

Hello!

O'Reilly Open Source Convention

July 23, 2001

San Diego, CA

Tutorial:

Getting started with mod_perl

by Stas Bekman

<http://stason.org/>

[<stas@stason.org>](mailto:stas@stason.org)

Senior Software Engineer, eXtropia.com

This document is originally written in **POD**, converted to **HTML**, **PostScript** and **PDF** by `Pod::HtmlPsPdf` 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
- Basic Configuration

- Basic Scripts and Handlers
- mod_perl and RDBMS
- Improving Performance.

1.2 Off-tutorial reading

- Perl Reference
- Getting Help

;o)

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.

;o)

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.**

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 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 C<httpd_docs>



8080 for C<httpd_perl>

- Now why would you want to use a proxy server (in the http 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:


$$42 \text{ KB} / (0.5 \text{ sec} * 7 \text{ KB/sec}) \sim 12$$

- 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.

;o)

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
- __END__ and __DATA__ tokens
- Output from system calls

- Terminating requests and processes
- `die()` and `mod_perl`
- 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:

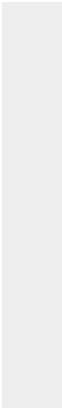
```
package Apache::ROOT::perl::conference::counter_2ep1;
use Apache qw(exit);
sub handler {
    BEGIN { $^W = 1;};
    use strict;
    print "Content-type: text/plain\r\n\r\n";

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

- **Conclusions:**
- Every scripts 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 sub, move into a separate file and `require()` it.
- For example:


```
mylib.pl:
-----
my $counter;
sub run{
    print "Content-type: text/plain\r\n\r\n";
    $counter = 0;
    for (1..5) {
        increment_counter();
    }
}
sub increment_counter{
    $counter++;
    print "Counter is equal to $counter !\r\n";
}
1;
```



```
counter.pl:
-----
use strict;
require "./mylib.pl";
run( );
```

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 `httd -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 `__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.

4.7 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.

4.8 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.

4.9 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.

4.10 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.

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

- Normally the switches are set via shebang line:

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

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

4.11.1 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 `$^W` to locally modify the warning behavior
- On the production server have the warnings off -- keep the *error_log* file small.

4.11.2 *Taint Mode*

- Perl's `-T` switch enables *Taint* mode.
- **Always** use this mode on production machines!!!
- (See the *perlsec* 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.

4.11.3 *Other switches*

- Finally, if you still need to 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)

5 RDBMS and mod_perl

5.1 Apache::DBI - Initiate a persistent database connection

- `mod_cgi` limitation -- its database connection is not persistent
- `Apache::DBI` removes this limitation.
- A database connection persists for the process' entire life.
- Available only under `mod_perl`
- No changes to your code required

5.1.1 Introduction

- The DBI module can make use of the `Apache::DBI` module.
- Detects `mod_perl` by testing `$ENV{MOD_PERL}`
- Forwards every `connect()` request to the `Apache::DBI` module.

```
if ($dbh->ping and
    matched($host,$username,$password,$arguments) ){
    return $dbh;
} else {
    connect() && cache();
}
```

- There is no need to delete the `disconnect()` statements from your code.

When should this module be used?

- **DO** use it if you use one connection (or a few) for all your users.
- **DO NOT** use it if you are opening a special connection for each of your users.
- Why: Each connection will stay persistent.
- $\text{\#conn} = \text{\#processes} * \text{\#users}$
- e.g. 20 procs * 100 users = 2000 connections
- The number of connections is limited by an SQL engine

- If you have both kinds of apps
- Run two Apache/mod_perl servers:
- One which uses `Apache::DBI`
- And one which does not.

5.1.2 Configuration

- Install the module
- Add to *httpd.conf*:



```
PerlModule Apache::DBI
```

- It is important to load this module before any other Apache*DBI module and before the DBI module itself!

5.1.3 *Preopening DBI connections*

- Open connection at a child startup:

```
Apache::DBI->connect_on_init  
( "DBI:mysql:myDB::myserver",  
  "username",  
  "passwd",  
  {  
    PrintError => 1, # warn() on errors  
    RaiseError => 0, # don't die on error  
    AutoCommit => 1, # commit executes immediately  
  }  
);
```

- Warning: if you call `connect_on_init()` and your database is down...
- Apache children will be delayed at server startup, trying to connect...
- They won't begin serving requests until either they are connected, or the connection attempt fails...
- Depending on your DBD driver, this can take several minutes!

5.1.4 *Debugging Apache::DBI*

- Enable Debug mode in the startup script if something is not working

```
$Apache::DBI::DEBUG = 2;
```

- After setting the DEBUG level view the *error_log* file

5.1.5 Opening connections with different parameters

- `connect()` method should use **identical** args to get the connection reused
- If one script sets `LongReadLen` and another one does not, `Apache::DBI` will make two different connections.
- So for example instead of having a maximum of 40 open connections, you get 80.
- Solution: modify the handle immediately after you get it from the cache.



```
$dbh->{LongReadLen} = 1;
```

- Always initiate connections using the same parameters and set LongReadLen (or whatever) afterwards.

5.1.6 Caching prepare() Statements

- Replace `prepare()` with `prepare_cached()`.

Pros:

- Makes sure that you have a good statement handle
- and you will get some caching benefit.

Cons:

- You are going to pay for DBI to parse your SQL
- and do a cache lookup every time you call `prepare_cached()`.

;o)

6 Performance Tuning

6.1 What we will learn in this chapter

- The Big Picture
- Essential Tools
- Choosing MaxClients
- KeepAlive
- Limiting the Size of the Processes
- Sharing Memory

- How Shared My Memory Is
- Preload Perl modules at server startup
- Preload Registry Scripts
- Modules Initialization
- Keeping the Shared Memory Limit
- Limiting the Resources Used by httpd Children
- Upload/Download of Big Files
- Global vs Fully Qualified Variables

- Forking or Executing subprocesses from mod_perl
- Sending plain HTML as a compressed output

6.2 The Big Picture

The goal: **User Experience**

- There are many factors which affect Web site usability
- But speed is one of the most important.

Which speed do we measure?

- Start: link has been clicked
- Finish: the resulting page has been rendered
- The rest is of a little interest to end user

Take into account more than just the code

- NIC and Network: a packet travels from a client and a server and backwards
- Machine's hardware: Each component can be a bottleneck
- Ultrafast webserver but slow modem connection defeats it all -- a need for a proxy server
- Of course the code itself

—
|
A Web service is
like a car,
if one of the
parts or mechanisms is broken the
car may ~ not ~ go smoothly and
it can even stop dead if pushed too
far without first fixing it !!!
___/ ___/

6.3 Essential Tools

Tools to measure performance

- Benchmarking Perl Code
- Benchmarking Response Time

6.3.1 *Benchmarking Perl Code*

- The `Benchmark` module
- The `Time::HiRes` module where you need better time precision (<10msec).

- `Benchmark.pm`:

```
use Benchmark;
```

```
timethis (1_000,  
  sub {  
    my $x = 100;  
    my $y = log ($x ** 100)   for (0..10000);  
  });
```

```
% perl benchmark.pl
```

```
timethis 1000: 25 wallclock secs (24.93 usr + 0.00 sys = 24.93 CPU)
```

- `Time::HiRes:`

```
use Time::HiRes qw(gettimeofday tv_interval);  
sub sub_that_takes_a_teeny_bit_of_time{1+1;};
```

```
my $start_time = [ gettimeofday ];  
sub_that_takes_a_teeny_bit_of_time();  
my $end_time = [ gettimeofday ];
```

```
my $elapsed = tv_interval($start_time,$end_time);  
print "The sub took $elapsed seconds.\n"
```

```
% perl hi-res.pl
```

```
The sub took 0.000262 seconds.
```

6.3.2 *Benchmarking Response Times*

Goals:

- Generate parallel requests,
- Process the responses and
- Print the results of the test.

Options:

- Re-use the existing stuff

- Develop your own.

6.3.2.1 ApacheBench

- ApacheBench (ab) comes bundled with Apache source distribution.
- Shows Requests/Sec capability of Apache with your code
- An example:
- Simulate 10 concurrent users.
- Each simulated user makes 10 requests.

```
% ./ab -n 100 -c 10 www.example.com:81/test/test.pl
```

Document Path: /perl/test.pl
Document Length: 319 bytes

Concurrency Level: 10
Time taken for tests: 0.715 seconds
Complete requests: 100
Failed requests: 0
Total transferred: 60700 bytes
HTML transferred: 31900 bytes
Requests per second: 139.86
Transfer rate: 84.90 kb/s received

Connection Times (ms)

	min	avg	max
Connect:	0	0	3
Processing:	13	67	71
Total:	13	67	74

6.3.2.2 httpperf

- httpperf was written by David Mosberger.

```
% httpperf --server hostname --port 80 --uri /test.html \  
--rate 150 --num-conn 27000 --num-call 1 --timeout 5
```


Total: connections 27000 requests 26701 replies 26701 test-duration 179.996 s

Connection rate: 150.0 conn/s (6.7 ms/conn, <=47 concurrent connections)

Connection time [ms]: min 1.1 avg 5.0 max 315.0 median 2.5 stddev 13.0

Connection time [ms]: connect 0.3

Request rate: 148.3 req/s (6.7 ms/req)

Request size [B]: 72.0

- ...more

```
Reply rate [replies/s]: min 139.8 avg 148.3 max 150.3 stddev 2.7 (36 samples)
Reply time [ms]: response 4.6 transfer 0.0
Reply size [B]: header 222.0 content 1024.0 footer 0.0 (total 1246.0)
Reply status: 1xx=0 2xx=26701 3xx=0 4xx=0 5xx=0

CPU time [s]: user 55.31 system 124.41 (user 30.7% system 69.1% total 99.8%)
Net I/O: 190.9 KB/s (1.6*10^6 bps)

Errors: total 299 client-timo 299 socket-timo 0 connrefused 0 connreset 0
Errors: fd-unavail 0 addrunavail 0 ftab-full 0 other 0
```

6.3.3 Using *LWP::Parallel::UserAgent*

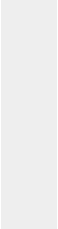
- Write your own tool with `LWP::Parallel::UserAgent`.
- This is a slightly adjusted code originally written by Michael Schilli
- It accepts more than one url to test
- The code is in your handouts

- Sample output:

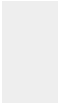
```
URL(s):          http://www.example.com:81/perl/access/access.cgi
Total Requests:  100
Parallel Agents: 10
Succeeded:       100 (100.00%)
Errors:          NONE
Total Time:      9.39 secs
Throughput:      10.65 Requests/sec
Latency:         0.85 secs/Request
```

6.4 Choosing MaxClients

- `MaxClients` sets the limit on the number of simultaneous requests that can be supported.
- We want it to be as small as possible


$$\text{MaxClients} = \frac{\text{Total RAM Dedicated to the Webserver}}{\text{MAX child's process size}}$$

- An example:


$$400\text{MB} / 10\text{MB} = 40 \text{ (clients)}$$

What if there are more concurrent requests than servers?

- This situation is accompanied by the following warning message in the `error_log`:

```
[Sun Jan 24 12:05:32 1999] [error] server reached MaxClients setting,  
consider raising the MaxClients setting
```

- Connections can be queued through the `ListenBacklog` directive.
- **It is an error** because clients are being put in the queue rather than getting served immediately, despite the fact that they do not get an error response.
- Try not to reach this condition

Real memory use

- Your children can share memory between them when the OS supports that.
- You must take action to allow the sharing to happen.
- Code should be preloaded at the server startup
- You can raise `MaxClients` when you get memory shared

$$\text{MaxClients} = \frac{\text{Total_RAM} + \text{Shared_RAM_per_Child} * (\text{MaxClients} - 1)}{\text{Max_Process_Size}}$$

which is:

$$\text{MaxClients} = \frac{\text{Total_RAM} - \text{Shared_RAM_per_Child}}{\text{Max_Process_Size} - \text{Shared_RAM_per_Child}}$$

- Let's roll some calculations:

```
Total_RAM           = 500Mb
Max_Process_Size     = 10Mb
Shared_RAM_per_Child = 4Mb
```

$$\text{MaxClients} = \frac{500 - 4}{10 - 4} = 82$$

- With no sharing in place

$$\text{MaxClients} = \frac{500}{10} = 50$$

- Conclusion: With sharing in place you can have 64% more servers without adding more RAM.

- If you improve sharing and keep the sharing level, let's say:

`Total_RAM` = 500Mb

`Max_Process_Size` = 10Mb

`Shared_RAM_per_Child` = 8Mb

$$\text{MaxClients} = \frac{500 - 8}{10 - 8} = 246$$

- 392% more servers!

6.5 KeepAlive

- If your mod_perl server's *httpd.conf* includes:

```
KeepAlive On
MaxKeepAliveRequests 100
KeepAliveTimeout 15
```

- You have a real performance penalty,
- The process will wait for `KeepAliveTimeout` seconds before closing the connection
- With this configuration you will need many more concurrent processes on a server with high traffic.

- Set it Off with:



KeepAlive Off

- the other two directives don't matter if `KeepAlive` is Off.

When you want it Enabled?

- A client requests more than one object from your server for a single HTML page.
- You save the HTTP connection overhead for all requests but the first one.
- Example: a page with 10 ad banners
- A server will work more effectively if a single process serves them all during a single connection.
- Your client will see a slightly slower response, though
- SSL connections benefit the most from `KeepAlive` in case you didn't configure the server to cache session ids.

6.5.1 *Be carefull with symbolic links*

- `Apache::Registry` caches the scripts based on their URI.
- Be ware of symlinks or you will get the same code cached twice

```
% ln -s /home/httpd/perl/news/news.pl /home/httpd/perl/news.pl
```

- Now the script can be reached through the both URIs `/news/news.pl` and `/news.pl`.
- The same code will be stored twice in the memory

Detection:

- Use the `/perl-status` (`Apache::Status`) handler
- `http://localhost/perl-status?rgysubs` you would see:

```
Apache::ROOT::perl::news::news_2ep1  
Apache::ROOT::perl::news_2ep1
```

- run the server in single mode (`httpd -X`) to debug.

6.6 Limiting the Size of the Processes

- `Apache::SizeLimit` allows you to kill off Apache httpd processes if they grow too large.
- *startup.pl*:

```
use Apache::SizeLimit; # 10MB
$Apache::SizeLimit::MAX_PROCESS_SIZE = 10000;
```

- *httpd.conf*:

```
PerlFixupHandler Apache::SizeLimit
```

- No need for `MaxRequestsPerChild` anymore!

6.7 Sharing Memory

- Saving memory by sharing it between child processes.
- It's possible only when preloading code at server startup
- During a child process' life, its memory pages becomes unshared
- This process is called **Copy-On-Write**
- Perl doesn't have strong typed variables.
- No clear separation between DATA and CODE memory pages.

- That's why the **Copy-On-Write** effect hits almost at random.
- which reduces the number of shared memory pages - thus enlarging the memory demands.
- Killing the child and respawning a new one, allows to get the pristine shared memory from the parent process again.
- Try using `MaxRequestsPerChild` to balance the memory that stays shared against the time
- You should do some measurements and you might see if this really makes a difference and what a reasonable number might be.
- The newly started child process has all of its memory shared.

- `MaxRequestsPerChild` doesn't have to be big (10000?)
- Having a child serving 300 requests on precompiled code is already a huge speedup

6.8 How Shared My Memory Is

- How much shared memory do you have?
- You can see it by either using the memory utils that comes with your system like `top(1)` and `ps(1)`.
- or you can deploy GTop module:

```
print "Shared memory of the current process: ",  
      GTop->new->proc_mem($$)->share, "\n";
```

```
print "Total shared memory: ",  
      GTop->new->mem->share, "\n";
```

6.9 Keeping the Shared Memory Limit

- `Apache::GTopLimit` module
- like `Apache::SizeLimit` (max memory limitation)
- plus shared memory limitation
- In *httpd.conf*:



```
PerlFixupHandler Apache::GTopLimit
```

- Configuration: *startup.pl*:

```
use Apache::GTopLimit;
```

```
# Control the life based on memory size
```

```
$Apache::GTopLimit::MAX_PROCESS_SIZE = 10000; # 10MB
```

```
# Control the life based on Shared memory size
```

```
$Apache::GTopLimit::MIN_PROCESS_SHARED_SIZE = 4000; # 4MB
```

```
# watch what happens
```

```
$Apache::GTopLimit::DEBUG = 1;
```


6.10 Preload Perl modules at server startup

- Use the `PerlRequire` and `PerlModule` directives to preload commonly used modules such as `CGI.pm`, `DBI...`

```
PerlModule CGI;  
PerlModule DBI;
```

- Do the same in Perl from *startup.pl*:

```
use DBI;  
use Carp;
```

- and require the IPreload Registry Scripts
 - `Apache::RegistryLoader` compiles `Apache::Registry` scripts at server startup.
 - precompile the scripts just like modules, and save memory

```
use Apache::RegistryLoader ();  
Apache::RegistryLoader->new->handler($url);
```

- Handouts: the code to recursively load of all scripts

6.11 Modules Initialization

- Let's see how one can calculate the actual improvements of modules preloading practice.
- We will use a very widely used module: DBI
- We will try to initialize DBI with the MySQL driver `DBD::mysql`
- And try to see how it minimizes memory use after forking the child processes.

- In order to have an easy measurement
- We will use only one child process
- Therefore we will use this setting in *httpd.conf*:

```
MinSpareServers 1
MaxSpareServers 1
StartServers 1
MaxClients 1
MaxRequestsPerChild 100
```

- or `httpd -X` (single server mode)

- We always preload these modules:

```
use Gtop();  
use Apache::DBI(); # preloads DBI as well
```

- We are going to run memory benchmarks on five different versions of the *startup.pl* file:

option 1

- Leave the file unmodified.

option 2

- Install MySQL driver (we will use MySQL RDBMS for our test):

```
DBI->install_driver("mysql");
```

- It's safe to use this method, since just like with `use()`, if it can't be installed it'll `die()`.

option 3

- Preload MySQL driver module:

```
use DBD::mysql;
```

option 4

- Use `Apache::DBI->connect_on_init()`

- No driver is preload before the child gets spawned!

```
Apache::DBI->connect_on_init(  
    'DBI:mysql:test::localhost',  
    "",  
    "",  
    {  
        PrintError => 1, # warn() on errors  
        RaiseError => 0, # don't die on error  
        AutoCommit => 1, # commit executes  
        # immediately  
    }  
)  
or die "Cannot connect to database: $DBI::errstr";
```

option 5

Options 2 and 4

- The `Apache::Registry` test script that was used for testing is in your handouts
- The script opens a connection to the database *'test'*
- issues a query to learn what tables the databases has.
- and prints the memory usage
- The server was restarted before each new test.

- The results sorted by the *Diff* column:

1.

After the first request:

2.

After the second request (all the subsequent request showed the same results):

Test type	Size	Shared	Diff
-----	-----	-----	-----
install_driver (2)	3465216	2621440	843776
install_driver & connect_on_init (5)	3461120	2609152	851968
preload driver (3)	3465216	2605056	860160
nothing added (1)	3461120	2494464	966656
connect_on_init (4)	3461120	2482176	978944

3.

After the second request (all the subsequent request showed the same results):

Test type	Size	Shared	Diff
-----	-----	-----	-----
install_driver (2)	3469312	2609152	860160
install_driver & connect_on_init (5)	3481600	2605056	876544
preload driver (3)	3469312	2588672	880640
nothing added (1)	3477504	2482176	995328
connect_on_init (4)	3481600	2469888	1011712

Conclusions:

- Only after a second reload we get the final memory footprint for a specific request in question.
- The test with preinstalled driver wins, since it allows to share more memory.
- Since we almost always want to use `connect_on_init()` we will go with option number 2.

- Given that we have 256M of memory dedicated to mod_perl processes
- The good old equation:

$$\text{N_of Procs} = \frac{\text{RAM} - \text{largest_shared_size}}{\text{Diff}}$$

$$(\text{ver } 2) \quad N = \frac{268435456 - 2609152}{860160} = 309$$

$$(\text{ver } 4) \quad N = \frac{268435456 - 2469888}{1011712} = 262$$

- So you can tell the difference
- 17% more child processes in the first version

6.12 Upload/Download of Big Files

- Don't use mod_perl servers for big file transfers!
- Upload/Download might take minutes
- The speed saved by mod_perl gives nothing.
- The server stays tied and cannot do other more important work.
- Solution: Use plain apache server and mod_cgi.

- Assumes that no mod_perl functionality is required (e.g. auth)

6.13 Global vs Fully Qualified Variables

- Stay away from global variables when possible.
- Sometimes you must have them...
- e.g.: `@ISA` or `$VERSION` variables (or fully qualified `@MyModule::ISA`).
- A combination of `strict` and `vars` pragmas keeps modules clean and reduces a bit of noise.
- However, `vars` pragma also creates aliases as the `Exporter` does, which eat up more memory.

- Use fully qualified names instead of use vars.
- Example:

```
package MyPackage;  
use strict;  
@MyPackage::ISA = qw(...);  
$MyPackage::VERSION = "1.00";
```

- VS.

```
package MyPackage;  
use strict;  
use vars qw(@ISA $VERSION);  
@ISA = qw(...);  
$VERSION = "1.00";
```

6.14 Forking or Executing subprocesses from mod_perl

- Process' forking might be expensive on some systems.
- To start a long running process (e.g. sending emails to many users: No SPAM please!) and let the parent continue:
- spawn a sub-process,
- hand it the information it needs to do the task,
- and have it detach (close STD* + `setsid()`)

```
$params=FreezeThaw::freeze(  
    [all data to pass to the other process]  
);  
system("program.pl", $params);
```

- and in *program.pl*:

```
use POSIX qw(setsid);  
@params=FreezeThaw::thaw(shift @ARGV);  
close STDIN;  
close STDOUT;  
close STDERR;  
setsid(); # to detach
```

- At this point, `program.pl` is running in the “background” while the `system()` returns and permits Apache to get on with life.

- This has obvious problems.
- Shell limitation on the size of `@params`.
- Also, the communication is only one way.
- For fast execution of the postprocess code use `PerlCleanupHandler`
- But you will keep the process busy, which is not a good idea...

Forking example:

```
if (fork){  
    #do nothing  
} else {  
    system("echo Hi");  
    CORE::exit(0);  
}
```

- Parent immediately continues with the code that comes up after the fork
- Child executes `system("echo Hi")` and then terminates itself.

- !!!: use `CORE::exit` (`exit()` == `Apache::exit` under Registry and friends)

...now the gory details:

6.14.1 Freeing the Parent Process

- Closing all the inherited communication pipes opened by the parent.
- allow the parent to complete the request and free itself for serving other requests.
- the spawned process also inherits the file descriptor that's tied to the socket through which all the communications between the server and the client happen.
- If this socket is not freed, the server cannot be restarted:

```
[Mon Dec 11 19:04:13 2000] [crit]  
(98)Address already in use: make_sock:  
could not bind to address 127.0.0.1 port 8000
```

- `Apache::SubProcess::cleanup_for_exec()` takes care of it:


```
use Apache::SubProcess;
defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
    # Parent runs this block
} else {
    # Child runs this block
    $r->cleanup_for_exec(); # untie the socket
    chdir '/' or die "Can't chdir to /: $!";
    close STDIN;
    close STDOUT;
    close STDERR;
    # some code comes here
    CORE::exit(0);
}
```

6.14.2 *Detaching the Forked Process*

- The parent cannot continue before the child detaches.
- If the parent is restarted, the child process gets killed

```
use POSIX 'setsid';

defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
    # Parent runs this block
} else {
    # Child runs this block
    setsid or die "Can't start a new session: $!";
    ...
}
```

- The spawned child process has a life of its own

6.14.3 *Avoiding Zombie Processes*

- Normally, every process has its parent.
- Many processes are children of the `init` process, whose `PID` equals to 1.
- When you fork a process you must `wait()` or `waitpid()` for it to finish.
- If you don't wait for it becomes a zombie.

- Zombie, is a process that doesn't have a father.
- When the child quits, it reports the termination to his parent.
- If no one `wait()`s to collect the exit status of the child, it gets “confused” and becomes a ghost process, that can be seen, but not killed.
- Ghostbuster: It will be killed only when you stop the `httpd` process that spawned it!
- `top(1)/ps(1)` mark those processes as `<defunc>`.
- Zombies counter in `top(1)` goes up

- These zombie processes can take up system resources and are generally undesirable.

The proper fork is:

```
defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
    waitpid($kid,0);
    print "Parent has finished\n";
} else {
    # do something
    CORE::exit(0);
}
```

- But in most cases the only reason you would want to fork is when you need to spawn a process that would take a lot of time to complete.
- So if the server child that spawns this process has to wait for it to finish, you gained nothing.

- You cannot neither wait for its completion, nor continue because you will get yet another zombie process.
- The simplest solution is to ignore your dead children:

```
$SIG{CHLD} = IGNORE;
```

- All the processes will be collected by the `init` process and prevent from them to become zombies.
- ... however this doesn't work everywhere.
- You cannot localize this setting with `local()`. If you do, it wouldn't take the desired effect.

- The child must close all the pipes to the connection socket that were opened by the parent process (`STDIN` and `STDOUT`)
- You may need to close and reopen a `STDERR` filehandler

So now the code would look like:

```
$SIG{CHLD} = IGNORE;

defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
    print "Parent has finished\n";
} else {
    close STDIN;
    close STDOUT;
    close STDERR;
    # do something
    CORE::exit(0);
}
```

- Notice that `waitpid()` call has gone

A double fork approach:

```
defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
    waitpid($kid,0);
} else {
    defined (my $grandkid = fork) or die "Kid cannot fork: $!\n";
    if ($grandkid) {
        CORE::exit(0);

    } else {
        # code here
        close STDIN;
        close STDOUT;
        close STDERR;
        # do something long lasting
        CORE::exit(0);
    }
}
```

- Grandkid becomes a "*child of init*" (parent process ID is 1).
- Note that the last two solutions do allow you to know the exit status of the process, but in our case we don't want to.

use a different *SIGCHLD* handler:

```
use POSIX 'WNOHANG';  
$SIG{CHLD} = sub { while( waitpid(-1,WNOHANG)>0 ) {} };
```

- Which is useful when you `fork()` more than once process.
- The arguments tell `waitpid()` to reap the next child that's available,
- and prevent the call from blocking if there happens to be no child ready from reaping.
- The handler will loop until `waitpid()` returns a negative number or zero, indicating that no more reapeable children remain.

6.14.4 A Complete Fork Example

- Your handouts include the complete example covering all the nuances discussed so far.

6.15 Sending plain HTML as a compressed output

- HTML files with JS code are huge!!!
- Compressing them will allow us to deliver them to users much faster
- java applets can be compressed into a jar and benefit from a faster download times.
- ASCII text can be compressed by a factor of 10.
- `Apache::GzipChain` comes to help you with this task.

- If a client (browser) understands `gzip` encoding this module compresses the output and sends it downstream.
- The client decompresses the data upon receive and renders the HTML as if it was a plain HTML fetch.
- For example to compress all html files on the fly, do:

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

- Watch an `access_log` file to see how many bytes were actually send, compare with a regular configuration send.
- Remember that it will work only if the browser claims to accept compressed input, thru `Accept-Encoding` header.

- `Apache::GzipChain` keeps a list of user-agents, thus it also looks at `User-Agent` header, for known to accept compressed output browsers.
- For example if you want to return compressed files which should pass in addition through `Embperl` module, you would write:

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

- See `perldoc Apache::GzipChain`
- Notice that the rightmost `PerlHandler` must be a content producer. Use `Apache::PassFile` or another similar module.

;o)