions to parse request data and puts references to the data in pnotes().

Once the request is past the PerlInit phase, all other phases can have access to form input and cookie data without parsing it themselves. This relieves some strain, especially when the GET or POST data is required by numerous handlers along the way.

See the Apache::RequestNotes manpage for more information.

1.9 Apache::PerlRun - Run unaltered CGI scripts under mod_perl

See Apache::PerlRun - a closer look.

1.10 Apache::RegistryNG -- Apache::Registry New Generation

Apache::RegistryNG is the same as Apache::Registry, aside from using filenames instead of URIs for namespaces. This feature ensures that if the same CGI script is requested from different URIs (e.g. different hostnames) it'll be compiled and cached only once, thus saving memory.

Apache::RegistryNG uses an Object Oriented interface.

```
PerlModule Apache::RegistryNG
<Location /perl>
    SetHandler perl-script
    PerlHandler Apache::RegistryNG->handler
</Location>
```

Apache::RegistryNG inherits from Apache::PerlRun, but the handler() is overridden. Aside from the handler(), the rest of Apache::PerlRun contains all the functionality of Apache::Registry broken down into several subclass-able methods. These methods are used by Apache::RegistryNG to implement the exact same functionality of Apache::Registry, using the Apache::PerlRun methods.

There is no compelling reason to use `Apache::RegistryNG` over `Apache::Registry`, unless you want to do add or change the functionality of the existing `Registry.pm` or if you want to use filenames instead of URIs for namespaces. For example, `Apache::RegistryBB` (Bare-Bones) is another subclass that skips the `stat()` call performed by `Apache::Registry` on each request.

1.11 Apache::RegistryBB -- Apache::Registry Bare Bones

It works just like `Apache::Registry`, but does not test the `x` bit (`-x` file test for executable mode), only compiles the file once (no `stat()` call is made per request), skips the `OPT_EXECCGI` checks and does not `chdir()` into the script parent directory. It uses the Object Oriented interface.

Configuration:

```
PerlModule Apache::RegistryBB
<Location /perl>
  SetHandler perl-script
  PerlHandler Apache::RegistryBB->handler
</Location>
```

1.12 Apache::OutputChain -- Chain Stacked Perl Handlers

`Apache::OutputChain` was written as a way of exploring the possibilities of stacked handlers in `mod_perl`. It ties `STDOUT` to an object which catches the output and makes it easy to build a chain of modules that work on output data stream.

Examples of modules that are build on this idea are `Apache::SSIChain`, `Apache::GzipChain` and `Apache::EmbperlChain` -- the first processes the SSI's in the stream, the second compresses the output on the fly, the last adds `Embperl` processing.

The syntax goes like this:

```
<Files *.html>
  SetHandler perl-script
  PerlHandler Apache::OutputChain Apache::SSIChain Apache::PassHtml
</Files>
```

The modules are listed in the reverse order of their execution -- here the `Apache::PassHtml` module simply picks a file's content and sends it to `STDOUT`, then it's processed by `Apache::SSIChain`, which sends its output to `STDOUT` again. Then it's processed by `Apache::OutputChain`, which finally sends the result to the browser.

An alternative to this approach is `Apache::Filter`, which has a more natural *forward* configuration order and is easier to interface with other modules.

It works with `Apache::Registry` as well, for example:

```
Alias /foo /home/httpd/perl/foo
<Location /foo>
  SetHandler "perl-script"
  Options +ExecCGI
  PerlHandler Apache::OutputChain Apache::GzipChain Apache::Registry
</Location>
```

It's really a regular `Apache::Registry` setup, except for the added modules in the `PerlHandler` line.

(`Apache::GzipChain` allows to compress the output on the fly.)

1.13 Apache::Filter - Alter the output of previous handlers

`Apache::Filter`, like `Apache::OutputChain`, allows you to chain stacked handlers. It's not very different from `Apache::OutputChain`, except for the way you configure the filters. A normal configuration with `Apache::Filter` would be the following:

```
PerlModule Apache::Filter Apache::RegistryFilter Apache::SSI Apache::Gzip
Alias /perl /home/httpd/perl
<Location /perl>
  SetHandler "perl-script"
  Options +ExecCGI
  PerlSetVar Filter On
  PerlHandler Apache::RegistryFilter Apache::SSI Apache::Gzip
</Location>
```

This accomplishes some things many CGI programmers want: you can output SSI code from your `Apache::Registry` scripts, have it parsed by `Apache::SSI`, and then compressed with `Apache::Gzip` (see `Apache::Gzip` below).

Thanks to `Apache::Filter`, you can also write your own filter modules, which allow you to read in the output from the previous handler in the chain and modify it. You would do something like this in your handler subroutine:

```
$r = $r->filter_register(); # Required
my $fh = $r->filter_input(); # Optional (you might not need the input FH)
while (<$fh>) {
  s/ something / something else /;
  print;
}
```

Another interesting thing to do with `Apache::Filter` would be to use it for XML output from your scripts (these modules are hypothetical, this is handled much better by `AxKit`, Matt Seargeant's XML application server for `mod_perl` (see <http://www.axkit.org/>)).

```
<Location /perl/xml-output>
  SetHandler perl-script
  Options +ExecCGI
  PerlSetVar Filter On
  PerlHandler Apache::RegistryFilter Apache::XSLT
</Location>
```

As you can see, you can get a lot of freedom by using stacked handlers, allowing you to separate various parts of your programs and leave those tasks up to other modules, which may already be available from CPAN (this is much better than the CGI time when your script would have to do *everything* itself, because you couldn't do much with its output).

1.14 Apache::GzipChain - compress HTML (or anything) in the OutputChain

Have you ever served a huge HTML file (e.g. a file bloated with JavaScript code) and wondered how could you send it compressed, thus dramatically cutting down the download times? After all Java applets can be compressed into a jar and benefit from faster download times. Why can't we do the same with plain ASCII (HTML, JS etc.)? ASCII text can often be compressed by a factor of 10.

Apache::GzipChain comes to help you with this task. If a client (browser) understands gzip encoding, this module compresses the output and sends it downstream. The client decompresses the data upon receipt and renders the HTML as if it were fetching plain HTML.

For example to compress all html files on the fly, do this:

```
<Files *.html>
  SetHandler perl-script
  PerlHandler Apache::OutputChain Apache::GzipChain Apache::PassFile
</Files>
```

Remember that it will work only if the browser claims to accept compressed input, by setting the Accept-Encoding header. Apache::GzipChain keeps a list of user-agents, thus it also looks at the User-Agent header to check for browsers known to accept compressed output.

For example if you want to return compressed files which will in addition pass through the Embperl module, you would write:

```
<Location /test>
  SetHandler perl-script
  PerlHandler Apache::OutputChain Apache::GzipChain Apache::EmbperlChain Apache::PassFile
</Location>
```

Hint: Watch the *access_log* file to see how many bytes were actually sent, and compare that with the bytes sent using a regular configuration.

(See also Apache::GzipChain).

Notice that the rightmost PerlHandler must be a content producer. Here we are using Apache::PassFile but you can use any module which creates output.

1.15 Apache::Gzip - Auto-compress web files with Gzip

Similar to `Apache::GzipChain` but works with `Apache::Filter`.

This configuration:

```
PerlModule Apache::Filter
<Files ~ "*\.html">
  SetHandler perl-script
  PerlSetVar Filter On
  PerlHandler Apache::Gzip
</Files>
```

will send all the `*.html` files compressed if the client accepts the compressed input.

And this one:

```
PerlModule Apache::Filter
Alias /home/http/perl /perl
<Location /perl>
  SetHandler perl-script
  PerlSetVar Filter On
  PerlHandler Apache::RegistryFilter Apache::Gzip
</Location>
```

will compress the output of the `Apache::Registry` scripts. Yes, you should use `Apache::RegistryFilter` instead of `Apache::Registry` for it to work.

You can use as many filters as you want:

```
PerlModule Apache::Filter
<Files ~ "*\.blah">
  SetHandler perl-script
  PerlSetVar Filter On
  PerlHandler Filter1 Filter2 Apache::Gzip
</Files>
```

You can test that it works by either looking at the size of the response in the `access.log` or by telnet:

```
panic% telnet localhost 8000
Trying 127.0.0.1
Connected to 127.0.0.1
Escape character is '^]'.
GET /perl/test.pl HTTP/1.1
Accept-Encoding: gzip
User-Agent: Mozilla
```

And you will get the data compressed if configured correctly.

1.16 Apache::PerlVINC - Allows Module Versioning in Location blocks and Virtual Hosts

With this module you can have different @INC for different virtual hosts, locations and equivalent configuration blocks.

Suppose two versions of Apache::Status are being hacked on the same server. In this configuration:

```
PerlModule Apache::PerlVINC

<Location /status-dev/perl>
  SetHandler      perl-script
  PerlHandler     Apache::Status

  PerlINC         /home/httpd/dev/lib
  PerlFixupHandler Apache::PerlVINC
  PerlVersion     Apache/Status.pm
</Location>

<Location /status/perl>
  SetHandler      perl-script
  PerlHandler     Apache::Status

  PerlINC         /home/httpd/prod/lib
  PerlFixupHandler Apache::PerlVINC
  PerlVersion     Apache/Status.pm
</Location>
```

The Apache::PerlVINC is loaded and then two different locations are specified for the same handler Apache::Status, whose development version resides in */home/httpd/dev/lib* and production version in */home/httpd/prod/lib*.

In case the */status/perl* request is issued (the latter configuration section), the fixup handler will internally do:

```
delete $INC{Apache/Status.pm};
unshift @INC, /home/httpd/prod/lib;
require "Apache/Status.pm";
```

which will load the production version of the module and it'll be used to process the request. If on the other hand if the request to the */status-dev/perl* location will be issued, as configured in the former configuration section, a similar thing will happen, but a different path (*/home/httpd/dev/lib*) will be prepended to @INC:

```
delete $INC{Apache/Status.pm};
unshift @INC, /home/httpd/dev/lib;
require "Apache/Status.pm";
```

It's important to be aware that a changed @INC is effective only inside the <Location> or a similar configuration directive. Apache::PerlVINC subclasses the PerlRequire directive, marking the file to be reloaded by the fixup handler, using the value of PerlINC for @INC. That's local to the fixup

handler, so you won't actually see @INC changed in your script.

In addition the modules with different versions can be unloaded at the end of request, using the `PerlCleanupHandler` handler:

```
<Location /status/perl>
  SetHandler      perl-script
  PerlHandler     Apache::Status

  PerlINC         /home/httpd/prod/lib
  PerlFixupHandler Apache::PerlVINC
  PerlCleanupHandler Apache::PerlVINC
  PerlVersion     Apache/Status.pm
</Location>
```

Also notice that `PerlVersion` effect things differently depending on where it was placed. If it was placed inside a `<Location>` or a similar block section, the files will only be reloaded on requests to that location. If it was placed in a server section, all requests to the server or virtual hosts will have these files reloaded.

As you can guess, this module slows the response time down because it reloads some modules on a per-request basis. Hence, this module should only be used in a development environment, not a production one.

1.17 Apache::LogSTDERR

When Apache's builtin syslog support is used, the `stderr` stream is redirected to `/dev/null`. This means that Perl warnings, any messages from `die()`, `croak()`, etc., will also end up in the black hole. The `HookStderr` directive will hook the `stderr` stream to a file of your choice, the default is shown in this example:

```
PerlModule Apache::LogSTDERR
HookStderr logs/stderr_log
```

[META: see <http://mathforum.org/epigone/modperl/vixquimwhen>]

1.18 Apache::RedirectLogFix

Because of the way `mod_perl` handles redirects, the status code is not properly logged. The `Apache::RedirectLogFix` module works around that bug until `mod_perl` can deal with this. All you have to do is to enable it in the `httpd.conf` file.

```
PerlLogHandler Apache::RedirectLogFix
```

For example, you will have to use it when doing:

```
$r->status(304);
```

and do some manual header sending, like this:

```
$r->status(304);
$r->send_http_header();
```

1.19 Apache::SubProcess

The output of `system()`, `exec()`, and `open(PIPE, "|program")` calls will not be sent to the browser unless your Perl was configured with `sfio`.

One workaround is to use backticks:

```
print `command here`;
```

But a cleaner solution is provided by the `Apache::SubProcess` module. It overrides the `exec()` and `system()` calls with calls that work correctly under `mod_perl`.

Let's see a few examples:

```
use Apache::SubProcess qw(system);
my $r = shift;
$r->send_http_header('text/plain');

system "/bin/echo hi there";
```

overrides built-in `system()` function and sends the output to the browser.

```
use Apache::SubProcess qw(exec);
my $r = shift;
$r->send_http_header('text/plain');

exec "/usr/bin/cal";

print "NOT REACHED\n";
```

overrides built-in `exec()` function and sends the output to the browser. As you can see the `print` statement after the `exec()` call will be never executed.

```
use Apache::SubProcess ();
my $r = shift;
$r->send_http_header('text/plain');

my $efh = $r->spawn_child(\&env);
$r->send_fd($efh);

sub env {
    my $r = shift;
    $r->subprocess_env(HELLO => 'world');
    $r->filename("/bin/env");
    $r->call_exec;
}
```

env() is a function that sets an environment variable that can be seen by the main and sub-processes, then it executes `/bin/env` program via `call_exec()`. The main code spawn a process, and tells it to execute the `env()` function. This call returns an output filehandler from the spawned child process. Finally it takes the output generated by the child process and sends it to the browser via `send_fd()`, that expects the filehandler as an argument.

```
use Apache::SubProcess ();
my $r = shift;
$r->send_http_header('text/plain');

my $fh = $r->spawn_child(\&banner);
$r->send_fd($fh);

sub banner {
    my $r = shift;
    # /usr/games/banner on many Unices
    $r->filename("/usr/bin/banner");
    $r->args("-w40+Hello%20World");
    $r->call_exec;
}
```

This example is very similar to the previous, but shows how can you pass arguments to the external process. It passes the string to print as a banner to via a subprocess.

```
use Apache::SubProcess ();
my $r = shift;
$r->send_http_header('text/plain');

use vars qw($String);
$String = "hello world";
my ($out, $in, $err) = $r->spawn_child(\&echo);
print $out $String;
$r->send_fd($in);

sub echo {
    my $r = shift;
    $r->subprocess_env(CONTENT_LENGTH => length $String);
    $r->filename("/tmp/pecho");
    $r->call_exec;
}
```

The last example shows how you can have a full access to `STDIN`, `STDOUT` and `STDERR` streams of the spawned sub process, so you can pipe data to a program and send its output to the browser. The `echo()` function is similar to the earlier example's `env()` function. The `/tmp/pecho` is as follows:

```
#!/usr/bin/perl
read STDIN, $buf, $ENV{CONTENT_LENGTH};
print "STDIN: '$buf' ($ENV{CONTENT_LENGTH})\n";
```

So in the last example a string is defined as a global variable, so it's length could be calculated in the `echo()` function. The subprocess reads from `STDIN`, to which the main process writes the string (*hello world*). It reads only a number of bytes specified by `CONTENT_LENGTH` passed to the external program via environment variable. Finally the external program prints the data that it read to `STDOUT`, the main program intercepts it and sends to the client's socket (browser in most cases).

1.20 Module::Use - Log and Load used Perl modules

Module::Use records the modules used over the course of the Perl interpreter's lifetime. If the logging module is able, the old logs are read and frequently used modules are automatically loaded.

For example if configured as:

```
<Perl>
    use Module::Use (Counting, Logger => "Debug");
</Perl>

PerlChildExitHandler Module::Use
```

it will only record the used modules when the child exists, logging everything (debug level).

1.21 Apache::ConfigFile - Parse an Apache style httpd.conf config file

This module parses *httpd.conf*, or any compatible config file, and provides methods for accessing the values from the parsed file.

See the module manpage for more information.

1.22 Apache::Admin::Config - Object oriented access to Apache style config files

Apache::Admin::Config provides an object oriented interface for reading and writing Apache-like configuration files without affecting comments, indentation, or truncated lines. You can easily extract informations from the apache configuration, or manage htaccess files.

See <http://rs.rhapsodyk.net/devel/apache-admin-config/> for more information.

1.23 Maintainers

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

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

1.24 Authors

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

1.24 Authors

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

Table of Contents:

1	Apache::* modules	1
1.1	Description	2
1.2	Apache::Session - Maintain session state across HTTP requests	2
1.3	Apache::DBI - Initiate a persistent database connection	4
1.4	Apache::Watchdog::RunAway - Hanging Processes Monitor and Terminator	4
1.5	Apache::VMonitor -- Visual System and Apache Server Monitor	4
1.5.0.1	Configuration	4
1.6	Apache::GTopLimit - Limit Apache httpd processes	5
1.7	Apache::Request (libapreq) - Generic Apache Request Library	5
1.8	Apache::RequestNotes - Allow Easy, Consistent Access to Cookie and Form Data Across Each Request Phase	6
1.9	Apache::PerlRun - Run unaltered CGI scripts under mod_perl	6
1.10	Apache::RegistryNG -- Apache::Registry New Generation	6
1.11	Apache::RegistryBB -- Apache::Registry Bare Bones	7
1.12	Apache::OutputChain -- Chain Stacked Perl Handlers	7
1.13	Apache::Filter - Alter the output of previous handlers	8
1.14	Apache::GzipChain - compress HTML (or anything) in the OutputChain	9
1.15	Apache::Gzip - Auto-compress web files with Gzip	10
1.16	Apache::PerlVINC - Allows Module Versioning in Location blocks and Virtual Hosts	11
1.17	Apache::LogSTDERR	12
1.18	Apache::RedirectLogFix	12
1.19	Apache::SubProcess	13
1.20	Module::Use - Log and Load used Perl modules	15
1.21	Apache::ConfigFile - Parse an Apache style httpd.conf config file	15
1.22	Apache::Admin::Config - Object oriented access to Apache style config files	15
1.23	Maintainers	15
1.24	Authors	15