

ApacheCon Europe
October 24, 2000
London, UK

Tutorial:
Getting Started with mod_perl

by Stas Bekman
<http://stason.org/>
<stas@stason.org>
JazzValley.com, CTO

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

Copyright © 1998-2000 Stas Bekman. All rights reserved. (Distributed under GPL license)

1 Agenda

1.1 Agenda

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

1.2 Off-tutorial reading

- Perl Reference
- Getting Help

;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
- March 2000 -- 612425 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. Having one binary and one configuration file (one big binary for mod_perl).
 2. Having two binaries and two configuration files (one big binary for mod_perl and one small binary for static objects like images.)

3. Any of the above plus a reverse proxy server in http accelerator mode.

- If you are a newbie, I would recommend that you start with the first option and work on getting your feet wet with apache and mod_perl.
- Later, you can decide whether to move to the second one which allows better tuning at the expense of more complicated administration,
- or to the third option which gives you even more power.

1.

- The first option will kill your production site if you serve a lot of static data from large (4 to 15MB) webserver processes.
- On the other hand, while testing you will have no other server interaction to mask or add to your errors.

2.

- This option allows you to tune the two servers individually, for maximum performance.
- However, you need to choose between running the two servers on multiple ports, multiple IPs, etc.,

- and you have the burden of administering more than one server.
 - You have to deal with proxying or fancy site design to keep the two servers in synchronization.
- 3.
- The third option (proxy in http accelerator mode), once correctly configured and tuned, improves the performance of any of the above three options by caching and buffering page results.

3.3 Standalone mod_perl Enabled Apache Server

- The first approach is to implement a straightforward mod_perl server.
- Just take your plain apache server and add mod_perl, like you add any other apache module.
- You continue to run it at the port it was running before.
- You probably want to try this before you proceed to more sophisticated and complex techniques.

The advantages:

- Simplicity. You just follow the installation instructions, configure it, restart the server and you are done.
- No network changes. You do not have to worry about using additional ports as we will see later.
- Speed. You get a very fast server, you see an enormous speedup from the first moment you start to use it.

The disadvantages:

- **Process size**
 - The process size of a mod_perl-enabled Apache server is huge (maybe 4Mb at startup and growing to 10Mb and more, depending on how you use it) compared to the typical plain Apache.
 - Of course if memory sharing is in place, RAM requirements will be smaller.
 - You probably have a few tens of child processes.
 - The additional memory requirements add up in direct relation to the number of child processes.

- Your memory demands are growing by an order of magnitude, but this is the price you pay for the additional performance boost of mod_perl.
- With memory prices so cheap nowadays, the additional cost is low -- especially when you consider the dramatic performance boost mod_perl gives to your services with every 100Mb of RAM you add.
- While you will be happy to have these monster processes serving your scripts with monster speed, you should be very worried about having them serve static objects such as images and html files.
- Each static request served by a mod_perl-enabled server means another large process running, competing for system resources such as memory and CPU cycles.

- The real overhead depends on static objects request rate.
- Remember that if your mod_perl code produces HTML code which includes images, each one will turn into another static object request.
- Having another plain webserver to serve the static objects solves this unpleasant obstacle.
- Having a proxy server as a front end, caching the static objects and freeing the mod_perl processes from this burden is another solution.

- **Serving slow clients**

- Another drawback of this approach is that when serving output to a client with a slow connection,
- the huge mod_perl-enabled server process (with all of its system resources) will be tied up until the response is completely written to the client.
- While it might take a few milliseconds for your script to complete the request,
- there is a chance it will be still busy for some number of seconds or even minutes if the request is from a slow connection client.
- As in the previous drawback, a proxy solution can solve this problem.

- Proxying dynamic content is not going to help much if all the clients are on a fast local net
- (for example, if you are administering an Intranet.)
- On the contrary, it can decrease performance.
- Still, remember that some of your Intranet users might work from home through slow modem links.

- If you are new to mod_perl, this is probably the best way to get yourself started.
- And of course, if your site is serving only mod_perl scripts (close to zero static objects, like images)
- This might be the perfect choice for you!

3.4 One Plain Apache and One mod_perl-enabled Apache Servers

- As I have mentioned before, when running scripts under mod_perl, you will notice that the httpd processes consume a huge amount of virtual memory, from 5Mb to 15Mb and even more.
- That is the price you pay for the enormous speed improvements under mod_perl.
- Using these large processes to serve static objects like images and html documents is overkill.

- A better approach is to run two servers:
- a very light, plain apache server to serve static objects
- and a heavier mod_perl-enabled apache server to serve requests for dynamic (generated) objects.
- From here on, I will refer to these two servers as **httpd_docs** and **httpd_perl**

The advantages:

- **Less memory**
 - The heavy mod_perl processes serve only dynamic requests, which allows the deployment of fewer of these large servers.
- **Better tuning**
 - MaxClients, MaxRequestsPerChild and related parameters can now be optimally tuned for both httpd_docs and httpd_perl servers, something we could not do before.
 - This allows us to fine tune the memory usage and get a better server performance.

- Now we can run many lightweight `httpd_docs` servers and just a few heavy `httpd_perl` servers.

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: using fully qualified URLs



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

The disadvantages:

- **An administration overhead.**
 - **Two sets of files**
 - The need for two different sets of configuration, log and other files.
 - We need a special directory layout to manage these.
 - While some directories can be shared between the two servers (like the `include` directory)
 - most of them should be separated and the configuration files updated to reflect the changes.

- **Two sets of scripts**

- The need for two sets of controlling scripts (startup/shutdown) and watchdogs.

- **Logs merging**

- If you are processing log files, now you probably will have to merge the two separate log files into one before processing them.

- **Serving Slow Clients**

- Just as in the one server approach, we still have the problem of a mod_perl process spending its precious time serving slow clients, when the processing portion of the request was completed a long time ago.

- Deploying a proxy solves this, and will be covered in the next section.
- As with the single server approach, this is not a major disadvantage if you are on a fast network (i.e. Intranet).
- It is likely that you do not want a buffering server in this case.

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 named them `httpd_docs` and `httpd_perl` respectively.

- The two servers coexist at the same IP address by listening to different ports:
- `httpd_docs` listens to port 80
- and `httpd_perl` listens to port 8080
- Now I am going to convince you that you **want** to use a proxy server (in the http accelerator mode).

The advantages:

- **Proxy cache**
 - Allow serving of static objects from the proxy's cache (objects that previously were entirely served by the `httpdocs` server).
- **Less IO**
 - You get less I/O activity reading static objects from the disk (proxy serves the most “popular” objects from RAM - of course you benefit more if you allow the proxy server to consume more RAM).
 - Since you do not wait for the I/O to be completed you are able to serve static objects much faster.

- **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.**
- 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:


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

- It will have to wait for another 12 secs!

- When it could serve 11 more dynamic requests in this time.



12 sec / 1 sec - 1 = 11

- But the generated 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**

- We are going to hide the details of the server's implementation.
- Users will never see ports in the URLs (more on that topic later).
- You can have a few boxes serving the requests, and only one serving as a front end, which spreads the jobs between the servers in a way that you can control.
- You can actually shut down a server, without the user even noticing, because the front end server will dispatch the jobs to other servers.
- (This is called a Load Ballancing and it's a pretty big issue, which will not be discussed here)

- **Security protection**

- For security reasons, using any httpd accelerator (or a proxy in httpd accelerator mode) is essential because you do not let your internal server get directly attacked by arbitrary packets from whomever.
- The httpd accelerator and internal server communicate in expected HTTP requests.
- 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, and while proxies are generally stable,
- You have to make sure to prepare proper startup and shutdown scripts, which are run at boot and reboot as appropriate.
- Also, you might want to set up the crontab to run a watchdog script.

- **Memory Usage**

- Proxy servers can be configured to be light or heavy, the admin must decide what gives the highest performance for his application.
- A proxy server like squid is light in the concept of having only one process serving all requests.
- But it can appear pretty heavy when it loads objects into memory for faster service.

- 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
- Name collisions with Modules and libs
- More package name related issues

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

4.2 Exposing `Apache::Registry` secrets


- `Apache::Registry` is a `mod_cgi` compatible module, 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.

- 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.

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.

- 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. Can 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:
- Firstly, the file */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";
    }
}
```


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();
}
```

- Keep your code in external libraries

- The main script simply calls the main function of the library in use

- Don't worry about nested subroutines effects anymore

- 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]

- 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

4.2.2 *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.

4.3 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.

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.
- 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 (httpd -X).

- To solve this particular */o* modifier problem refer to the *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.
- 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/modules`

- **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
```

4.5 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?

4.5.1 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.


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 *Reloading handlers*

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



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

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


4.6 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 lose 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 `perldoc 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.

4.7 More package name related issues

- If you have the following:




```
PerlHandler Apache::Work::Foo
PerlHandler Apache::Work::Foo::Bar
```

- And you make a request that pulls in `Apache/Work/Foo/Bar.pm` first,
- then the `Apache::Work::Foo` package gets defined,
- so `mod_perl` does not try to pull in `Apache/Work/Foo.pm`

4.8 `__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.

4.9 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.

4.10 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 = $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.

4.11 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` trappes the `__DIE__` signal,
- `o` logs the error message
- `o` 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(); }
```

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

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

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

- There is also an optimization built into `Apache::print()`.
- If any of the arguments to the method are scalar references to strings, they are automatically dereferenced for you.
- This avoids needless copying of large strings when passing them to subroutines.
- For example:



```
$long_string = "A" x 100000000;  
$r->print(\$long_string);
```

4.13 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 harder
- You will have to make certain that all the variables are initialized before they are used.
- 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.

4.14 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.

4.14.1 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.

4.14.2 *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 *perl*sec 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.

4.14.3 *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`.

;o)

