

ApacheCon US
April 4, 2001
Santa Clara, CA

Tutorial:

Getting started with mod_perl

by Stas Bekman
<http://stason.org/>
[<stas@stason.org>](mailto:stas@stason.org)
eXtropia.com, Senior Engineer

This document is originally written in **POD**, converted to **HTML**,
PostScript and **PDF** by Pod: :HtmlPsdF Perl module.

1 Agenda

1.1 Agenda

- mod_perl Introduction
- Basic Configuration
- Basic Scripts and Handlers
- Server Setup Strategies
- CGI to mod_perl Porting
- mod_perl Coding Guidelines

1.2 Off-tutorial reading

- Perl Reference
- Getting Help

```
;0
```


2 Getting Started Fast

2.1 mod_perl in Four Slides

- Installation
- Configuration
- The “mod_perl rules” Apache::Registry Scripts
- The “mod_perl rules” Apache Perl Module

2.2 What is mod_perl?

Solves numerous mod_cgi shortcomings:

- Embedded Perl Interpreter -- no loading overhead
- Code compiled only once per process life -- no compilation overhead
- No forking per request -- process reuse
- Response processing is now reduced to running your code.
- Response times improve by a factor of 10 to 100

- A bigger size, but just a few processes can handle a much bigger load
- `mod_cgi` compatibility preserved (Apache::Registry and Apache::PerlRun modules)
- Persistent database connections

Extended mod_cgi's functionality:

- A complete Perl API added to the Apache core
- Handling of all phases of request processing in Perl.
- Writing complete Apache modules in Perl
- Complete server configuration in Perl.
- Numerous 3rd party modules are available

Logistics:

- Developed by Doug MacEachern
- Licensed under the Apache Software License.
- Home page <http://perl.apache.org>
- Mailing list: send an empty email to modperl-subscribe@apache.org
- January 2001 -- 2 Million mod_perl hosts (according to <http://perl.apache.org/netcraft/>)

2.3 Installation

```
% lwp-download \  
http://www.apache.org/dist/apache_x.x.x.tar.gz  
% lwp-download \  
http://perl.apache.org/dist/mod_perl-x.xx.tar.gz  
% tar xzvf apache_x.x.x.tar.gz  
% tar xzvf mod_perl-x.xx.tar.gz  
% cd mod_perl-x.xx  
% perl Makefile.PL APACHE_SRC=../apache_x.x.x/src \  
DO_HTTPD=1 USE_APACI=1 EVERYTHING=1  
% make && make test && make install  
% cd ../apache_x.x.x && make install
```

- **That's all!**

2.4 Configuration

- Add the following to the configuration file *httpd.conf*:

```
# for Apache::Registry mode
Alias /perl/ /home/httpd/perl/

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

2.5 The "mod_perl rules" Apache::Registry Scripts

- You can write plain perl/CGI scripts just as under mod_cgi:

```
mod_perl_rules1.pl
-----
print "Content-type: text/plain\r\n\r\n";
print "mod_perl rules!\n";
```

- Of course you can write them in the Apache Perl API:

```
mod_perl_rules2.pl
-----
my $r = shift;
$r->send_http_header('text/plain');
$r->print("mod_perl rules!\n");
```

- Save both files under the */home/httpd/perl* directory
- Make them executable and readable by server,
- and issue these requests using your favorite browser:

```
http://localhost/perl/mod_perl_rules1.pl  
http://localhost/perl/mod_perl_rules2.pl
```
- In both cases you will see on the following response:

```
mod_perl rules!
```

2.6 The "mod_perl rules" Apache Perl Module

- To create an Apache Perl module, all you have to do is to wrap the code into a `handler` subroutine:

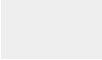
```
ModPerl/Rules.pm
-----
package ModPerl::Rules;
use Apache::Constants;

sub handler{
    my $r = shift;
    $r->send_http_header('text/plain');
    print "mod_perl rules!\n";
    return OK;
}
1;
```

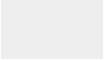
- Create a directory called *ModPerl* under one of the directories in `@INC`
- and put *Rules.pm* into it.
- Then add the following snippet to *httpd.conf*:

```
PerlModule ModPerl::Rules  
<Location /mod_perl_rules>  
    SetHandler perl-script  
    PerlHandler ModPerl::Rules  
</Location>
```

- Now you can issue a request to:

 `http://localhost/mod_perl_rules`

- and just as with our *mod_perl_rules.pl* scripts you will see:

 `mod_perl rules!`

- as the response.

2.7 Is That All I Need To Know About `mod_perl`?

- Definitely not!
- These slides are intended to show you that you can install and start using a `mod_perl` server within 30 minutes of downloading the sources.
- There is much more to `mod_perl` than this.
- Fortunately, there are many resources and lots of help freely available to you.

- See the last chapter of this tutorial for the help references.

```
;0
```


3 Server Setup Strategies

3.1 What we will learn in this chapter

- mod_perl Deployment Overview
- Standalone mod_perl Enabled Apache Server
- One Plain Apache and One mod_perl-enabled Apache Servers
- Adding a Proxy Server in http Accelerator Mode

3.2 mod_perl Deployment Overview

- There are several different ways to build, configure and deploy your mod_perl enabled server.
- Some of them are:
 1. 1 mod_perl server
 2. 1 light Apache and 1 mod_perl servers
 3. Any of the above plus a reverse proxy server in http accelerator mode.

One server:

- (+) while learning you will have no other server interaction to mask or add to your errors.
- (-) will kill your production site if you serve a lot of static data from large webserver processes.

Two servers:

- (+) allows you to tune the two servers individually, for maximum performance.
- (-) you need to choose between running the two servers on multiple ports, multiple IPs, etc.,
- (-) you have the burden of administering more than one server.
- (-) you have to deal with proxying or fancy site design to keep the two servers in synchronization.

Opt1 or opt2 plus proxy:

- (+) once correctly configured and tuned, improves the performance of any of the above two options by caching and buffering page results.
- (-) more configuration to do (one time)

3.3 Standalone mod_perl Enabled Apache Server

The advantages:

- Simplicity. Copy-n-paste instructions and you are done.
- No network/IPs/ports changes.
- Blazing speed.

The disadvantages:

- **Process size**
 - (-) usually 5-10MB and more
 - (+) but memory sharing helps a lot!!!
 - (-) many processes, more memory
 - (+) but less processes than with mod_cgi
 - (+) memory is cheap
 - (-) a waste if serving static objects

- **Serving slow clients**

- serving output to a client with a slow connection,
- a process is tied before all of the response is sent to a client.
- output generation: 20-500 millisec
- sending output: 10-60 sec (10,000 - 60,000 millisec)

Conclusions:

- Best to start with if you are new!
- Perfect choice if you server only mod_perl scripts
- A good choice for Intranet sites (very fast delivery!)

3.4 One Plain Apache and One mod_perl-enabled Apache Servers

The advantages:

- **Less memory**
 - Fewer processes -- less total memory used
- **Better tuning**
 - Optimal tuning of `MaxClients`, `MaxRequestsPerChild` etc. for each of the servers to utilize better the resources.

- **Many lightweight `httpd_docs` servers**
- **just a few heavy `httpd_perl` servers.**

A catch: **Relative URLs:**

```
http://www.example.com:8080/perl/example.pl
```

- The above URL returns a page with relative links to images
- Where the images will be brought from?
- Of course from the mod_perl heavy server
http://www.example.com:8080 --
- Solution: fully qualified URIs

```
s{/img/foo\.gif}{http://www.example.com/img/foo.gif};
```

The disadvantages:

- Two sets of configuration files
- Two sets of controlling scripts + watchdogs
- Logs merging
- Still having a problem of serving slow clients

3.5 Adding a Proxy Server in http Accelerator Mode

- At the beginning there were 2 servers:
- One plain apache server, which was *very light*, and configured to serve static objects,
- The other mod_perl enabled (*very heavy*) and configured to serve mod_perl scripts.
- We name them `httpd_docs` and `httpd_perl` respectively.
- Usually the two servers coexist at the same IP address by listening to different ports:

- 80 for `httpd_docs`
- 8080 for `httpd_perl`
- Now why would you want to use a proxy server (in the `httpd_accelerator` mode).

The advantages:

- **Proxy cache**
 - Serve static objects from cache
 - Less IO -- higher throughput

- **Output Buffering**

- The proxy server acts as a sort of output buffer for the dynamic content.
- The mod_perl server sends the entire response to the proxy and is then free to deal with other requests.
- The proxy server is responsible for sending the response to the browser.
- So if the transfer is over a slow link, the mod_perl server is not waiting around for the data to move.
- Using numbers is always more convincing :)

- 56 kbps connection => **7 Kbytes/sec**. (1 byte = 8 bit)
- An average generated HTML page to be of **42kb**
- An average script that generates this output in 0.5 second
- How long will the server wait before the user gets the whole output response?
- A simple calculation reveals pretty scary numbers:
 - **42 KB / (0.5 sec * 7 KB/sec) ~ 12**
- 11 (12-1) other dynamic requests could be served during this time

- Usually pages are generally much bigger than 42Kb
- and users tend to open more than one browser at the same time
- Result: The waiting time can grow 10 times and more

- **Hiding Implementation Details**

- Users will never see ports in the URLs
- Being able to shut down one server and tell the front end to send request to another machine.
- Load ballancing transparent to users

- **Security protection**

- Makes your internal server inaccessible from outside -- listens on 127.0.0.1 (or NAT).
- Prevents from getting directly attacked by arbitrary packets from whomever.
- This allows for only your public “bastion” accelerating www server to get hosed in a successful attack, while leaving your internal data safe.

The disadvantages

- **Administration overhead**
 - You have another daemon to worry about
 - One more startup/shutdown/watchdog script
- **Memory Usage**
 - Adding to memory usage: Proxy servers can be configured to be light or heavy
 - Squid runs only a single process (threaded) but might consume a lot of memory

- Have I succeeded in convincing you that you want a proxy server?
- If you are on a local area network (LAN), then the big benefit of the proxy buffering the output and feeding a slow client is gone.
- You are probably better off sticking with a straight `mod_perl` server in this case.

3.6 Implementations of Proxy Servers

- This section is located in your handouts and was left as an off-tutorial reading.

```
;0
```


4 Porting from CGI Scripts and mod_perl Coding Guidelines.

4.1 What we will learn in this chapter

- Exposing Apache::Registry secrets
- Sometimes it Works, Sometimes it Doesn't
- @INC and mod_perl
- Reloading Modules and Required Files
- Name collisions with Modules and libs
- __END__ and __DATA__ tokens

- Output from system calls
- Terminating requests and processes
- `die()` and `mod_perl`
- `Apache::print()` and `CORE::print()`
- Global Variables Persistence
- Command line Switches (`-w`, `-T`, etc)

4.2 Exposing Apache::Registry secrets

- `Apache::Registry` is a `mod_cgi` compatible module
- It doesn't like sloppy programming style
- There are a few hidden issues to know about.
- Let's start with some simple code
- Detect bugs and debug them,
- Discuss possible pitfalls and how to avoid them.

- A simple CGI script, that initializes a `$counter` to 0, and prints its value while incrementing it.

```
counter.pl:
-----
use strict;
print "Content-type: text/plain\r\n\r\n";

my $counter = 0;
for (1..5) {
    increment_counter();
}
sub increment_counter{
    $counter++;
    print "Counter is equal to $counter !\r\n";
}
```

- You would expect to see the output:

```
Counter is equal to 1 !  
Counter is equal to 2 !  
Counter is equal to 3 !  
Counter is equal to 4 !  
Counter is equal to 5 !
```

- And that's what you see when you execute this script the first time
- But let's reload it a few times...

- Suddenly after a few reloads the counter doesn't start its count from 1 any more.

```
Counter is equal to 6 !  
Counter is equal to 7 !  
Counter is equal to 8 !  
Counter is equal to 9 !  
Counter is equal to 10 !
```

- We continue to reload and see that it keeps on growing, but not steadily starting almost randomly at 10, 10, 10, 15, 20... Weird...

We saw two anomalies in this very simple script:

- **Unexpected increment of our counter over 5**
- **Inconsistent growth over reloads.**

4.2.1 The First Mystery

- The `error_log` file says:

```
Variable "$counter" will not stay shared  
at /home/httpd/perl/conference/counter.pl line 13.
```

- Add `'use diagnostics;'` to see the long version of the warning.
- A named nested subroutine that refers to a lexically scoped variable defined outside this nested subroutine? Where?
- Perl sees the script in a different way?

- A debugger to rescue!!
- How?
- Special debugger for mod_perl -- via `Apache::DB`
- Interactive and non-interactive debugging
- We go for non-interactive debug here

- *httpd.conf*:

```
PerlSetEnv PERLDB_OPTS \  
"NonStop=1 LineInfo=/tmp/db.out AutoTrace=1 frame=2"  
PerlModule Apache::DB  
<Location /perl>  
    PerlFixupHandler Apache::DB  
    SetHandler perl-script  
    PerlHandler Apache::Registry  
    Options ExecCGI  
    PerlSendHeader On  
</Location>
```

- Restart the server
- ...request...
- Firstly, */tmp/db.out* was written, with a complete trace of the code that was executed.
- Secondly, *error_log* now contains the real code that was actually executed.
- This is produced as a side effect of reporting the '*Variable "\$counter" will not stay shared at...*' warning that we saw earlier.

- Here is the code that was actually executed:

```
package Apache::ROOT::perl::conference::counter_2ep1;
use Apache qw(exit);
sub handler {
    BEGIN { $^W = 1;};
    use strict;
    print "Content-type: text/plain\r\n\r\n";

    my $counter = 0;
    for (1..5) {
        increment_counter();
    }
    sub increment_counter{
        $counter++;
        print "Counter is equal to $counter !\r\n";
    }
}
```

- **Conclusions:**
- Every script is placed into a unique package
- The code is wrapped into a `handler()` subroutine.
- `increment_counter()` becomes a nested subroutine under `Apache::Registry`.

- **Solution:**

- Put your code into a library or module, which the main script `require()`'s or `use()`'s.

- For example:

```
mylib.pl:
-----
sub increment_counter{
    $counter++;
    print "Counter is equal to $counter !\r\n";
}
1;
```

```
counter.pl:
-----
#!/usr/bin/perl -w

use strict;
require "./mylib.pl";
print "Content-type: text/plain\r\n\r\n";
my $counter = 0;
for (1..5) {
    increment_counter();
}
```

Postmortem:

- Keep your subs in external libraries (modules)
- Don't worry about nested subroutines effects anymore
- Help Perl to help you -- keep the warnings mode **On**:
 - Variable "\$counter" will not stay shared at ...
- Watch *error_log*

- The above example was pretty boring.
- Once upon a time I wrote a simple user registration program.

```
use CGI;
$q = new CGI;
my $name = $q->param( 'name' );
print_respond();

sub print_respond{
    print "Content-type: text/plain\r\n\r\n";
    print "Thank you, $name!";
}
```

- A cool nice program, which happily went into production.
- My boss does the verification
- Expects: “Thank you, boss”
- Sees: “Thank you, Stas!” .
- But I’ve tested the script a lot on development machine and it worked.
- What’s the catch?

4.2.2 *The Second Mystery*

- Back to our original example
- Why did we see inconsistent results over numerous reloads?
- Apache is serving requests in Round Robin fashion.
- If you have 10 httpd children alive -- 10 first reloads are fine.
- Subsequent reloads then return unexpected results.
- Requests can appear at random and children don't always run the same scripts.

- Why couldn't we see the problem?
- We didn't look at *error_log*
- If we did, there were many warnings -- too hard to see any real errors.
- We had too many children running to notice the problem.

- **Solution:**
- To run the server as a single process. (`httpd -X`).
- The problems reveals itself on the second reload.
- Warnings should be turned On
- `error_log` shouldn't be clobbered with multiply warnings.

4.3 Sometimes it Works, Sometimes it Doesn't

- Some code is behaving differently from execution to execution?
- We just saw such an example with the counter script.
- Run the server in the single mode `httpd -X` to nail these bugs.
- Generally the problem you have is of using global variables.
- Global variables are persistent through the process life.

4.3.1 Regular Expression Memory

- Be careful, using the `/o` regular expression modifier
- It compiles a regular expression once, on its first execution, and never compiles it again.

```
my $pat = $q->param( "keyword" );
foreach( @list ) {
    print if /$pat/o;
}
```

- To catch this bug, use `httpd -X`

- Also see *Compiled Regular Expressions* section of the *Perl Reference* chapter at the end of the handout.

4.4 @INC and mod_perl

- Under mod_perl, once the server is up, @INC is frozen and cannot be updated from the code.
- Temp modification is possible while the script or the module are loaded and compiled for the first time.
- After that its value is reset to the original one.
- The only way to change @INC permanently is to modify it at Apache startup.

Two ways to alter @INC at server startup:

- In the configuration file.

```
PerlSetEnv PERL5LIB /home/httpd/perl:/home/httpd/mymodules
```

- In the startup file:

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

```
httpd.conf:
```

```
-----
```

```
PerlRequire /path/to/startup.pl
```

- Note that you cannot use `FindBin` under `mod_perl`, since it gets compiled only once.

4.5 Reloading Modules and Required Files

- We want modules under development to be reloaded on every modification
- Doesn't happen under `mod_perl -- optimized` for speed.
- Only Registry scripts get reloaded if modified.
- Solutions?

4.5.1 Restarting the server

- Restart the server after every change
- Not convenient at all

4.5.2 Using Apache::StatINC for the Development Process

- Help comes from the `Apache::StatINC` module.
- When Perl pulls a file via `require()`, it stores the full pathname as a value in the global hash `%INC` with the file name as the key.
- `Apache::StatINC` looks through `%INC` and it immediately reloads any files it finds in there if it sees that they have been updated on disk.
- To enable this module just add two lines to `httpd.conf`.

```
PerlModule Apache::StatINC
PerlInitHandler Apache::StatINC
```

- Enable the Debug mode to be sure that it works.

```
PerlModule Apache::StatINC
<Location /perl>
    SetHandler perl-script
    PerlHandler Apache::Registry
    Options ExecCGI
    PerlSendHeader On
    PerlInitHandler Apache::StatINC
    PerlSetVar StatINCDebug On
</Location>
```

4.5.3 Using Apache::Reload

- Apache::Reload comes as a drop-in replacement for Apache::StatINC.
- It provides extra functionality and better flexibility.
- The default is the *Check all* mode:

PerlInitHandler Apache::Reload

Register modules implicitly:

```
PerlInitHandler Apache::Reload  
PerlSetVar ReloadAll Off
```

- and add:

```
use Apache::Reload;
```

- to every module that you want to be reloaded on change.

Register Modules Explicitly:

```
PerlInitHandler Apache::Reload
```

```
PerlSetVar ReloadModules "My::Foo My::Bar Foo::Bar::Test"
```

- You can register groups of modules using the metacharacter (*).

```
PerlSetVar ReloadModules "Foo::* Bar::*"
```

Use a "Touch" File:

```
PerlsetVar ReloadTouchFile /tmp/reload_modules
```

- when an update is performed

```
% touch /tmp/reload_modules
```

- Modified modules get reloaded
- Useful in production
- The only overhead is a single stat call on every request.

- This module might have a problem with reloading single modules that contain multiple packages that all use pseudo-hashes.

4.5.4 Reloading handlers

- Reloading PerlHandler on each invocation:

```
PerlHandler "sub { do 'MyTest.pm' ; MyTest::handler(shift) }"
```

- `do()` reloads `MyTest.pm` on every request.

4.6 Name collisions with Modules and libs

A reminder:

- The `%INC` hash stores information about compiled modules.
- The keys are the names of the modules and files passed as arguments to `require()` and `use()`.
- The values are the full or relative paths to these modules and files
- Each child process has its own `%INC`

- Let's look at three scenarios with name space problems.
- For the following discussion we will consider just one individual child process.

Scenario 1:

- You can't have two identical module names running under the same server!
- Only the first one found in a `use()` or `require()` statement will be loaded.
- For example:

```
./perl/tool1/Foo.pm  
./perl/tool1/tool1.pl  
./perl/tool2/Foo.pm  
./perl/tool2/tool2.pl
```

Where a sample code could be:

```
./perl/tool1/tool1.pl
-----
use Foo;
print "Content-type: text/plain\r\n\r\n";
print "I'm Script number One\n";
foo();

./perl/tool1/Foo.pm
-----
sub foo{
    print "<B>I'm Module Number One!</B>\n";
}
1;
```

```
./perl/tool2/tool2.pl
-----
use Foo;
print "Content-type: text/plain\r\n\r\n";
print "I'm script number Two\n";
foo();
```

```
./perl/tool2/Foo.pm
-----
sub foo{
    print "<B>I'm Module Number Two!\n";
}
1;
```

- Both scripts call `use Foo;`.
- Only the first one called will know about `Foo`.
- The second script will say:

Undefined subroutine

```
&Apache::ROOT::perl::tool2::tool2_2ep1::foo called at  
/home/httpd/perl/tool2/tool2.pl line 4.
```

- `httpd -X` easily detects this

Scenario 2:

- If the files do not declare a package, the above is true for files you `require()` as well:
- Suppose the content of the scripts and `config.pl` files is exactly like in the example above, and you have a directory structure like this:

```
• /perl/tool1/config.pl
• /perl/tool1/tool1.pl
• /perl/tool2/config.pl
• /perl/tool2/tool2.pl
```

and both scripts contain

```
use lib qw(.);  
require "config.pl";
```

- The second scenario is not different from the first

Scenario 3:

- This setup fails too:

```
./perl/tool/config.pl  
./perl/tool/tool1.pl  
./perl/tool/tool2.pl
```

- where `tool1.pl` and `tool2.pl` both do:

```
require "config.pl";
```

There are three solutions for this:

Solution 1: Different path prefixes

- Solves only the first two scenarios
- The file system layout will be something like:

```
./perl/tool1/Tool1/Foo.pm
./perl/tool1/tool1.pl
./perl/tool2/Tool2/Foo.pm
./perl/tool2/tool2.pl
```

- And modify the scripts:

```
use Tool1::Foo;  
use Tool2::Foo;
```

- For `require()` (scenario number 2) use the following:

```
./perl/tool1/tool1-lib/config.pl  
./perl/tool1/tool1.pl  
./perl/tool2/tool2-lib/config.pl  
./perl/tool2/tool2.pl
```

- And each script contains respectively:

```
use lib qw(.);  
require "tool1-lib/config.pl";
```

- and:

```
use lib qw(.);  
require "tool2-lib/config.pl";
```

Caveats:

- It works but it's not very good.
- You have to remember to choose different dir names.
- If you add another script that wants to use the same module or `config.pl` file, it would fail as we saw in the third scenario.

Solution 2: Using the Full Path

```
 require "/full/path/to/the/config.pl";
```

- This solution solves the problem of all three scenarios.

Caveats:

- With this solution you lose some portability.
- Hard to move the code around filesystem

Solution 3: Package Declaration

- It should be unique to the rest of the package names you use.
- `%INC` will then use the unique package name for the key.
- Use at least two-level package names for your private modules
- `MyProject::Carp` and not `Carp`
- Since the latter will collide with an existing standard package.
- New standard modules get added to the Perl distribution.
- The collision might happen when upgrading Perl.

- What are the implications of package declaration?

Without package declarations:


- It is very convenient to `use()` or `require()` files
- All the variables and subroutines are part of the `main::` package.
- Any of them can be used as if they are part of the main script.

With package declarations things are more awkward:

- Using the `Package::function()` method to call a subroutine
from `Package`
- Using `$Package::foo` to access a global variable `$foo` from
outside.

- Lexically defined variables (`my`) inside Package will be inaccessible from outside the package.

- Possible workaround: importing symbols

```
 use Module qw( :mysubs sub_b $var1 :myvars );
```

- You can import both subroutines and global variables.
- Importing symbols consumes more memory!!!
- See `perldoc Exporter` for information about exporting other variables and symbols.
- See also the `perlmodlib` and `perlmodmanpages`.

Conclusions:

- Declaring packages solves all the problems.
- You cannot run development and production versions of the tools using the same apache server!
- You have to run a separate server for each.
- They can be on the same machine, but the servers will use different ports.
- Another solution is to use virtual hosts with `Apache::PerlVINC`

4.7 `__END__` and `__DATA__` tokens

- `Apache::Registry` scripts cannot contain `__END__` or `__DATA__` tokens.
- Remember that:

```
print "Content-type: text/plain\r\n\r\n";  
print "Hi";
```

- Under Apache: `:Registry` becomes:

```
package Apache::ROOT::perl::test_2ep1;
use Apache qw(exit);
sub handler {
    print "Content-type: text/plain\r\n\r\n";
    print "Hi";
}
```

- So if you happen to put an `__END__` tag, like:

```
print "Content-type: text/plain\r\n\r\n";
print "Hi";
__END__
```

Some text that wouldn't be normally executed

- It will be turned into:

```
package Apache::ROOT::perl::test_2ep1;
use Apache qw(exit);
sub handler {
    print "Content-type: text/plain\r\n\r\n";
    print "Hi";
    ___END___
    Some text that wouldn't be normally executed
}
```

- When run:

Missing right bracket at line 4, at end of line

- Perl cuts everything after the `__END__` tag.
- The same applies to the `__DATA__` tag.

4.8 Output from system calls

- The output of `system()`, `exec()`, and `open(PIPE, " | program")` calls will not be sent to the browser
- unless your Perl was configured with `sfio`.
- You can use backticks as a possible workaround:

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

- But you're throwing performance out the window either way.
- It's best not to fork at all if you can avoid it.

4.9 Terminating requests and processes

- Perl's `exit()` built-in function cannot be used in `mod_perl` code.
- Using it defeats the object of using `mod_perl`.
- The `Apache::exit()` function should be used instead.
- To make the script work under `mod_perl` and `mod_cgi`, do:

```
BEGIN { $USE_MOD_PERL = $ENV{MOD_PERL} ? 1 : 0; }
use subs qw(exit);

sub exit{
    $USE_MOD_PERL ? Apache::exit(0) : CORE::exit(0);
}
```

- You can leave `exit()` calls in `Apache::Registry`.

```
Apache::exit(-2)
# or
Apache::exit(Apache::Constants::DONE) >
```

- will cause the server to exit gracefully, completing the logging functions and protocol requirements etc.

- To shut down the child cleanly after the request was completed use the `$r->child_terminate` method.

4.10 die() and mod_perl

■ `open FILE, "foo" or die "Cannot open foo file for reading: $!";`

- will not kill the server if the `die()` will be called!
- When the `die()` gets triggered:
- Apache logs the error message
- and calls `Apache::exit()` instead of real `die()`.
- Thus the script stops, but the process doesn't quit.

4.11 `Apache::print()` and `CORE::print()`

- The `STDOUT` filehandle is tied to the *Apache* module.
- `CORE::print()` will redirect its data to `Apache::print()`
- This allows us to run CGI scripts unmodified under `Apache::Registry`
- And chain the output of one content handler to the input of the other handler.
- `Apache::print()` behaves mostly like the built-in `print()` function.

- In addition it sets a timeout so that if the client connection is broken the handler won't wait forever trying to print data downstream to the client.

- There is also an optimization built into `Apache::print()`.
- Automatically dereferencing of references to scalars
- This avoids needless copying of large strings when passing them to subroutines.
- For example:

```
$long_string = "A" x 10000000;  
$r->print(\$long_string);
```

4.12 Global Variables Persistence

- The child process doesn't exit
- Global variables persist inside the same process from request to request.
- Don't rely on the value of the global variable unless it was initialized at the beginning of the request processing.
- Avoid using global variables unless it's impossible without them
- They makes code development and debugging harder
- Use `my ()` scoped variables wherever you can.

- You should be especially careful with Perl special variables which cannot be lexically scoped.
- You have to use `local()` instead.

4.13 Command line Switches (-w, -T, etc)

- Normally the switches are set via sheband line:

```
#!/usr/bin/perl -Tw
```

- mod_perl ignores all switches, but `-w` if shebang exists.
- Most command line switches have a special variable equivalent.

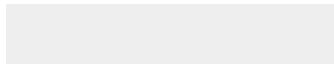
4.13.1 Warnings

There are three ways to enable warnings under `mod_perl`:

- **Locally to a block**
 - This code turns warnings mode **On** for the scope of the block.

```
{  
    local $^W = 1;  
    # some code  
}
```

- This turns it **Off**:

```
  
{  
    local $^W = 0;  
    # some code  
}
```

- Without `local()`, `$^W` will affect all the requests.

```
  
$^W = 0;
```

- will turn the warnings *Off* process wide

- To turn warnings *On* for the scope of the whole file, as with
-w, add:

```
local $^W = 1;
```

- at the beginning of the file.

- **Globally to all Processes**

```
perlwarn On
```

- in `httpd.conf` will turn warnings **On** in any script.
- You can then fine tune your code, turning warnings **Off** and **On** by setting the `$^W` variable in your scripts.

- **Locally to a script**



```
#!/usr/bin/perl -w
```

- will turn warnings **On** for the scope of the script.
- use `use $^W` to locally modify the warning behavior
- On the production server have the warnings off `-- keep the error_log file small.`

4.13.2 Taint Mode

- Perl's `-T` switch enables *Taint* mode.
- **Always** use this mode on production machines!!!
- (See the *perl*sec manpage for more information)
- Perl doesn't have an equivalent variable for `-T`
- under `mod_perl` the `PerlTaintCheck` directive controls this mode.

PerlTaintCheck On

- turns the mode on
- If you use the `-T` switch, Perl will warn you that you should use the `PerlTaintCheck` configuration directive and will otherwise ignore it.

4.13.3 *Other switches*

- Finally, if you still need to set additional perl startup flags such as `-d` and `-D`, you can use an environment variable `PERL5OPT`.
- Also consult the `perlvar` manpage for the details about other switches equivalents.

```
;O
```