

The ApacheCon 2000
March 8, 2000
Orlando, Florida

Tutorial:
Getting Started with mod_perl
(Part II of II)

By Stas Bekman
Internet and Intranet programmer
<http://stason.org/>
<stas@stason.org>

This document is originally written in **POD**, converted to **HTML** by **pod2html** utility and then to **PostScript** by **html2ps** utility.

Copyright © 1998, 1999 Stas Bekman. All rights reserved.

1 Getting Started Fast

1.1 mod_perl in Four Slides

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

1.2 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 “Artistic License” as Perl itself.
- Home page <http://perl.apache.org>
- Mailing list: send email to modperl-subscribe@apache.org with the string “*subscribe modperl*” in the body.
- December 1999 -- 412000 mod_perl hosts (according to <http://perl.apache.org/netcraft/>)

1.3 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!

1.4 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>
```

1.5 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!

1.6 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
-----
use Apache::Constants;

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

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/perl/mod_perl_rules

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

mod_perl rules!

as the response.

1.7 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)

2 Perl Reference

2.1 What we will learn in this chapter

- Tracing Warnings Reports
- `my()` Scoped Variable in Nested Subroutines
- When You Cannot Get Rid of The Inner Subroutine
- `use()`, `require()`, `do()`, `%INC` and `@INC` Explained
- Using Global Variables and Sharing Them Between Modules/Packages

- The Scope of the Special Perl Variables
- Compiled Regular Expressions
- perldoc's Rarely Known But Very Useful Options

2.2 Time is money :)

- Unfortunately we hardly have time for mod_perl specific issues.
- You have to know the material in these sections in order to successfully program under mod_perl.
- This chapter is left as an exercise for you.
- Please skip to the next chapter.

;o)

3 CGI to mod_perl Porting. mod_perl Coding guidelines.

3.1 What we will learn in this chapter

- Before you start to code
- 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
- Using `format()` and `write()`
- Terminating requests and processes, the `exit()` and `child_terminate()` functions
- `die()` and `mod_perl`
- I/O is different
- `STDIN`, `STDOUT` and `STDERR` streams
- Global Variables Persistence

- **Generating correct HTTP Headers**
- **NPH (Non Parsed Headers) scripts**
- **BEGIN blocks**
- **END blocks**
- **Command line Switches (-w, -T, etc)**
- **The strict pragma**
- **Passing ENV variables to CGI**
- **Apache and syslog**

- Filehandlers and locks leakages
- The Script Is Too Dirty, But It Does The Job And I Cannot Afford To Rewrite It.
- Apache::PerlRun--a closer look

3.2 Before you start to code

- You need to know Perl before you start with mod_perl :-)
- mod_perl doesn't tolerate sloppy programming.
- This chapter relies on a certain level of Perl knowledge.
- This will allow me to concentrate on pure mod_perl issues and make them more prominent to the experienced Perl programmer, which would otherwise be lost in the sea of Perl background notes.

Additional resources:

- **Perl Module Mechanics**

This page describes the mechanics of creating, compiling, releasing, and maintaining Perl modules.

http://world.std.com/~swmcd/steven/perl/module_mechanics.html

The information is very relevant to a mod_perl developer.

- **The Eagle Book**

“Writing Apache Modules with Perl and C” is a “must have” book!

See the details at <http://www.modperl.com> .

- **"Programming Perl" Book**

- **"Perl Cookbook" Book**

3.3 Exposing Apache::Registry secrets

- Apache::Registry is a mod_cgi compatible module, but you should avoid the sloppy programming style with it.
- There are a few hidden issues to know about.
- Let's start with some simple code and see what can go wrong with it, detect bugs and debug them, discuss possible pitfalls and how to avoid them.

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

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

- See, 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.**

Let's investigate this script.

3.3.1 *The First Mystery*

- The `error_log` file says:

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

- This warning is generated when the script contains a named nested subroutine that refers to a lexically scoped variable defined outside this nested subroutine.
- Add `'use diagnostics;'` to see the long version of the warning.
- I cannot see a nested subroutine. Do you see it?

- Maybe the Perl interpreter sees the script in a different way
- Maybe the code goes through some changes before it actually gets executed?
- The easiest way to check what's actually happening is to run the script with a debugger.
- A normal debugger wouldn't help, because the debugger has to be invoked from within the webserver.
- Luckily Doug MacEachern wrote the `Apache::DB` module
- While `Apache::DB` allows you to debug the code interactively, we will do it non-interactively.

- Modify the `httpd.conf` file in the following way:

```
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
- Issue a request to *counter.pl* as before.

- On the surface nothing has changed--we still see the correct output as before, but two things happened in the background:
- First, the file */tmp/db.out* was written, with a complete trace of the code that was executed.
- Second, *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";
    }
}
```

What do we learn from this?

- Every cgi script is cached under a package whose name is formed from the `Apache::ROOT::prefix` and the relative part of the script's URL
(`perl::conference::counter_2ep1`) by replacing all occurrences of `/` with `::`.
- Now you see why the `diagnostics` pragma talked about an inner (nested) subroutine `-- increment_counter` is actually a nested subroutine.

- Each subroutine in every `Apache::Registry` script is nested inside the `handler` subroutine.
- If you put your code into a library or module, which the main script `require()`'s or `use()`'s, this effect doesn't occur.

- 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();  
}
```

- Personally, unless the script is very short, I tend to write all the code in external libraries, and to have only a few lines in the main script.
- Generally the main script simply calls the main function of my library.
- I don't worry about nested subroutines effects anymore (unless I create them myself :).
- Keep the warnings mode On and Perl will gladly tell you whenever you have this effect, by saying:

Variable "\$counter" will not stay shared at ... [snipped]

- Just don't forget to check your *error_log* file, before going into production!

- The above example was pretty boring.
- In my first days of using mod_perl, I wrote a simple user registration program.
- I'll give a very simple representation of this 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 to production.
- When my boss decided to test the production version and registered as let's say "The Boss", he saw the response "Thank you, Stas!".
- But I've tested the script a lot on development machine and it worked. What's the catch?
- We will see in a minute

3.3.2 *The Second Mystery*

- Back to our original example
- Why did we see inconsistent results over numerous reloads?
- Every time a server gets a request to process, it hands it over one of the children, generally in a round robin fashion.
- So if you have 10 httpd children alive
- The first 10 reloads might seem to be correct because the effect we've just talked about starts to appear from the second re-invocation.

- Subsequent reloads then return unexpected results.
- Moreover, requests can appear at random and children don't always run the same scripts.

- Now you see why we didn't notice the problem with the user registration system in the example.
- First, we didn't look at the `error_log`.
- (As a matter of fact we did, but there were so many warnings in there that we couldn't tell what were the important ones and what were not).
- Second, we had too many server children running to notice the problem.

- A workaround is to run the server as a single process.
(`httpd -X`).
- Since there are no other servers (children) running, you will see the problem on the second reload.
- Warnings should be turned On
- `error_log` shouldn't be clobbered with multiply warnings.
- Clean these up, so you can distinguish the real problems from stupid non-relevant warnings.

3.4 Sometimes it Works, Sometimes it Doesn't

- Have you ever seen your code 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 don't change from one script invocation to another unless you change them.

Let's look at three real world examples:

3.4.1 *An Easy Break-in*

- Imagine that you enter some site where you have an account, perhaps a free email account. Now you want to peek at other users' emails.
- Type in a username you want to peek at and a dummy password and try to enter the account. You would be surprised but this might work.
- Consider the following code:

```
use vars ($authenticated);
my $q = new CGI;
my $username = $q->param('username');
my $passwd = $q->param('passwd');
authenticate($username, $passwd);
    # failed, break out
unless ($authenticated) {
    print "Wrong passwd";
    exit;
}
# user is OK, fetch user's data
show_user($username);
sub authenticate {
    my ($username, $passwd) = @_;
        # some checking
    $authenticated = 1 if SOME_USER_PASSWD_CHECK_IS_OK;
}
```

- I can type in any valid username and any dummy passwd and enter that user's account, if someone has successfully entered his account before me using the same child process!
- Since `$authenticated` is global--if it becomes 1 once, it'll stay 1 for the remainder of the child's life!!
- The solution is trivial--reset `$authenticated` to 0 at the beginning of the program.

- A cleaner solution of course is not to rely on global variables, but rely on the return value from the function.

```
my $q = new CGI;
my $username = $q->param( 'username' );
my $passwd = $q->param( 'passwd' );
my $authenticated = authenticate($username, $passwd);
# failed, break out
unless ($authenticated) {
    print "Wrong passwd";
    exit;
}
# user is OK, fetch user's data
show_user($username);
sub authenticate{
    my ($username, $passwd) = @_;
        # some checking
    return (SOME_USER_PASSWD_CHECK_IS_OK) ? 1 : 0;
}
```

- **Of course this example is trivial--but believe me it happens!**

3.4.2 Thinking *mod_cgi*

- Just another little one liner that can spoil your day, assuming you forgot to reset the `$allowed` variable.
- It works perfectly OK in plain `mod_cgi`:

```
$allowed = 1 if $username eq 'admin' ;
```

- But under `mod_perl` you gain the admin access if you are served by the same process, the admin was served by.
- The obvious fix is:

```
$allowed = $username eq 'admin' ? 1 : 0;
```

3.4.3 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.
- An example of such a case would be:

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

- To make sure you don't miss these bugs always test your CGI in single process mode (`httd -X`).

- To solve this particular *!o* modifier problem refer to the *Compiled Regular Expressions* section of the *Perl Reference* chapter we have skipped before.

3.5 @INC and mod_perl

- Under mod_perl, once the server is up, @INC is frozen and cannot be updated.
- The only opportunity to **temporarily** modify @INC is 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.**

- For example:

```
PerlSetEnv PERL5LIB /home/httpd/perl
```

- Or

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

- **In the startup file directly alter the @INC.**

- For example

```
startup.pl  
-----  
use lib qw( /home/httpd/perl /home/httpd/mymodules );
```

- and load the startup file from the configuration file by:

PerlRequire /path/to/startup.pl

3.6 Reloading Modules and Required Files

- During code developing process you want the server to reload the code when it gets changed.
- This is not happening under mod_perl as it's optimized for the production environment and not development.
- Only Registry scripts get reloaded if modified.
- Solutions?

3.6.1 *Restarting the server*

- The simplest approach is to restart the server each time you apply some change to your code.
- After restarting the server about 100 times, you will tire of it and you will look for other solutions.

3.6.2 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>
```

3.6.3 Reloading handlers

- If you want to reload a perhandler on each invocation, the following trick will do it:

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

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

3.7 Name collisions with Modules and libs

- This sections requires an indepth understanding of `use()`, `require()`, `do()`, `%INC` and `@INC`.
- Each child process has its own `%INC` hash which is used to store information about its compiled modules.
- The keys of the hash 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.

- Let's look at three scripts with faults related to name space.
- 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 compiled into the package, the request for the other module will be skipped, since the server will think that it's already compiled.

- 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 Tool 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 Tool Number Two!</B>\n";
}
1;
-----
```

- Both scripts call `use Foo;`.
- Only the first one called will know about `Foo`.
- When you call the second script it will not know about `Foo` at all--it's like you've forgotten to write `use Foo;`.
- You will see the following in the `error_log` file:

`Undefined subroutine`

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

- Run the server in the single server mode (`httpd -X`) to detect this kind of bug immediately.

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
- There is almost no difference between `use()` and `require()` if you don't have to import some symbols into a calling script.
- Only the first script served will actually do the `require()`, for the same reason as the example above. `%INC` already includes the key "`config.pl`"!

Scenario 3

- It is interesting that the following scenario will fail too!
 - `./perl/tool/config.pl`
 - `./perl/tool/tool1.pl`
 - `./perl/tool/tool2.pl`
- where `tool1.pl` and `tool2.pl` both `require()` the **same** `config.pl`.

There are three solutions for this:

Solution 1

- Solves only the first two scenarios
- By placing your library modules in a subdirectory structure so that they have different path prefixes.
- 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";
```

- This solution isn't good, since while it might work for you now, 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.
- Let's see some better solutions.

Solution 2

- Another option is to use a full path to the script, so it will be used as a key in `%INC`;
`require "full/path/to/the/config.pl";`
- This solution solves the problem of all three scenarios.
- With this solution you loose some portability.
- If you move the tool around in the file system you will have to change the base directory

Solution 3

- Declare a package in the required files!
- 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.
- Foresee problems like this and save yourself future trouble.

- 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:

- You have to use the `Package::function()` method to call a subroutine from `Package`

- To access a global variable `$foo` inside the same package you have to write `$Package::foo`.
- Lexically defined variables, those declared with `my()` inside `Package` will be inaccessible from outside the package.

- You can leave your scripts unchanged if you import the names of the global variables and subroutines into the namespace of package **main::** like this:

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

- You can export both subroutines and global variables.
- Note however that this method has the disadvantage of consuming more memory for the current process.
- See `perl doc Exporter` for information about exporting other variables and symbols.

- See also the `perlmodlib` and `perlmodmanpages`.
- From the above discussion it should be clear that 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.

3.8 `__END__` and `__DATA__` tokens

- `Apache::Registry` scripts cannot contain `__END__` or `__DATA__` tokens.
- Because `Apache::Registry` scripts are being wrapped into a subroutine called `handler`, like the script at `URI/perl/test.pl`:

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

- When the script is being executed under `Apache::Registry` handler, it actually 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
}
```

- and you try to execute this script, you will receive the following warning:

Missing right bracket at line 4, at end of line

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

3.9 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.

3.10 Using `format()` and `write()`

- The interface to filehandles which are linked to variables with Perl's `tie()` function is not yet complete.
- The `format()` and `write()` functions are missing.
- If you configure Perl with `sfi`, `write()` and `format()` should work just fine.

3.11 Terminating requests and processes, the `exit()` and `child_terminate()` functions

- Perl's `exit()` built-in function cannot be used in `mod_perl` scripts.
- Calling it causes the `mod_perl` process to exit (which 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 {
    # Auto-detect if we are running under mod_perl or CGI.
    $USE_MOD_PERL = ( (exists $ENV{'GATEWAY_INTERFACE'}
        and $ENV{'GATEWAY_INTERFACE'} =~ /CGI-Perl/)
        or exists $ENV{'MOD_PERL'} ) ? 1 : 0;
}
use subs qw(exit);
# select the correct exit function
#####
sub exit{
    $USE_MOD_PERL ? Apache::exit(0) : CORE::exit(0);
}

```

- Now the correct `exit()` will be always chosen, whether you run the script under `mod_perl`, ordinary CGI or from the shell.
- Note that scripts running under `Apache::Registry` shouldn't worry about `exit()` as it overrides the Perl core built-in function.

- `Apache::exit(-2)` or `Apache::exit(Apache::Constants::DONE)` will cause the server to exit gracefully, completing the logging functions and protocol requirements etc.
- If you need to shut down the child cleanly after the request was completed, use the `$r->child_terminate` method.

3.12 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:
 - `mod_perl` traps the `__DIE__` signal,
 - logs the error message
 - and calls `Apache::exit()` instead of real `die()`.
- Thus the script stops, but the process doesn't quit.

- This is an example of a trapping code, not the real code:

```
$SIG{__DIE__} = sub { print STDERR @_; Apache::exit(); }
```

3.13 I/O is different

- If you are using Perl 5.004 or better, most CGI scripts can run under `mod_perl` untouched.
- If you're using 5.003, Perl's built-in `read()` and `print()` functions do not work as they do under CGI.
- If you're using `CGI.pm`, use `$query->print` instead of plain `print()`.

3.14 STDIN, STDOUT and STDERR streams

- In `mod_perl` both `STDIN` and `STDOUT` are tied to the socket the request came from. `STDERR` is tied to the `error_log` file.
- To print to `STDOUT` you can either use a regular `print()` (which is automatically tied to the socket) or the `$r->print` method.

3.15 Global Variables

Persistence

- We saw already that the global variables are persistent!
- You must never rely on the value of the global variable if it wasn't initialized at the beginning of the request processing.
- You should avoid using global variables unless it's impossible without them
- Because it will make the code development harder
- And you will have to make very sure that all the variables are initialized before being used.

- Use `my ()` scoped variables everywhere you can.
- You should be especially careful with Perl Special Variables, like `$|`, `$/` and other.

3.16 Generating correct HTTP Headers

- If you don't want to send the header yourself, you should have a

`PerlSendHeader` On

directive in the location configuration,

- The only header that you must send is a document MIME type.

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

- If you use `CGI.pm`'s `header()` function to generate HTTP headers,
- you do not need to activate `PerlSendHeader` directive
- because `CGI.pm` detects `mod_perl` and calls `send_http_header()` for you.

3.17 NPH (Non Parsed Headers) scripts

- To run a Non Parsed Header CGI script under mod_perl, simply add to your code:

```
local $| = 1;
```

- And if you normally set `PerlSendHeader On`, add this to your server's configuration file:

```
<Files */nph-*>  
    PerlSendHeader Off  
</Files>
```

3.18 BEGIN blocks

- Perl executes `BEGIN` blocks as soon as possible, at the time of compiling the code.
- The same is true under `mod_perl`.
- However, since `mod_perl` normally only compiles scripts and modules once, either in the parent server or once per-child, `BEGIN` blocks in that code will only be run once.
- As the `perlmod` manpage explains, once a `BEGIN` block has run, it is immediately undefined.
- In the `mod_perl` environment, this means that `BEGIN` blocks will not be run during the response to an incoming request unless that request happens to be the one that causes the

compilation of the code.

`BEGIN` blocks in modules and files pulled in via `require()` or `use()` will be executed:

- Only once, if pulled in by the parent process.
- Once per-child process if not pulled in by the parent process.
- An additional time, once per child process if the module is pulled in off a disk again via `Apache::StatINC`.
- An additional time, in the parent process on each restart if `PerlFreshRestart` is On.
- Unpredictable if you fiddle with `%INC` yourself.

BEGIN blocks in Apache::Registry scripts will be executed, as above plus:

- **Only once, if pulled in by the parent process via Apache::RegistryLoader.**
- **Once per-child process if not pulled in by the parent process.**
- **An additional time, once per child process, each time the script file changes on disk.**
- **An additional time, in the parent process on each restart if pulled in by the parent process via Apache::RegistryLoader and PerlFreshRestart is On.**

3.19 END blocks

- As the `perlmod` manpage explains, an `END` subroutine is executed as late as possible, that is, when the interpreter exits.
- In the `mod_perl` environment, the interpreter does not exit until the server shuts down.
- However, `mod_perl` does make a special case for `Apache::Registry` scripts and executes it every time the request is completed.
- All other `END` blocks encountered during other `Perl*Handler` call-backs, e.g. `PerlChildInitHandler`, will be suspended while the process is running and called during `child_exit()` when the process is shutting down.

- **If you write a module and want a `END` behavior use**
`$r->register_cleanup()`
- `$r->register_cleanup()` is called at the `CleanUp` processing phase of each request and thus can be used to emulate plain perl's `END{ }` block behavior.

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

- Normally when you run perl from the command line, you have the shell invoke it with `#!/bin/perl` (sometimes referred to as a shebang line).
- In scripts running under `mod_cgi`, you may use perl execution switch arguments as described in the `perlrun` manpage, such as `-w`, `-T` or `-d`.
- Since scripts running under `mod_perl` don't need the shebang line, all switches except `-w` are ignored by `mod_perl`.

- This feature was added for a backward compatibility with CGI scripts.
- Most command line switches have a special variable equivalent.
- Consult the `perlvar` manpage for more details.

3.20.1 Warnings

There are three ways to enable warnings:

- **Globally to all Processes**
 - Setting:

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. You can turn them **Off** and **On** in the script by setting the `$_^W` variable as noted above.

- **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  
}
```

- Note, that if you forget the `local` operator this code will affect the child processing the current request, and all the subsequent requests processed by that child.

- Thus

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

- will turn the warnings *Off*, no matter what.

- If you want to turn warnings *On* for the scope of the whole file, as in the previous item, you can do this by adding:

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

- at the beginning of the file.
 - Since a file is effectively a block,
 - file scope behaves like a block's curly braces { }
 - and `local $^W` at the start of the file will be effective for the whole file.
- While having warning mode turned **On** is a must for a development server, you should turn it globally **Off** in a production server.

- Since if every served request generates only one warning,
- and your server serves millions of requests per day,
- your log file will eat up all of your disk space and your system will die.

3.20.2 Taint Mode

- Perl's `-T` switch enables *Taint* mode.
- If you aren't forcing all your scripts to run under **Taint** mode you are looking for trouble from malicious users.
- (See the *perlsec* manpage for more information)
- Since the `-T` switch doesn't have an equivalent perl variable, `mod_perl` provides the `PerlTaintCheck` directive to turn on taint checks.
- In `httpd.conf`, enable this mode with:

PerlTaintCheck On

- Now any code compiled inside httpd will be taint checked.
- If you use the `-T` switch, Perl will warn you that you should use the `PerlTaintCheck` configuration directive and will otherwise ignore it.

3.20.3 *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`.

3.21 The strict pragma

- It's `_absolutely_` mandatory (at least for development) to start all your scripts with:

`use strict;`
- If needed, you can always turn off the 'strict' pragma or a part of it inside the block, e.g:

```
{  
    no strict 'refs';  
    ... some code  
}
```

- It's more important to have `strict` pragma enabled under `mod_perl` than anywhere else.
- While it's not required by the language, its use cannot be too strongly recommended.
- It will save you a great deal of time.
- And, of course, clean scripts will still run under `mod_cgi` (plain CGI)!

3.22 Passing ENV variables to CGI

- To pass an environment variable from a configuration file, add to it:

```
PerlsetEnv key val
```

```
PerlPassEnv key
```

- e.g.:

```
PerlsetEnv PERLDB_OPTS "LineInfo=/tmp/db.out AutoTrace=1"
```

will set \$ENV{PERLDB_OPTS}, and it will be accessible in every child.

- `%ENV` is only set up for CGI emulation.
- If you are using the API, you should use `$r->subprocess_env`, `$r->notes` or `$r->pnotes` for passing data around between handlers.
- `%ENV` is slow because it must update the underlying C environment table.
- This also exposes the data on systems which allow users to see the environment with `ps`.
- In any case, `%ENV` and the tables used by those methods are all cleared after the request is served so that `$ENV{SESSION_ID}` will not be swapped or reused by different http requests.

3.23 Apache and syslog

- When native syslog support is enabled, the stderr stream will be redirected to `/dev/null`!
- It has nothing to do with `mod_perl` (plain Apache does the same). Doug wrote the `Apache::LogSTDERR` module to work around this.

3.24 Filehandlers and locks leakages

- When you write a script running under mod_cgi, you can get away with sloppy programming,
- like opening a file and letting the interpreter close it for you when the script had finished its run:

```
open IN, "in.txt" or die "Cannot open in.txt for reading : $!\n";
```

- For mod_perl, before the end of the script you **must** `close()` any files you opened!

`close IN;`

- If you forget to `close()`, you might get file descriptor leakage and (if you `flock()`ed on this file descriptor) unlock problems.
- Even if you do close the files, but for some reason the interpreter was stopped before the `close()` call, the leakage is still there.
- For example when a user presses the Stop button.
- After a long run without restarting Apache your machine might run out of file descriptors, and worse, files might be left locked and unusable.

- What can you do?

Use `IO::File` (and the other `IO::*` modules).

- This allows you to assign the file handler to variable which can be `my()` (lexically) scoped.
- When this variable goes out of scope the file or other file system entity will be properly closed (and unlocked if it was locked).
- If the variable was lexically defined inside some internal block, it will go out of scope at the end of the block. For example:

```
{  
    my $fh = new IO::File("filename") or die $!;  
    # read from $fh  
    # ...$fh is closed automatically at end of block, without leaks.  
}
```

- A script in a file is effectively written in a block with the same scope as the file, so you can simply write:

```
my $fh = new IO::File("filename") or die $!;  
# read from $fh  
# ...$fh is closed automatically at end of script, without leaks.
```

- Using a `{ BLOCK }` makes sure is that the file is closed the moment that the end of the block is reached.
- An even faster and lighter technique is to use `Symbol.pm`:

```
my $fh = Symbol::gensym();  
open $fh, "filename" or die $!;
```

- Use these approaches to ensure you have no leakages, but don't be too lazy to write `close()` statements. Make it a habit.

3.25 The Script Is Too Dirty, But It Does The Job And I Cannot Afford To Rewrite It.

- You still can win from using `mod_perl`.
- One approach is to replace the `Apache::Registry` handler with `Apache::PerlRun` and define a new location. The script can reside in the same directory on the disk.

```
# srm.conf
Alias /cgi-perl/ /home/httpd/cgi/

# httpd.conf
<Location /cgi-perl>
```

```
#AllowOverride None
SetHandler perl-script
PerlHandler Apache::PerlRun
Options ExecCGI
allow from all
PerlSendHeader On
</Location>
```

- Another “bad”, but workable method is to set `MaxRequestsPerChild` to 1, which will force each child to exit after serving only one request.
- You will get the preloaded modules, etc., but the script will be compiled for each request, then thrown away.

- This isn't good for "high-traffic" sites, as the parent server will need to fork a new child each time one is killed.
- You can fiddle with `MaxStartServers` and `MinSpareServers`, so that the parent pre-spawns more servers than actually required and the killed one will immediately be replaced with a fresh one.
- Probably that's not what you want.

3.26 Apache::`PerlRun`--a closer look

- Apache::`PerlRun` gives you the benefit of preloaded Perl and its modules.
- This module's handler emulates the CGI environment, allowing programmers to write scripts that run under CGI or `mod_perl` without any change.
- Unlike Apache::`Registry`, the Apache::`PerlRun` handler does not cache the script inside a subroutine.
- Scripts will be “compiled” on each request.

- After the script has run, its name space is flushed of all variables and subroutines.
- Still, you don't have the overhead of loading the Perl interpreter and the compilation time of the standard modules.
- If your script is very light, but uses lots of standard modules, you will see no difference between `Apache::PerlRun` and `Apache::Registry!`
`;o)`

4 Controlling and Monitoring the Server

4.1 What we will learn in this chapter

- Restarting techniques
- Implications of sending TERM, HUP, and USR1 to the server
- Using apachectl to control the server
- Safe Code Updates on a Live Production Server
- SUID start-up scripts
- Preparing for Machine Reboot

- **Monitoring the Server. A watchdog.**
- **Running server in a single mode**

4.2 Restarting techniques

- Using `kill(1)`
- Need to know PID
- Find it in the file specified by `httpd.conf`: `PidFile` directive

`PidFile /usr/local/var/httpd_per1/run/httpd.pid`

- Another way is to use the `ps` and `grep` utilities:

```
% ps auxc | grep httpd_per1
```

- or maybe:

```
% ps -ef | grep httpd_perl
```

- You will see something like this

```
root      1302  0.0  4.3 10260 8380 ?    S    14:54   0:00 httpd_perl
nobody    1304  0.0  4.7 10676 9184 ?    S    14:54   0:00 httpd_perl
nobody    1305  0.0  4.7 10672 9180 ?    S    14:54   0:00 httpd_perl
nobody    1306  0.0  4.7 10672 9180 ?    S    14:54   0:00 httpd_perl
```

- Get the one with *root* ownership, use it to control the server and its children.
- There are three signals that you can send the parent: **TERM**, **HUP**, and **USR1**.

4.3 Implications of sending TERM, HUP, and USR1 to the server

- We will concentrate here on the implications of sending these signals to a mod_perl enabled server.
- For plain Apache behavior see <http://www.apache.org/docs/stopping.html> .

TERM Signal: stop now

- Sending the **TERM** signal to the parent causes it to immediately attempt to kill off all of its children.
- Takes several seconds to complete, following which the parent itself exits.
- Any requests in progress are terminated, and no further requests are served.
- That's the moment that the accumulated **END** blocks will be executed!
- Note that if you use `Apache::Registry` or `Apache::PerlRun`, then **END** blocks are being executed upon each request (at the end).

HUP Signal: restart now

- Just like **TERM** but the parent doesn't exit.
- Re-reads its configuration files,
- Re-opens any log files.
- Flush all the compiled and preloaded modules,
- Rerun any startup files.
- Continues serving hits.
- Spawns a new set of children

- It's equivalent to stopping, then restarting a server.
- If your configuration file has errors in it when you issue a restart then your parent will not restart but exit with an error.

USR1 Signal: graceful restart

- The **USR1** signal causes the parent process to advise the children to exit after their current request (or to exit immediately if they're not serving anything).
- The parent re-reads its configuration files and re-opens its log files.
- As each child dies off the parent replaces it with a child from the new generation of the configuration, which begins serving new requests immediately.
- The only difference between **USR1** and **HUP** is that **USR1** allows children to complete any in-progress request prior to killing them off.

- By default, if a server is restarted (with **USR1** or **HUP** signals), Perl scripts and modules are not reloaded.
- To reload modules and flush the `Apache::Registry` cache, do:

PerlFreshRestart On (in httpd.conf)

- It causes a lot of problem since not all modules can stand the reload and should be avoided if possible.

- Restart or termination can sometimes take quite a lot of time.
- `PERL_DESTRUCT_LEVEL=-1` speeds this up and leads to more robust operation in the face of problems, like running out of memory.
- Usable only if no significant cleanup has to be done by perl `END` blocks and `DESTROY` methods when the child terminates
- What constitutes significant cleanup?
- Any change of state outside of the current process that would not be handled by the operating system itself.
- So committing database transactions is significant

- **But closing an ordinary file isn't.**

- Numeric values of signals `kill(1)` man page

- `/usr/include/sys/signal.h`:

```
#define SIGHUP    1    /* hangup, generated when terminal disconnects */
#define SIGTERM  15   /* software termination signal */
#define SIGUSR1  30   /* user defined signal 1 */
```

- `'kill -HUP 1234' == 'kill -1 1234'`

4.4 Using `apachectl` to control the server

- Apache's distribution provides a nice script to control the server.
- It's called `apachectl` and it's installed into the same location with `httpd`.
- In my installation it's `/usr/local/sbin/httpd_perl/apachectl`.
- Start `httpd`:

```
% /usr/local/sbin/httpd_perl/apachectl start

• Stop httpd:

% /usr/local/sbin/httpd_perl/apachectl stop

• Restart httpd if running by sending a SIGHUP or start if not
  running:

% /usr/local/sbin/httpd_perl/apachectl restart

• Do a graceful restart by sending a SIGUSR1 or start if not
  running:

% /usr/local/sbin/httpd_perl/apachectl graceful
```

- Do a configuration syntax test:

```
% /usr/local/sbin/httpd_perl/apachectl configtest
```

- There are other options for **apachectl**, use `help` option to see them all.
- **apachectl** is a rc file
- Copy it to */etc/rc.d/rc3.d/S99apache* or similar
- Create a symlink, but be careful that `apachectl` is writable only by root -- the startup scripts have root privileges during init processing, and you don't want to be opening any security holes.)

4.5 Safe Code Updates on a Live Production Server

- You have prepared a new version of code
- Uploaded it into a production server, restarted it
- and... it doesn't work.-- it's a live server!
- What could be worse than that?
- Roll everything back?
- Cannot go back, because the good working code was overwritten

- Easy to prevent -- don't overwrite the previous working version!

- How I do it:
- Assume `/home/httpd/perl/re1` is a root dir and we are located in this directory
- Put the new version in *beta* directory
- Last sanity checks (file permissions (rx for server),
- Testing again the new modules with `perl -c`
- Ready to swicth

- Now I do:

```
% cd /home/httpd/perl  
% mv rel old && mv beta rel && stop && sleep 3 && restart && err
```

- Which is in fact does

```
% mv rel old  
% mv beta rel  
% /usr/local/apache/bin/apachectl stop  
% sleep 3  
% /usr/local/apache/bin/apachectl restart  
% tail -f /usr/local/apache/logs/error_log
```

- I use aliases to make things faster:

```
% alias | grep apachectl
restart /usr/local/apache/bin/apachectl restart
start /usr/local/apache/bin/apachectl start
stop /usr/local/apache/bin/apachectl stop

% alias err
tail -f /usr/local/apache/logs/error_log
```

- `&&` ensures that if any command has failed the rest won't be executed.
- All-in-one ensures that if I suddenly get a connection lost (sadly but that happens sometimes) I wouldn't leave the server down if only the `stop` command squeezed in.

- 'sleep 3' is required to let Apache terminate its processes.
- If you don't wait and start the server right away you might see:

`Address already in use: make_sock: could not bind to port 8080`

- Your server may not need it, or need to wait even for more seconds.
- If you use `restart`, it will patiently wait for the server to quit and then will cleanly start it.

- Executing 'err' immediately indicates whether there are any problems.

- If there are problems we must roll back. ASAP!!!

```
% mv rel bad && mv old rel && stop && sleep 3 && restart && err
```

- Just as before it's:

```
% mv rel bad
% mv old rel
% /usr/local/apache/bin/apachectl stop
% sleep 3
% /usr/local/apache/bin/apachectl restart
% tail -f /usr/local/apache/logs/error_log
```

- And 99.9% that everything would be alright,
- You have had only about 10 secs of downtime, which is pretty good!

4.6 SUID start-up scripts

- If you don't want or cannot become a *root* in order to control your webserver, suid script is the only solution.
- The code is in your handouts, there is nothing fancy about it
- You need a perl version that recognizes and emulates the suid bits in order for this to work.
- The script will do different things depending on whether it is named `start_http`, `stop_http` or `restart_http`.
- You can use symbolic links for this purpose.

4.7 Preparing for Machine Reboot

- Dev box can get away without automatic server startup/shutdown rc scripts.
- Not the production server!
- After building mod_perl enabled Apache, don't forget to install the rc files!!!
- The simplest solution is to copy there the apachectl script
- If you have more than one Apache server, you have to put a script for each one, of course renaming them on the way.

- This is the setup I have:

```
/etc/rc.d/init.d/httpd_docs
```

```
/etc/rc.d/init.d/httpd_perl
```

```
/etc/rc.d/rc3.d/s86httpd_docs -> ../init.d/httpd_docs
```

```
/etc/rc.d/rc3.d/s87httpd_perl -> ../init.d/httpd_perl
```

```
/etc/rc.d/rc6.d/K86httpd_docs -> ../init.d/httpd_docs
```

```
/etc/rc.d/rc6.d/K87httpd_perl -> ../init.d/httpd_perl
```

- On startup OS enters level 3 and executes the rc files from rc3.d

```
/etc/rc.d/init.d/httpd_perl start
```

```
/etc/rc.d/init.d/httpd_docs start
```

- On shutdown, OS enters level 6 and executes the rc files from rc6.d

```
/etc/rc.d/init.d/httpd_perl stop  
/etc/rc.d/init.d/httpd_docs stop
```

- Your OS can have a different idea about rc files, but the principle is the same.

4.8 Monitoring the Server. A watchdog.

- Your webserver might die when you won't be around
- The processes might hang
- You need a watchdog to look after your service.

Modified apachectl as a watchdog

- I've named it *apache.watchdog*
- Call it from the crontab every 30 mites or 1 minute.
- A sample crontab entry:

```
0,30 * * * * /path/to/the/apache.watchdog >/dev/null 2>&1
```

- The script is in your handouts
- If everything is OK it'll do nothing
- But if the server is down it will attempt to restart it

- You will receive the status of restart (pass|fail)

A watchdog based on the LWP module

- Using LWP to make a better testing by requesting some script served by your webserver
- Why is it more practical?
- Because, while the server can be up as a process, it can be stuck and not working
- Failing to get the document will trigger restart, and “probably” the problem will go away.
- Call it from crontab
- Execute it frequently. I use one minute, the highest cron’s time resolution

- Why so frequent?
- If your server starts to spin and trash your disk's space with multiply error messages,
- In a 5 minutes you might run out of free space, which might bring your system to its knees.
- You are running `mod_perl`! Adding one more request per second won't be felt at all!
- So we end up with crontab entry:

```
* * * * * /path/to/the/watchdog.pl >/dev/null 2>&1
```

- The code as usual in your handouts :)

4.9 Running server in a single mode

- Developing new code? Test it while running in the single process mode.
- Running in single process mode inhibits the server from “daemonizing”, allowing you to run it more easily under debugger control.

```
% /usr/local/sbin/httpd_perl/httpd_perl -X
```

- The server will run in the foreground of the shell you have called it from.

- So to kill you just kill it with **Ctrl-C**.

- In `-x` mode the server will run very slowly while fetching images. under Netscape

- Netscape tries to open multiple connections and keep them open.

- Because there is only one server process listening, each connection has to time-out before the next succeeds.

- Solution: Turn off `KeepAlive` in `httpd.conf` to avoid this effect while testing in the single mode

- When running with `-x` You won't see any control messages that the parent server normally writes to the `error_log`.

- Since `httpd -X` causes the server to handle all requests itself, without forking any children, there is no controlling parent to write status messages.

;o)

5 mod_perl and Relational Databases

5.1 What we will learn in this chapter

- Why Relational (SQL) Databases
- Apache::DBI - Initiate a persistent database connection
- Introduction
- Configuration
- Preopening DBI connections
- Debugging Apache::DBI

- Troubleshooting

5.2 Why Relational (SQL) Databases

- Nowadays you cannot imagine a website without a DB behind it.
- DBI module is used to drive DBs from within Perl/CGI scripts and modules.
- Actually it's only a bridge between your code and DBD module, that actually talks to the DB server itself.

5.3 Apache::DBI

- `mod_cgi` limitation: DB connection overhead for each request
- `mod_perl` allows data persistence at the server side
- Apache::DBI under `mod_perl` overcomes `mod_cgi` limitation
- Provides a single persistent database connection per process.
- The Connection persists for the process' entire life.
- Scripts need neither to initiate connection nor terminate it.

5.3.1 Introduction

- Apache::DBI must be loaded before DBI.
- DBI is Apache::DBI aware (if `$ENV{GATEWAY_INTERFACE} eq CGI-Perl`)
- DBI forwards every `connect()` request to `Apache::DBI`
- Apache::DBI uses the `ping()` method to look for a database handle from a previous `connect()` request, and tests if this handle is still valid.
- If these two conditions are fulfilled it just returns the database handle.

- **If `ping()` fails, `Apache::DBI` establishes a new connection and stores the handle for later re-use.**

delete()

- There is no need to delete the `disconnect()` statements from your code.
- Apache::DBI module overloads the `disconnect()` method with NOP.

- When this module should be used and when shouldn't?
- Important: The connection parameters must be the same!
- *host, username, password, etc*
- If there aren't a new connection will be opened.
- So if you use the same handle to access the DB
Apache::DBI is for you
- If you open a different connection per user (with different connection parameters) Apache::DBI will make your DB run out of connections very soon.
- If you have both situations on one machine, the only solution is to run two Apache/mod_perl servers, one which uses Apache::DBI and one which does not.

5.3.2 Configuration

- After installing this module, the configuration is simple
- Add the following directive to `httpd.conf`:

PerlModule Apache::DBI

- Note that it is important to load this module before any other `Apache*DBI` module and `DBI` module itself!
- You can skip preloading `DBI`, since `Apache::DBI` does that.
- But there is no harm in leaving it in, as long as it is loaded after `Apache::DBI`.

5.3.3 Preopening DBI connections

- You can preopen connections when the child process is spawned
- In the startup file add:

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

- Be warned though, that 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!

5.3.4 Debugging Apache::DBI

- Enable Debug mode to see it working correctly by adding in the startup script:

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

- For a full trace you set:

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

Another approach is to add to `httpd.conf` (which does the same):

```
PerlModule Apache::DebugDBI
```

- All the log messages go to the *error_log* file

5.3.5 Troubleshooting

5.3.5.1 The Morning Bug

- The SQL server keeps a connection to the client open for a limited period of time.
- Many developers were bitten by so called **Morning bug**, when every morning the first users to use the site received a No Data Returned message, but after that everything worked fine.
- The error is caused by `Apache: :DBI` returning a handle of the invalid connection (the server closed it because of a timeout), and the script was dying on that error.
- The infamous `ping()` method was introduced to solve this problem, but still people were being bitten by this problem.

- Another solution was found - to increase the timeout parameter when starting the SQL server.
- Currently I startup MySQL server with a script `safe_mysql`, so I have modified it to use this option:

```
nohup $ledir/mysqld [snipped other options] -O wait_timeout=172800
```

- 172800 seconds is equal to 48 hours. This change solves the problem.
- Note that as from version 0.82, Apache::DBI implements `ping()` inside the `eval` block.
- This means that if the handle has timed out it should be reconnected automatically, and avoid the morning bug.

5.3.5.2 Opening connections with different parameters

- It's important to stress again the importance of using the exact connecting parameters
- Apache : :DBI insists that the new connection be opened in exactly the same way as the cached connection.
- If I one script sets `LongReadLen` and another does not, Apache : :DBI will make two different connections.
- So instead of having a maximum of 40 open connections, I can end up with 80.

- However, you are free to modify the handle immediately after you get it from the cache.
- So always initiate connections using the same parameters and set `LongReadLen` (or whatever) afterwards.

5.3.5.3 Debugging code which deploys DBI

- To log a trace of DBI statement execution, you must set the `DBI_TRACE` environment variable.
- The `PerlSetEnv DBI_TRACE` directive must appear before you load `Apache::DBI` and `DBI`.
- For example if you use `Apache::DBI`, modify your `httpd.conf` with:

```
PerlSetEnv DBI_TRACE "3=/tmp/dbitrace.log"  
PerlModule Apache::DBI
```

- The traces from each request will be appended to `/tmp/dbitrace.log`.

- Note that the logs might interleave if requests are processed concurrently.
- Replace 3 with the TRACE level you want.
- Within your code you can control trace generation with the `trace()` method:

```
DBI->trace($trace_level)
```

```
DBI->trace($trace_level, $trace_filename)
```

Trace Levels:

- 0 - trace disabled.
- 1 - trace DBI method calls returning with results.

- 2 - trace method entry with parameters and exit with results.
- 3 - as above, adding some high-level information from the driver also adds some internal information from the DBI.
- 4 - as above, adding more detailed information from the driver also includes DBI mutex information when using threaded perl.
- 5 and above - as above but with more and more obscure information.
;o)

6 mod_perl and dbm files

6.1 What we will learn in this chapter

- Where and Why to use dbm files
- mod_perl and dbm
- Locking dbm handlers

6.2 Where and Why to use dbm files

Requirements:

- A light database
- An easy API
- Use of simple key-value pairs to store and manipulate the records

Solution:

- Enter the DBM world
- The maximum practical size depends on your hardware and the desired response times
- As a rough guide consider 5000 to 10000 records to be reasonable.
- Berkeley DB is the most powerful dbm implementation.
- The whole database is rarely read into a memory.
- Smart storage techniques
- dbm files can be manipulated much faster than their flat file brothers.

- Flat file databases can become very slow on insert, update and delete operations,

Storage algorithms:

- The HASH algorithm
- $O(1)$ complexity of search and update,
- fast insert and delete, but a slow sort.
- The B TREE algorithm
- allows arbitrary key/value pairs to be stored in a sorted, balanced binary tree,
- allows to get a sorted sequence of data pairs in $O(1)$,
- but at the expense of much slower insert, update, delete operations than is the case with HASH.

- The RECNO algorithm
- enables both fixed-length and variable-length flat text files to be manipulated using the same key/value pair interface as in HASH and BTREE.
- In this case the key will consist of a record (line) number.

- Most often you will want to use the `HASH` method
- Your choice depends very much on your application.
- **dbm** databases are not limited to storing key/value pairs.
- `MLDBM` module use to store more complicated data structures.
- This module can dump and restore the whole symbol table of your script, including arrays, hashes and other complicated data structures.

6.3 mod_perl and dbm

- Where does mod_perl fit into the picture?
- Read only dbm file which is kept open (tied) all the time.
- Read/write dbm as well but you need to add locking and data flushing to avoid data corruption.
- Allows much faster access because of removed need to `tie()`/`untie()`

Hazards

- What happens if user aborts the script by pressing 'Stop' button?
- What happens if the process `die()`'s?
- Stale lock files when using external lock files.
- A deadlock situation if two processes simultaneously try to acquire locks on two separate databases.

6.4 Locking dbm handlers

- Let's make the lock status a global variable
- It will persist from request to request.
- If we request a lock - *READ* (shared) or *WRITE* (exclusive), we obtain the current lock status first.

READ lock request:

- Granted as soon as the file becomes unlocked
- Or if it is already *READ* locked.

- The lock status becomes *READ* on success.

WRITE lock request:

- Granted as soon as the file becomes unlocked.
- The lock status becomes *WRITE* on success.
- The treatment of the *WRITE* lock request is most important.

Write Starvation:

- If the DB is *READ* locked, a process that makes a *WRITE* request will poll until there are no reading or writing processes left.
- The following diagram represents a possible scenario where everybody can read but no one can write:

```
[ -p1- ]           [ --p1-- ]  
[ --p2-- ]  
[ -----p3----- ]  
                    [ -----p4----- ]  
[ --p5-- ]       [ -----p5----- ]
```

- The result is a starving process

- Will timeout the request, and it will fail to update the DB.

6.4.1 *Tie::DB_Lock*

- `Tie::DB_Lock` ties hashes to databases using shared and exclusive locks.
- This module, by Ken Williams, solves the problems raised in the previous section.
- `Tie::DB_Lock` copies a dbm file on read.
- Reading processes do not have to keep the file locked while they read it.
- Writing processes can still access the file while others are reading.

- This works best when you have lots of long-duration reading, and a few short bursts of writing.
- The drawback of this module is the heavy IO performed when every reader makes a fresh copy of the DB.
- With big dbm files this can be quite a disadvantage and can slow the server down considerably.

6.4.2 Locking techniques that work with dbm files

6.4.2.1 Flawed methods which must not be used

- **Caution:** The suggested locking methods in the Camel book and DB_File man page (at least before the version 1.72) are flawed.
- If you use them in an environment where more than one process can modify the dbm file, it can get corrupted!!
- The following is an explanation of why this happens.
- You may not use a tied file's filehandle for locking
- You get the filehandle after the file has been already tied.

- It's too late to lock.
- The problem is that the database file is locked **after** it is opened.
- When the database is opened, the first 4k (in my dbm library) are read and then cached in memory.
- Therefore, a process can open the database file, cache the first 4k, and then block while another process writes to the file.
- If the second process modifies the first 4k of the file, when the original process gets the lock is now has an inconsistent view of the database.

- If it writes using this view it may easily corrupt the database on disk.

6.4.2.2 Lock on tie (only supported by a few operating systems)

- On some Operating Systems like FreeBSD, it's possible to lock on tie:

```
tie my %t, 'DB_File', $TOK_FILE, O_RDWR | O_EXLOCK, 0664;
```

- and only release the lock by untying the file.
- Notice the `O_EXLOCK` flag, which is not available on all Operating Systems.

6.4.2.3 DB_File::Lock

- In your handouts you will find a module which does the locking by using an external lockfile.
- There are examples of usage as well
- There is a new DB_File::Lock module available from CPAN, by David Harris.

;o)

7 Getting Help and Further Learning

7.1 What we will learn in this chapter

- Getting help
- Get help with mod_perl
- Get help with Perl
- Get help with Perl/CGI
- Get help with Apache
- Get help with DBI

- **Get help with Squid**

7.2 Getting help

1. Searchable Mailing list archive
2. The Books and Online Documentation
3. Mailing List as a last resort.

7.3 Get help with mod_perl

- **mod_perl home**

<http://perl.apache.org>

- **mod_perl Garden project**

<http://modperl.sourceforge.org>

- **mod_perl Books**

- **'Apache Modules' Book**

<http://www.modperl.com> is the home site of The Apache Modules Book, a book about creating Web server modules using the Apache API, written by Lincoln Stein and Doug MacEachern.

Now you can purchase the book at your local bookstore or from the online dealer. O'Reilly lists this book as:

Writing Apache Modules with Perl and C
By Lincoln Stein & Doug MacEachern
1st Edition March 1999
1-56592-567-X, Order Number: 567X
746 pages, \$34.95

- **'Enabling web services with mod_perl' Book**
<http://www.modperlbook.com> is the home site of the new mod_perl book, that Eric Cholet and Stas Bekman are co-authoring together. We expect the book to be published in fall 2000.

Ideas, suggestions and comments are welcome. You may send them to info@modperlbook.com .

- **mod_perl Guide**

by Stas Bekman at <http://perl.apache.org/guide>

- **mod_perl FAQ**

by Frank Cringle at <http://perl.apache.org/faq/> .

- **mod_perl performance tuning guide**

by Vivek Kherra at <http://perl.apache.org/tuning/> .

- **mod_perl plugin reference guide**

by Doug MacEachern at http://perl.apache.org/src/mod_perl.html .

- **Quick guide for moving from CGI to mod_perl**
at http://perl.apache.org/dist/cgi_to_mod_perl.html .
- **mod_perl_traps, common traps and solutions for mod_perl users**
at http://perl.apache.org/dist/mod_perl_traps.html .
- **mod_perl Quick Reference Card**
<http://www.refcards.com> (Apache and other refcards are available from this link)

- **mod_perl Resources Page**

http://www.perlreference.com/mod_perl/

- **mod_perl mailing list**

The Apache/Perl mailing list (modperl@apache.org) is available for mod_perl users and developers to share ideas, solve problems and discuss things related to mod_perl and the Apache::* modules. To subscribe to this list, send mail to modperl-subscribe@apache.org with empty Subject and with Body:

```
subscribe modperl
```

A **searchable** mod_perl mailing list archive available at <http://forum.swarthmore.edu/epigone/modperl> . We owe it to Ken Williams.

More archives available:

- <http://www.geocrawler.com/lists/3/web/182/0/>
- <http://www.bitmechanic.com/mail-archives/modperl/>
- <http://www.mail-archive.com/modperl%40apache.org/>
- <http://www.davin.ottawa.on.ca/archive/modperl/>
- <http://www.progressive-comp.com/Lists/?l=apache-modperl&r=1&w=2#apache-modperl>
- <http://www.egroups.com/group/modperl/>

7.4 Get help with Perl

- **The Perl FAQ**

<http://www.perl.com/CPAN/doc/FAQs/FAQ/PerlFAQ.html>

- **The Perl home**

<http://www.perl.com/>

- **The Perl Journal**

<http://www.tpj.com/>

- **Perl Module Mechanics**

http://world.std.com/~swmcd/steven/perl/module_mechanics.html

- This page describes the mechanics of creating, compiling, releasing and maintaining Perl modules.

7.5 Get help with Perl/CGI

- **Perl/CGI FAQ**

at <http://www.perl.com/CPAN/doc/FAQs/cgi/perl-cgi-faq.html>

- **Answers to some bothering Perl and Perl/CGI questions**

<http://www.stason.org/TULARC/webmaster/myfaq.html>

- **Idiot's Guide to CGI programming**

<http://www.perl.com/CPAN/doc/FAQs/cgi/idiots-guide.html>

- **WWW Security FAQ**

<http://www.w3.org/Security/Faq/www-security-faq.html>

- **CGI/Perl Taint Mode FAQ**

<http://www.gunther.web66.com/FAQS/taintmode.html> (by
Gunther Birznieks)

7.6 Get help with Apache

- **Apache Project's Home**

<http://www.apache.org>

- **Apache Quick Reference Card**

<http://www.refcards.com> (Apache and other refcards are available from this link)

- **The Apache FAQ**

<http://www.apache.org/docs/misc/FAQ.html>

- **Apache Server Documentation**

<http://www.apache.org/docs/>

- **Apache Handlers**

<http://www.apache.org/docs/handler.html>

- **mod_rewrite Guide**

<http://www.engelschall.com/pw/apache/rewriteguide/>

7.7 Get help with DBI

- **Perl DBI examples**

<http://www.saturn5.com/~jwb/dbi-examples.html> (by Jeffrey William Baker).

- **DBI Homepage**

<http://www.symbolstone.org/technology/perl/DBI/>

- **DBI mailing list information**

<http://www.fugue.com/dbi/>

- **DBI mailing list archives**

<http://outside.organic.com/mail-archives/dbi-users/>

<http://www.xray.mpe.mpg.de/mailling-lists/dbi/>

- **Persistent connections with mod_perl**

http://perl.apache.org/src/mod_perl.html#PERSISTENT_DATABASE_CONNECTIONS

7.8 Get help with Squid - Internet Object Cache

- Home page - <http://squid.nlanr.net/>
 - FAQ - <http://squid.nlanr.net/Squid/FAQ/FAQ.html>
 - Users Guide - <http://squid.nlanr.net/Squid/Users-Guide/>
 - Mailing lists - <http://squid.nlanr.net/Squid/mailling-lists.html>
- ;o)