

O'Reilly Open Source Convention
July 23, 2001
San Diego, CA

Tutorial: Getting started with mod_perl

by **Stas Bekman**
<http://stason.org/>
<stas@stason.org>
Senior Software Engineer, eXtropia.com

This talk is available from: <http://stason.org/talks/>

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

(you will find a Table of Contents at the end of the Tutorial)

1 Agenda

1.1 Agenda



I will start the presentation with a very basic introduction into mod_perl, 10 lines installation instructions, a simple configuration and a few code examples. These should help you get your feet wet if you are really new to mod_perl.



Afterwards I'll talk about the machine setups most popular servers use. I'll explain the incentives for having the light Apache and the heavy mod_perl servers serving different kinds of requests. We will see the two major setups, one using squid as a front-end machine, and the other plain Apache server with mod_proxy.



Then we will see some mod_perl peculiarities you should know about, will talk about the modules which allows you to run your CGI scripts unaltered.



Afterwards I'll talk about boosting a performance of web applications working with RDBMS databases under mod_perl. We will see what modules allow us to make the work with database faster.



Finally we will see some performance improvement tips, these should get you programmers produce more efficient code. We will how one should measure performance and DO's and DON'T's to make the code run faster and use less memory.



There are two more sections left for the post-conference reading. The first one is a Perl reference. It's talking about Perl stuff which is very important to know when coding for mod_perl. And the other one includes additional information about mod_perl and related products resources. You should use it to find your way to find the answer to the questions that you might need to get answered, on your way to becoming a mod_perl guru or when you need some general help.

;o)

2 Getting Started Fast

2.1 mod_perl in Four Slides

Each tutorial will concentrate on different aspects of running a mod_perl server and mod_perl programming. In case you don't know how to get started with it, or you think it's a difficult task, these slides will take away any worries you might have had when you came to this tutorial.

In just four slides you will be able to install and configure a mod_perl server. And, of course, to write new code and reuse the existing code under mod_perl.

The four slides (sections) are:

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

2.2 What is mod_perl?

But before we go any further, there is a chance that you don't know what mod_perl is. So let's make a little introduction to mod_perl.

Everybody knows that Perl scripts running under mod_cgi have numerous shortcomings. There are many of them, but code recompilation and Perl interpreter loading overhead at each request is the hardest one to overcome.

Among various attempts to improve on mod_cgi's shortcomings, mod_perl has proved to be one of the better ones and has been widely adopted by CGI developers. According to the <http://perl.apache.org/netcraft/> as of January 2001 about 2 million hosts use mod_perl. Doug MacEachern fathered the core code of this Apache module and licensed it under the Apache Software License.

mod_perl does away with mod_cgi's forking by reusing the existing child processes. In this new model, the child process doesn't exit anymore when it has processed a request. The Perl interpreter is loaded only once, when the process is started. Since the interpreter is persistent throughout the process' lifetime, all code is loaded and compiled only once, the first time it is seen. This makes all subsequent requests run much faster because everything is already loaded and compiled. Response processing is now reduced to running your code. This improves response times by a factor of 10 to 100, depending on the code being executed.

Doug didn't stop here, he went and extended mod_cgi's functionality by adding a complete Perl API to the Apache core. This makes it possible to write a complete Apache module in Perl, a feat that used to require coding in C. From then on mod_perl enabled the programmer to handle all phases of request processing in Perl.

The new Perl API also allows complete server configuration in Perl. This has which made the lives of many server administrators much easier, as they could now benefit from dynamically generating the configuration, freed from hunting for bugs in huge configuration files full of similar directives for virtual hosts and the like.

To provide backwards compatibility for plain CGI scripts that used to be run under `mod_cgi`, while still benefiting from a preloaded perl and modules, a few special handlers were written, each allowing a different level of proximity to pure `mod_perl` functionality. Some take full advantage of `mod_perl`, while others only a partial one.

`mod_perl` embeds a copy of the Perl interpreter into the Apache `httpd` executable, providing complete access to Perl functionality within Apache. This enables a set of `mod_perl`-specific configuration directives, all of which start with the string `Perl*`. Most, but not all, of these directives are used to specify handlers for various phases of the request.

It might occur to you that sticking a large executable (Perl) into another large executable (Apache) makes a very, very large program. `mod_perl` certainly makes `httpd` significantly bigger and you will need more RAM on your production server to be able to run many `mod_perl` processes, but in reality the situation is different. Since `mod_perl` processes requests much faster, the number of the processes needed to handle the same request rate is much lower relative to the `mod_cgi` approach. Generally you need slightly more memory available, and the speed improvements you will see are well worth every megabyte of memory you can add.

Now let's get back to the *All-In-Four-Slides...*

2.3 Installation

Did you know that it takes about 10 minutes to build and install a `mod_perl` enabled Apache server on a computer with a pretty average processor and a decent amount of system memory? It goes like this:

```
% cd /usr/src
% 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!

- Of course you must replace `x.x.x` with the actual version numbers of the `mod_perl` and Apache releases that you use.
- The GNU `tar` utility knows how to uncompress a gzipped tar archive (use the `z` option).

All that's left is to add a few configuration lines to a *httpd.conf*, an Apache configuration file, start the server and enjoy `mod_perl`.

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>
```

This configuration causes every URI starting with */perl* to be handled by the Apache `mod_perl` module. It will use the handler from the Perl module `Apache::Registry`.

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 and return the status to the server.

```
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, you will need to plan your study around the projects you want to implement. Fortunately, there are many resources and lots of help freely available to you.

At the end of this tutorial you will find a chapter describing the available resources and pointers to them.

;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
- Implementations of Proxy Servers

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.

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:



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. We will discuss both below.



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. More on proxies later.

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. (Again -- shared memory keeps the real memory that is being used much smaller :)

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 (aka CGI).

From here on, I will refer to these two servers as **httpd_docs** (vanilla apache) and **httpd_perl** (mod_perl enabled apache).

The advantages:

-

The heavy mod_perl processes serve only dynamic requests, which allows the deployment of fewer of these large servers.

-

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.

An **important** note: When a user browses static pages and the base URL in the **Location** window points to the static server, for example `http://www.nowhere.com/index.html` -- all relative URLs (e.g. ``) are being served by the light plain apache server. But this is not the case with dynamically generated pages. For example when the base URL in the **Location** window points to the dynamic server -- (e.g.

`http://www.nowhere.com:8080/perl/index.pl`) all relative URLs in the dynamically generated HTML will be served by the heavy mod_perl processes. You must use fully qualified URLs and not relative ones! `http://www.nowhere.com/icons/arrow.gif` is a full URL, while `/icons/arrow.gif` is a relative one. Using `<BASE`

`HREF="http://www.nowhere.com/">` in the generated HTML is another way to handle this problem. Also the httpd_perl server could rewrite the requests back to httpd_docs (much slower) and you still need the attention of the heavy servers. This is not an issue if you hide the internal port implementations, so the client sees only one server running on port 80.

The disadvantages:

-

An administration overhead.

-

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

-

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

-

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 (e.g. `http://www.nowhere.com/images/test.gif`) and `httpd_perl` listens to port 8080 (e.g. `http://www.nowhere.com:8080/perl/test.pl`). Note that I did not write `http://www.nowhere.com:80` for the first example, since port 80 is the default port for the http service. Later on, I will be changing the configuration of the `httpd_docs` server to make it listen to port 81.

Now I am going to convince you that you **want** to use a proxy server (in the http accelerator mode). The advantages are:

-

Allow serving of static objects from the proxy's cache (objects that previously were entirely served by the `httpd_docs` server).

-

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.

-

And the extra functionality provided by the http-acceleration mode, which makes the proxy server act 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 than just words. So we are going to show a simple example from the real world.

First let's explain the abbreviation used in the networking world. If someone claims to have a 56 kbps connection -- it means that the connection is of 56 killo-bits per second (~56000 bits/sec). It's not 56 killo-bytes per second, but 7 killo-bytes per second, because 1 byte equals to 8 bits. So don't let the merchants fool you--your modem gives you 7 killo-bytes per second connection at most and not 56 killo-bytes per second as one might think.

So here is the real world example. Let's look at the typical scenario with a user connected to your site with 56Kbps modem. It means that the speed of the user's link is $56/8 = 7\text{KB/s}$ per sec. Let's assume an average generated HTML page to be of 42KB and an average mod_perl script that generates this response in 0.5 second. How many responses this script could produce during the time it took for the output to be delivered to user? A simple calculation reveals pretty scary numbers:

$$42\text{KB} / (0.5\text{s} * 7\text{KB/s}) = 12$$

12 other dynamic requests could be served at the same time, if we could put mod_perl to do only what it's best at: generating responses.

This very simple example shows us that we need only one twelfth the number of children running, which means that we will need only one twelfth of the memory (not quite true because some parts of the code are shared).

But you know that nowadays scripts often return pages which are blown up with javascript code and similar, which can make them 100kb size and the download time will be of the order of... (This calculation is left to you as an exercise :)

Moreover many users like to open many browser windows and do many things at once (download files and browse graphically *heavy* sites). So in the speed of 7KB/sec we have assumed before, may in reality be 5-10 times slower.

●

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.

●

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 are:

-

Of course there are drawbacks. Luckily, these are not functionality drawbacks, but they are more administration hassle. 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.

-

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

As of this writing, two proxy implementations are known to be widely used with mod_perl - **squid** proxy server and **mod_proxy** which is a part of the apache server. Let's compare them.

3.6.1 *The Squid Server*

The Advantages:

-

Caching of static objects. These are served much faster, assuming that your cache size is big enough to keep the most frequently requested objects in the cache.

-

Buffering of dynamic content, by taking the burden of returning the content generated by mod_perl servers to slow clients, thus freeing mod_perl servers from waiting for the slow clients to download the data. Freed servers immediately switch to serve other requests, thus your number of required servers goes down dramatically.

-

Non-linear URL space / server setup. You can use Squid to play some tricks with the URL space and/or domain based virtual server support.

The Disadvantages:

-

Proxying dynamic content is not going to help much if all the clients are on a fast local net. Also, a message on the squid mailing list implied that squid only buffers in 16k chunks so it would not allow a mod_perl to complete immediately if the output is larger.

-

Speed. Squid is not very fast today when compared with the plain file based web servers available. Only if you are using a lot of dynamic features such as mod_perl or similar is there a reason to use Squid, and then only if the application and the server are designed with caching in mind.

-

Memory usage. Squid uses quite a bit of memory.

-

HTTP protocol level. Squid is pretty much a HTTP/1.0 server, which seriously limits the deployment of HTTP/1.1 features.

-

HTTP headers, dates and freshness. The squid server might give out stale pages, confusing downstream/client caches.(You update some documents on the site, but squid will still serve the old ones.)

-

Stability. Compared to plain web servers, Squid is not the most stable.

The pros and cons presented above lead to the idea that you might want to use squid for its dynamic content buffering features, but only if your server serves mostly dynamic requests. So in this situation, when performance is the goal, it is better to have a plain apache server serving static objects, and squid proxying the mod_perl enabled server only.

3.6.2 Apache's mod_proxy

I do not think the difference in speed between apache's **mod_proxy** and **squid** is relevant for most sites, since the real value of what they do is buffering for slow client connections. However, squid runs as a single process and probably consumes fewer system resources.

The trade-off is that mod_rewrite is easy to use if you want to spread parts of the site across different back end servers, while mod_proxy knows how to fix up redirects containing the back-end server's idea of the location. With squid you can run a redirector process to proxy to more than one back end, but there is a problem in fixing redirects in a way that keeps the client's view of both server names and port numbers in all cases.

The difficult case is where:

- **You have DNS aliases that map to the same IP address and**

- **You want the redirect to port 80 and**
- **The server is on a different port and**
- **You want to keep the specific name the browser has already sent, so that it does not change in the client's Location window.**

The Advantages:

-

No additional server is needed. We keep the one plain plus one mod_perl enabled apache servers. All you need is to enable mod_proxy in the httpd_docs server and add a few lines to httpd.conf file.

-

The ProxyPass and ProxyPassReverse directives allow you to hide the internal redirects, so if `http://nowhere.com/modperl/` is actually `http://localhost:81/modperl/`, it will be absolutely transparent to the user. ProxyPass redirects the request to the mod_perl server, and when it gets the response, ProxyPassReverse rewrites the URL back to the original one, e.g:

```
ProxyPass      /modperl/ http://localhost:81/modperl/
ProxyPassReverse /modperl/ http://localhost:81/modperl/
```

-

It does mod_perl output buffering like squid does.

-

It even does caching. You have to produce correct Content-Length, Last-Modified and Expires http headers for it to work. If some of your dynamic content does not change frequently, you can dramatically increase performance by caching it with ProxyPass.

-

ProxyPass happens before the authentication phase, so you do not have to worry about authenticating twice.

-

Apache is able to accelerate secure HTTP requests completely, while also doing accelerated HTTP. With Squid you have to use an external redirection program for that.

-

The latest (apache 1.3.6 and later) Apache proxy accelerated mode is reported to be very stable.

;o)

4 Porting from CGI Scripts and mod_perl Coding Guidelines.

4.1 What we will learn in this chapter

- Exposing Apache::Registry secrets
- Sometimes it Works, Sometimes it Doesn't
- @INC and mod_perl
- Reloading Modules and Required Files
- __END__ and __DATA__ tokens
- Output from system calls
- Terminating requests and processes
- die() and mod_perl
- Global Variables Persistence
- Command line Switches (-w, -T, etc)

4.2 Exposing Apache::Registry secrets

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.

I will use a simple CGI script, that initializes a \$counter to 0, and prints its value to the browser while incrementing it.

```
counter.pl:
-----
use strict;

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

my $counter = 0;

for (1..5) {
    increment_counter();
}

sub increment_counter{
    $counter++;
    print "Counter is equal to $counter !\r\n";
}
```

You would expect to see the output:

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

And that's what you see when you execute this script the first time. But let's reload it a few times... See, suddenly after a few reloads the counter doesn't start its count from 1 any more. We continue to reload and see that it keeps on growing, but not steadily starting almost randomly at 10, 10, 10, 15, 20... Weird...

```
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 saw two anomalies in this very simple script: Unexpected increment of our counter over 5 and inconsistent growth over reloads. Let's investigate this script.

4.2.1 The First Mystery

First let's peek into the `error_log` file. Since we have enabled the warnings what we see is:

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

The *Variable "\$counter" will not stay shared* warning is generated when the script contains a named nested subroutine (a named - as opposed to anonymous - subroutine defined inside another subroutine) that refers to a lexically scoped variable defined outside this nested subroutine. This effect is explained in the Perl Reference section at the end of this handout.

Do you see a nested named subroutine in my script? I don't! What's going on? Maybe it's a bug? But wait, 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.

But since we must debug it when it's being executed by the webserver, a normal debugger won't help, because the debugger has to be invoked from within the webserver. Luckily Doug MacEachern wrote the `Apache::DB` module and we will use this to debug my script. 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 and 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_2epl;
use Apache qw(exit);
sub handler {
    BEGIN {
        $^W = 1;
    };
    $^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";
    }
}
```

The code in the `error.log` wasn't indented. I've indented it for you to stress that the code was wrapped inside the `handler()` subroutine.

What do we learn from this?

Well firstly that 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_2epl`) by replacing all occurrences of `/` with `::` and `.` with `_2e`. That's how `mod_perl` knows what script should be fetched from the cache--each script is just a package with a single subroutine named `handler`.

If we were to add `use diagnostics` to the script we would also see a reference in the error text to an inner (nested) subroutine--`increment_counter` is actually a nested subroutine.

With `mod_perl`, each subroutine in every `Apache::Registry` script is nested inside the `handler` subroutine.

It's important to understand that the *inner subroutine* effect happens only with code that `Apache::Registry` wraps with a declaration of 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 if we move the code from the script into the subroutine *run*, place the subroutines into the *mylib.pl* file, save it in the same directory as the script itself and `require()` it, there will be no problem at all. (Don't forget the `1;` at the end of the library or the `require()` might fail.)

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

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

This solution provides the easiest and the fastest way to solve the nested subroutines problem, since all you have to do is to move the code into a separate file, by first wrapping the initial code into some function that you later will call from the script and keeping the lexically scoped variables that could cause the problem out of this function.

But as a general rule of thumb, 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. Usually I call it `init()` or `run()`. I don't worry about nested subroutine effects anymore (unless I create them myself :).

The section *Remedies for Inner Subroutines* in the Perl Reference chapter discusses other possible workarounds for this problem.

You shouldn't be intimidated by this issue at all, since Perl is your friend. Just 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!

By the way, 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 = CGI->new;
my $name = $q->param('name');
print_response();

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

```

My boss and I checked the program at the development server and it worked OK. So we decided to put it in production. Everything was OK, but my boss decided to keep on checking by submitting variations of his profile. Imagine the surprise when after submitting his name (let's say "The Boss" :), he saw the response "Thank you, Stas Bekman!".

What happened is that I tried the production system as well. I was new to mod_perl stuff, and was so excited with the speed improvement that I didn't notice the nested subroutine problem. It hit me. At first I thought that maybe Apache had started to confuse connections, returning responses from other people's requests. I was wrong of course.

Why didn't we notice this when we were trying the software on our development server? Keep reading and you will understand why.

4.2.2 *The Second Mystery*

Let's return to our original example and proceed with the second mystery we noticed. Why did we see inconsistent results over numerous reloads?

That's very simple. 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. At any given moment one of the children could have served the same script more times than any other, and another may never have run it. That's why we saw the strange behavior.

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. You achieve this by invoking the server with the `-X` parameter (`httpd -X`). Since there are no other servers (children) running, you will see the problem on the second reload.

But before that, let the `error_log` help you detect most of the possible errors--most of the warnings can become errors, so you should make sure to check every warning that is detected by perl, and probably you should write your code in such a way that no warnings appear in the `error_log`. If your `error_log` file is filled up with hundreds of lines on every script invocation, you will have difficulty noticing and locating real problems--and on a production server you'll soon run out of disk space if your site is popular.

Of course none of the warnings will be reported if the warning mechanism is not turned **On**.

4.3 Sometimes it Works, Sometimes it Doesn't

When you start running your scripts under `mod_perl`, you might find yourself in a situation where a script seems to work, but sometimes it screws up. And the more it runs without a restart, the more it screws up. Often the problem is easily detectable and solvable. You have to test your script under a server running in single process mode (`httpd -X`).

Generally the problem is the result of using global variables. Because global variables don't change from one script invocation to another unless you change them, you can find your scripts do strange things.

4.3.1 Regular Expression Memory

Another good example is usage of the `/o` regular expression modifier, which compiles a regular expression once, on its first execution, and never compiles it again. This problem can be difficult to detect, as after restarting the server each request you make will be served by a different child process, and thus the regex pattern for that child will be compiled afresh. Only when you make a request that happens to be served by a child which has already cached the regex will you see the problem. Generally you miss that. When you press reload, you see that it works (with a new, fresh child). Eventually it doesn't, because you get a child that has already cached the regex and won't recompile because of the `/o` modifier.

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.

4.4 @INC and mod_perl

When running 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 add:

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

or

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

Note that this setting will be ignored if you have the `PerlTaintMode` mode turned on.

•

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

When you develop plain CGI scripts, you can just change the code, and rerun the CGI from your browser. Since the script isn't cached in memory, the next time you call it the server starts up a new perl process, which recompiles it from scratch. The effects of any modifications you've applied are immediately present.

The situation is different with `Apache::Registry`, since the whole idea is to get maximum performance from the server. By default, the server won't spend time checking whether any included library modules have been changed. It assumes that they weren't, thus saving a few milliseconds to `stat()` the source file (multiplied by however many modules/libraries you `use()` and/or `require()` in your script.)

The only check that is done is to see whether your main script has been changed. So if you have only scripts which do not `use()` or `require()` other perl modules or packages, there is nothing to worry about. If, however, you are developing a script that includes other modules, the files you `use()` or `require()` aren't checked for modification and you need to do something about that.

So how do we get our modperl-enabled server to recognize changes in library modules? Well, there are a couple of techniques:

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 immediately reloads any files that have been updated on disk.

To enable this module just add two lines to `httpd.conf`.

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

To be sure it really works, turn on debug mode on your development box by adding `PerlSetVar StatINCDebug On` to your config file. You end up with something like this:

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

Be aware that only the modules located in `@INC` are reloaded on change, and you can change `@INC` only before the server has been started (in the startup file).

Nothing you do in your scripts and modules which are pulled in with `require()` after server startup will have any effect on `@INC`.

When you write:

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

`@INC` is changed only for the time the code is being parsed and compiled. When that's done, `@INC` is reset to its original value.

To make sure that you have set `@INC` correctly, configure `/perl-status` location (the `Apache::Status` module), fetch `http://www.example.com/perl-status?inc` and look at the bottom of the page, where the contents of `@INC` will be shown.

Notice the following trap:

While “.” is in `@INC`, perl knows to `require()` files with pathnames given relative to the current (script) directory. After the script has been parsed, the server doesn't remember the path!

So you can end up with a broken entry in `%INC` like this:

```
$INC{bar.pl} eq "bar.pl"
```

If you want `Apache::StatINC` to reload your script--modify `@INC` at server startup, or use a full path in the `require()` call.

4.5.3 Using *Apache::Reload*

Apache::Reload comes as a drop-in replacement for *Apache::StatINC*. It provides extra functionality and better flexibility.

If you want *Apache::Reload* to check all the loaded modules on each request, you just add to *httpd.conf*:

```
PerlInitHandler Apache::Reload
```

If you want to reload only specific modules when these get changed, you have a few ways to do that:

- **Register modules implicitly**

Turn *Off* the *ReloadAll* variable, which is *On* by default

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

and add:

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

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

- **Register Modules Explicitly**

Explicitly specify modules to be reloaded in *httpd.conf*:

```
PerlInitHandler Apache::Reload
PerlSetVar ReloadModules "My::Foo My::Bar Foo::Bar::Test"
```

Note that these are split on whitespace, but the module list **must** be in quotes, otherwise Apache tries to parse the parameter list.

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

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

In the above example all modules starting with *Foo::* and *Bar::* will become registered. This features allows you to assign the whole project modules tree in one pattern.

- **Using a special "touch" file**

You can also set a file that you can `touch(1)` that causes the reloads to be performed. If you set this, and don't `touch(1)` the file, the reloads don't happen (no matter how have you registered the modules to be reloaded).

```
PerlSetVar ReloadTouchFile /tmp/reload_modules
```

Now when you're happy with your changes, simply go to the command line and type:

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

This feature is very convenient in a production server environment, but compared to a full restart, the benefits of preloaded modules memory sharing are lost, since each child will get it's own copy of the reloaded modules.

4.5.3.1 Caveats

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

4.6 __END__ and __DATA__ tokens

Apache::Registry scripts cannot contain __END__ or __DATA__ tokens.

Why? 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_2epl;
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_2epl;
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.

Also, remember that whatever applies to `Apache::Registry` scripts, in most cases applies to `Apache::PerlRun` scripts.

4.7 Output from system calls

The output of `system()`, `exec()`, and `open(PIPE, "|program")` calls will not be sent to the browser unless your Perl was configured with `sfio`.

You can use backticks as a possible workaround:

```

print `command here`;

```

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

4.8 Terminating requests and processes

Perl's `exit()` built-in function cannot be used in `mod_perl` scripts. Calling it causes the `mod_perl` process to exit (which defeats the purpose of using `mod_perl`). The `Apache::exit()` function should be used instead.

You might start your scripts by overriding the `exit()` subroutine (if you use `Apache::exit()` directly, you will have a problem testing the script from the shell, unless you put `use Apache ();` into your code.) I use the following code:

```

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 always be chosen, whether you run the script under `mod_perl`, ordinary CGI or from the shell.

Note that if you run the script under `Apache::Registry`, **The Apache function `exit()` overrides the Perl core built-in function**. While you see `exit()` listed in the `@EXPORT_OK` list of the Apache package, `Apache::Registry` does something you don't see and imports this function for you. This means that if your script is running under the `Apache::Registry` handler you don't have to worry about `exit()`. The same applies to `Apache::PerlRun`.

If you use `CORE::exit()` in scripts running under `mod_perl`, the child will exit, but neither a proper exit nor logging will happen on the way. `CORE::exit()` cuts off the server's legs.

Note that `Apache::exit(Apache::Constants::DONE)` will cause the server to exit gracefully, completing the logging functions and protocol requirements etc. (`Apache::Constants::DONE == -2`, `Apache::Constants::OK == 0`.)

If you need to shut down the child cleanly after the request was completed, use the `$r->child_terminate` method. You can call it anywhere in the code, and not just at the "end". This sets the value of the `MaxRequestsPerChild` configuration variable to 1 and clears the `keepalive` flag. After the request is serviced, the current connection is broken, because of the `keepalive` flag, and the parent tells the child to cleanly quit, because `MaxRequestsPerChild` is smaller than the number of requests served.

In an `Apache::Registry` script you would do:

```
Apache->request->child_terminate;
```

or in `httpd.conf`:

```
PerlFixupHandler "sub { shift->child_terminate }"
```

You would want to use the latter example only if you wanted the child to terminate every time the registered handler is called. Probably this is not what you want.

Even if you don't need to call `child_terminate()` at the end of the request if you want the process to quit afterwards, here is an example of assigning the postprocessing handler. You might do this if you wanted to execute your own code a moment before the process quits.

```
my $r = shift;
$r->post_connection(&exit_child);
sub exit_child{
    # some logic here if needed
    $r->child_terminate;
}
```

The above is the code that is used by the `Apache::SizeLimit` module which terminates processes that grow bigger than a value you choose.

`Apache::GTopLimit` (based on *libgtop* and *GTop.pm*) is a similar module. It does the same thing, plus you can configure it to terminate processes when their shared memory shrinks below some specified size.

4.9 die() and mod_perl

When you write:

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

in a perl script and execute it--the script would `die()` if it is unable to open the file, by aborting the script execution, printing the death reason and quitting the Perl interpreter.

You will hardly find a properly written Perl script that doesn't have at least one `die()` statement in it, if it has to cope with system calls and the like.

A CGI script running under `mod_cgi` exits on its completion. The Perl interpreter exits as well. So it doesn't really matter whether the interpreter quits because the script died by natural death (when the last statement was executed) or was aborted by a `die()` statement.

In `mod_perl` we don't want the interpreter to quit. We already know that when the script completes its chores the interpreter won't quit. There is no reason why it should quit when the script has stopped because of `die()`. As a result calling `die()` won't quit the process.

And this is how it works--when the `die()` gets triggered, it's `mod_perl`'s `$SIG{__DIE__}` handler that logs the error message and calls `Apache::exit()` instead of `CORE::die()`. Thus the script stops, but the process doesn't quit.

Here is an example of such trapping code, although it isn't the real code:

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

4.10 Global Variables Persistence

Since the child process generally doesn't exit before it has serviced several requests, global variables persist inside the same process from request to request. This means that 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 code development harder and 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.

4.11 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 the 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 which allows them to be set/unset in code. Consult the `perlvar` manpage for more details.

4.11.1 Warnings

There are three ways to enable warnings:

- **Locally to a block**

This code turns warnings mode **On** for the scope of the block.

```
{
    local $^W = 1;
    # some code
}
# $^W assumes its previous value here
```

This turns it **Off**:

```
{
    local $^W = 0;
    # some code
}
# $^W assumes its previous value here
```

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.

- **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.

While having warning mode turned **On** is essential for a development server, you should turn it globally **Off** in a production server, since, for example, 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.

4.11.2 *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)

If you have some scripts that won't run under Taint mode, run only the ones that run under `mod_perl` with Taint mode enabled and the rest on another server with Taint mode disabled -- this can be either a `mod_cgi` in the front-end server or another back-end `mod_perl` server. You can use the `mod_rewrite` module and redirect requests based on the file extensions. For example you can use *.tcgi* for the taint-clean scripts, and *cgi* for the rest.

When you have this setup you can start working toward cleaning the rest of the scripts, to make them run under the Taint mode. Just because you have a few dirty scripts doesn't mean that you should jeopardize your whole service.

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.

4.11.3 *Other switches*

Finally, if you still need to to set additional perl startup flags such as `-d` and `-D`, you can use an environment variable `PERL5OPT`. Switches in this variable are treated as if they were on every Perl command line.

Only the `-[DIMUdmw]` switches are allowed.

When the `PerlTaintCheck` variable is turned on, the value of `PERL5OPT` will be ignored.

;o)

5 RDBMS and mod_perl

5.1 Apache::DBI - Initiate a persistent database connection

When people started to use the web, they found that they needed to write web interfaces to their databases. CGI is the most widely used technology for building such interfaces. The main limitation of a CGI script driving a database is that its database connection is not persistent - on every request the CGI script has to re-connect to the database, and when the request is completed the connection is closed.

Apache::DBI was written to remove this limitation. When you use it, you have a database connection which persists for the process' entire life. So when your mod_perl script needs to use a database, Apache::DBI provides a valid connection immediately and your script starts work right away without having to initiate a database connection first.

This is possible only with CGI running under a mod_perl enabled server, since in this model the child process does not quit when the request has been served.

It's almost as straightforward as it sounds; there are just a few things to know about and we will cover them in this section.

5.1.1 Introduction

The DBI module can make use of the Apache::DBI module. When it loads, the DBI module tests if the environment variable `$ENV{MOD_PERL}` is set, and if the Apache::DBI module has already been loaded. If so, the DBI module will forward every `connect()` request to the Apache::DBI module. 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 there is no appropriate database handle or if the `ping()` method fails, Apache::DBI establishes a new connection and stores the handle for later re-use. When the script is run again by a child that is still connected, Apache::DBI just checks the cache of open connections by matching the *host*, *username* and *password* parameters against it. A matching connection is returned if available or a new one is initiated and then returned.

There is no need to delete the `disconnect()` statements from your code. They won't do anything because the Apache::DBI module overloads the `disconnect()` method with an empty one.

When should this module be used and when shouldn't it be used?

You will want to use this module if you are opening several database connections to the server. Apache::DBI will make them persistent per child, so if you have ten children and each opens two different connections (with different `connect()` arguments) you will have in total twenty opened and persistent connections. After the initial `connect()` you will save the connection time for every `connect()` request from your DBI module. This can be a huge benefit for a server with a high volume of database traffic.

You must **not** use this module if you are opening a special connection for each of your users. Each connection will stay persistent and in a short time the number of connections will be so big that your machine will scream in agony and die.

If you want to use `Apache::DBI` but you have both situations on one machine, at the time of writing the only solution is to run two `Apache/mod_perl` servers, one which uses `Apache::DBI` and one which does not.

5.1.2 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 before the `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`.

5.1.3 Preopening DBI connections

If you want to make sure that a connection will already be opened when your script is first executed after a server restart, then you should use the `connect_on_init()` method in the startup file to preload every connection you are going to use. For example:

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

As noted above, use this method only if you want all of apache to be able to connect to the database server as one user (or as a very few users), i.e. if your `user(s)` can effectively share the connection. Do **not** use this method if you want for example one unique connection per user.

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!

5.1.4 Debugging Apache::DBI

If you are not sure if this module is working as advertised, you should enable Debug mode in the startup script by:

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

Starting with `ApacheDBI-0.84`, setting `$Apache::DBI::DEBUG = 1` will produce only minimal output. For a full trace you should set `$Apache::DBI::DEBUG = 2`.

After setting the `DEBUG` level you will see entries in the `error_log` both when `Apache::DBI` initializes a connection and when it returns one from its cache. Use the following command to view the log in real time (your `error_log` might be located at a different path, it is set in the Apache configuration files):

```
tail -f /usr/local/apache/logs/error_log
```

I use `alias` (in `tcsh`) so I do not have to remember the path:

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

5.1.5 Opening connections with different parameters

When it receives a connection request, before it decides to use an existing cached connection, `Apache::DBI` insists that the new connection be opened in exactly the same way as the cached connection. If I have one script that sets `LongReadLen` and one that 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.

5.1.6 Caching `prepare()` Statements

You can also benefit from persistent connections by replacing `prepare()` with `prepare_cached()`. That way you will always be sure that you have a good statement handle and you will get some caching benefit. The downside is that you are going to pay for `DBI` to parse your SQL and do a cache lookup every time you call `prepare_cached()`.

Be warned that some databases (e.g. PostgreSQL and Sybase) don't support caches of prepared plans. With Sybase you could open multiple connections to achieve the same result, although this is at the risk of getting deadlocks depending on what you are trying to do!

;o)

6 Performance Tuning

6.1 What we will learn in this chapter

- The Big Picture
- Essential Tools
- Choosing MaxClients
- KeepAlive
- Limiting the Size of the Processes
- Sharing Memory
- How Shared My Memory Is
- Preload Perl modules at server startup
- Preload Registry Scripts
- Modules Initialization
- Keeping the Shared Memory Limit
- Limiting the Resources Used by httpd Children
- Upload/Download of Big Files
- Global vs Fully Qualified Variables
- Forking or Executing subprocesses from mod_perl
- Sending plain HTML as a compressed output

6.2 The Big Picture

To make the user's Web browsing experience as painless as possible, every effort must be made to wring the last drop of performance from the server. There are many factors which affect Web site usability, but speed is one of the most important. This applies to any webserver, not just Apache, and it is very important for you to understand it.

How do we measure the speed of a server? Since the user (and not the computer) is the one that interacts with the Web site, one good speed measurement is the time elapsed between the moment when she clicks on a link or presses a *Submit* button to the moment when the resulting page is rendered complete.

The requests and replies are broken into packets. A request may be made up of several packets, a reply may be many thousands. Each packet has to make its own way from one machine to another, perhaps passing through many interconnection nodes. We must measure the time starting from when the first packet of the request leaves our user's machine to when the last packet of the reply arrives back there.

A webserver is only one of the elements the packets see along their way. If we follow them from browser to server and back again, they may travel by different routes through many different entities. Before they are processed by your server the packets might have to go through proxy (accelerator) servers and if the request contains more than one packet they will all have to wait for the last one so that the full request message can be reassembled at the server. Then the whole process is repeated in reverse.

You could work hard to fine tune your webserver's performance, but a slow Network Interface Card (NIC) or a slow network connection from your server might defeat it all. That's why it's important to think about the Big Picture and to be aware of possible bottlenecks between the server and the Web. Of course there is little that you can do if the user has a slow connection.

You might tune your scripts and webserver to process incoming requests ultra fast, so you will need only a small number of working servers, but you might find that the server processes are all busy waiting for slow clients to accept their responses. You will see more examples in this chapter.

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

6.3 Essential Tools

In order to improve performance we need measurement tools. We use benchmarking for this purpose. We can benchmark the code and we can benchmark the response time which in addition to the code execution measures the request arrival and response delivery time amongst other things.

6.3.1 *Benchmarking Perl Code*

If you are going to write your own benchmarking utility, use the Benchmark module and the Time::HiRes module where you need better time precision (<10msec).

An example of the Benchmark.pm module usage:

```
benchmark.pl
-----
use Benchmark;

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

% perl benchmark.pl
timethis 1000: 25 wallclock secs (24.93 usr + 0.00 sys = 24.93 CPU)
```

An example of the Time::HiRes module usage:

```

hi-res.pl
-----
use Time::HiRes qw(gettimeofday tv_interval);
sub sub_that_takes_a_teeny_bit_of_time{1+1;};
my $start_time = [ gettimeofday ];
&sub_that_takes_a_teeny_bit_of_time();
my $end_time = [ gettimeofday ];
my $elapsed = tv_interval($start_time,$end_time);
print "The sub took $elapsed seconds.\n"

```

```

% perl hi-res.pl
The sub took 0.000262 seconds.

```

6.3.2 Benchmarking Response Times

To measure response times all we need is a client that will generate parallel requests, process the responses and print the results of the test. You can use either an existing tool that performs this task or you can develop your own.

6.3.2.1 ApacheBench

From existing tools you can try ApacheBench (ab) that comes bundled with Apache source distribution. It is designed to give you an idea of the performance that your current Apache installation can give. In particular, it shows you how many requests per second your Apache server is capable of serving.

Let's try it. We will simulate 10 users concurrently requesting a very light script at `www.example.com:81/test/test.pl`. Each simulated user makes 10 requests.

```

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

```

The results are:

```

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

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

Connection Times (ms)
      min      avg      max
Connect:    0       0       3
Processing: 13      67      71
Total:      13      67      74

```

6.3.2.2 httpperf

httpperf is a utility written by David Mosberger. Just like ApacheBench, it measures the performance of the webserver.

A sample command line is shown below:

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

This command causes httpperf to use the web server on the host with IP name hostname, running at port 80. The web page being retrieved is */test.html* and, in this simple test, the same page is retrieved repeatedly. The rate at which requests are issued is 150 per second. The test involves initiating a total of 27,000 TCP connections and on each connection one HTTP call is performed. A call consists of sending a request and receiving a reply.

The timeout option defines the number of seconds that the client is willing to wait to hear back from the server. If this timeout expires, the tool considers the corresponding call to have failed. Note that with a total of 27,000 connections and a rate of 150 per second, the total test duration will be approximately 180 seconds (27,000/150), independent of what load the server can actually sustain. Here is a result that one might get:

```
Total: connections 27000 requests 26701 replies 26701 test-duration 179.996 s  
  
Connection rate: 150.0 conn/s (6.7 ms/conn, <=47 concurrent connections)  
Connection time [ms]: min 1.1 avg 5.0 max 315.0 median 2.5 stddev 13.0  
Connection time [ms]: connect 0.3  
  
Request rate: 148.3 req/s (6.7 ms/req)  
Request size [B]: 72.0  
  
Reply rate [replies/s]: min 139.8 avg 148.3 max 150.3 stddev 2.7 (36 samples)  
Reply time [ms]: response 4.6 transfer 0.0  
Reply size [B]: header 222.0 content 1024.0 footer 0.0 (total 1246.0)  
Reply status: 1xx=0 2xx=26701 3xx=0 4xx=0 5xx=0  
  
CPU time [s]: user 55.31 system 124.41 (user 30.7% system 69.1% total 99.8%)  
Net I/O: 190.9 KB/s (1.6*10^6 bps)  
  
Errors: total 299 client-timo 299 socket-timo 0 connrefused 0 connreset 0  
Errors: fd-unavail 0 addrunavail 0 ftab-full 0 other 0
```

6.3.3 Using LWP::Parallel::UserAgent

You can use LWP::Parallel::UserAgent to write your own bechmarking utility.

This is another crashme suite originally written by Michael Schilli (and used to be located at <http://www.linux-magazin.de/ausgabe.1998.08/Pounder/pounder.html>, but it's has gone). I made a few modifications, mostly adding my() operators. I also allowed it to accept more than one url to test, since sometimes you want to test more than one script.

The tool provides the same results as **ab** above but it also allows you to set the timeout value, so requests will fail if not served within the time out period. You also get values for **Latency** (seconds per request) and **Throughput** (requests per second). It can do a complete simulation of your favorite

Netscape browser :) and give you a better picture.

I have noticed while running these two benchmarking suites, that **ab** gave me results from two and a half to three times better. Both suites were run on the same machine, with the same load and the same parameters, but the implementations were different.

Sample output:

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

And the code:

```

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

use LWP::Parallel::UserAgent;
use Time::HiRes qw(gettimeofday tv_interval);
use strict;

###
# Configuration
###

my $nof_parallel_connections = 10;
my $nof_requests_total = 100;
my $timeout = 10;
my @urls = (
    'http://www.example.com:81/perl/faq_manager/faq_manager.pl',
    'http://www.example.com:81/perl/access/access.cgi',
);

#####
# Derived Class for latency timing
#####

package MyParallelAgent;
@MyParallelAgent::ISA = qw(LWP::Parallel::UserAgent);
use strict;

###
# Is called when connection is opened
###
sub on_connect {
    my ($self, $request, $response, $entry) = @_;
    $self->{__start_times}->{$entry} = [Time::HiRes::gettimeofday];
}

###
# Are called when connection is closed
###
sub on_return {
    my ($self, $request, $response, $entry) = @_;
    my $start = $self->{__start_times}->{$entry};
    $self->{__latency_total} += Time::HiRes::tv_interval($start);
}

sub on_failure {
    on_return(@_); # Same procedure
}

###
# Access function for new instance var
###
sub get_latency_total {
    return shift->{__latency_total};
}

```

```
#####
package main;
#####
###
# Init parallel user agent
###
my $ua = MyParallelAgent->new();
$ua->agent("pounder/1.0");
$ua->max_req($nof_parallel_connections);
$ua->redirect(0);    # No redirects

###
# Register all requests
###
foreach (1..$nof_requests_total) {
    foreach my $url (@urls) {
        my $request = HTTP::Request->new('GET', $url);
        $ua->register($request);
    }
}

###
# Launch processes and check time
###
my $start_time = [gettimeofday];
my $results = $ua->wait($timeout);
my $total_time = tv_interval($start_time);

###
# Requests all done, check results
###

my $succeeded      = 0;
my %errors = ();

foreach my $entry (values %$results) {
    my $response = $entry->response();
    if($response->is_success()) {
        $succeeded++; # Another satisfied customer
    } else {
        # Error, save the message
        $response->message("TIMEOUT") unless $response->code();
        $errors{$response->message}++;
    }
}
}
```


You will be wondering what will happen to your server if there are more concurrent users than `MaxClients` at any time. This situation is accompanied by the following warning message in the `error_log`:

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

There is no problem -- any connection attempts over the `MaxClients` limit will normally be queued, up to a number based on the `ListenBacklog` directive. When a child process is freed at the end of a different request, the connection will be served.

It **is an error** because clients are being put in the queue rather than getting served immediately, despite the fact that they do not get an error response. The error can be allowed to persist to balance available system resources and response time, but sooner or later you will need to get more RAM so you can start more child processes. The best approach is to try not to have this condition reached at all, and if you reach it often you should start to worry about it.

It's important to understand how much real memory a child occupies. Your children can share memory between them when the OS supports that. You must take action to allow the sharing to happen. If you do this, the chances are that your `MaxClients` can be even higher. But it seems that it's not so simple to calculate the absolute number. If you come up with solution please let us know! If the shared memory was of the same size throughout the child's life, we could derive a much better formula:

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

which is:

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

Let's roll some calculations:

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

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

With no sharing in place

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

With sharing in place you can have 64% more servers without buying more RAM.

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

```
Total_RAM           = 500Mb
Max_Process_Size     = 10Mb
Shared_RAM_per_Child = 8Mb
```

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

392% more servers! Now you can feel the importance of having as much shared memory as possible.

6.5 KeepAlive

If your mod_perl server's *httpd.conf* includes the following directives:

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

you have a real performance penalty, since after completing each request processing, the process will wait for `KeepAliveTimeout` seconds before closing the connection and thus not serving other requests at this time. With this configuration you will need many more concurrent processes on a server with high traffic.

If you use some server status reporting tools, you will see the process in *K* status when it's in `KeepAlive` status.

The chances are that you don't want this feature enabled. Set it Off with:

```
KeepAlive Off
```

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

You might want to consider enabling this option if the client's browser needs to request more than one object from your server for a single HTML page. If this is the situation then by setting `KeepAlive Off` for each page you save the HTTP connection overhead for all requests but the first one.

For example if you have a page with 10 ad banners, which is not uncommon today, your server will work more effectively if a single process serves them all during a single connection. However, your client will see a slightly slower response, since banners will be brought one at a time and not concurrently as is the case if each `IMG` tag opens a separate connection.

Since keepalive connections will not incur the additional three-way TCP handshake, turning it off will be kinder to the network.

SSL connections benefit the most from KeepAlive in case you didn't configure the server to cache session ids.

You have probably followed the advice to send all the requests for static objects to a plain Apache server. Since most pages include more than one unique static image, you should keep the default KeepAlive setting of the non-mod_perl server, i.e. keep it On. It will probably be a good idea also to reduce the timeout a little.

One option would be for the proxy/accelerator to keep the connection open to the client but make individual connections to the server, read the response, buffer it for sending to the client and close the server connection. Obviously you would make new connections to the server as required by the client's requests.

Also you should know that KeepAlive requests only work with responses that contain a Content-Length header. To send this header do:

```
$r->header_out('Content-Length', $length);
```

6.6 Be carefull with symbolic links

As you know Apache::Registry caches the scripts based on their URI. If you have the same script that can be reached by different URIs, which is possible if you have used symbolic links, you will get the same script cached twice!

For example:

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

Now the script can be reached through the both URIs /news/news.pl and /news.pl. It doesn't really matter until you advertise the two URIs, and users reach the same script from both of them.

To detect this, use the /perl-status (Apache::Status) handler to see all the compiled scripts and their packages. In our example, when requesting: http://localhost/perl-status?rgysubs you would see:

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

after the both URIs have been requested from the same child process that happened to serve your request. To make the debugging easier see run the server in single mode.

6.7 Limiting the Size of the Processes

Apache::SizeLimit allows you to kill off Apache httpd processes if they grow too large.

Configuration:

In your *startup.pl*:

```
use Apache::SizeLimit;
$Apache::SizeLimit::MAX_PROCESS_SIZE = 10000;
# in KB, so this is 10MB
```

In your *httpd.conf*:

```
PerlFixupHandler Apache::SizeLimit
```

See `perldoc Apache::SizeLimit` for more details.

By using this module, you should be able to avoid using the Apache configuration directive `MaxRequestsPerChild`, although for some folks, using both in combination does the job.

6.8 Sharing Memory

A very important point is the sharing of memory. If your OS supports this (and most sane systems do), you might save more memory by sharing it between child processes. This is only possible when you preload code at server startup. However during a child process' life, its memory pages becomes unshared and there is no way we can control perl to make it allocate memory so (dynamic) variables land on different memory pages than constants, that's why the **copy-on-write** effect (will explain in a moment) will hit almost at random. If you are pre-loading many modules you might be able to balance the memory that stays shared against the time for an occasional fork by tuning the `MaxRequestsPerChild` to a point where you restart before too much becomes unshared. In this case the `MaxRequestsPerChild` is very specific to your scenario. You should do some measurements and you might see if this really makes a difference and what a reasonable number might be. Each time a child reaches this upper limit and restarts it should release the unshared copies and the new child will inherit pages that are shared until it scribbles on them.

It is very important to understand that your goal is not to have `MaxRequestsPerChild` to be 10000. Having a child serving 300 requests on precompiled code is already a huge speedup, so if it is 100 or 10000 it does not really matter if it saves you the RAM by sharing. Do not forget that if you preload most of your code at the server startup, the fork to spawn a new child will be very very fast, because it inherits most of the preloaded code and the perl interpreter from the parent process. But then, during the work of the child, its memory pages (which aren't really its yet, it uses the parent's pages) are getting dirty (originally inherited and shared variables are getting updated/modified) and the **copy-on-write** happens, which reduces the number of shared memory pages - thus enlarging the memory demands. Killing the child and respawning a new one, allows to get the pristine shared memory from the parent process again.

The conclusion is that `MaxRequestsPerChild` should not be too big, otherwise you loose the benefits of the memory sharing.

6.9 How Shared My Memory Is

You've probably noticed that the word shared is being repeated many times in many things related to `mod_perl`. Indeed, shared memory might save you a lot of money, since with sharing in place you can run many more servers than without it.

How much shared memory do you have? You can see it by either using the memory utils that comes with your system or you can deploy GTop module:

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

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

When you watch the output of the `top` utility, don't confuse **RSS** (or **RES**) column with **SHARE** column -- **RES** is a RESident memory, which is a size of pages currently swapped in.

6.10 Keeping the Shared Memory Limit

Apache::GTopLimit module allows you to kill off Apache httpd processes if they grow too large (just like Apache::SizeLimit) or have too little of shared memory.

Configuration:

In your *startup.pl*:

```
use Apache::GTopLimit;

# Control the life based on memory size
# in KB, so this is 10MB
$Apache::GTopLimit::MAX_PROCESS_SIZE = 10000;

# Control the life based on Shared memory size
# in KB, so this is 4MB
$Apache::GTopLimit::MIN_PROCESS_SHARED_SIZE = 4000;

# watch what happens
$Apache::GTopLimit::DEBUG = 1;
```

In your *httpd.conf*:

```
PerlFixupHandler Apache::GTopLimit
```

6.11 Preload Perl modules at server startup

Use the `PerlRequire` and `PerlModule` directives to load commonly used modules such as `CGI.pm`, `DBI` and etc., when the server is started. On most systems, server children will be able to share the code space used by these modules. Just add the following directives into *httpd.conf*:

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

But even a better approach is to create a separate startup file (where you code in plain perl) and put there things like:

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

Then you `require()` this startup file with help of `PerlRequire` directive from `httpd.conf`, by placing it before the rest of the `mod_perl` configuration directives:

```
PerlRequire /path/to/start-up.pl
```

`CGI.pm` is a special case. Ordinarily `CGI.pm` autoloads most of its functions on an as-needed basis. This speeds up the loading time by deferring the compilation phase. However, if you are using `mod_perl`, `FastCGI` or another system that uses a persistent Perl interpreter, you will want to precompile the methods at initialization time. To accomplish this, call the package function `compile()` like this:

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

The arguments to `compile()` are a list of method names or sets, and are identical to those accepted by the `use()` and `import()` operators. Note that in most cases you will want to replace `:all` with tag names you really use in your code, since generally only a subset of subs is actually being used.

6.11.0.0.1 Modules Initialization

The first example is the DBI module. As you know DBI works with many database drivers falling into the `DBD::` category, e.g. `DBD::mysql`. It's not enough to preload DBI, you should initialize DBI with `driver(s)` that you are going to use (usually a single driver is used), if you want to minimize memory use after forking the child processes. Note that you want to do this under `mod_perl` and other environments where the shared memory is very important. Otherwise you shouldn't initialize drivers.

You probably know already that under `mod_perl` you should use the `Apache::DBI` module to get the connection persistence, unless you open a separate connection for each user--in this case you should not use this module. `Apache::DBI` automatically loads DBI and overrides some of its methods, so you should continue coding like there is only a DBI module.

Just as with modules preloading our goal is to find the startup environment that will lead to the smallest "*difference*" between the shared and normal memory reported, therefore a smaller total memory usage.

And again in order to have an easy measurement we will use only one child process, therefore we will use this setting in `httpd.conf`:

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

We always preload these modules:

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

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

option 1

Leave the file unmodified.

option 2

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

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

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

option 3

Preload MySQL driver module:

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

option 4

Tell `Apache::DBI` to connect to the database when the child process starts (`ChildInitHandler`), no driver is preload before the child gets spawned!

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

option 5

Options 2 and 4: using `connect_on_init()` and `install_driver()`.

Here is the `Apache::Registry` test script that we have used:

```
preload_dbi.pl
-----
use strict;
use GTop ();
use DBI ();

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

my $r = shift;
$r->send_http_header('text/plain');

my $do_sql = "show tables";
my $sth = $dbh->prepare($do_sql);
$sth->execute();
my @data = ();
while (my @row = $sth->fetchrow_array){
    push @data, @row;
}
print "Data: @data\n";
$dbh->disconnect(); # NOP under Apache::DBI

my $proc_mem = GTop->new->proc_mem($$);
my $size = $proc_mem->size;
my $share = $proc_mem->share;
my $diff = $size - $share;
printf "%8s %8s %8s\n", qw(Size Shared Diff);
printf "%8d %8d %8d (bytes)\n", $size, $share, $diff;
```

The script opens a connection to the database `'test'` and issues a query to learn what tables the database has. When the data is collected and printed the connection would be closed in the regular case, but `Apache::DBI` overrides it with empty method. When the data is processed a familiar to you already code to print the memory usage follows.

The server was restarted before each new test.

So here are the results of the five tests that were conducted, sorted by the *Diff* column:

- 1.

After the first request:

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

2.

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

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

Now what do we conclude from looking at these numbers. First we see that only after a second reload we get the final memory footprint for a specific request in question (if you pass different arguments the memory usage might and will be different).

But both tables show the same pattern of memory usage. We can clearly see that the real winner is the *startup.pl* file's version where the MySQL driver was installed (2). Since we want to have a connection ready for the first request made to the freshly spawned child process, we generally use the version (5) which uses somewhat more memory, but has almost the same number of shared memory pages. The version (3) only preloads the driver which results in smaller shared memory. The last two versions having nothing initialized (1) and having only the `connect_on_init()` method used (4). The former is a little bit better than the latter, but both significantly worse than the first two versions.

To remind you why do we look for the smallest value in the column *diff*, recall the real memory usage formula:

```
RAM_dedicated_to_mod_perl = diff * number_of_processes
                           + the_processes_with_largest_shared_memory
```

Notice that the smaller the *diff* is, the bigger the number of processes you can have using the same amount of RAM. Therefore every 100K difference counts, when you multiply it by the number of processes. If we take the number from the version (2) vs. (4) and assume that we have 256M of memory dedicated to `mod_perl` processes we will get the following numbers using the formula derived from the above formula:

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

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

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

So you can tell the difference (17% more child processes in the first version).

6.12 Preload Registry Scripts

`Apache::RegistryLoader` compiles `Apache::Registry` scripts at server startup. It can be a good idea to preload the scripts you are going to use as well. So the code will be shared among the children.

Here is an example of the use of this technique. This code is included in a `PerlRequire`'d file, and walks the directory tree under which all registry scripts are installed. For each `.pl` file encountered, it calls the `Apache::RegistryLoader::handler()` method to preload the script in the parent server (before pre-forking the child processes):

```
use File::Find 'finddepth';
use Apache::RegistryLoader ();
{
    my $perl_dir = "perl/";
    my $rl = Apache::RegistryLoader->new;
    finddepth(sub {
        return unless /\.pl$/;
        my $url = "/" . $File::Find::dir . $_;
        print "pre-loading $url\n";

        my $status = $rl->handler($url);
        unless($status == 200) {
            warn "pre-load of '$url' failed, status=$status\n";
        }
    }, $perl_dir);
}
```

Note that we didn't use the second argument to `handler()` here, as module's manpage suggests. To make the loader smarter about the `uri->filename` translation, you might need to provide a `trans()` function to translate the `uri` to `filename`. `URI` to `filename` translation normally doesn't happen until `HTTP` request time, so the module is forced to roll its own translation. If `filename` is omitted and a `trans()` routine was not defined, the loader will try using the `URI` relative to **ServerRoot**.

6.13 Upload/Download of Big Files

You don't want to tie up your precious `mod_perl` backend server children doing something as long and dumb as transferring a file. The user won't really see any important performance benefits from `mod_perl` anyway, since the upload may take up to several minutes, and the overhead saved by `mod_perl` is typically under one second.

If some particular script's main functionality is the uploading or downloading of big files, you probably want it to be executed on a plain `apache` server under `mod_cgi`.

This of course assumes that the script requires none of the functionality of the `mod_perl` server, such as custom authentication handlers.

6.14 Global vs Fully Qualified Variables

It's always a good idea to stay away from global variables when possible. Some variables must be global so Perl can see them, such as a module's `@ISA` or `$VERSION` variables (or fully qualified `@MyModule::ISA`). In common practice, a combination of `strict` and `vars` pragmas keeps modules clean and reduces a bit of noise. However, `vars` pragma also creates aliases as the `Exporter` does, which eat up more memory. When possible, try to use fully qualified names instead of use `vars`. Example:

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

vs.

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

6.15 Forking or Executing subprocesses from `mod_perl`

Generally you should not fork from your `mod_perl` scripts, since when you do -- you are forking the entire apache web server, lock, stock and barrel. Not only is your perl code being duplicated, but so is `mod_ssl`, `mod_rewrite`, `mod_log`, `mod_proxy`, `mod_spelling` or whatever modules you have used in your server, all the core routines and so on.

A much wiser approach would be to spawn a sub-process, hand it the information it needs to do the task, and have it detach (`close STD* + setsid()`). This is wise only if the parent who spawns this process, immediately continues, you do not wait for the sub-process to complete. This approach is suitable for a situation when you want to trigger a long time taking process through the web interface, like processing some data, sending email to thousands of subscribed users and etc. Otherwise, you should convert the code into a module, and use its functions or methods to call from CGI script.

Just making a `system()` call defeats the whole idea behind `mod_perl`, perl interpreter and modules should be loaded again for this external program to run.

Basically, you would do:

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

and in `program.pl` :

```
use POSIX qw(setsid);
@params=FreezeThaw::thaw(shift @ARGV);
# check that @params is ok
close STDIN;
close STDOUT;
close STDERR;
# you might need to reopen the STDERR
# open STDERR, ">/dev/null";
setsid(); # to detach
```

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

This has obvious problems. Not the least of which is that `@params` must not be bigger than whatever your architecture’s limit is (could depend on your shell).

Also, the communication is only one way.

However, you might want be trying to do the “wrong thing”. If what you want is to send information to the browser and then do some post-processing, look into `PerlCleanupHandler`.

If you are interested in more deep level details, this is what actually happens when you fork and make a system call, like

```
system("echo Hi"),CORE::exit(0) unless fork();
```

which is might be more familiar in this form:

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

What happens is that `fork()` gives you 2 execution paths and the child gets virtual memory sharing a copy of the program text (read only) and sharing a copy of the data space copy-on-write (remember why you pre-load modules in `mod_perl?`). In the above code a parent will immediately continue with the code that comes up after the fork, while the forked process will execute `system("echo Hi")` and then terminate itself.

Notice that I use `CORE::exit` and not `exit` which would be automatically overridden by `Apache::exit` if used in conjunction with `Apache::Registry` and friends.

The only work is setting up the page tables for the virtual memory and the second process goes on its separate way.

Next, Perl will find `/bin/echo` along the search path, and invoke it directly. Perl `system()` is **not** `system(3)` [C-library]. Only when the command has shell meta-chars does Perl invoke a real shell. That’s a **very** nice optimization.

Only if you do:

```
system "sh -c 'echo foo'"
```

OS actually parses your command with a shell so you `exec()` a copy of `/bin/sh`, but since one is almost certainly already running somewhere, the system will notice that (via the disk inode reference) and replace your virtual memory page table with one pointed at the already-loaded program code plus your own data space. Then the shell parses the passed command.

Since it is `echo`, it will execute it as a built-in in the latter example or a `/bin/echo` in the former and be done, but this is only an example. You aren't calling `system("echo Hi")` in your `mod_perl` scripts, right? Since most other real things (heavy programs executed as a subprocess) would involve repeating the process to load the specified command or script (it might involve some actual demand paging from the program file if you execute new code).

The only place you see real overhead from this scheme is when the parent process is huge (unfortunately like `mod_perl`...) and the page table becomes large as a side effect. The whole point of `mod_perl` is to avoid having to `fork()` / `exec()` something on every hit, though. Perl can do just about anything by itself. However, you probably won't get in trouble until you hit about 30 forks/sec on a so-so pentium.

Now let's get to the gory details of forking.

6.15.1 *Freeing the Parent Process*

In the child code you must also close all the pipes to the connection socket that were opened by the parent process (i.e. `STDIN` and `STDOUT`) and inherited by the child, so the parent will be able to complete the request and free itself for serving other requests. If you need the `STDIN` and/or `STDOUT` streams you should re-open them. You may need to close or re-open the `STDERR` filehandle. It's opened to append to the `error_log` file as inherited from its parent, so chances are that you will want to leave it untouched.

Under `mod_perl`, the spawned process also inherits the file descriptor that's tied to the socket through which all the communications between the server and the client happen. Therefore we need to free this stream in the forked process. If we don't do that, the server cannot be restarted while the spawned process is still running. If an attempt is made to restart the server you will get the following error:

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

`Apache::SubProcess` comes to help and provides a method `cleanup_for_exec()` which takes care of closing this file descriptor.

So the simplest way to freeing the parent process is to close all three `STD*` streams if we don't need them and untie the Apache socket. In addition you may want to change process' current directory to / so the forked process won't keep the mounted partition busy, if this is to be unmounted at a later time. To summarize all this issues, here is an example of the fork that takes care of freeing the parent process.

```

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

    # some code comes here

    CORE::exit(0);
}
# possibly more code here usually run by the parent

```

Of course between the freeing the parent code and child process termination the real code is to be placed.

6.15.2 Detaching the Forked Process

Now what happens if the forked process is running and we decided that we need to restart the web-server? This forked process will be aborted, since when parent process will die during the restart it'll kill its child processes as well. In order to avoid this we need to detach the process from its parent session, by opening a new session with help of `setsid()` system call, provided by the POSIX module:

```

use POSIX 'setsid';

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

```

Now the spawned child process has a life of its own, and it doesn't depend on the parent anymore.

6.15.3 Avoiding Zombie Processes

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

Zombie, is a process that doesn't have a father. When the child quits, it reports the termination to his parent. If no one `wait()`s to collect the exit status of the child, it gets "confused" and becomes a ghost process, that can be seen, but not killed. It will be killed only when you stop the `httpd` process that spawned it! (generally `top()`/`ps()` utilities display these processes with `<defunc>` tag, and you will see an increment of the zombies counter reported when doing `top()`.) These zombie processes can take up system resources and are generally undesirable.

So the proper fork is:

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

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

But in most cases the only reason you would want to fork is when you need to spawn a process that would take a lot of time to complete. So if the server child that spawns this process has to wait for it to finish, you gained nothing. You cannot neither wait for its completion, nor continue because you will get yet another zombie process.

The simplest solution is to ignore your dead children (this doesn't work everywhere, however).

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

When you set CHLD signal handler to IGNORE, all the processes will be collected by the `init` process and prevent from them to become zombies.

Note, that you cannot localize this setting with `local()`. If you do, it wouldn't take the desired effect.

The other thing that you must do -- is to close all the pipes to the connection socket that were opened by the parent process (a STDIN and a STDOUT) and inherited by the child, so the parent will be able to complete the request and free itself for serving other requests. You may need to close and reopen a STDERR filehandler (It's opened to append to the `error_log` file as inherited by parent, so chances are that you want it to leave untouched).

So now the code would look like:

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

$SIG{CHLD} = IGNORE;

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


Another more portable, but slightly more expensive solution is to use a double fork approach.

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

defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
    waitpid($kid,0);
} else {
    defined (my $grandkid = fork) or die "Kid cannot fork: $!\n";
    if ($grandkid) {
        CORE::exit(0);
    } else {
        # code here
        close STDIN;
        close STDOUT;
        close STDERR;
        # do something long lasting
        CORE::exit(0);
    }
}
```

Grandkid becomes a "*child of init*" (parent process ID is 1).

Note that the last two solutions do allow you to know the exit status of the process, but in our case we don't want to.

One more solution is to use a different *SIGCHLD* handler:

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

Which is useful when you `fork()` more than once process. The handler could call `wait()` as well, but for a variety of reasons involving the handling of stopped processes and the rare event in which two children exit at nearly the same moment, the best technique is to call `waitpid()` in a tight loop with a first argument of `-1` and a second argument of `WNOHANG`. Together these arguments tell `waitpid()` to reap the next child that's available, and prevent the call from blocking if there happens to be no child ready from reaping. The handler will loop until `waitpid()` returns a negative number or zero, indicating that no more reapeable children remain.

You will probably want to open your own log file in the spawned process and log some info so you know what have happened there. At least while debugging your code.

6.15.4 A Complete Fork Example

Now let's put all the bits of code together and show a well written fork code that solves all the problems discussed so far. We will use an `<Apache::Registry>` script for this purpose:

```

proper_fork1.pl
-----
use strict;
use POSIX 'setsid';
use Apache::SubProcess;

my $r = shift;
$r->send_http_header("text/plain");

$SIG{CHLD} = 'IGNORE';
defined (my $kid = fork) or die "Cannot fork: $!\n";
if ($kid) {
    print "Parent $$ has finished, kid's PID: $kid\n";
} else {
    $r->cleanup_for_exec(); # untie the socket
    chdir '/' or die "Can't chdir to /: $!";
    open STDIN, '/dev/null' or die "Can't read /dev/null: $!";
    open STDOUT, '>/dev/null'
        or die "Can't write to /dev/null: $!";
    open STDERR, '>/tmp/log' or die "Can't write to /tmp/log: $!";
    setsid or die "Can't start a new session: $!";

    my $oldfh = select STDERR;
    local $| = 1;
    select $oldfh;
    warn "started\n";
    # do something time-consuming
    sleep 1, warn "$_\n" for 1..20;
    warn "completed\n";

    CORE::exit(0); # terminate the process
}

```

The script starts with the usual declaration of the strict mode, loading the POSIX and Apache::SubProcess modules and importing of the setsid() symbol from the POSIX package.

The HTTP header is sent next, with the *Content-type* of *text/plain*. The gets ready to ignore the child, to avoid zombies and the fork is called.

The program gets its personality split after fork and the if conditional evaluates to a true value for the parent process, and to a false value for the child process, therefore the first block is executed by the parent and the second by the child.

The parent process announces his PID and the PID of the spawned process and finishes its block. If there will be any code outside it will be executed by the parent as well.

The child process starts its code by disconnecting from the socket, changing its current directory to /, opening the STDIN and STDOUT streams to */dev/null*, which in effect closes them both before opening. In fact in this example we don't need neither of these, so we could just close() both. The child process completes its disengagement from the parent process by opening the STDERR stream to */tmp/log*, so it could write there, and creating a new session with help of setsid(). Now the child process has nothing to do with the parent process and can do the actual processing that it has to do. In our example it performs a simple series of warnings, which are logged into */tmp/log*:

```
my $oldfh = select STDERR;
local $| = 1;
select $oldfh;
warn "started\n";
# do something time-consuming
sleep 1, warn "$_\n" for 1..20;
warn "completed\n";
```

The localized setting of `$|=1` unbuffers the `STDERR` stream, so we can immediately see the debug output generated by the program. In fact this setting is not required when the output is generated by `warn()`.

Finally the child process terminates by calling:

```
CORE::exit(0);
```

which make sure that it won't get out of the block and run some code that it's not supposed to run.

This code example will allow you to verify that indeed the spawned child process has its own life, and its parent is free as well. Simply issue a request that will run this script, watch that the warnings are started to be written into the `/tmp/log` file and issue a complete server stop and start. If everything is correct, the server will successfully restart and the long term process will still be running. You will know that it's still running, if the warnings will still be printed into the `/tmp/log` file. You may need to raise the number of warnings to do above 20, to make sure that you don't miss the end of the run.

If there are only 5 warnings to be printed, you should see the following output in this file:

```
started
1
2
3
4
5
completed
```

6.16 Sending plain HTML as a compressed output

Have you ever served a huge HTML file (e.g. a file bloated with JavaScript code) and wondered how could you send it compressed, thus dramatically cutting down the download times. After all java applets can be compressed into a jar and benefit from a faster download times. Why cannot we do the same with a plain ASCII (HTML,JS and etc), it is a known fact that ASCII text can be compressed by a factor of 10.

`Apache::GzipChain` comes to help you with this task. If a client (browser) understands `gzip` encoding this module compresses the output and sends it downstream. A client decompresses the data upon receive and renders the HTML as if it was a plain HTML fetch.

For example to compress all html files on the fly, do:

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

Remember that it will work only if the browser claims to accept compressed input, thru Accept-Encoding header. Apache::GzipChain keeps a list of user-agents, thus it also looks at User-Agent header, for known to accept compressed output browsers.

For example if you want to return compressed files which should pass in addition through Embperl module, you would write:

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

Hint: Watch an access_log file to see how many bytes were actually send, compare with a regular configuration send.

(See perldoc Apache::GzipChain).

Notice that the rightmost PerlHandler must be a content producer. Use Apache::PassFile or another similar module.

;o)

7 Perl Reference

7.1 What we will learn in this chapter

- perldoc's Rarely Known But Very Useful Options
- Tracing Warnings Reports
- Variables Globally, Lexically Scoped And Fully Qualified
- 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
- Exception Handling for mod_perl

7.2 perldoc's Rarely Known But Very Useful Options

First of all, I want to stress that you cannot become a Perl hacker without knowing how to read Perl documentation and search through it. Books are good, but an easily accessible and searchable Perl reference is at your fingertips and is a great time saver.

While you can use online Perl documentation at the Web, the `perldoc` utility provides you with access to the documentation installed on your system. To find out what Perl manpages are available execute:

```
% perldoc perl
```

To find what functions perl has, execute:

```
% perldoc perlfunc
```

To learn the syntax and to find examples of a specific function, you would execute (e.g. for `open()`):

```
% perldoc -f open
```

Note: In perl5.00503 and earlier, there is a bug in this and the `-q` options of `perldoc`. It won't call `pod2man`, but will display the section in POD format instead. Despite this bug it's still readable and very useful.

To search through the Perl FAQ (*perlfaq* manpage) sections you would (e.g for the `open` keyword) execute:

```
% perldoc -q open
```

This will show you all the matching Q&A sections, still in POD format.

To read the *perldoc* manpage you execute:

```
% perldoc perldoc
```

7.3 Tracing Warnings Reports

Sometimes it's very hard to understand what a warning is complaining about. You see the source code, but you cannot understand why some specific snippet produces that warning. The mystery often results from the fact that the code can be called from different places if it's located inside a subroutine.

Here is an example:

```
warnings.pl
-----
#!/usr/bin/perl -w

correct();
incorrect();

sub correct{
    print_value("Perl");
}

sub incorrect{
    print_value();
}

sub print_value{
    my $var = shift;
    print "My value is $var\n";
}
```

In the code above, `print_value()` prints the passed value, `correct()` passes the value to print and in `incorrect()` we forgot to pass it. When we run the script:

```
% ./warnings.pl
```

we get the warning:

```
Use of uninitialized value at ./warnings.pl line 16.
```

Perl complains about an undefined variable `$var` at the line that attempts to print its value:

```
print "My value is $var\n";
```

But how do we know why it is undefined? The reason here obviously is that the calling function didn't pass the argument. But how do we know who was the caller? In our example there are two possible callers, in the general case there can be many of them, perhaps located in other files.

We can use the `caller()` function, which tells who has called us, but even that might not be enough: it's possible to have a longer sequence of called subroutines, and not just two. For example, here it is `sub third()` which is at fault, and putting `sub caller()` in `sub second()` would not help us very much:

```
sub third{
    second();
}
sub second{
    my $var = shift;
    first($var);
}
sub first{
    my $var = shift;
    print "Var = $var\n"
}
```

The solution is quite simple. What we need is a full calls stack trace to the call that triggered the warning.

The Carp module comes to our aid with its `cluck()` function. Let's modify the script by adding a couple of lines. The rest of the script is unchanged.

```
warnings2.pl
-----
#!/usr/bin/perl -w

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

correct();
incorrect();

sub correct{
    print_value("Perl");
}

sub incorrect{
    print_value();
}

sub print_value{
    my $var = shift;
    print "My value is $var\n";
}
```

Now when we execute it, we see:

```
Use of uninitialized value at ./warnings2.pl line 19.
main::print_value() called at ./warnings2.pl line 14
main::incorrect() called at ./warnings2.pl line 7
```


Take a moment to understand the calls stack trace. The deepest calls are printed first. So the second line tells us that the warning was triggered in `print_value()`; the third, that `print_value()` was called by `incorrect()` subroutine.

```
script => incorrect() => print_value()
```

We go into `incorrect()` and indeed see that we forgot to pass the variable. Of course when you write a subroutine like `print_value` it would be a good idea to check the passed arguments before starting execution. We omitted that step to contrive an easily debugged example.

Sure, you say, I could find that problem by simple inspection of the code!

Well, you're right. But I promise you that your task would be quite complicated and time consuming if your code has some thousands of lines. In addition, under `mod_perl`, certain uses of the `eval` operator and "here documents" are known to throw off Perl's line numbering, so the messages reporting warnings and errors can have incorrect line numbers.

Getting the trace helps a lot.

7.4 Variables Globally, Lexically Scoped And Fully Qualified

Also see the clarification of `my()` vs. `use vars` - Ken Williams writes:

```
Yes, there is quite a bit of difference!  With use vars(), you are
making an entry in the symbol table, and you are telling the
compiler that you are going to be referencing that entry without an
explicit package name.
```

```
With my(), NO ENTRY IS PUT IN THE SYMBOL TABLE.  The compiler
figures out C<at compile time> which my() variables (i.e. lexical
variables) are the same as each other, and once you hit execute time
you cannot go looking those variables up in the symbol table.
```

And `my()` vs. `local()` - Randal Schwartz writes:

```
local() creates a temporal-limited package-based scalar, array,
hash, or glob -- when the scope of definition is exited at runtime,
the previous value (if any) is restored.  References to such a
variable are *also* global... only the value changes.  (Aside: that
is what causes variable suicide. :)
```

```
my() creates a lexically-limited non-package-based scalar, array, or
hash -- when the scope of definition is exited at compile-time, the
variable ceases to be accessible.  Any references to such a variable
at runtime turn into unique anonymous variables on each scope exit.
```

7.5 my() Scoped Variable in Nested Subroutines

Before we proceed let's make the assumption that we want to develop the code under the `strict` pragma. We will use lexically scoped variables (with help of the `my()` operator) whenever it's possible.

7.5.1 The Poison

Let's look at this code:

```
nested.pl
-----
#!/usr/bin/perl

use strict;

sub print_power_of_2 {
    my $x = shift;

    sub power_of_2 {
        return $x ** 2;
    }

    my $result = power_of_2();
    print "$x^2 = $result\n";
}

print_power_of_2(5);
print_power_of_2(6);
```

Don't let the weird subroutine names fool you, the `print_power_of_2()` subroutine should print the square of the passed number. Let's run the code and see whether it works:

```
% ./nested.pl

5^2 = 25
6^2 = 25
```

Ouch, something is wrong. May be there is a bug in Perl and it doesn't work correctly with number 6? Let's try again using the 5 and 7:

```
print_power_of_2(5);
print_power_of_2(7);
```

And run it:

```
% ./nested.pl

5^2 = 25
7^2 = 25
```

Wow, does it work only for 5? How about using 3 and 5:

```
print_power_of_2(3);  
print_power_of_2(5);
```

and the result is:

```
% ./nested.pl  
  
3^2 = 9  
5^2 = 9
```

Now we start to understand--only the first call to the `print_power_of_2()` function works correctly. Which makes us think that our code has some kind of memory for results of the first execution, or it ignores the arguments in subsequent executions.

7.5.2 *The Diagnosis*

Let's follow the guidelines and use the `-w` flag. Now execute the code:

```
% ./nested.pl  
  
Variable "$x" will not stay shared at ./nested.pl line 9.  
5^2 = 25  
6^2 = 25
```

We have never seen such a warning message before and we don't quite understand what it means. The `diagnostics` pragma will certainly help us. Let's prepend this pragma before the `strict` pragma in our code:

```
#!/usr/bin/perl -w  
  
use diagnostics;  
use strict;
```

And execute it:

```
% ./nested.pl

Variable "$x" will not stay shared at ./nested.pl line 10 (#1)

(W) An inner (nested) named subroutine is referencing a lexical
variable defined in an outer subroutine.

When the inner subroutine is called, it will probably see the value of
the outer subroutine's variable as it was before and during the
*first* call to the outer subroutine; in this case, after the first
call to the outer subroutine is complete, the inner and outer
subroutines will no longer share a common value for the variable. In
other words, the variable will no longer be shared.

Furthermore, if the outer subroutine is anonymous and references a
lexical variable outside itself, then the outer and inner subroutines
will never share the given variable.

This problem can usually be solved by making the inner subroutine
anonymous, using the sub {} syntax. When inner anonymous subs that
reference variables in outer subroutines are called or referenced,
they are automatically rebound to the current values of such
variables.

5^2 = 25
6^2 = 25
```

Well, now everything is clear. We have the **inner** subroutine `power_of_2()` and the **outer** subroutine `print_power_of_2()` in our code.

When the inner `power_of_2()` subroutine is called for the first time, it sees the value of the outer `print_power_of_2()` subroutine's `$x` variable. On subsequent calls the `$x` variable won't be updated, no matter what the value of it in the outer subroutine. There are two copies of the `$x` variable, no longer a single one shared by the two routines.

7.5.3 The Remedy

The `diagnostics` pragma suggests that the problem can be solved by making the inner subroutine anonymous.

An anonymous subroutine can act as a *closure* with respect to lexically scoped variables. Basically this means that if you define a subroutine in a particular **lexical** context at a particular moment, then it will run in that same context later, even if called from outside that context. The upshot of this is that when the subroutine **runs**, you get the same copies of the lexically scoped variables which were visible when the subroutine was **defined**. So you can pass arguments to a function when you define it, as well as when you invoke it.

Let's rewrite the code to use this technique:

```

anonymous.pl
-----
#!/usr/bin/perl

use strict;

sub print_power_of_2 {
    my $x = shift;

    my $func_ref = sub {
        return $x ** 2;
    };

    my $result = &$func_ref();
    print "$x^2 = $result\n";
}

print_power_of_2(5);
print_power_of_2(6);

```

Now `$func_ref` contains a reference to an anonymous function, which we later use when we need to get the power of two. (In Perl, a function is the same thing as a subroutine.) Since it is anonymous, the function will automatically be rebound to the new value of the outer scoped variable `$x`, and the results will now be as expected.

Let's verify:

```

% ./anonymous.pl

5^2 = 25
6^2 = 36

```

Indeed, *anonymous.pl* worked as we expected.

7.6 When You Cannot Get Rid of The Inner Subroutine

First you might wonder, why in the world will someone need to define an inner subroutine? Well, for example to reduce some of Perl's script startup overhead you might decide to write a daemon that will compile the scripts and modules only once, and cache the pre-compiled code in memory. When some script is to be executed, you just tell the daemon the name of the script to run and it will do the rest and do it much faster.

Seems like an easy task, and it is. The only problem is once the script is compiled, how do you execute it? Or let's put it the other way: after it was executed for the first time and it stays compiled in the daemon memory, how do you call it again? If you could get all developers to code the scripts so each has a subroutine called `run()` that will actually execute the code in the script then you have half of the problem solved.

But how does the daemon know to refer to some specific script if they all run in the `main::` name space? One solution might be to ask the developers to declare a package in each and every script, and for the package name to be derived from the script name. However, since there is chance that there will be more than one script with the same name but residing in different directories, then in order to prevent name-space collisions the directory has to be a part of the package name too. And don't forget

that script may be moved from one directory to another, so you will have to make sure that the package name is corrected every time the script gets moved.

But why enforce these strange rules on developers, when we can arrange for our daemon to do this work? For every script that daemon is about to execute for the first time, it should be wrapped inside the package whose name is constructed from the mangled path to the script and a subroutine called `run()`. For example if the daemon is about to execute the script `/tmp/hello.pl`:

```
hello.pl
-----
#!/usr/bin/perl
print "Hello\n";
```

Prior to running it, the daemon will change the code to be:

```
wrapped_hello.pl
-----
package cache::tmp::hello_2epl;

sub run{
    #!/usr/bin/perl
    print "Hello\n";
}
```

The package name is constructed from the prefix `cache::`, each directory separation slash is replaced with `::`, and non alphanumeric characters are encoded so that for example `.` (a dot) becomes `_2e` (an underscore followed by the ASCII code for a dot in hex representation).

```
% perl -e 'printf "%x",ord(".".)'
```

prints: `2e`. The underscore is the same you see in URL encoding where `%` character is used instead (`%2E`), but since `%` has a special meaning in Perl (prefix of hash variable) it couldn't be used.

Now when the daemon is requested to execute the script `/tmp/hello.pl`, all it has to do is to build the package name as before based on the location of the script and call its `run()` subroutine:

```
use cache::tmp::hello_2epl;
cache::tmp::hello_2epl::run();
```

We have just written a partial prototype of the daemon we desired. The only method now remaining undefined is how to pass the path to the script to the daemon. This detail is left to the reader as an exercise.

If you are familiar with the `Apache::Registry` module, you know that it works in almost the same way. It uses a different package prefix and the generic function is called `handler()` and not `run()`. The scripts to run are passed through the HTTP protocol's headers.

Now you understand that there are cases where your normal subroutines can become inner, since if your script was a simple:

```

simple.pl
-----
#!/usr/bin/perl
sub hello { print "Hello" }
hello();

```

Wrapped into a `run()` subroutine it becomes:

```

simple.pl
-----
package cache::simple_2epl;

sub run{
    #!/usr/bin/perl
    sub hello { print "Hello" }
    hello();
}

```

Therefore, `hello()` is an inner subroutine and if you have used `my()` scoped variables defined and altered outside and used inside `hello()`, it won't work as you expect starting from the second call, as was explained in the previous section.

7.6.1 Remedies for Inner Subroutines

First of all there is nothing to worry about, as long as you don't forget to turn the warnings On. If you do happen to have the “`my()` Scoped Variable in Nested Subroutines” problem, Perl will always alert you.

Given that you have a script that has this problem, what are the ways to solve it? There are many of them and we will discuss some of them here.

We will use the following code to show the different solutions.

```

multirun.pl
-----
#!/usr/bin/perl -w

use strict;

for (1..3){
    print "run: [time $_]\n";
    run();
}

sub run {

    my $counter = 0;

    increment_counter();
    increment_counter();

    sub increment_counter{
        $counter++;
        print "Counter is equal to $counter !\n";
    }

} # end of sub run

```

This code executes the `run()` subroutine three times, which in turn initializes the `$counter` variable to 0, every time it executed and then calls the inner subroutine `increment_counter()` twice. Sub `increment_counter()` prints `$counter`'s value after incrementing it. One might expect to see the following output:

```

run: [time 1]
Counter is equal to 1 !
Counter is equal to 2 !
run: [time 2]
Counter is equal to 1 !
Counter is equal to 2 !
run: [time 3]
Counter is equal to 1 !
Counter is equal to 2 !

```

But as we have already learned from the previous sections, this is not what we are going to see. Indeed, when we run the script we see:

```
% ./multirun.pl
```

```

Variable "$counter" will not stay shared at ./nested.pl line 18.
run: [time 1]
Counter is equal to 1 !
Counter is equal to 2 !
run: [time 2]
Counter is equal to 3 !
Counter is equal to 4 !
run: [time 3]
Counter is equal to 5 !
Counter is equal to 6 !

```


Obviously, the `$counter` variable is not reinitialized on each execution of `run()`. It retains its value from the previous execution, and `sub increment_counter()` increments that.

One of the workarounds is to use globally declared variables, with the `vars` pragma.

```
multirun1.pl
-----
#!/usr/bin/perl -w

use strict;
use vars qw($counter);

for (1..3){
    print "run: [time $_]\n";
    run();
}

sub run {

    $counter = 0;

    increment_counter();
    increment_counter();

    sub increment_counter{
        $counter++;
        print "Counter is equal to $counter !\n";
    }

} # end of sub run
```

If you run this and the other solutions offered below, the expected output will be generated:

```
% ./multirun1.pl

run: [time 1]
Counter is equal to 1 !
Counter is equal to 2 !
run: [time 2]
Counter is equal to 1 !
Counter is equal to 2 !
run: [time 3]
Counter is equal to 1 !
Counter is equal to 2 !
```

By the way, the warning we saw before has gone, and so has the problem, since there is no `my()` (lexically defined) variable used in the nested subroutine.

Another approach is to use fully qualified variables. This is better, since less memory will be used, but it adds a typing overhead:

```

multirun2.pl
-----
#!/usr/bin/perl -w

use strict;

for (1..3){
    print "run: [time $_]\n";
    run();
}

sub run {

    $main::counter = 0;

    increment_counter();
    increment_counter();

    sub increment_counter{
        $main::counter++;
        print "Counter is equal to $main::counter !\n";
    }

} # end of sub run

```

You can also pass the variable to the subroutine by value and make the subroutine return it after it was updated. This adds time and memory overheads, so it may not be good idea if the variable can be very large, or if speed of execution is an issue.

Don't rely on the fact that the variable is small during the development of the application, it can grow quite big in situations you don't expect. For example, a very simple HTML form text entry field can return a few megabytes of data if one of your users is bored and wants to test how good is your code. It's not uncommon to see users Copy-and-Paste 10Mb core dump files into a form's text fields and then submit it for your script to process.

```

multirun3.pl
-----
#!/usr/bin/perl -w

use strict;

for (1..3){
    print "run: [time $_]\n";
    run();
}

sub run {

    my $counter = 0;

    $counter = increment_counter($counter);
    $counter = increment_counter($counter);

    sub increment_counter{
        my $counter = shift || 0 ;

        $counter++;
        print "Counter is equal to $counter !\n";

        return $counter;
    }

} # end of sub run

```

Finally, you can use references to do the job. The version of `increment_counter()` below accepts a reference to the `$counter` variable and increments its value after first dereferencing it. When you use a reference, the variable you use inside the function is physically the same bit of memory as the one outside the function. This technique is often used to enable a called function to modify variables in a calling function.

```

multirun4.pl
-----
#!/usr/bin/perl -w

use strict;

for (1..3){
    print "run: [time $_]\n";
    run();
}

sub run {

    my $counter = 0;

    increment_counter(\$counter);
    increment_counter(\$counter);

    sub increment_counter{
        my $r_counter = shift || 0;

        $$r_counter++;
        print "Counter is equal to $$r_counter !\n";
    }

} # end of sub run

```

Here is yet another and more obscure reference usage. We modify the value of `$counter` inside the subroutine by using the fact that variables in `@_` are aliases for the actual scalar parameters. Thus if you called a function with two arguments, those would be stored in `$_[0]` and `$_[1]`. In particular, if an element `$_[0]` is updated, the corresponding argument is updated (or an error occurs if it is not updatable).

```

multirun5.pl
-----
#!/usr/bin/perl -w

use strict;

for (1..3){
    print "run: [time $_]\n";
    run();
}

sub run {

    my $counter = 0;

    increment_counter($counter);
    increment_counter($counter);

    sub increment_counter{
        $_[0]++;
        print "Counter is equal to $_[0] !\n";
    }

} # end of sub run

```

Now you have at least five workarounds to choose from.

For more information please refer to `perlref` and `perlsub` manpages.

7.7 `use()`, `require()`, `do()`, `%INC` and `@INC` Explained

7.7.1 *The @INC array*

`@INC` is a special Perl variable which is the equivalent of the shell's `PATH` variable. Whereas `PATH` contains a list of directories to search for executables, `@INC` contains a list of directories from which Perl modules and libraries can be loaded.

When you `use()`, `require()` or `do()` a filename or a module, Perl gets a list of directories from the `@INC` variable and searches them for the file it was requested to load. If the file that you want to load is not located in one of the listed directories, you have to tell Perl where to find the file. You can either provide a path relative to one of the directories in `@INC`, or you can provide the full path to the file.

7.7.2 *The %INC hash*

`%INC` is another special Perl variable that is used to cache the names of the files and the modules that were successfully loaded and compiled by `use()`, `require()` or `do()` functions. Before attempting to load a file or a module, Perl checks whether it's already in the `%INC` hash. If it's there, the loading and therefore the compilation are not performed at all. Otherwise the file is loaded into memory and an attempt is made to compile it.

If the file is successfully loaded and compiled, a new key-value pair is added to `%INC`. The key is the name of the file or module as it was passed to the one of the three functions we have just mentioned, and if it was found in any of the `@INC` directories except `"."` the value is the full path to it in the file system.

The following examples will make it easier to understand the logic.

First, let's see what are the contents of `@INC` on my system:

```
% perl -e 'print join "\n", @INC'
/usr/lib/perl5/5.00503/i386-linux
/usr/lib/perl5/5.00503
/usr/lib/perl5/site_perl/5.005/i386-linux
/usr/lib/perl5/site_perl/5.005
.
```

Notice the `.` (current directory) is the last directory in the list.

Now let's load the module `strict.pm` and see the contents of `%INC`:

```
% perl -e 'use strict; print map {"$_ => $INC{$_}\n"} keys %INC'

strict.pm => /usr/lib/perl5/5.00503/strict.pm
```

Since `strict.pm` was found in `/usr/lib/perl5/5.00503/` directory and `/usr/lib/perl5/5.00503/` is a part of `@INC`, `%INC` includes the full path as the value for the key `strict.pm`.

Now let's create the simplest module in `/tmp/test.pm`:

```
test.pm
-----
1;
```

It does nothing, but returns a true value when loaded. Now let's load it in different ways:

```
% cd /tmp
% perl -e 'use test; print map {"$_ => $INC{$_}\n"} keys %INC'

test.pm => test.pm
```

Since the file was found relative to `.` (the current directory), the relative path is inserted as the value. If we alter `@INC`, by adding `/tmp` to the end:

```
% cd /tmp
% perl -e 'BEGIN{push @INC, "/tmp"} use test; \
print map {"$_ => $INC{$_}\n"} keys %INC'

test.pm => test.pm
```

Here we still get the relative path, since the module was found first relative to `"."`. The directory `/tmp` was placed after `.` in the list. If we execute the same code from a different directory, the `"."` directory won't match,

```
% cd /
% perl -e 'BEGIN{push @INC, "/tmp"} use test; \
print map {"$_ => $INC{$_}\n"} keys %INC'

test.pm => /tmp/test.pm
```

so we get the full path. We can also prepend the path with `unshift()`, so it will be used for matching before `"."` and therefore we will get the full path as well:

```
% cd /tmp
% perl -e 'BEGIN{unshift @INC, "/tmp"} use test; \
print map {"$_ => $INC{$_}\n"} keys %INC'

test.pm => /tmp/test.pm
```

The code:

```
BEGIN{unshift @INC, "/tmp"}
```

can be replaced with the more elegant:

```
use lib "/tmp";
```

Which executes the BEGIN block above exactly.

These approaches to modifying @INC can be labor intensive, since if you want to move the script around in the file-system you have to modify the path. This can be painful, for example, when you move your scripts from development to a production server.

There is a module called `FindBin` which solves this problem in the plain Perl world, but unfortunately it won't work under `mod_perl`, since it's a module and as any module it's loaded only once. So the first script using it will have all the settings correct, but the rest of the scripts will not if located in a different directory from the first.

For a completeness of this section, I'll present this module anyway.

If you use this module, you don't need to write a hard coded path. The following snippet does all the work for you (the file is `/tmp/load.pl`):

```
load.pl
-----
#!/usr/bin/perl

use FindBin ();
use lib "$FindBin::Bin";
use test;
print "test.pm => $INC{'test.pm'}\n";
```

In the above example `$FindBin::Bin` is equal to `/tmp`. If we move the script somewhere else... e.g. `/tmp/x` in the code above `$FindBin::Bin` equals `/home/x`.

```
% /tmp/load.pl

test.pm => /tmp/test.pm
```

Just like with `use lib` but no hard coded path required.

You can use this workaround to make it work under `mod_perl`.

```
do 'FindBin.pm';
unshift @INC, "$FindBin::Bin";
require test;
#maybe test::import( ... ) here if need to import stuff
```

You will have a slight overhead because you will load from disk and recompile the `FindBin` module on each request. So it can be not worth it.

7.7.3 Modules, Libraries and Files

Before we proceed, let's define what we mean by *module*, and *library* or *file*.

- **The Library or the File**

A file which contains perl subroutines and other code.

It generally doesn't include a package declaration.

Its last statement returns true.

It can be named in any way desired, but generally its extension is *.pl* or *.ph*.

Examples:

```
config.pl
-----
$dir = "/home/httpd/cgi-bin";
$cgi = "/cgi-bin";
1;
```

```
mysubs.pl
-----
sub print_header{
    print "Content-type: text/plain\r\n\r\n";
}
1;
```

- **the Module**

A file which contains perl subroutines and other code.

It generally declares a package name at the beginning of it.

Its last statement returns true.

The naming convention requires it to have a *.pm* extension.

Example:

```
MyModule.pm
-----
package My::Module;
$My::Module::VERSION = 0.01;

sub new{ return bless {}, shift;}
END { print "Quitting\n"}
1;
```

7.7.4 *require()*

`require()` reads a file containing Perl code and compiles it. Before attempting to load the file it looks up the argument in `%INC` to see whether it has already been loaded. If it has, `require()` just returns without doing a thing. Otherwise an attempt will be made to load and compile the file.

`require()` has to find the file it has to load. If the argument is a full path to the file, it just tries to read it. For example:

```
require "/home/httpd/perl/mylibs.pl";
```

If the path is relative, `require()` will attempt to search for the file in all the directories listed in `@INC`. For example:

```
require "mylibs.pl";
```

If there is more than one occurrence of the file with the same name in the directories listed in `@INC` the first occurrence will be used.

The file must return *TRUE* as the last statement to indicate successful execution of any initialization code. Since you never know what changes the file will go through in the future, you cannot be sure that the last statement will always return *TRUE*. That's why the suggestion is to put `1;` at the end of file.

Although you should use the real filename for most files, if the file is a module, you may use the following convention instead:

```
require My::Module;
```

This is equal to:

```
require "My/Module.pm";
```

If `require()` fails to load the file, either because it couldn't find the file in question or the code failed to compile, or it didn't return *TRUE*, then the program would `die()`. To prevent this the `require()` statement can be enclosed into an `eval()` block, as in this example:

```
require.pl
-----
#!/usr/bin/perl -w

eval { require "/file/that/does/not/exists" };
if ($?) {
    print "Failed to load, because : $@"
}
print "\nHello\n";
```

When we execute the program:

```
% ./require.pl

Failed to load, because : Can't locate /file/that/does/not/exists in
@INC (@INC contains: /usr/lib/perl5/5.00503/i386-linux
/usr/lib/perl5/5.00503 /usr/lib/perl5/site_perl/5.005/i386-linux
/usr/lib/perl5/site_perl/5.005 .) at require.pl line 3.

Hello
```

We see that the program didn't `die()`, because *Hello* was printed. This *trick* is useful when you want to check whether a user has some module installed, but if she hasn't it's not critical, perhaps the program can run without this module with reduced functionality.

If we remove the `eval()` part and try again:

```
require.pl
-----
#!/usr/bin/perl -w

require "/file/that/does/not/exists";
print "\nHello\n";

% ./require1.pl

Can't locate /file/that/does/not/exists in @INC (@INC contains:
/usr/lib/perl5/5.00503/i386-linux /usr/lib/perl5/5.00503
/usr/lib/perl5/site_perl/5.005/i386-linux
/usr/lib/perl5/site_perl/5.005 .) at require1.pl line 3.
```

The program just `die()`s in the last example, which is what you want in most cases.

For more information refer to the `perlfunc` manpage.

7.7.5 *use()*

`use()`, just like `require()`, loads and compiles files containing Perl code, but it works with modules only. The only way to pass a module to load is by its module name and not its filename. If the module is located in *MyCode.pm*, the correct way to `use()` it is:

```
use MyCode
```

and not:

```
use "MyCode.pm"
```

`use()` translates the passed argument into a file name replacing `::` with `/` and appending `.pm` at the end. So `My::Module` becomes *My/Module.pm*.

`use()` is exactly equivalent to:

```
BEGIN { require Module; import Module LIST; }
```

Internally it calls `require()` to do the loading and compilation chores. When `require()` finishes its job, `import()` is called unless `()` is the second argument. The following pairs are equivalent:

```
use MyModule;
BEGIN {require MyModule; import MyModule; }

use MyModule qw(foo bar);
BEGIN {require MyModule; import MyModule ("foo","bar"); }

use MyModule ();
BEGIN {require MyModule; }
```

The first pair exports the default tags. This happens if the module sets `@EXPORT` to a list of tags to be exported by default. The module manpage generally describes what modules are exported by default.

The second pair exports all the tags passed as arguments. No default tags are exported unless explicitly told to.

The third pair describes the case where the caller does not want any symbols to be imported.

`import()` is not a builtin function, it's just an ordinary static method call into the "MyModule" package to tell the module to import the list of features back into the current package. See the Exporter manpage for more information.

When you write your own modules, always remember that it's better to use `@EXPORT_OK` instead of `@EXPORT`, since the former doesn't export symbols unless it was asked to. Exports pollute the namespace of the module user. Also avoid short or common symbol names to reduce the risk of name clashes.

When functions and variables aren't exported you can still access them using their full names, like `$My::Module::bar` or `$My::Module::foo()`. By convention you can use a leading underscore on names to informally indicate that they are *internal* and not for public use.

There's a corresponding "no" command that un-imports symbols imported by `use`, i.e., it calls `unimport Module LIST` instead of `import()`.

7.7.6 *do()*

While `do()` behaves almost identically to `require()`, it reloads the file unconditionally. It doesn't check `%INC` to see whether the file was already loaded.

If `do()` cannot read the file, it returns `undef` and sets `$!` to report the error. If `do()` can read the file but cannot compile it, it returns `undef` and sets an error message in `$@`. If the file is successfully compiled, `do()` returns the value of the last expression evaluated.

7.8 Using Global Variables and Sharing Them Between Modules/Packages

7.8.1 Making Variables Global

When you first wrote `$x` in your code you created a global variable. It is visible everywhere in the file you have used it. If you defined it inside a package, it is visible inside the package. But it will work only if you do not use `strict` pragma and you **HAVE** to use this pragma if you want to run your scripts under `mod_perl`.

7.8.2 Making Variables Global With *strict* Pragma On

First you use :

```
use strict;
```

Then you use:

```
use vars qw($scalar %hash @array);
```

Starting from this moment the variables are global only in the package where you defined them. If you want to share global variables between packages, here is what you can do.

7.8.3 Using *Exporter.pm* to Share Global Variables

Assume that you want to share the `CGI.pm` object (I will use `$q`) between your modules. For example, you create it in `script.pl`, but you want it to be visible in `My::HTML`. First, you make `$q` global.

```
script.pl:
-----
use vars qw($q);
use CGI;
use lib qw(.);
use My::HTML qw($q); # My/HTML.pm is in the same dir as script.pl
$q = new CGI;

My::HTML::printmyheader();
-----
```

Note that we have imported `$q` from `My::HTML`. And `My::HTML` does the export of `$q`:

```

My/HTML.pm
-----
package My::HTML;
use strict;

BEGIN {
    use Exporter ();

    @My::HTML::ISA      = qw(Exporter);
    @My::HTML::EXPORT    = qw();
    @My::HTML::EXPORT_OK = qw($q);
}

use vars qw($q);

sub printmyheader{
    # Whatever you want to do with $q... e.g.
    print $q->header();
}
1;
-----

```

So the `$q` is shared between the `My::HTML` package and `script.pl`. It will work vice versa as well, if you create the object in `My::HTML` but use it in `script.pl`. You have true sharing, since if you change `$q` in `script.pl`, it will be changed in `My::HTML` as well.

What if you need to share `$q` between more than two packages? For example you want `My::Doc` to share `$q` as well.

You leave `My::HTML` untouched, and modify *script.pl* to include:

```
use My::Doc qw($q);
```

Then you write `My::Doc` exactly like `My::HTML` - except of course that the content is different :).

One possible pitfall is when you want to use `My::Doc` in both `My::HTML` and *script.pl*. Only if you add

```
use My::Doc qw($q);
```

into `My::HTML` will `$q` be shared. Otherwise `My::Doc` will not share `$q` any more. To make things clear here is the code:

```

script.pl:
-----
use vars qw($q);
use CGI;
use lib qw(.);
use My::HTML qw($q); # My/HTML.pm is in the same dir as script.pl
use My::Doc qw($q); # Ditto
$q = new CGI;

My::HTML::printmyheader();
-----

```

```

My/HTML.pm
-----
package My::HTML;
use strict;

BEGIN {
    use Exporter ();

    @My::HTML::ISA      = qw(Exporter);
    @My::HTML::EXPORT    = qw();
    @My::HTML::EXPORT_OK = qw($q);
}

use vars    qw($q);
use My::Doc qw($q);

sub printmyheader{
    # Whatever you want to do with $q... e.g.
    print $q->header();

    My::Doc::printtitle('Guide');
}
1;
-----

```

```

My/Doc.pm
-----
package My::Doc;
use strict;

BEGIN {
    use Exporter ();

    @My::Doc::ISA      = qw(Exporter);
    @My::Doc::EXPORT    = qw();
    @My::Doc::EXPORT_OK = qw($q);
}

use vars qw($q);

sub printtitle{
    my $title = shift || 'None';

    print $q->h1($title);
}
1;
-----

```

7.8.4 Using the Perl Aliasing Feature to Share Global Variables

As the title says you can import a variable into a script or module without using `Exporter.pm`. I have found it useful to keep all the configuration variables in one module `My::Config`. But then I have to export all the variables in order to use them in other modules, which is bad for two reasons: polluting other packages' name spaces with extra tags which increase the memory requirements; and adding the overhead of keeping track of what variables should be exported from the configuration module and what imported, for some particular package. I solve this problem by keeping all the vari-

ables in one hash %c and exporting that. Here is an example of My::Config:

```
package My::Config;
use strict;
use vars qw(%c);
%c = (
    # All the configs go here
    scalar_var => 5,

    array_var => [
        foo,
        bar,
    ],

    hash_var => {
        foo => 'Foo',
        bar => 'BARRR',
    },
);
1;
```

Now in packages that want to use the configuration variables I have either to use the fully qualified names like \$My::Config::test, which I dislike or import them as described in the previous section. But hey, since we have only one variable to handle, we can make things even simpler and save the loading of the Exporter.pm package. We will use the Perl aliasing feature for exporting and saving the keystrokes:

```
package My::HTML;
use strict;
use lib qw(.);
# Global Configuration now aliased to global %c
use My::Config (); # My/Config.pm in the same dir as script.pl
use vars qw(%c);
*c = \%My::Config::c;

# Now you can access the variables from the My::Config
print $c{scalar_val};
print $c{array_val}[0];
print $c{hash_val}{foo};
```

Of course \$c is global everywhere you use it as described above, and if you change it somewhere it will affect any other packages you have aliased \$My::Config::c to.

Note that aliases work either with global or local() vars - you cannot write:

```
my *c = \%My::Config::c;
```

Which is an error. But you can write:

```
local *c = \%My::Config::c;
```

7.9 The Scope of the Special Perl Variables

Special Perl variables like `$|` (buffering), `$^T` (time), `$^W` (warnings), `$/` (input record separator), `$\` (output record separator) and many more are all global variables. This means that you cannot scope them with `my()`. Only `local()` is permitted to do that. Since the child server doesn't usually exit, if in one of your scripts you modify a global variable it will be changed for the rest of the process' life and will affect all the scripts executed by the same process.

We will demonstrate the case on the input record separator variable. If you undefine this variable, a diamond operator will suck in the whole file at once if you have enough memory. Remembering this you should never write code like the example below.

```
$/ = undef;
open IN, "file" ....
    # slurp it all into a variable
    $all_the_file = <IN>;
```

The proper way is to have a `local()` keyword before the special variable is changed, like this:

```
local $/ = undef;
open IN, "file" ....
    # slurp it all inside a variable
    $all_the_file = <IN>;
```

But there is a catch. `local()` will propagate the changed value to any of the code below it. The modified value will be in effect until the script terminates, unless it is changed again somewhere else in the script.

A cleaner approach is to enclose the whole of the code that is affected by the modified variable in a block, like this:

```
{
    local $/ = undef;
    open IN, "file" ....
        # slurp it all inside a variable
        $all_the_file = <IN>;
}
```

That way when Perl leaves the block it restores the original value of the `$/` variable, and you don't need to worry elsewhere in your program about its value being changed here.

7.10 Compiled Regular Expressions

When using a regular expression that contains an interpolated Perl variable, if it is known that the variable (or variables) will not change during the execution of the program, a standard optimization technique is to add the `/o` modifier to the regexp pattern. This directs the compiler to build the internal table once, for the entire lifetime of the script, rather than every time the pattern is executed. Consider:


```

my $pat = '^foo$'; # likely to be input from an HTML form field
foreach( @list ) {
    print if /$pat/o;
}

```

This is usually a big win in loops over lists, or when using `grep()` or `map()` operators.

In long-lived `mod_perl` scripts, however, the variable can change according to the invocation and this can pose a problem. The first invocation of a fresh `httpd` child will compile the regex and perform the search correctly. However, all subsequent uses by that child will continue to match the original pattern, regardless of the current contents of the Perl variables the pattern is supposed to depend on. Your script will appear to be broken.

There are two solutions to this problem:

The first is to use `eval q//`, to force the code to be evaluated each time. Just make sure that the `eval` block covers the entire loop of processing, and not just the pattern match itself.

The above code fragment would be rewritten as:

```

my $pat = '^foo$';
eval q{
    foreach( @list ) {
        print if /$pat/o;
    }
}

```

Just saying:

```

foreach( @list ) {
    eval q{ print if /$pat/o; };
}

```

is going to be a horribly expensive proposition.

You can use this approach if you require more than one pattern match operator in a given section of code. If the section contains only one operator (be it an `m//` or `s//`), you can rely on the property of the null pattern, that reuses the last pattern seen. This leads to the second solution, which also eliminates the use of `eval`.

The above code fragment becomes:

```

my $pat = '^foo$';
"something" =~ /$pat/; # dummy match (MUST NOT FAIL!)
foreach( @list ) {
    print if //;
}

```

The only gotcha is that the dummy match that boots the regular expression engine must absolutely, positively succeed, otherwise the pattern will not be cached, and the `//` will match everything. If you can't count on fixed text to ensure the match succeeds, you have two possibilities.

If you can guarantee that the pattern variable contains no meta-characters (things like *, +, ^, \$...), you can use the dummy match:

```
"$pat" =~ /\Q$pat\E/; # guaranteed if no meta-characters present
```

If there is a possibility that the pattern can contain meta-characters, you should search for the pattern or the non-searchable \377 character as follows:

```
"\377" =~ /$pat|^\377$/; # guaranteed if meta-characters present
```

Another approach:

It depends on the complexity of the regexp to which you apply this technique. One common usage where a compiled regexp is usually more efficient is to “*match any one of a group of patterns*” over and over again.

Maybe with a helper routine, it’s easier to remember. Here is one slightly modified from Jeffery Friedl’s example in his book “*Mastering Regex*”.

```
#####
# Build_MatchMany_Function
# -- Input:  list of patterns
# -- Output: A code ref which matches its $_[0]
#           against ANY of the patterns given in the
#           "Input", efficiently.
#
sub Build_MatchMany_Function {
    my @R = @_;
    my $expr = join '||', map { "\$_[0] =~ m/\$R[\$_]/o" } ( 0..$#R );
    my $matchsub = eval "sub { $expr }";
    die "Failed in building regex @R: $@" if $@;
    $matchsub;
}
```

Example usage:

```
@some_browsers = qw(Mozilla Lynx MSIE AmigaVoyager lwp libwww);
$Known_Browser=Build_MatchMany_Function(@some_browsers);

while (<ACCESS_LOG>) {
    # ...
    $browser = get_browser_field($_);
    if ( ! &$Known_Browser($browser) ) {
        print STDERR "Unknown Browser: $browser\n";
    }
    # ...
}
```

7.11 Exception Handling for mod_perl

Provided here are some guidelines for clean(er) exception handling for mod_perl usage, although the technique presented here applies to all of your Perl programming.

The reasoning behind this document is the current broken status of `$SIG{__DIE__}` in the perl core - see both the perl5-porters and mod_perl mailing list archives for details on this discussion. (It's broken in at least Perl v5.6.0 and probably in later versions as well.)

7.11.1 Trapping Exceptions in Perl

To trap an exception in Perl we use the `eval{ }` construct. Many people initially make the mistake that this is the same as the `eval EXPR` construct, which compiles and executes code at run time, but that's not the case. `eval{ }` compiles at compile time, just like the rest of your code, and has next to zero run-time penalty.

When in an eval block, if the code executing `die()`'s for some reason, rather than terminating your code, the exception is *caught* and the program is allowed to examine that exception and make decisions based on it. The full construct looks like this:

```
eval
{
    # Some code here
}; # Note important semi-colon there
if ($@) # $@ contains the exception that was thrown
{
    # Do something with the exception
}
else # optional
{
    # No exception was thrown
}
```

Most of the time when you see these exception handlers there is no else block, because it tends to be OK if the code didn't throw an exception.

7.11.2 Alternative Exception Handling Techniques

An often suggested method for handling global exceptions in mod_perl, and other perl programs in general, is a `__DIE__` handler, which can be setup by either assigning a function name as a string to `$SIG{__DIE__}` (not particularly recommended, because of the possible namespace clashes) or assigning a code reference to `$SIG{__DIE__}`, the usual way of doing so is to use an anonymous subroutine:

```
$SIG{__DIE__} = sub { print "Eek - we died with:\n", $_[0]; };
```

The current problem with this is that `$SIG{__DIE__}` is a global setting in your script, so while you can potentially hide away your exceptions in some external module, the execution of `$SIG{__DIE__}` is fairly magical, and interferes not just with your code, but with all code in every module you import. Beyond the magic involved, `$SIG{__DIE__}` actually interferes with perl's normal exception handling mechanism, the `eval{ }` construct. Witness:

```

$SIG{__DIE__} = sub { print "handler\n"; };

eval {
    print "In eval\n";
    die "Failed for some reason\n";
};
if ($@) {
    print "Caught exception: $@";
}

```

The code unfortunately prints out:

```

In eval
handler

```

Which isn't quite what you would expect, especially if that `$SIG{__DIE__}` handler is hidden away deep in some other module that you didn't know about. There are work arounds however. One is to localise `$SIG{__DIE__}` in every exception trap you write:

```

eval {
    local $SIG{__DIE__};
    ...
};

```

Obviously this just doesn't scale - you don't want to be doing that for every exception trap in your code, and it's a slow down. A second work around is to check in your handler if you are trying to catch this exception:

```

$SIG{__DIE__} = sub {
    die $_[0] if $^S;
    print "handler\n";
};

```

However this won't work under `Apache::Registry` - you're always in an eval block there!

You should warn people about this danger of `$SIG{__DIE__}` and inform them of better ways to code. The following material is an attempt to just that.

7.11.3 *Better Exception Handling*

The `eval{ }` construct in itself is a fairly weak way to handle exceptions as strings. There's no way to pass more information in your exception, so you have to handle your exception in more than one place - at the location the error occurred, in order to construct a sensible error message, and again in your exception handler to de-construct that string into something meaningful (unless of course all you want your exception handler to do is dump the error to the browser).

A little known fact about exceptions in perl 5.005 is that you can call `die` with an object. The exception handler receives that object in `$@`. This is how you are advised to handle exceptions now, as it provides an extremely flexible and scalable exceptions solution.

7.11.3.1 A Little Housekeeping

First though, before we delve into the details, a little housekeeping is in order. Most, if not all, `mod_perl` programs consist of a main routine that is entered, and then dispatches itself to a routine depending on the parameters passed and/or the form values. In a normal C program this is your `main()` function, in a `mod_perl` handler this is your `handler()` function/method.

In order for you to be able to use exception handling to its best extent you need to change your script to have some sort of global exception handling. This is much more trivial than it sounds. If you're using `Apache::Registry` to emulate CGI you might consider wrapping your entire script in one big `eval` block, but I would discourage that. A better method would be to modularise your script into discrete function calls, one of which should be a dispatch routine:

```
#!/usr/bin/perl -w
# Apache::Registry script

eval {
    dispatch();
};
catch($@);

sub dispatch {
    ...
}

sub catch {
    my $exception = shift;
    ...
}
```

This is easier with an ordinary `mod_perl` handler as it is natural to have separate functions, rather than a long run-on script:

```
MyHandler.pm
-----
sub handler {
    my $r = shift;

    eval {
        dispatch($r);
    };
    catch ($@);
}

sub dispatch {
    my $r = shift;
    ...
}

sub catch {
    my $exception = shift;
    ...
}
```

Now that the skeleton code is setup, let's create an exception class, making use of Perl 5.005's ability to throw exception objects.

7.11.3.2 An Exception Class

This is a really simple exception class, that does nothing but contain information. A better implementation would probably also handle its own exception conditions, but that would be more complex, requiring separate packages for each exception type.

```
My/Exception.pm
-----
package My::Exception;

sub AUTOLOAD {
    my ($package, $filename, $line) = caller;
    no strict 'refs', 'subs';
    if ($AUTOLOAD =~ /\.*::([A-Z]\w+)$/ ) {
        my $exception = $1;
        *{$AUTOLOAD} =
            sub {
                shift;
                push @_, caller => {
                    package => $package,
                    filename => $filename,
                    line => $line,
                };
                bless { @_ }, "My::Exception::$exception";
            };
        goto &{$AUTOLOAD};
    }
    else {
        die "No such exception class: $AUTOLOAD\n";
    }
}

1;
```

OK, so this is all highly magical, but what does it do? It creates a simple package that we can import and use as follows:

```
use My::Exception;

die My::Exception->SomeException( foo => "bar" );
```

The exception class tracks exactly where we died from using the `caller()` mechanism, it also caches exception classes so that `AUTOLOAD` is only called the first time (in a given process) an exception of a particular type is thrown (particularly relevant under `mod_perl`).

7.11.4 Catching Uncaught Exceptions

What about exceptions that are thrown outside of your control? We can fix this using one of two possible methods. The first is to override `die` globally using the old magical `$SIG{__DIE__}`, and the second, is the cleaner non-magical method of overriding the global `die()` method to your own `die()` method that throws an exception that makes sense to your application.

7.11.4.1 Using \$SIG{__DIE__}

Overloading using \$SIG{__DIE__} in this case is rather simple, here's some code:

```
$SIG{__DIE__} = sub {  
    my $err = shift;  
    if(!ref $err) {  
        $err = My::Exception->UnCaught(text => $err);  
    }  
    die $err;  
};
```

All this does is catch your exception and re-throw it. It's not as dangerous as we stated earlier that \$SIG{__DIE__} can be, because we're actually re-throwing the exception, rather than catching it and stopping there.

There's only one slight buggette left, and that's if some external code die() 'ing catches the exception and tries to do string comparisons on the exception, as in:

```
eval {  
    ... # some code  
    die "FATAL ERROR!\n";  
};  
if ($@) {  
    if ($@ =~ /^FATAL ERROR/) {  
        die $@;  
    }  
}
```

In order to deal with this, we can overload stringification for our My::Exception::UnCaught class:

```
{  
    package My::Exception::UnCaught;  
    use overload '""' => \&str;  
  
    sub str {  
        shift->{text};  
    }  
}
```

We can now let other code happily continue.

7.11.4.2 Overriding the Core die() Function

So what if we don't want to touch \$SIG{__DIE__} at all? We can overcome this by overriding the core die function. This is slightly more complex than implementing a \$SIG{__DIE__} handler, but is far less magical, and is the right thing to do, according to the perl5-porters mailing list.

Overriding core functions has to be done from an external package/module. So we're going to add that to our My::Exception module. Here's the relevant parts:

```

use vars qw/@ISA @EXPORT/;
use Exporter;

@EXPORT = qw/die/;
@ISA = 'Exporter';

sub import {
    my $pkg = shift;
    $pkg->export('CORE::GLOBAL', 'die');
    Exporter::import($pkg,@_);
}

sub die {
    if (!ref($_[0])) {
        CORE::die My::Exception->UnCaught(text => join('', @_));
    }
    CORE::die $_[0];
}

```

That wasn't so bad, was it? We're relying on Exporter's export function to do the hard work for us, exporting the die() function into the CORE::GLOBAL namespace. Along with the above overloaded stringification, we now have a complete exception system (well, mostly complete. Exception die-hards would argue that there's no "finally" clause, and no exception stack, but that's another topic for another time).

7.11.5 *Some Uses*

I'm going to come right out and say now: I abuse this system horribly! I throw exceptions all over my code, not because I've hit an exceptional bit of code, but because I want to get straight back out of the current function, without having to have every single level of function call check error codes. One way I use this is to return Apache return codes:

```

# paranoid security check
die My::Exception->RetCode(code => 204);

```

Returns a 204 error code (HTTP_NO_CONTENT), which is caught at my top level exception handler:

```

if ($@->isa('My::Exception::RetCode')) {
    return $@->{code};
}

```

That last return statement is in my handler() method, so that's the return code that Apache actually sends. I have other exception handlers in place for sending Basic Authentication headers and Redirect headers out. I also have a generic My::Exception::OK class, which gives me a way to back out completely from where I am, but register that as an OK thing to do.

Why do I go to these extents? After all, code like slashcode (the code behind <http://slashdot.org>) doesn't need this sort of thing, so why should my web site? Well it's just a matter of scalability and programmer style really. There's lots of literature out there about exception handling, so I suggest doing some research.

7.11.6 Conclusions

Here I've demonstrated a simple and scalable (and useful) exception handling mechanism, that fits perfectly with your current code, and provides the programmer with excellent means to determine what has happened in his code. Some users might be worried about the overhead of such code. However in use I've found accessing the database to be a much more significant overhead, and this is used in some code delivering to thousands of users.

For similar exception handling techniques, see the section “Other Implementations”.

7.11.7 The `My::Exception` class in its entirety

```

package My::Exception

use vars qw/@ISA @EXPORT $AUTOLOAD//;
use Exporter;
@ISA = 'Exporter';
@EXPORT = qw/die/;

sub import {
    my $pkg = shift;
    $pkg->export('CORE::GLOBAL', 'die');
    Exporter::import($pkg,@_);
}

sub die {
    if (!ref($_[0])) {
        CORE::die My::Exception->UnCaught(text => join('', @_));
    }
    CORE::die $_[0];
}

{
    package My::Exception::UnCaught;
    use overload '""' => \&str;

    sub str {
        shift->{text};
    }
}

sub AUTOLOAD {
    no strict 'refs', 'subs';
    if ($AUTOLOAD =~ /\.*::([A-Z]\w+)/) {
        my $exception = $1;
        *{$AUTOLOAD} =
            sub {
                shift;
                my ($package, $filename, $line) = caller;
                push @_, caller => {
                    package => $package,
                    filename => $filename,
                    line => $line,
                };
                bless { @_ }, "My::Exception::$exception";
            };
        goto &{$AUTOLOAD};
    }
    else {
        die "No such exception class: $AUTOLOAD\n";
    }
}

1;

```

7.11.8 Other Implementations

Some users might find it very useful to have a more C++/Java like interface of try/catch functions. These are available in several forms that all work in slightly different ways. See the documentation for each module for details:

- **Error.pm**

Graham Barr's excellent OO styled "try, throw, catch" module (from CPAN).

- **Exception.pm and StackTrace.pm**

by Autarch (from <ftp://ftp.urth.org/pub/>).

`Exception` a bit cleaner than the `AUTOLOAD` method from the above examples as it can catch typos later on. Plus it lets you create actual class hierarchies for your exceptions, which could be nice if you want to create exception classes that do more stuff and then inherit from them.

- **Try.pm**

Tony Olekshy's. Adds an unwind stack. Not on CPAN (yet?).

- **Exceptions.pm**

Peter Seibel's `Exceptions` module is totally non-functional with modern Perl and has been superseded by Graham Barr's `Error` module.

;o)

8 Getting Help and Further Learning

8.1 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

8.2 Getting help

If after reading this guide and other documents listed in this section, you feel that your question is not yet answered, please ask the apache/mod_perl mailing list to help you. But first try to browse the mailing list archive. Most of the time you will find the answer for your question by searching the mailing archive, since there is a big chance someone else has already encountered the same problem and found a solution for it. If you ignore this advice, do not be surprised if your question will be left unanswered - it bores people to answer the same question more than once. It does not mean that you should avoid asking questions. Just do not abuse the available help and **RTFM** before you call for **HELP**. (You have certainly heard the infamous fable of the shepherd boy and the wolves)

8.3 Get help with mod_perl

- **mod_perl home**

<http://perl.apache.org>

- **News and Resources**

Take23: News and Resources for the mod_perl world <http://take23.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

- **'Managing and Programming 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. We expect the book to be published in 2001.

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

.

- **mod_perl Quick Reference Card**

mod_perl Pocket Reference by Andrew Ford was published by O'Reilly and Associates
<http://www.oreilly.com/catalog/modperlpr/>

You should probably get also the *Apache Pocket Reference* by the same author and the same publisher: <http://www.oreilly.com/catalog/apachepr/>

See also Andrew's collection of reference card for Apache and other programs:
<http://www.refcards.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 Khera 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 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/>

8.4 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.

8.5 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://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)

8.6 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**

8.7 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

8.8 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)

Table of Contents:

Tutorial: Getting started with mod_perl	1
mod_perl Tutorial: Agenda	3
1 Agenda	3
1.1 Agenda	4
mod_perl Tutorial: Getting Started Fast	5
2 Getting Started Fast	5
2.1 mod_perl in Four Slides	6
2.2 What is mod_perl?	6
2.3 Installation	7
2.4 Configuration	8
2.5 The "mod_perl rules" Apache::Registry Scripts	8
2.6 The "mod_perl rules" Apache Perl Module	9
2.7 Is That All I Need To Know About mod_perl?	9
mod_perl Tutorial: Server Setup Strategies	11
3 Server Setup Strategies	11
3.1 What we will learn in this chapter	12
3.2 mod_perl Deployment Overview	12
3.3 Standalone mod_perl Enabled Apache Server	12
3.4 One Plain Apache and One mod_perl-enabled Apache Servers	14
3.5 Adding a Proxy Server in http Accelerator Mode	15
3.6 Implementations of Proxy Servers	17
3.6.1 The Squid Server	17
3.6.2 Apache's mod_proxy	18
mod_perl Tutorial: Porting from CGI Scripts and mod_perl Coding Guidelines.	20
4 Porting from CGI Scripts and mod_perl Coding Guidelines.	20
4.1 What we will learn in this chapter	21
4.2 Exposing Apache::Registry secrets	21
4.2.1 The First Mystery	22
4.2.2 The Second Mystery	25
4.3 Sometimes it Works, Sometimes it Doesn't	26
4.3.1 Regular Expression Memory	26
4.4 @INC and mod_perl	26
4.5 Reloading Modules and Required Files	27
4.5.1 Restarting the server	27
4.5.2 Using Apache::StatINC for the Development Process	27
4.5.3 Using Apache::Reload	29
4.5.3.1 Caveats	30
4.6 __END__ and __DATA__ tokens	30
4.7 Output from system calls	31
4.8 Terminating requests and processes	31
4.9 die() and mod_perl	33
4.10 Global Variables Persistence	33
4.11 Command line Switches (-w, -T, etc)	33
4.11.1 Warnings	34
4.11.2 Taint Mode	35
4.11.3 Other switches	35
mod_perl Tutorial: RDBMS and mod_perl	37
5 RDBMS and mod_perl	37

5.1 Apache::DBI - Initiate a persistent database connection	38
5.1.1 Introduction	38
5.1.2 Configuration	39
5.1.3 Preopening DBI connections	39
5.1.4 Debugging Apache::DBI	39
5.1.5 Opening connections with different parameters	40
5.1.6 Caching prepare() Statements	40
mod_perl Tutorial: Performance Tuning	41
6 Performance Tuning	41
6.1 What we will learn in this chapter	42
6.2 The Big Picture	42
6.3 Essential Tools	43
6.3.1 Benchmarking Perl Code	43
6.3.2 Benchmarking Response Times	44
6.3.2.1 ApacheBench	44
6.3.2.2 httpperf	45
6.3.3 Using LWP::Parallel::UserAgent	45
6.4 Choosing MaxClients	49
6.5 KeepAlive	51
6.6 Be carefull with symbolic links	52
6.7 Limiting the Size of the Processes	52
6.8 Sharing Memory	53
6.9 How Shared My Memory Is	53
6.10 Keeping the Shared Memory Limit	54
6.11 Preload Perl modules at server startup	54
6.11.0.1 Modules Initialization	55
6.12 Preload Registry Scripts	59
6.13 Upload/Download of Big Files	59
6.14 Global vs Fully Qualified Variables	60
6.15 Forking or Executing subprocesses from mod_perl	60
6.15.1 Freeing the Parent Process	62
6.15.2 Detaching the Forked Process	63
6.15.3 Avoiding Zombie Processes	63
6.15.4 A Complete Fork Example	65
6.16 Sending plain HTML as a compressed output	67
mod_perl Tutorial: Perl Reference	69
7 Perl Reference	69
7.1 What we will learn in this chapter	70
7.2 perldoc's Rarely Known But Very Useful Options	70
7.3 Tracing Warnings Reports	71
7.4 Variables Globally, Lexically Scoped And Fully Qualified	73
7.5 my() Scoped Variable in Nested Subroutines	74
7.5.1 The Poison	74
7.5.2 The Diagnosis	75
7.5.3 The Remedy	76
7.6 When You Cannot Get Rid of The Inner Subroutine	77
7.6.1 Remedies for Inner Subroutines	79
7.7 use(), require(), do(), %INC and @INC Explained	85
7.7.1 The @INC array	85
7.7.2 The %INC hash	85

7.7.3 Modules, Libraries and Files	87
7.7.4 require()	88
7.7.5 use()	90
7.7.6 do()	91
7.8 Using Global Variables and Sharing Them Between Modules/Packages	92
7.8.1 Making Variables Global	92
7.8.2 Making Variables Global With strict Pragma On	92
7.8.3 Using Exporter.pm to Share Global Variables	92
7.8.4 Using the Perl Aliasing Feature to Share Global Variables	94
7.9 The Scope of the Special Perl Variables	96
7.10 Compiled Regular Expressions	96
7.11 Exception Handling for mod_perl	99
7.11.1 Trapping Exceptions in Perl	99
7.11.2 Alternative Exception Handling Techniques	99
7.11.3 Better Exception Handling	100
7.11.3.1 A Little Housekeeping	101
7.11.3.2 An Exception Class	102
7.11.4 Catching Uncaught Exceptions	102
7.11.4.1 Using \$SIG{__DIE__}	103
7.11.4.2 Overriding the Core die() Function	103
7.11.5 Some Uses	104
7.11.6 Conclusions	105
7.11.7 The My::Exception class in its entirety	105
7.11.8 Other Implementations	106
mod_perl Tutorial: Getting Help and Further Learning	108
8 Getting Help and Further Learning	108
8.1 What we will learn in this chapter	109
8.2 Getting help	109
8.3 Get help with mod_perl	109
8.4 Get help with Perl	111
8.5 Get help with Perl/CGI	112
8.6 Get help with Apache	112
8.7 Get help with DBI	113
8.8 Get help with Squid - Internet Object Cache	113