

# 1 mod\_perl Configuration

## 1.1 Description

This section documents the various configuration options available for Apache and `mod_perl`, as well as the Perl startup files, and more esoteric possibilities such as configuring Apache with Perl.

## 1.2 Server Configuration

The next step after building and installing your new `mod_perl` enabled Apache server is to configure the server. There are two separate parts to configure: Apache and `mod_perl`. Each has its own set of directives.

To configure your `mod_perl` enabled Apache server, the only file that you should need to edit is *httpd.conf*. By default, *httpd.conf* is put into the *conf* directory under the server root directory. The default server root is `/usr/local/apache/` on many UNIX platforms, but within reason it can be any directory you choose. If you are new to Apache and `mod_perl`, you will probably find it helpful to keep to the directory layouts we use in this Guide if you can.

Apache versions 1.3.4 and later are distributed with the configuration directives in a single file -- *httpd.conf*. This Guide uses the same approach in its examples. Prior to version 1.3.4, the default Apache installation used three configuration files -- *httpd.conf*, *srm.conf*, and *access.conf*. If you wish you can still use all three files, by setting the `AccessConfig` and `ResourceConfig` directives in *httpd.conf*. You will also see later on that we use other files, for example *perl.conf* and *startup.pl*. This is just for our convenience, you could still do everything in *httpd.conf* if you wished.

## 1.3 Apache Configuration

Apache configuration can be confusing. To minimize the number of things that can go wrong, it can be a good idea first to configure Apache itself without `mod_perl`. This will give you the confidence that it works and maybe that you have some idea how to configure it.

There is a warning in the *httpd.conf* distributed with Apache about simply editing *httpd.conf* and running the server, without understanding all the implications. This is another warning. Modifying the configuration file and adding new directives can introduce security problems, and have performance implications.

The Apache distribution comes with an extensive configuration manual, and in addition each section of the distributed configuration file includes helpful comments explaining how every directive should be configured and what the default values are.

If you haven't moved Apache's directories around, the installation program will have configured everything for you. You can just start the server and test it. To start the server use the `apachectl` utility which comes bundled with the Apache distribution. It resides in the same directory as `httpd`, the Apache server itself. Execute:

```
/usr/local/apache/bin/apachectl start
```

Now you can test the server, for example by accessing `http://localhost` from a browser running on the same host.

### 1.3.1 Configuration Directives

For a basic setup there are just a few things to configure. If you have moved any directories you have to update them in `httpd.conf`. There are many of them, here are just a couple of examples:

```
ServerRoot    "/usr/local/apache"  
DocumentRoot "/home/httpd/docs"
```

If you want to run it on a port other than port 80 edit the `Port` directive:

```
Port 8080
```

You might want to change the user and group names the server will run under. Note that if started as the `root` user (which is generally the case), the parent process will continue to run as `root`, but its children will run as the user and group you have specified. For example:

```
User httpd  
Group httpd
```

There are many other directives that you might need to configure as well. In addition to directives which take a single value there are whole sections of the configuration (such as the `<Directory>` and `<Location>` sections) which apply only to certain areas of your Web space. As mentioned earlier you will find them all in `httpd.conf`.

### 1.3.2 `.htaccess` files

If there is a file with the name `.htaccess` in any directory, Apache scans it for further configuration directives which it then applies only to that directory (and its subdirectories). The name `.htaccess` is confusing because it can contain any configuration directives, not just those related to access to resources. You will not be surprised to find that a configuration directive can change the names of the files used in this way.

Note that if there is a

```
<Directory />  
  AllowOverride None  
</Directory>
```

directive in `httpd.conf`, Apache will not try to look for `.htaccess` at all.

### 1.3.3 `<Directory>`, `<Location>` and `<Files>` Sections

I'll explain just the basics of the `<Directory>`, `<Location>` and `<Files>` sections. Remember that there is more to know and the rest of the information is available in the Apache documentation. The information I'll present here is just what is important for understanding the `mod_perl` configuration sections.

Apache considers directories and files on your machine all to be resources. For each resource you can determine a particular behaviour which will apply to every request for information from that particular resource.

Obviously the directives in <Directory> sections apply to specific directories on your host machine, and those in <Files> sections apply only to specific files (actually groups of files with names which have something in common). In addition to these sections, Apache has the concept of a <Location>, which is also just a resource. <Location> sections apply to specific URIs. Locations are based at the document root, directories are based at the filesystem root. For example, if you have the default server directory layout where the server root is */usr/local/apache* and the document root is */usr/local/apache/htdocs* then static files in the directory */usr/local/apache/htdocs/pub* are in the location */pub*.

It is up to you to decide which directories on your host machine are mapped to which locations. You should be careful how you do it, because the security of your server may be at stake.

Locations do not necessarily have to refer to existing physical directories, but may refer to virtual resources which the server creates for the duration of a single browser request. As you will see, this is often the case for a *mod\_perl* server.

When a browser asks for a resource from your server, Apache determines from its configuration whether or not to serve the request, whether to pass the request to another server, what (if any) authorization is required for access to the resource, and how to reply. For any given resource, the various sections in your configuration may provide conflicting information. For example you may have a <Directory> section which tells Apache that authorization is required for access to the resource but you may have a <Files> section which says that it is not. It is not always obvious which directive takes precedence in these cases. This can be a trap for the unwary.

- **<Directory directoryPath> ... </Directory>**

Can appear in server and virtual host configurations.

<Directory> and </Directory> are used to enclose a group of directives which will apply only to the named directory and sub-directories of that directory. Any directive which is allowed in a directory context (see the Apache documentation) may be used.

The path given in the <Directory> directive is either the full path to a directory, or a wild-card string. In a wild-card string, ? matches any single character, \* matches any sequence of characters, and [ ] matches character ranges. (This is similar to the shell's file globs.) None of the wildcards will match a / character. For example:

```
<Directory /home/httpd/docs>
  Options Indexes
</Directory>
```

If you want to use a regular expression to match then you should use the syntax <DirectoryMatch regex> ... </DirectoryMatch>.

If multiple (non-regular expression) directory sections match the directory (or its parents) containing a document, then the directives are applied in the order of shortest match first, interspersed with the directives from any *.htaccess* files. For example, with

```
<Directory />
  AllowOverride None
</Directory>

<Directory /home/httpd/docs/*>
  AllowOverride FileInfo
</Directory>
```

for access to the document */home/httpd/docs/index.html* the steps are:

- **Apply directive `AllowOverride None` (disabling *.htaccess* files).**
- **Apply directive `AllowOverride FileInfo` for directory */home/httpd/docs/* (which now enables *.htaccess* in */home/httpd/docs/* and its sub-directories).**
- **Apply any `FileInfo` directives in */home/httpd/docs/.htaccess*.**
- **<Files filename> ... </Files>**

Can appear in server and virtual host configurations, and *.htaccess* files as well.

The <Files> directive provides for access control by filename. It is comparable to the <Directory> and <Location> directives. It should be closed with the </Files> directive. The directives given within this section will be applied to any object with a basename (last component of filename) matching the specified filename.

<Files> sections are processed in the order they appear in the configuration file, after the <Directory> sections and *.htaccess* files are read, but before <Location> sections. Note that <Files> can be nested inside <Directory> sections to restrict the portion of the filesystem they apply to. <Files> cannot be nested inside <Location> sections however.

The filename argument should include a filename, or a wild-card string, where ? matches any single character, and \* matches any sequence of characters. Extended regular expressions can also be used, simply place a tilde character ~ between the directive and the regular expression. The regular expression should be in quotes. The dollar symbol refers to the end of the string. The pipe character indicates alternatives. Special characters in extended regular expressions must be escaped with a backslash. For example:

```
<Files ~ "\.(gif|jpe?g|png)$">
```

would match most common Internet graphics formats. Alternatively you can use the <FilesMatch regex> ... </FilesMatch> syntax.

- **<Location URL> ... </Location>**

Can appear in server and virtual host configurations.

The `<Location>` directive provides for access control by URL. It is similar to the `<Directory>` directive, and starts a section which is terminated with the `</Location>` directive.

`<Location>` sections are processed in the order they appear in the configuration file, after the `<Directory>` sections, `.htaccess` files and `<Files>` sections are read.

The `<Location>` section is the directive that is used most often with `mod_perl`.

URLs *do not* have to refer to real directories or files within the filesystem at all, `<Location>` operates completely outside the filesystem. Indeed it may sometimes be wise to ensure that `<Location>`s do not match real paths to avoid confusion.

The URL may use wildcards. In a wild-card string, `?` matches any single character, and `*` matches any sequences of characters, `[]` groups characters to match. For regular expression matches use the `<LocationMatch regex> ... </LocationMatch>` syntax.

The `<Location>` functionality is especially useful when combined with the `SetHandler` directive. For example to enable status requests, but allow them only from browsers at `example.com`, you might use:

```
<Location /status>
  SetHandler server-status
  order deny,allow
  deny from all
  allow from .example.com
</Location>
```

### 1.3.4 How Directory, Location and Files Sections are Merged

When configuring the server, it's important to understand the order in which the rules of each section apply to requests. The order of merging is:

1. **`<Directory>` (except regular expressions) and `.htaccess` are processed simultaneously, with `.htaccess` overriding `<Directory>`**
2. **`<DirectoryMatch>`, and `<Directory>` with regular expressions**
3. **`<Files>` and `<FilesMatch>` are processed simultaneously**
4. **`<Location>` and `<LocationMatch>` are processed simultaneously**

Apart from `<Directory>`, each group is processed in the order that it appears in the configuration files. `<Directory>` (group 1 above) is processed in the order shortest directory component to longest. If multiple `<Directory>` sections apply to the same directory then they are processed in the configuration file order.

Sections inside `<VirtualHost>` sections are applied as if you were running several independent servers. The directives inside `<VirtualHost>` sections do not interact with each other. They are applied after first processing any sections outside the virtual host definition. This allows virtual host configurations to override the main server configuration.

Later sections override earlier ones.

### 1.3.5 Sub-Grouping of <Location>, <Directory> and <Files> Sections

Let's say that you want all files, except for a few of the files in a specific directory and below, to be handled in the same way. For example if you want all the files in `/home/http/docs` to be served as plain files, but any files with ending `.html` and `.txt` to be processed by the content handler of your `Apache::MyFilter` module.

```
<Directory /home/httpd/docs>
  <FilesMatch "\.(html|txt)$">
    SetHandler perl-script
    PerlHandler Apache::MyFilter
  </FilesMatch>
</Directory>
```

Thus it is possible to embed sections inside sections to create subgroups which have their own distinct behavior. Alternatively you could use a <Files> section inside an `.htaccess` file.

Note that you can't put <Files> or <FilesMatch> sections inside a <Location> section, but you can put them inside a <Directory> section.

### 1.3.6 Options Directive

Normally, if multiple `Options` directives apply to a directory, then the most specific one is taken complete; the options are not merged.

However if all the options on the `Options` directive are preceded by a `+` or `-` symbol, the options are merged. Any options preceded by `+` are added to the options currently in force, and any options preceded by `-` are removed.

For example, without any `+` and `-` symbols:

```
<Directory /home/httpd/docs>
  Options Indexes FollowSymLinks
</Directory>
<Directory /home/httpd/docs/shtml>
  Options Includes
</Directory>
```

then only `Includes` will be set for the `/home/httpd/docs/shtml` directory. However if the second `Options` directive uses the `+` and `-` symbols:

```
<Directory /home/httpd/docs>
  Options Indexes FollowSymLinks
</Directory>
<Directory /home/httpd/docs/shtml>
  Options +Includes -Indexes
</Directory>
```

then the options `FollowSymLinks` and `Includes` are set for the `/home/httpd/docs/shtml` directory.

## 1.4 mod\_perl Configuration

When you have tested that the Apache server works on your machine, it's time to configure `mod_perl`. Some of the configuration directives are already familiar to you, but `mod_perl` introduces a few new ones.

It can be a good idea to keep all the `mod_perl` related configuration at the end of the configuration file, after the native Apache configuration directives.

To ease maintenance and to simplify multiple server installations, the Apache/`mod_perl` configuration system allows you several alternative ways to keep your configuration directives in separate places. The `Include` directive in `httpd.conf` allow you to include the contents of other files, just as if the information were all contained in `httpd.conf`. This is a feature of Apache itself. For example if you want all your `mod_perl` configuration to be placed in a separate file `mod_perl.conf` you can do that by adding to `httpd.conf` this directive:

```
Include conf/mod_perl.conf
```

If you want to include this configuration conditionally, depending on whether your apache has been compiled with `mod_perl`, you can use the `IfModule` directive:

```
<IfModule mod_perl.c>
  Include conf/mod_perl.conf
</IfModule>
```

`mod_perl` adds two further directives: `<Perl>` sections allow you to execute Perl code from within any configuration file at server startup time, and as you will see later, a file containing any Perl program can be executed (also at server startup time) simply by mentioning its name in a `PerlRequire` or `PerlModule` directive.

### 1.4.1 Alias Configurations

The `ScriptAlias` and `Alias` directives provide a mapping of a URI to a file system directory. The directive:

```
Alias /foo /home/httpd/foo
```

will map all requests starting with `/foo` onto the files starting with `/home/httpd/foo/`. So when Apache gets a request `http://www.example.com/foo/test.pl` the server will map this into the file `test.pl` in the directory `/home/httpd/foo/`.

In addition `ScriptAlias` assigns all the requests that match the URI (i.e. `/cgi-bin`) to be executed under `mod_cgi`.

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

is actually the same as:

```
Alias /cgi-bin/ /home/httpd/cgi-bin
<Location /cgi-bin>
    SetHandler cgi-script
    Options +ExecCGI
</Location>
```

where latter directive invokes `mod_cgi`. You shouldn't use the `ScriptAlias` directive unless you want the request to be processed under `mod_cgi`. Therefore when you configure `mod_perl` sections use `Alias` instead.

Under `mod_perl` the `Alias` directive will be followed by two further directives. The first is the `SetHandler perl-script` directive, which tells Apache to invoke `mod_perl` to run the script. The second directive (for example `PerlHandler`) tells `mod_perl` which handler (Perl module) the script should be run under, and hence for which phase of the request. Refer to the section `Perl*Handlers` for more information about handlers for the various request phases.

When you have decided which methods to use to run your scripts and where you will keep them, you can add the configuration directive(s) to `httpd.conf`. They will look like those below, but they will of course reflect the locations of your scripts in your file-system and the decisions you have made about how to run the scripts:

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

In the examples above all the requests issued for URIs starting with `/cgi-bin` will be served from the directory `/home/httpd/cgi-bin/`, and starting with `/perl` from the directory `/home/httpd/perl/`.

### 1.4.1.1 Running CGI, PerlRun, and Registry Scripts Located in the Same Directory

```
# Typical for plain cgi scripts:
ScriptAlias /cgi-bin/ /home/httpd/perl/

# Typical for Apache::Registry scripts:
Alias       /perl/    /home/httpd/perl/

# Typical for Apache::PerlRun scripts:
Alias       /cgi-perl/ /home/httpd/perl/
```

In the examples above we have mapped the three different URIs (`http://www.example.com/perl/test.pl`, `http://www.example.com/cgi-bin/test.pl` and `http://www.example.com/cgi-perl/test.pl`) all to the same file `/home/httpd/perl/test.pl`. This means that we can have all our CGI scripts located at the same place in the file-system, and call the script in any of three ways simply by changing one component of the URI (`cgi-bin/perl/cgi-perl`).

This technique makes it easy to migrate your scripts to `mod_perl`. If your script does not seem to be working while running under `mod_perl`, then in most cases you can easily call the script in straight `mod_cgi` mode or under `Apache::PerlRun` without making any script changes. Simply change the URL you use to invoke it.

Although in the configuration above we have configured all three *Aliases* to point to the same directory within our file system, you can of course have them point to different directories if you prefer.

You should remember that it is undesirable to run scripts in plain `mod_cgi` mode from a `mod_perl`-enabled server--the resource consumption is too high. It is better to run these on a plain Apache server. See [Standalone mod\\_perl Enabled Apache Server](#).

## 1.4.2 <Location> Configuration

The <Location> section assigns a number of rules which the server should follow when the request's URI matches the *Location*. Just as it is the widely accepted convention to use */cgi-bin* for your `mod_cgi` scripts, it is conventional to use */perl* as the base URI of the perl scripts which you are running under `mod_perl`. Let's review the following very widely used <Location> section:

```
Alias /perl/ /home/httpd/perl/
PerlModule Apache::Registry
<Location /perl>
    SetHandler perl-script
    PerlHandler Apache::Registry
    Options ExecCGI
    allow from all
    PerlSendHeader On
</Location>
```

This configuration causes all requests for URIs starting with */perl* to be handled by the `mod_perl` Apache module with the handler from the `Apache::Registry` Perl module. Let's review the directives inside the <Location> section in the example:

```
<Location /perl>
```

Remember the `Alias` from the above section? We use the same `Alias` here; if you were to use a <Location> that does not have the same `Alias`, the server would fail to locate the script in the file system. You need the `Alias` setting only if the code that should be executed is located in the file. So `Alias` just provides the URI to filepath translation rule.

Sometimes there is no script to be executed. Instead there is some module whose method is being executed, similar to */perl-status*, where the code is stored in an Apache module. In such cases we don't need `Alias` settings for those <Location>s.

```
SetHandler perl-script
```

This assigns the `mod_perl` Apache module to handle the content generation phase.

```
PerlHandler Apache::Registry
```

Here we tell Apache to use the `Apache::Registry` Perl module for the actual content generation.

```
Options ExecCGI
```

The `Options` directive accepts various parameters (options), one of which is `ExecCGI`. This tells the server that the file is a program and should be executed, instead of just being displayed like a static file (like HTML file). If you omit this option then the script will either be rendered as plain text or else it will trigger a *Save-As* dialog, depending on the client's configuration.

```
allow from all
```

This directive is used to set access control based on domain. The above settings allow clients from any domain to run the script.

```
PerlSendHeader On
```

`PerlSendHeader On` tells the server to send an HTTP headers to the browser on every script invocation. You will want to turn this off for `nph` (non-parsed-headers) scripts.

The `PerlSendHeader On` setting invokes the Apache's `ap_send_http_header()` method after parsing the headers generated by the script. It is only meant for emulation of `mod_cgi` behavior with regard to headers.

To send the HTTP headers it's always better either to use the `$r->send_http_header` method using the Apache Perl API or to use the `$q->header` method from the `CGI.pm` module.

```
</Location>
```

Closes the `<Location>` section definition.

Note that sometimes you will have to preload the module before using it in the `<Location>` section. In the case of `Apache::Registry` the configuration will look like this:

```
PerlModule Apache::Registry
<Location /perl>
  SetHandler perl-script
  PerlHandler Apache::Registry
  Options ExecCGI
  allow from all
  PerlSendHeader On
</Location>
```

`PerlModule` is equivalent to Perl's native `use()` function call.

No changes are required to the `/cgi-bin` location (`mod_cgi`), since it has nothing to do with `mod_perl`.

Here is another very similar example, this time using `Apache::PerlRun` (For more information see `Apache::PerlRun`):

```
<Location /cgi-perl>
  SetHandler perl-script
  PerlHandler Apache::PerlRun
  Options ExecCGI
  allow from all
  PerlSendHeader On
</Location>
```

The only difference from the `Apache::Registry` configuration is the argument of the `PerlHandler` directive, where `Apache::Registry` has been replaced with `Apache::PerlRun`.

### 1.4.3 Overriding <Location> Setting in "Sub-Location"

So if you have:

```
<Location /foo>
  SetHandler perl-script
  PerlHandler My::Module
</Location>
```

If you want to remove a `mod_perl` handler setting from a location beneath a location where the handler was set (i.e. `/foo/bar`), all you have to do is to reset it, like this:

```
<Location /foo/bar>
  SetHandler default-handler
</Location>
```

Now, all the requests starting with `/foo/bar` would be served by Apache's default handler.

### 1.4.4 PerlModule and PerlRequire Directives

As we saw earlier, a module should be loaded before it is used. `PerlModule` and `PerlRequire` are the two `mod_perl` directives which are used to load modules and code. They are almost equivalent to Perl's `use()` and `require()` functions respectively and called from the Apache configuration file. You can pass one or more module names as arguments to `PerlModule`:

```
PerlModule Apache::DBI CGI DBD::Mysql
```

Generally the modules are preloaded from the startup script, which is usually called `startup.pl`. This is a file containing plain Perl code which is executed through the `PerlRequire` directive. For example:

```
PerlRequire /home/httpd/perl/lib/startup.pl
```

A `PerlRequire` file name can be absolute or relative to `ServerRoot` or a path in `@INC`.

As with any file with Perl code that gets `use()`'d or `require()`'d, it must return a `true` value. To ensure that this happens don't forget to add `1;` at the end of `startup.pl`.

Notice that unless `mod_perl` is compiled as DSO and unless `PerlFreshRestart` is set to `On`, one needs to fully stop and start Apache for any changes to take affect, if the files and modules have been modified.

### 1.4.5 Perl\*Handlers

As you probably know Apache traverses a loop for each HTTP request it receives.

After you have compiled and installed mod\_perl, your Apache mod\_perl configuration directives tell Apache to invoke the module mod\_perl as the handler for some request which it receives. Although it could in fact handle all the phases of the request loop, usually it does not. You tell mod\_perl which phases it is to handle (and so which to leave to other modules, or to the default Apache routines) by putting Perl\*Handler directives in the configuration files.

Because you need the Perl interpreter to be present for your Perl script to do any processing at all, there is a slight difference between the way that you configure Perl and C handlers to handle parts of the request loop. Ordinarily a C module is written, compiled and configured to hook into a specific phase of the request loop. For a Perl handler you compile mod\_perl itself to hook into the appropriate phases, as if it were to handle the phases itself. Then you put Perl\*Handler directives in your configuration file to tell mod\_perl that it is to pass the responsibility for handling that part of the request phase to your Perl module.

mod\_perl is an Apache module written in C. As most programmers will only need to handle the response phase, in the default compilation most of the Perl\*Handlers are disabled. When you configure the *Makefile.PL* file for its compilation, you must specify whether or not you will want to handle parts of the request loop other than the usual content generation phase. If so you need to specify which parts. See the "Callback Hooks" section for how to do this.

Apache specifies about eleven phases of the request loop, namely (and in order of processing): Post-Read-Request, URI Translation, Header Parsing, Access Control, Authentication, Authorization, MIME type checking, FixUp, Response (also known as the Content handling phase), Logging and finally Cleanup. These are the stages of a request where the Apache API allows a module to step in and do something. There is a dedicated Perl\*Handler for each of these stages plus a couple of others which don't correspond to parts of the request loop.

We call them Perl\*Handler directives because the names of the many mod\_perl handler directives for the various phases of the request loop all follow the same format. The \* in Perl\*Handler is a placeholder to be replaced by something which identifies the phase to be handled. For example PerlLogHandler is a Perl Handler which (fairly obviously) handles the logging phase.

The slight exception is PerlHandler, which you can think of as PerlResponseHandler. It is the content generation handler and so it is probably the one that you will use most frequently.

Note that it is mod\_perl which recognizes these directives, and not Apache. They are mod\_perl directives, and an ordinary Apache does not recognize them. If you get error messages about these directives being "*perhaps mis-spelled*" it is a sure sign that the appropriate part of mod\_perl (or the entire mod\_perl module!) is not present in your copy of Apache executable.

The full list of Perl\*Handlers follows. They are in the order that they are processed by Apache and mod\_perl:

```
PerlChildInitHandler
PerlPostReadRequestHandler
PerlInitHandler
PerlTransHandler
PerlHeaderParserHandler
PerlAccessHandler
```

```
PerlAuthenHandler  
PerlAuthzHandler  
PerlTypeHandler  
PerlFixupHandler  
PerlHandler  
PerlLogHandler  
PerlCleanupHandler  
PerlChildExitHandler  
PerlDispatchHandler  
PerlRestartHandler
```

`PerlChildInitHandler` and `PerlChildExitHandler` do not refer to parts of the request loop, they are to allow your modules to initialize data structures and to clean up at the child process start-up and shutdown respectively, for example by allocating and deallocating memory.

All `<Location>`, `<Directory>` and `<Files>` sections contain a physical path specification. Like `PerlChildInitHandler` and `PerlChildExitHandler`, the directives `PerlPostReadRequestHandler` and `PerlTransHandler` cannot be used in these sections, nor in `.htaccess` files, because it is not until the end of the Translation Handler (`PerlTransHandler`) phase that the path translation is completed and a physical path is known.

`PerlInitHandler` changes its behaviour depending upon where it is used. In any case it is the first handler to be invoked in serving a request. If found outside any `<Location>`, `<Directory>` or `<Files>` section (at the top level), it is an alias for `PerlPostReadRequestHandler`. When inside any such section it is an alias for `PerlHeaderParserHandler`.

Starting from `PerlHeaderParserHandler` the requested URI has been mapped to a physical server pathname, and thus it can be used to match a `<Location>`, `<Directory>` or `<Files>` configuration section, or to look in a `.htaccess` file if such a file exists in the specified directory in the translated path.

`PerlDispatchHandler` and `PerlRestartHandler` do not correspond to parts of the Apache API, but allow you to fine-tune the `mod_perl` API.

The Apache documentation will tell you all about these stages and what your modules can do. By default, most of these hooks are disabled at compile time, see the "Callback Hooks" section for information on enabling them.

## 1.4.6 *The handler subroutine*

By default the `mod_perl` API expects a subroutine called `handler()` to handle the request in the registered `Perl*Handler` module. Thus if your module implements this subroutine, you can register the handler with `mod_perl` like this:

```
Perl*Handler Apache::Foo
```

Replace `Perl*Handler` with the name of a specific handler from the list given above. `mod_perl` will preload the specified module for you. Please note that this approach will not preload the module at startup. To make sure it gets loaded you have three options: you can explicitly preload it with the `PerlModule` directive:

```
PerlModule Apache::Foo
```

You can preload it at the startup file:

```
use Apache::Foo ();
```

Or you can use a nice shortcut that the `Perl*Handler` syntax provides:

```
Perl*Handler +Apache::Foo
```

Note the leading `+` character. This directive is equivalent to:

```
PerlModule Apache::Foo
Perl*Handler Apache::Foo
```

If you decide to give the handler routine a name other than `handler`, for example `my_handler`, you must preload the module and explicitly give the name of the handler subroutine:

```
PerlModule Apache::Foo
Perl*Handler Apache::Foo::my_handler
```

As you have seen, this will preload the module at server startup.

If a module needs to know which handler is currently being run, it can find out with the `current_callback` method. This method is most useful to *PerlDispatchHandlers* which wish to take action for certain phases only.

```
if ($r->current_callback eq "PerlLogHandler") {
    $r->warn("Logging request");
}
```

## 1.4.7 Stacked Handlers

With the `mod_perl` stacked handlers mechanism, during any stage of a request it is possible for more than one `Perl*Handler` to be defined and run.

`Perl*Handler` directives (in your configuration files) can define any number of subroutines. For example:

```
PerlTransHandler OneTrans TwoTrans RedTrans BlueTrans
```

With the method `Apache->push_handlers()`, callbacks (handlers) can be added to a stack *at runtime* by `mod_perl` scripts.

`Apache->push_handlers()` takes the callback hook name as its first argument and a subroutine name or reference as its second.

Here's an example:

```
use Apache::Constants qw(:common);
sub my_logger {
    #some code here
    return OK;
}
Apache->push_handlers("PerlLogHandler", \&my_logger);
```

Here's another one:

```
use Apache::Constants qw(:common);
$r->push_handlers("PerlLogHandler", sub {
    print STDERR "__ANON__ called\n";
    return OK;
});
```

After each request, this stack is erased.

All handlers will be called unless a handler returns a status other than OK or DECLINED.

Example uses:

CGI.pm maintains a global object for its plain function interface. Since the object is global, it does not go out of scope, DESTROY is never called. CGI->new can call:

```
Apache->push_handlers("PerlCleanupHandler", \&CGI::_reset_globals);
```

This function will be called during the final stage of a request, refreshing CGI.pm's globals before the next request comes in.

Apache::DCELogin establishes a DCE login context which must exist for the lifetime of a request, so the DCE::Login object is stored in a global variable. Without stacked handlers, users must set

```
PerlCleanupHandler Apache::DCELogin::purge
```

in the configuration files to destroy the context. This is not "user-friendly". Now, Apache::DCELogin::handler can call:

```
Apache->push_handlers("PerlCleanupHandler", \&purge);
```

Persistent database connection modules such as Apache::DBI could push a PerlCleanupHandler handler that iterates over %Connected, refreshing connections or just checking that connections have not gone stale. Remember, by the time we get to PerlCleanupHandler, the client has what it wants and has gone away, so we can spend as much time as we want here without slowing down response time to the client (although the process itself is unavailable for serving new requests before the operation is completed).

PerlTransHandlers (e.g. Apache::MysqlProxy) may decide, based on the URI or some arbitrary condition, whether or not to handle a request. Without stacked handlers, users must configure it themselves:

```
PerlTransHandler Apache::MsqlProxy::translate
PerlHandler      Apache::MsqlProxy
```

PerlHandler is never actually invoked unless `translate()` sees that the request is a proxy request (`$r->proxyreq`). If it is a proxy request, `translate()` sets `$r->handler("perl-script")`, and only then will PerlHandler handle the request. Now users do not have to specify PerlHandler Apache::MsqlProxy, the `translate()` function can set it with `push_handlers()`.

Imagine that you want to include footers, headers, etc., piecing together a document, without using SSI. The following example shows how to implement it. First we prepare the code as follows:

```
Test/Compose.pm
-----
package Test::Compose;
use Apache::Constants qw(:common);

sub header {
    my $r = shift;
    $r->content_type("text/plain");
    $r->send_http_header;
    $r->print("header text\n");
    return OK;
}
sub body   { shift->print("body text\n") ; return OK}
sub footer { shift->print("footer text\n") ; return OK}
1;
__END__

# in httpd.conf or perl.conf
PerlModule Test::Compose
<Location /foo>
    SetHandler "perl-script"
    PerlHandler Test::Compose::header Test::Compose::body Test::Compose::footer
</Location>
```

Parsing the output of another PerlHandler? This is a little more tricky, but consider:

```
<Location /foo>
    SetHandler "perl-script"
    PerlHandler OutputParser SomeApp
</Location>

<Location /bar>
    SetHandler "perl-script"
    PerlHandler OutputParser AnotherApp
</Location>
```

Now, OutputParser goes first, but it `untie()`'s `*STDOUT` and `re-tie()`'s it to its own package like so:

```
package OutputParser;

sub handler {
    my $r = shift;
    untie *STDOUT;
```

```

    tie *STDOUT => 'OutputParser', $r;
}

sub TIEHANDLE {
    my ($class, $r) = @_;
    bless { r => $r}, $class;
}

sub PRINT {
    my $self = shift;
    for (@_) {
        #do whatever you want to $_ for example:
        $self->{r}->print($_ . "[insert stuff]");
    }
}

1;
__END__

```

To build in this feature, configure with:

```
% perl Makefile.PL PERL_STACKED_HANDLERS=1 [ ... ]
```

If you want to test whether your running mod\_perl Apache can stack handlers, the method `Apache->can_stack_handlers` will return TRUE if mod\_perl was configured with `PERL_STACKED_HANDLERS=1`, and FALSE otherwise.

## 1.4.8 Perl Method Handlers

If a Perl \*Handler is prototyped with \$\$, this handler will be invoked as a method. For example:

```

package MyClass;
@ISA = qw(BaseClass);

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

package BaseClass;

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

1;

```

Configuration:

```
PerlHandler MyClass
```

or

```
PerlHandler MyClass->handler
```

Since the handler is invoked as a method, it may inherit from other classes:

```
PerlHandler MyClass->method
```

In this case, the `MyClass` class inherits this method from `BaseClass`. This means that any method of `MyClass` or any of its parent classes can serve as a `mod_perl` handler, and that you can apply good OO methodology within your `mod_perl` handlers.

For instance, you could have this base class:

```
package ServeContent;

use Apache::Constants qw(OK);

sub handler($$) {
    my ($class, $r) = @_;

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

    return OK;
}

sub get_content {
    return 'Hello World';
}

1;
```

And then use the same base class for different contents:

```
package HelloWorld;

use ServeContent;
@ISA = qw(ServeContent);

sub get_content {
    return 'Hello, happy world!';
}

package GoodbyeWorld;

use ServeContent;
@ISA = qw(ServeContent);

sub get_content {
    return 'Goodbye, cruel world!';
}

1;
```

Now you can keep the same handler subroutine for a group of modules which are similar. The following configuration will enable the handlers from the subclasses:

```
<Location /hello>
    SetHandler perl-script
    PerlHandler HelloWorld->handler
</Location>

<Location /bye>
    SetHandler perl-script
    PerlHandler GoodbyeWorld->handler
</Location>
```

To build in this feature, configure with:

```
% perl Makefile.PL PERL_METHOD_HANDLERS=1 [ ... ]
```

## 1.4.9 PerlFreshRestart

To reload `PerlRequire`, `PerlModule` and other `use()`'d modules, and to flush the `Apache::Registry` cache on server restart, add to *httpd.conf*:

```
PerlFreshRestart On
```

Make sure you read Evil things might happen when using `PerlFreshRestart`.

Starting from `mod_perl` version 1.22 `PerlFreshRestart` is ignored when `mod_perl` is compiled as a DSO. But it almost doesn't matter, since `mod_perl` as a DSO will do a full tear-down (`perl_destruct()`). So it's still a *FreshRestart*, just fresher than static (non-DSO) `mod_perl` :)

But note that even if you have

```
PerlFreshRestart Off
```

and `mod_perl` as a DSO you will still get a *FreshRestart*.

## 1.4.10 PerlSetEnv and PerlPassEnv

```
PerlSetEnv key val
PerlPassEnv key
```

`PerlPassEnv` passes, `PerlSetEnv` sets and passes *ENVIRONMENT* variables to your scripts. You can access them in your scripts through `%ENV` (e.g. `$ENV{"key"}`). These commands are useful to pass information to your handlers or scripts, or to any modules you use that require some additional configuration.

For example, the Oracle RDBMS requires a number of `ORACLE_*` environment variables to be set so that you can connect to it through `DBI`. So you might want to put this in your *httpd.conf*:

```
PerlSetEnv ORACLE_BASE /oracle
PerlSetEnv ORACLE_HOME /oracle
:
```

You can then use DBI to access your oracle server without having to set the environment variables in your handlers.

PerlPassEnv proposes another approach: you might want to set the corresponding environment variables in your shell, and not reproduce the information in your *httpd.conf*. For example, you might have this in your *.bash\_profile*:

```
ORACLE_BASE=/oracle
ORACLE_HOME=/oracle
export ORACLE_BASE ORACLE_HOME
```

However, Apache (or mod\_perl) don't pass on environment variables from the shell by default; you'll have to specify these using either the standard PassEnv or mod\_perl's PerlPassEnv directives.

```
PerlPassEnv ORACLE_BASE ORACLE_HOME
```

One thing to be aware of is that when you start Apache under a shell different than the one you are logged in from, the environment variables could be totally different, so don't be surprised if you get a different value when using Passenv/PerlPassEnv or none at all. Check the environment Apache is started from. Often it's started from a special account like *apache*, or *nobody*, and can be anything else. Check the value of User variable in *httpd.conf* to find out the right answer. Once you figure that out, make sure that the shell Apache starts from has the desired environment variables right. And may be it's a better idea not to rely on the shell variables, but instead set those explicitly using Setenv/PerlSetEnv.

Regarding the setting of PerlPassEnv PERL5LIB in *httpd.conf*: if you turn on taint checks (PerlTaintCheck On), \$ENV{PERL5LIB} will be ignored (unset). See the 'Switches -w, -T' section.

While the Apache's SetEnv/PassEnv and mod\_perl's PerlSetEnv/PerlPassEnv apparently do the same thing, the former doesn't happen until the fixup phase, the latter happens as soon as possible, so those variables are available before then, e.g. in PerlAuthenHandler for \$ENV{ORACLE\_HOME} (or another environment variable that you need in these early request processing stages).

## 1.4.11 PerlSetVar and PerlAddVar

PerlSetVar is very similar to PerlSetEnv; however, variables set using PerlSetVar are only available through the mod\_perl API, and is thus more suitable for configuration. For example, environment variables are available to all, and might show up on casual "print environment" scripts, which you might not like. PerlSetVar is well-suited for modules needing some configuration, but not wanting to implement first-class configuration handlers just to get some information.

```
PerlSetVar foo bar
```

or

```
<Perl>
  push @{$Location{"/"}->{PerlSetVar} }, [ foo => 'bar' ];
</Perl>
```

and in the code you read it with:

```
my $r = Apache->request;
print $r->dir_config('foo');
```

The above prints:

```
bar
```

Note that you cannot do this:

```
push @{$Location{"/"}->{PerlSetVar} }, [ foo => \%bar ];
```

All values are treated as strings, so you will get a stringified reference to a hash as a value (something which will look like "HASH(0x87a5108)"). This cannot be turned back into a reference and therefore into the original hash upon retrieval.

However you can use the `PerlAddVar` directive to push more values into the variable, emulating arrays. For example:

```
PerlSetVar foo bar
PerlAddVar foo bar1
PerlAddVar foo bar2
```

or the equivalent:

```
PerlAddVar foo bar
PerlAddVar foo bar1
PerlAddVar foo bar2
```

To retrieve the values use the `$r->dir_config->get()` method:

```
my @foo = $r->dir_config->get('foo');
```

or

```
my %foo = $r->dir_config->get('foo');
```

Make sure that you use an even number of elements if you store the retrieved values in a hash, like this:

```
PerlAddVar foo key1
PerlAddVar foo value1
PerlAddVar foo key2
PerlAddVar foo value2
```

Then `%foo` will have a structure like this:

```
%foo = (
    key1 => 'value1',
    key2 => 'value2',
);
```

There are some things you should know about sub requests and `$r->dir_config`. For `$r->lookup_uri`, everything works as expected, because all normal phases are run. You can then retrieve variables set in the server scope of the configuration, in `<VirtualHost>` sections, in `<Location>` sections, etc.

However, when using the `$r->lookup_file` method, you are effectively skipping the URI translation phase. This means that the URI won't be known by Apache, only the file name to retrieve. As such, `<Location>` sections won't be applied. This means that if you were using:

```
Alias /perl-subr/ /home/httpd/perl-subr/
<Location /perl-subr>
    PerlSetVar foo bar
    PerlSetVar foo2 bar2
</Location>
```

And issue a subrequest using `$r->lookup_file` and try to retrieve its directory configuration (Apache::SubRequest class is just a subclass of Apache):

```
my $subr = $r->lookup_file('/home/httpd/perl-subr/script.pl');
print $subr->dir_config('foo');
```

You won't get the results you wanted.

As a side note: the issue we discussed here means that `/perl-subr/script.pl` won't even run under `mod_perl` if configured in the normal `Apache::Registry` way (using a `<Location>` section), because the `<Location>` blocks won't be applied. You'd have to use a `<Directory>` or `<Files>` section configuration to achieve the desired effect. As to the `PerlSetVar` discussion, using `<Directory>` or `<Files>` section would solve the problem.

## 1.4.12 PerlSetupEnv

`PerlSetupEnv` On will allow you to access the environment variables like `$ENV{REQUEST_URI}`, which are available under CGI. However, when programming handlers, there are always better ways to access these variables through the Apache API. Therefore, it is recommended to turn it Off except for scripts which absolutely require it. See `PerlSetupEnv Off`.

## 1.4.13 PerlWarn and PerlTaintCheck

For `PerlWarn` and `PerlTaintCheck` directives see the 'Switches -w, -T' section.

### ***1.4.14 MinSpareServers MaxSpareServers StartServers MaxClients MaxRequestsPerChild***

`MinSpareServers`, `MaxSpareServers`, `StartServers` and `MaxClients` are standard Apache configuration directives that control the number of servers that will be launched at server startup and kept alive during the server's operation.

`MaxRequestsPerChild` lets you specify the maximum number of requests which each child will be allowed to serve. When a process has served `MaxRequestsPerChild` requests the parent kills it and replaces it with a new one. There may also be other reasons why a child is killed, so it does not mean that each child will in fact serve this many requests, only that it will not be allowed to serve more than that number.

These five directives are very important for achieving the best performance from your server. The section 'Performance Tuning by Tweaking Apache Configuration' provides all the details.

## **1.5 The Startup File**

At server startup, before child processes are spawned to receive incoming requests, there is more that can be done than just preloading files. You might want to register code that will initialize a database connection for each child when it is forked, tie read-only dbm files, etc.

The `startup.pl` file is an ideal place to put the code that should be executed when the server starts. Once you have prepared the code, load it in `httpd.conf` before the rest of the `mod_perl` configuration directives like this:

```
PerlRequire /home/httpd/perl/lib/startup.pl
```

I must stress that all the code that is run at server initialization time is run with root privileges if you are executing it as the root user (which you have to do unless you choose to run the server on an unprivileged port, above 1024). This means that anyone who has write access to a script or module that is loaded by `PerlModule` or `PerlRequire` effectively has root access to the system. You might want to take a look at the new and experimental `PerlOpmask` directive and `PERL_OPMASK_DEFAULT` compile time option to try to disable some of the more dangerous operations.

Since the startup file is a file written in plain Perl, one can validate its syntax with:

```
% perl -c /home/httpd/perl/lib/startup.pl
```

### ***1.5.1 The Sample Startup File***

Let's look at a real world startup file:

```
startup.pl
-----
use strict;

# Extend @INC if needed
```

```

use lib qw(/dir/foo /dir/bar);

# Make sure we are in a sane environment.
$ENV{MOD_PERL} or die "not running under mod_perl!";

# For things in the "/perl" URL
use Apache::Registry;

# Load Perl modules of your choice here
# This code is interpreted *once* when the server starts
use LWP::UserAgent ();
use Apache::DBI ();
use DBI ();

# Tell me more about warnings
use Carp ();
$SIG{__WARN__} = \&Carp::cluck;

# Load CGI.pm and call its compile() method to precompile
# (but not to import) its autoloading methods.
use CGI ();
CGI->compile(':all');

# Initialize the database connections for each child
Apache::DBI->connect_on_init
("DBI:mysql:database=test;host=localhost",
 "user", "password",
 {
   PrintError => 1, # warn() on errors
   RaiseError => 0, # don't die on error
   AutoCommit => 1, # commit executes immediately
 }
);

1;

```

Now we'll review the code explaining why each line is used.

```
use strict;
```

This pragma is worth using in every script longer than half a dozen lines. It will save a lot of time and debugging later on.

```
use lib qw(/dir/foo /dir/bar);
```

The only chance to permanently modify @INC before the server is started is with this command. Later the running code can modify @INC just for the moment it require()'s some file, and then @INC's value gets reset to what it was originally.

```
$ENV{MOD_PERL} or die "not running under mod_perl!";
```

A sanity check, if Apache/mod\_perl wasn't properly built, the above code will abort the server startup.

```
use Apache::Registry;
use LWP::UserAgent ();
use Apache::DBI ();
use DBI ();
```

Preload the modules that get used by our Perl code serving the requests. Unless you need the symbols (variables and subroutines) exported by the modules you preload to accomplish something within the startup file, don't import them, since it's just a waste of startup time. Instead use the empty list `()` to tell the `import()` function not to import anything.

```
use Carp ();
$SIG{__WARN__} = \&Carp::cluck;
```

This is a useful snippet to enable extended warnings logged in the `error_log` file. In addition to basic warnings, a trace of calls is added. This makes the tracking of the potential problem a much easier task, since you know who called whom. For example, with normal warnings you might see:

```
Use of uninitialized value at
  /usr/lib/perl5/site_perl/5.005/Apache/DBI.pm line 110.
```

but you have no idea where it was called from. When we use `Carp` as shown above we might see:

```
Use of uninitialized value at
  /usr/lib/perl5/site_perl/5.005/Apache/DBI.pm line 110.
Apache::DBI::connect(undef, 'mydb::localhost', 'user',
  'passwd', 'HASH(0x87a5108)') called at
  /usr/lib/perl5/site_perl/5.005/i386-linux/DBI.pm line 382
DBI::connect('DBI', 'DBI:mysql:mydb::localhost', 'user',
  'passwd', 'HASH(0x8375e4c)') called at
  /usr/lib/perl5/site_perl/5.005/Apache/DBI.pm line 36
Apache::DBI::__ANON__('Apache=SCALAR(0x87a50c0)') called at
  PerlChildInitHandler subroutine
  'Apache::DBI::__ANON__' line 0
eval {...} called at PerlChildInitHandler subroutine
  'Apache::DBI::__ANON__' line 0
```

we clearly see that the warning was triggered by `eval()`uating the `Apache::DBI::__ANON__` which called `DBI::connect` (with the arguments that we see as well), which in turn called the `Apache::DBI::connect` method. Now we know where to look for our problem.

```
use CGI ();
CGI->compile(':all');
```

Some modules create their subroutines at run time to improve their load time. This helps when the module includes many subroutines, but only a few are actually used. `CGI.pm` falls into this category. Since with `mod_perl` the module is loaded only once, it might be a good idea to precompile all or a part of its methods.

`CGI.pm`'s `compile()` method performs this task. Notice that this is a proprietary function of this module, other modules can implement this feature or not and use this or some other name for this functionality. As with all modules we preload in the startup file, we don't import symbols from them as they will be lost when they go out of the file's scope.

Note that starting with CGI.pm version 2.46, the recommended method to precompile the code in CGI.pm is:

```
use CGI qw(-compile :all);
```

But the old method is still available for backward compatibility.

```
1;
```

As *startup.pl* is run through Perl's `require()`, it has to return a true value so that Perl can make sure it has been successfully loaded. Don't forget this (it's very easy to forget it).

See also the 'Apache::Status -- Embedded interpreter status information' section.

## ***1.5.2 What Modules You Should Add to the Startup File and Why***

Every module loaded at server startup will be shared among the server children, saving a lot of RAM on your machine. Usually I put most of the code I develop into modules and preload them.

You can even preload your CGI script with `Apache::RegistryLoader` (See Preload Perl modules at server startup) and you can get the children to preopen their database connections with `Apache::DBI`.

## ***1.5.3 The Confusion with use() in the Server Startup File***

Some people wonder why you need to duplicate the `use()` clause in the startup file and in the script itself. The confusion arises due to misunderstanding the `use()` function. `use()` normally performs two operations, namely `require()` and `import()`, called within a `BEGIN` block. See the section "use()" for a detailed explanation of the `use()`, `require()` and `import()` functions.

In the startup file we don't want to import any symbols since they will be lost when we leave the scope of the startup file anyway, i.e. they won't be visible to any of the child processes which run our `mod_perl` scripts. Instead we want to preload the module in the startup file and then import any symbols that we actually need in each script individually.

Normally when we write `use MyModule;`, `use()` will both load the module and import its symbols; so for the startup file we write `use MyModule ();` and the empty parentheses will ensure that the module is loaded but that no symbols are imported. Then in the actual `mod_perl` script we write `use()` in the standard way, e.g. `use MyModule;`. Since the module has already been preloaded, the only action taken is to import the symbols. For example in the startup file you write:

```
use CGI ();
```

since you probably don't need any symbols to be imported there. But in your code you would probably write:

```
use CGI qw(:html);
```

For example, if you have `use()`'d `Apache::Constants` in the startup file, it does not mean you can have the following handler:

```
package MyModule;
sub handler {
    my $r = shift;
    ## Cool stuff goes here
    return OK;
}
1;
```

You would either need to add:

```
use Apache::Constants qw( OK );
```

Or use the fully qualified name:

```
return Apache::Constants::OK;
```

If you want to use the function interface without exporting the symbols, use fully qualified function names, e.g. `CGI::param`. The same rule applies to variables, you can import variables and you can access them by their full name. e.g. `$My::Module::bar`. When you use the object oriented (method) interface you don't need to export the method symbols.

Technically, you aren't required to supply the `use()` statement in your (handler?) code if it was already loaded during server startup (i.e. by `'PerlRequire startup.pl'`). When writing your code, however, you should not assume the module code has been preloaded. In the future, you or someone else will revisit this code and will not understand how it is possible to use a module's methods without first loading the module itself.

Read the `Exporter` and `perlmod` manpages for more information about `import()`.

## 1.6 Apache Configuration in Perl

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

### 1.6.1 Usage

`<Perl>` sections can contain *any* and as much Perl code as you wish. These sections are compiled into a special package whose symbol table `mod_perl` can then walk and grind the names and values of Perl variables/structures through the Apache core configuration gears. Most of the configuration directives can be represented as scalars (`$scalar`) or lists (`@list`). A `@list` inside these sections is simply converted into a space delimited string for you. Here is an example:

```
httpd.conf
-----
<Perl>
@PerlModule = qw(Mail::Send Devel::Peek);

#run the server as whoever starts it
```

```

$User = getpwuid($>) || $>;
$Group = getgrgid($>) || $>;

$ServerAdmin = $User;

</Perl>

```

Block sections such as `<Location>..</Location>` are represented in a `%Location` hash, e.g.:

```

<Perl>

$Location{"/~doug/"} = {
    AuthUserFile => '/tmp/htpasswd',
    AuthType => 'Basic',
    AuthName => 'test',
    DirectoryIndex => [qw(index.html index.htm)],
    Limit => {
        METHODS => 'GET POST',
        require => 'user dougm',
    },
};

</Perl>

```

If an Apache directive can take two or three arguments you may push strings (the lowest number of arguments will be shifted off the `@list`) or use an array reference to handle any number greater than the minimum for that directive:

```

push @Redirect, "/foo", "http://www.foo.com/";

push @Redirect, "/imdb", "http://www.imdb.com/";

push @Redirect, [qw(temp "/here" "http://www.there.com")];

```

Other section counterparts include `%VirtualHost`, `%Directory` and `%Files`.

To pass all environment variables to the children with a single configuration directive, rather than listing each one via `PassEnv` or `PerlPassEnv`, a `<Perl>` section could read in a file and:

```

push @PerlPassEnv, [$key => $val];

```

or

```

Apache->httpd_conf("PerlPassEnv $key $val");

```

These are somewhat simple examples, but they should give you the basic idea. You can mix in any Perl code you desire. See *eg/httpd.conf.pl* and *eg/perl\_sections.txt* in the `mod_perl` distribution for more examples.

Assume that you have a cluster of machines with similar configurations and only small distinctions between them: ideally you would want to maintain a single configuration file, but because the configurations aren't *exactly* the same (e.g. the `ServerName` directive) it's not quite that simple.

<Perl> sections come to rescue. Now you have a single configuration file and the full power of Perl to tweak the local configuration. For example to solve the problem of the `ServerName` directive you might have this <Perl> section:

```
<Perl>
$ServerName = `hostname`;
</Perl>
```

For example if you want to allow personal directories on all machines except the ones whose names start with *secure*:

```
<Perl>
$ServerName = `hostname`;
if ( $ServerName !~ /^secure/) {
    $UserDir = "public.html";
} else {
    $UserDir = "DISABLED";
}
</Perl>
```

Behind the scenes, `mod_perl` defines a package called `Apache::ReadConfig`. Here it keeps all the variables that you define inside the <Perl> sections. Therefore it's not necessarily to configure the server within the <Perl> sections. Actually what you can do is to write the Perl code to configure the server just like you'd do in the <Perl> sections, but instead place it into a separate file that should be called during the configuration parsing with either `PerlModule` or `PerlRequire` directives, or from within the startup file. All you have to do is to declare the package `Apache::ReadConfig` within this file. Using the last example:

```
apache_config.pl
-----
package Apache::ReadConfig;

$ServerName = `hostname`;
if ( $ServerName !~ /^secure/) {
    $UserDir = "public.html";
} else {
    $UserDir = "DISABLED";
}

1;

httpd.conf
-----
PerlRequire /home/httpd/perl/lib/apache_config.pl
```

## 1.6.2 Enabling

To enable <Perl> sections you should build `mod_perl` with `perl Makefile.PL PERL_SECTIONS=1 [ ... ]`.

### 1.6.3 Caveats

Be careful when you declare package names inside `<Perl>` sections, for example this code has a problem:

```
<Perl>
package My::Trans;
use Apache::Constants qw(:common);
sub handler{ OK }

$PerlTransHandler = "My::Trans";
</Perl>
```

When you put code inside a `<Perl>` section, by default it actually goes into the `Apache::ReadConfig` package, which is already declared for you. This means that the `PerlTransHandler` we have tried to define above is actually undefined. If you define a different package name within a `<Perl>` section you must make sure to close the scope of that package and return to the `Apache::ReadConfig` package when you want to define the configuration directives, like this:

```
<Perl>
package My::Trans;
use Apache::Constants qw(:common);
sub handler{ OK }

package Apache::ReadConfig;
$PerlTransHandler = "My::Trans";
</Perl>
```

### 1.6.4 Verifying

This section shows how to check and dump the configuration you have made with the help of `<Perl>` sections in `httpd.conf`.

To check the `<Perl>` section syntax outside of `httpd`, we make it look like a Perl script:

```
<Perl>
# !perl
# ... code here ...
__END__
</Perl>
```

Now you may run:

```
perl -cx httpd.conf
```

In a running `httpd` you can see how you have configured the `<Perl>` sections through the URI `/perl-status`, by choosing *Perl Section Configuration* from the menu. In order to make this item show up in the menu you should set `$Apache::Server::SaveConfig` to a true value. When you do that the `Apache::ReadConfig` namespace (in which the configuration data is stored) will not be flushed, making configuration data available to Perl modules at request time.

Example:

```
<Perl>
$Apache::Server::SaveConfig = 1;

$DocumentRoot = ...
...
</Perl>
```

At request time, the value of **\$DocumentRoot** can be accessed with the fully qualified name **\$Apache::ReadConfig::DocumentRoot**.

You can dump the configuration of <Perl> sections like this:

```
<Perl>
use Apache::PerlSections();
...
# Configuration Perl code here
...
print STDERR Apache::PerlSections->dump();
</Perl>
```

Alternatively you can store it in a file:

```
Apache::PerlSections->store("httpd_config.pl");
```

You can then `require()` that file in some other <Perl> section.

## 1.6.5 *Strict <Perl> Sections*

If the Perl code doesn't compile, the server won't start. If the generated Apache config is invalid, <Perl> sections have always just logged an error and carried on, since there might be globals in the section that are not intended for the config.

The variable `$Apache::Server::StrictPerlSections` has been added in `mod_perl` version 1.22. If you set this variable to a true value, for example

```
$Apache::Server::StrictPerlSections = 1;
```

then `mod_perl` will not tolerate invalid Apache configuration syntax and will `croak` (die) if this is the case. At the time of writing the default value is 0.

## 1.6.6 *Debugging*

If you compile `mod_perl` with `PERL_TRACE=1` and set the environment variable `MOD_PERL_TRACE` then you should see some useful diagnostics when `mod_perl` is processing <Perl> sections.

## 1.6.7 Perl Section Tricks

- The Perl `%ENV` is cleared during startup, but the C environment is left intact and so you can use it to set `@PassEnv`.

## 1.6.8 References

For more info see *Writing Apache Modules with Perl and C*, Chapter 8: <http://modperl.com:9000/book/chapters/ch8.html>

## 1.7 Validating the Configuration Syntax

`apachectl configtest` tests the configuration file without starting the server. You can safely validate the configuration file on your production server, if you run this test before you restart the server with `apachectl restart`. Of course it is not 100% perfect, but it will reveal any syntax errors you might have made while editing the file.

'`apachectl configtest`' is the same as '`httpd -t`' and it doesn't just parse the code in `startup.pl`, it actually executes it. `<Perl>` configuration has always started Perl during the configuration read, and `Perl{Require,Module}` do so as well.

Of course we assume that the code that gets called during this test cannot cause any harm to your running production environment. The following hint shows how to prevent the code in the startup script and `<Perl>` from being executed during the syntax check, if that's what you want.

If you want your startup code to get control over the `-t` (`configtest`) server launch, start the server configuration test with:

```
httpd -t -Dsyntax_check
```

and, if for example you want to prevent your startup code from being executed, at the top of the code add:

```
return if Apache->define('syntax_check');
```

## 1.8 Enabling Remote Server Configuration Reports

The nifty `mod_info` module displays the complete server configuration in your browser. In order to use it you have compile it in or, if the server was compiled with DSO mode enabled, load it as an object. Then just uncomment the ready-prepared section in the `httpd.conf` file:

```
<Location /server-info>
  SetHandler server-info
  Order deny,allow
  Deny from all
  Allow from www.example.com
</Location>
```

Now restart the server and issue the request:

```
http://www.example.com/server-info
```

## 1.9 Publishing Port Numbers other than 80

If you are using a two-server setup, with a mod\_perl server listening on a high port, it is advised that you do not publish the number of the high port number in URLs. Rather use a proxying rewrite rule in the non-mod\_perl server:

```
RewriteEngine      On
RewriteLogLevel    0
RewriteRule        ^/perl/(.*) http://localhost:8080/perl/$1 [P]
ProxyPassReverse   /          http://localhost/
```

I was told one problem with publishing high port numbers is that IE 4.x has a bug when re-posting data to a non-port-80 URL. It drops the port designator, and uses port 80 anyway.

Another reason is that firewalls probably will have the high port closed, therefore users behind the firewalls will be unable to reach your service, running on the blocked port.

## 1.10 Configuring Apache + mod\_perl with mod\_macro

mod\_macro is an Apache module written by Fabien Coelho that lets you define and use macros in the Apache configuration file.

mod\_macro can be really useful when you have many virtual hosts, and where each virtual host has a number of scripts/modules, most of them with a moderately complex configuration setup.

First download the latest version of mod\_macro from [http://www.cri.ensmp.fr/~coelho/mod\\_macro/](http://www.cri.ensmp.fr/~coelho/mod_macro/), and configure your Apache server to use this module.

Here are some useful macros for mod\_perl users:

```
# set up a registry script
<Macro registry>
SetHandler "perl-script"
PerlHandler Apache::Registry
Options +ExecCGI
</Macro>

# example
Alias /stuff /usr/www/scripts/stuff
<Location /stuff>
Use registry
</Location>
```

If your registry scripts are all located in the same directory, and your aliasing rules consistent, you can use this macro:

```
# set up a registry script for a specific location
<Macro registry $location $script>
Alias /$location /home/httpd/perl/scripts/$script
<Location /$location>
SetHandler "perl-script"
PerlHandler Apache::Registry
Options +ExecCGI
</Location>
</Macro>

# example
Use registry stuff stuff.pl
```

If you're using content handlers packaged as modules, you can use the following macro:

```
# set up a mod_perl content handler module
<Macro modperl $module>
SetHandler "perl-script"
Options +ExecCGI
PerlHandler $module
</Macro>

#examples
<Location /perl-status>
PerlSetVar StatusPeek On
PerlSetVar StatusGraph On
PerlSetVar StatusDumper On
Use modperl Apache::Status
</Location>
```

The following macro sets up a Location for use with `HTML::Embperl`. Here we define all ".html" files to be processed by `Embperl`.

```
<Macro embperl>
SetHandler "perl-script"
Options +ExecCGI
PerlHandler HTML::Embperl
PerlSetEnv EMBPERL_FILESMATCH \.html$
</Macro>

# examples
<Location /mrtg>
Use embperl
</Location>
```

Macros are also very useful for things that tend to be verbose, such as setting up Basic Authentication:

```
# Sets up Basic Authentication
<Macro BasicAuth $realm $group>
Order deny,allow
Satisfy any
AuthType Basic
AuthName $realm
AuthGroupFile /usr/www/auth/groups
AuthUserFile /usr/www/auth/users
Require group $group
```

```
Deny from all
</Macro>

# example of use
<Location /stats>
Use BasicAuth WebStats Admin
</Location>
```

Finally, here is a complete example that uses macros to set up simple virtual hosts. It uses the BasicAuth macro defined previously (yes, macros can be nested!).

```
<Macro vhost $ip $domain $docroot $admingroup>
<VirtualHost $ip>
ServerAdmin webmaster@$domain
DocumentRoot /usr/www/htdocs/$docroot
ServerName www.$domain
<Location /stats>
Use BasicAuth Stats-$domain $admingroup
</Location>
</VirtualHost>
</Macro>

# define some virtual hosts
Use vhost 10.1.1.1 example.com example example-admin
Use vhost 10.1.1.2 example.net examplenet examplenet-admin
```

mod\_macro is also useful in a non vhost setting. Some sites for example have lots of scripts which people use to view various statistics, email settings and etc. It is much easier to read things like:

```
use /forwards email/showforwards
use /webstats web/showstats
```

The actual macros for the last example are left as an exercise to reader. These can be easily constructed based on the examples presented in this section.

## 1.11 General Pitfalls

### *1.11.1 My CGI/Perl Code Gets Returned as Plain Text Instead of Being Executed by the Webserver*

Check your configuration files and make sure that the ExecCGI is turned on in your configurations.

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

### ***1.11.2 My Script Works under mod\_cgi, but when Called via mod\_perl I Get a 'Save-As' Prompt***

Did you put `PerlSendHeader On` in the configuration part of the `<Location foo></Location>`.

### ***1.11.3 Is There a Way to Provide a Different startup.pl File for Each Individual Virtual Host***

No. Any virtual host will be able to see the routines from a `startup.pl` loaded for any other virtual host.

### ***1.11.4 Is There a Way to Modify @INC on a Per-Virtual-Host or Per-Location Basis.***

You can use `PerlSetEnv PERL5LIB ...` or a `PerlFixupHandler` with the `lib` pragma (use `lib qw(...)`).

A better way is to use `Apache::PerlVINC`

### ***1.11.5 A Script From One Virtual Host Calls a Script with the Same Path From the Other Virtual Host***

This has been a bug before, last fixed in 1.15\_01, i.e. if you are running 1.15, that could be the problem. You should set this variable in a startup file (which you load with `PerlRequire` in `httpd.conf`):

```
$Apache::Registry::NameWithVirtualHost = 1;
```

But, as we know sometimes a bug turns out to be a feature. If the same script is running for more than one Virtual host on the same machine, this can be a waste, right? Set it to 0 in a startup script if you want to turn it off and have this bug as a feature. (Only makes sense if you are sure that there will be no *other* scripts with the same path/name). It also saves you some memory as well.

```
$Apache::Registry::NameWithVirtualHost = 0;
```

### ***1.11.6 the Server no Longer Retrieves the DirectoryIndex Files for a Directory***

The problem was reported by users who declared `mod_perl` configuration inside a `<Directory>` section for all files matching `*.pl`. The problem went away after placing the directives in a `<Files>` section.

The `mod_alias` and `mod_rewrite` are both `Trans` handlers in the normal case. So in the setup where both are used, if `mod_alias` runs first and matches it will return OK and `mod_rewrite` won't see the request.

The opposite can happen as well, where `mod_rewrite` rules apply but the `Alias` directives are completely ignored.

The behavior is not random, but depends on the Apache modules loading order. Apache modules are being executed in *reverse* order, i.e. module that was *Added* first will be executed last.

The solution is not to mix `mod_rewrite` and `mod_alias`. `mod_rewrite` does everything `mod_alias` does--except for `ScriptAlias` which is not really relevant to `mod_perl` anyway. Don't rely on the module ordering, but use explicitly disjoint URL namespaces for `Alias` and `Rewrite`. In other words any URL regex that can potentially match a `Rewrite` rule should not be used in an `Alias`, and vice versa. Given that `mod_rewrite` can easily do what `mod_alias` does, it's no problem.

Here is one of the examples where `Alias` is replaced with `RedirectMatch`. This is a snippet of configuration at the light non-`mod_perl` Apache server:

```
RewriteEngine      on
RewriteLogLevel    0
RewriteRule        ^/(perl.*)$    http://127.0.0.1:8045/$1    [P,L]
RewriteRule        ^/(mail.*)$    http://127.0.0.1:8045/$1    [P,L]
NoCache            *
ProxyPassReverse   / http://www.example.com/

RedirectMatch permanent ^/$        /pages/index
RedirectMatch permanent ^/foo$     /pages/bar
```

This configuration works fine because any URI that matches one of the redirects will never match one of the rewrite rules.

In the above setup we proxy requests starting with `/perl` or `/mail` to the `mod_perl` server, forbid proxy requests to the external sites, and make sure that the proxied requests will use the `http://www.example.com/` as their URL on the way back to the client.

The `RedirectMatch` settings work exactly like if you'd write:

```
Alias /           /pages/index
Alias /foo       /pages/bar
```

But as we told before we don't want to mix the two.

Here is another example where the redirect is done by a rewrite rule:

```
RewriteEngine      on
RewriteLogLevel    0
RewriteMap         lowercase int:tolower
RewriteRule        ^/(perl.*)$    http://127.0.0.1:8042/$1    [P,L]
RewriteRule        ^/$            /pages/welcome.htm         [R=301,L]
RewriteRule        ^(.*)$         ${lowercase:$1}
NoCache            *
ProxyPassReverse   / http://www.example.com/
```

If we omit the rewrite rule that matches `^/$`, and instead use a redirect, it will never be called, because the URL is still matched by the last rule `^(.*)$`. This is a somewhat contrived example because that last regex could be rewritten as `^(/.+)$` and all would be well.

### ***1.11.7 Do Perl\* Directives Affect Code Running under mod\_cgi?***

No, they don't.

So for example if you do:

```
PerlSetEnv foo bar
```

It'll be seen from `mod_perl`, but not `mod_cgi` or any other module.

## **1.12 Configuration Security Concerns**

The more modules you have in your web server, the more complex the code.

The more complex the code in your web server, the more chances for bugs.

The more chances for bugs, the more chance that some of those bugs may involve security breaches.

### ***1.12.1 Choosing User and Group***

Because `mod_perl` runs within an `httpd` child process, it runs with the `User` ID and `Group` ID specified in the `httpd.conf` file. This `User/Group` should have the lowest possible privileges. It should only have access to world readable files, even better only files that belongs to this user. Even so, careless scripts can give away information. You would not want your `/etc/passwd` file to be readable over the net, for instance, even if you use shadow passwords.

When a handler needs write permissions, make sure that only the user, the server is running under, has write permissions to the files. Sometimes you need group write permissions, but be very careful, because a buggy or malicious code run in the server may destroy files writable by the server.

### ***1.12.2 Taint Checking***

Make sure to run the server under:

```
PerlTaintCheck On
```

setting in the `httpd.conf` file. Taint checking doesn't ensure that your code is completely safe from external hacks, but it does forces you to improve your code to prevent many potential security problems.

### 1.12.3 Exposing Information About the Server's Component

It is better not to expose the `mod_perl` server to the outside world, for it creates a potential security risk by revealing which Apache modules used by the server and the OS the server is running on.

You can see what information is revealed by your server, by telnetting to it and issuing some request. For example:

```
% telnet localhost 8080
Trying 127.0.0.1
Connected to localhost
Escape character is '^]'.
HEAD / HTTP1.0

HTTP/1.1 200 OK
Date: Sun, 16 Apr 2000 11:06:25 GMT
Server: Apache/1.3.12 (Unix) mod_perl/1.22 mod_ssl/2.6.2 OpenSSL/0.9.5
[more lines snipped]
```

So as you see that a lot of information is revealed and a `Full ServerTokens` has been used.

We never were completely sure why the default of the `ServerTokens` directive in Apache is `Full` rather than `Minimal`. Seems like you would only make it `Full` if you are debugging. Probably the reason for using the `ServerTokens Full` is for a show-off, so Netcraft (<http://netcraft.com>) and other similar survey services will count more Apache servers, which is good for all of us, but you really want to reveal as little information as possible to the potential crackers.

Another approach is to modify `httpd` sources to reveal no unwanted information, so all responses will return an empty or phony `Server:` field.

From the other point of view, security by obscurity is a lack of security. Any determined cracker will eventually figure out what version of Apache run and what third party modules you have built in.

An even better approach is to completely hide the `mod_perl` server behind a front-end or a proxy server, so the server cannot be accessed directly.

## 1.13 Apache Restarts Twice On Start

When the server is restarted, the configuration and module initialization phases are called twice in total before the children are forked. The second restart is done in order to ensure that future restarts will work correctly, by making sure that all modules can survive a restart (`SIGHUP`). This is very important if you restart a production server.

You can control what code will be executed on the start or restart by checking the value of `$Apache::Server::Starting` and `$Apache::Server::ReStarting` respectively. The former variable is `true` when the server is starting and the latter is `true` when it's restarting.

For example:

```
<Perl>
print STDERR "Server is Starting\n"    if $Apache::Server::Starting;
print STDERR "Server is ReStarting\n"  if $Apache::Server::ReStarting;
</Perl>
```

The *startup.pl* file and similar loaded via `PerlModule` or `PerlRequire` are compiled only once. Because once the module is compiled it enters the special `%INC` hash. When Apache restarts--Perl checks whether the module or script in question is already registered in `%INC` and won't try to compile it again.

So the only code that you might need to protect from running on restart is the one in the `<Perl>` sections. But since one usually uses the `<Perl>` sections mainly for on the fly configuration creation, there shouldn't be a reason why it'd be undesirable to run the code more than once.

## 1.14 Knowing the proxy\_pass'ed Connection Type

Let's say that you have a frontend server running `mod_ssl`, `mod_rewrite` and `mod_proxy`. You want to make sure that your user is using a secure connection for some specific actions like login information submission. You don't want to let the user login unless the request was submitted through a secure port.

Since you have to proxy\_pass the request between front and backend servers, you cannot know where the connection has come from. Neither is using the HTTP headers reliable.

A possible solution for this problem is to have the `mod_perl` server listen on two different ports (e.g. 8000 and 8001) and have the `mod_rewrite` proxy rule in the regular server redirect to port 8000 and the `mod_rewrite` proxy rule in the SSL virtual host redirect to port 8001. In the `mod_perl` server just check the `PORT` variable to tell if the connection is secure.

## 1.15 Adding Custom Configuration Directives

This is covered in the Eagle Book in a great detail. This is just a simple example, showing how to add your own Configuration directives.

```
Makefile.PL
-----
package Apache::TestDirective;

use ExtUtils::MakeMaker;

use Apache::ExtUtils qw(command_table);
use Apache::src ();

my @directives = ({
    name          => 'Directive4',
    errmsg        => 'Anything',
    args_how      => 'RAW_ARGS',
    req_override => 'OR_ALL',
});
```

## 1.15 Adding Custom Configuration Directives

```
command_table(\@directives);

WriteMakefile(
    NAME      => 'Apache::TestDirective',
    VERSION_FROM => 'TestDirective.pm',
    INC       => Apache::src->new->inc,
);

TestDirective.pm
-----
package Apache::TestDirective;

use strict;
use Apache::ModuleConfig ();
use DynaLoader ();

if ($ENV{MOD_PERL}) {
    no strict;
    $VERSION = '0.01';
    @ISA = qw(DynaLoader);
    __PACKAGE__->bootstrap($VERSION); #command table, etc.
}

sub Directive4 {

    warn "Directive4 @_ \n";
}

1;
__END__
```

In the `mod_perl` source tree, add this to `t/docs/startup.pl`:

```
use blib qw(/home/doug/m/test/Apache/TestDirective);
```

and at the bottom of `t/conf/httpd.conf`:

```
PerlModule Apache::TestDirective
Directive4 hi
```

Test it:

```
% make start_httpd
% make kill_httpd
```

You should see:

```
Directive4 Apache::TestDirective=HASH(0x83379d0)
Apache::CmdParms=SCALAR(0x862b80c) hi
```

And in the error log file:

```
% grep Directive4 t/logs/error_log
Directive4 Apache::TestDirective=HASH(0x83119dc)
Apache::CmdParms=SCALAR(0x8326878) hi
```

If it didn't work as expected try building mod\_perl with `PERL_TRACE=1`, then do:

```
setenv MOD_PERL_TRACE all
```

before starting the server. Now you should get some useful diagnostics.

## 1.16 Maintainers

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

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

## 1.17 Authors

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

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



## Table of Contents:

1	mod_perl Configuration	1
1.1	Description	2
1.2	Server Configuration	2
1.3	Apache Configuration	2
1.3.1	Configuration Directives	3
1.3.2	.htaccess files	3
1.3.3	<Directory>, <Location> and <Files> Sections	3
1.3.4	How Directory, Location and Files Sections are Merged	6
1.3.5	Sub-Grouping of <Location>, <Directory> and <Files> Sections	7
1.3.6	Options Directive	7
1.4	mod_perl Configuration	8
1.4.1	Alias Configurations	8
1.4.1.1	Running CGI, PerlRun, and Registry Scripts Located in the Same Directory	9
1.4.2	<Location> Configuration	10
1.4.3	Overriding <Location> Setting in "Sub-Location"	12
1.4.4	PerlModule and PerlRequire Directives	12
1.4.5	Perl*Handlers	12
1.4.6	The handler subroutine	14
1.4.7	Stacked Handlers	15
1.4.8	Perl Method Handlers	18
1.4.9	PerlFreshRestart	20
1.4.10	PerlSetEnv and PerlPassEnv	20
1.4.11	PerlSetVar and PerlAddVar	21
1.4.12	PerlSetupEnv	23
1.4.13	PerlWarn and PerlTaintCheck	23
1.4.14	MinSpareServers MaxSpareServers StartServers MaxClients MaxRequestsPerChild	24
1.5	The Startup File	24
1.5.1	The Sample Startup File	24
1.5.2	What Modules You Should Add to the Startup File and Why	27
1.5.3	The Confusion with use() in the Server Startup File	27
1.6	Apache Configuration in Perl	28
1.6.1	Usage	28
1.6.2	Enabling	30
1.6.3	Caveats	31
1.6.4	Verifying	31
1.6.5	Strict <Perl> Sections	32
1.6.6	Debugging	32
1.6.7	Perl Section Tricks	33
1.6.8	References	33
1.7	Validating the Configuration Syntax	33
1.8	Enabling Remote Server Configuration Reports	33
1.9	Publishing Port Numbers other than 80	34
1.10	Configuring Apache + mod_perl with mod_macro	34
1.11	General Pitfalls	36

1.11.1 My CGI/Perl Code Gets Returned as Plain Text Instead of Being Executed by the Webservice . . . . .	36
1.11.2 My Script Works under mod_cgi, but when Called via mod_perl I Get a 'Save-As' Prompt	37
1.11.3 Is There a Way to Provide a Different startup.pl File for Each Individual Virtual Host . . . . .	37
1.11.4 Is There a Way to Modify @INC on a Per-Virtual-Host or Per-Location Basis. . . . .	37
1.11.5 A Script From One Virtual Host Calls a Script with the Same Path From the Other Virtual Host . . . . .	37
1.11.6 the Server no Longer Retrieves the DirectoryIndex Files for a Directory . . . . .	37
1.11.7 Do Perl* Directives Affect Code Running under mod_cgi? . . . . .	39
1.12 Configuration Security Concerns . . . . .	39
1.12.1 Choosing User and Group . . . . .	39
1.12.2 Taint Checking . . . . .	39
1.12.3 Exposing Information About the Server's Component . . . . .	40
1.13 Apache Restarts Twice On Start . . . . .	40
1.14 Knowing the proxy_pass'ed Connection Type . . . . .	41
1.15 Adding Custom Configuration Directives . . . . .	41
1.16 Maintainers . . . . .	43
1.17 Authors . . . . .	43