1 CGI to mod_perl Porting. mod_perl Coding guidelines.

1 CGI to mod_perl Porting. mod_perl Coding guidelines.

1.1 Description

This chapter is relevant to both writing a new CGI script or perl handler from scratch and migrating an application from plain CGI to mod_perl.

It also addresses the situation where the CGI script being ported does the job, but is too dirty to be altered easily to run as a mod_perl program. (`Apache::PerlRun` mode)

If you are at the porting stage, you can use this chapter as a reference for possible problems you might encounter when running an existing CGI script in the new mode.

If your project schedule is tight, I would suggest converting to mod_perl in the following steps: Initially, run all the scripts in the `Apache::PerlRun` mode. Then as time allows, move them into `Apache::Registry` mode. Later if you need Apache Perl API functionality you can always add it.

If you are about to write a new CGI script from scratch, it would be a good idea to learn about possible mod_perl related pitfalls and to avoid them in the first place.

If you don't need mod_cgi compatibility, it's a good idea to start writing using the mod_perl API in first place. This will make your application a little bit more efficient and it will be easier to use the full mod_perl feature set, which extends the core Perl functionality with Apache specific functions and overridden Perl core functions that were reimplemented to work better in mod_perl environment.

1.2 Before you start to code

It can be a good idea to tighten up some of your Perl programming practices, since mod_perl doesn't tolerate sloppy programming.

This chapter relies on a certain level of Perl knowledge. Please read through the Perl Reference chapter and make sure you know the material covered there. This will allow me to concentrate on pure mod_perl issues and make them more prominent to the experienced Perl programmer, which would otherwise be lost in the sea of Perl background notes.

Additional resources:

- **Perl Module Mechanics**

  This page describes the mechanics of creating, compiling, releasing, and maintaining Perl modules. http://world.std.com/~swmcd/steven/perl/module_mechanics.html

  The information is very relevant to a mod_perl developer.

- **The Eagle Book**

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

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

- **"Programming Perl" Book**
- **"Perl Cookbook" Book**
- **"Object Oriented Perl" Book**

1.3 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:
----------
#!/usr/bin/perl -w
use strict;

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

my $counter = 0; # Explicit initialization technically redundant

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.

1.3.1 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 my() Scoped Variable in Nested Subroutines.

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, if you have loaded the `Carp` module already, *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. To load the Carp module, you can add:

```
use Carp;
```

in your *startup.pl* file or in the executed code.

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; # Explicit initialization technically redundant

  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 all your code into modules, which the main script `use()`s, this effect doesn't occur.

Do not use Perl4-style libraries. Subroutines in such libraries will only be available to the first script in any given interpreter thread to `require()` a library of any given name. This can lead to confusing sporadic failures.

The easiest and the fastest way to solve the nested subroutines problem is to switch every lexically scoped variable foe which you get the warning for to a package variable. The `handler` subroutines are never called re-entrantly and each resides in a package to itself. Most of the usual disadvantates of package scoped variables are, therefore, not a concern. Note, however, that whereas explicit initialization is not always necessary for lexical variables it is usually necessary for these package variables as they persist in subsequent executions of the handler and unlike lexical variables, don't get automatically destroyed at the end of each handler.

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

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

# In Perl <5.6 our() did not exist, so:
#   use vars qw($counter);
our $counter = 0; # Explicit initialization now necessary

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

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

If the variable contains a reference it may hold onto lots of unecessary memory (or worse) if the reference is left to hang about until the next call to the same handler. For such variables you should use `local` so that the value is removed when the `handler` subroutine exits.

```
my $query = CGI->new;
```

becomes:

```
local our $query = CGI->new;
```

All this is very interesting 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' discusses many 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.

1.3.2 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`. Refer to the section "Tracing Warnings Reports" to learn about warnings in general and to the "Warnings" section to learn how to turn them on and off under mod_perl.

1.4 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.

Let’s look at three real world examples:

1.4.1 An Easy Break-in

The first example is amazing--Web Services. Imagine that you enter some site where you have an account, perhaps a free email account. Having read your own mail you decide to take a look at someone else’s.

You type in the username you want to peek at and a dummy password and try to enter the account. On some services this will work!!!

You say, why in the world does this happen? The answer is simple: **Global Variables**. You have entered the account of someone who happened to be served by the same server child as you. Because of sloppy programming, a global variable was not reset at the beginning of the program and voila, you can easily peek into someone else’s email! Here is an example of sloppy code:

```
use vars ($authenticated);
my $q = new CGI;
my $username = $q->param('username');
my $passwd   = $q->param('passwd');
authenticate($username,$passwd);
  # failed, break out
unless ($authenticated){
  print "Wrong passwd";
  exit;
}
```

```
  # user is OK, fetch user's data
show_user($username);

sub authenticate{
  my ($username,$passwd) = @_;
  # some checking
  $authenticated = 1 if SOME_USER_PASSWD_CHECK_IS_OK;
}
```

Do you see the catch? With the code above, I can type in any valid username and any dummy password and enter that user's account, provided she has successfully entered her account before me using the same child process! Since `$authenticated` is global--if it becomes 1 once, it'll stay 1 for the remainder of the child's life!!! The solution is trivial--reset `$authenticated` to 0 at the beginning of the program.

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

```
my $q = CGI->new;
my $username = $q->param('username');
my $passwd   = $q->param('passwd');
my $authenticated = authenticate($username,$passwd);
  # failed, break out
unless ($authenticated){
  print "Wrong passwd";
  exit;
}
  # user is OK, fetch user's data
show_user($username);

sub authenticate{
  my ($username,$passwd) = @_;
  # some checking
  return (SOME_USER_PASSWD_CHECK_IS_OK) ? 1 : 0;
}
```

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

1.4.2 Thinking mod_cgi

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

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

But using mod_perl, and if your system administrator with superuser access rights has previously used the system, anybody who is lucky enough to be served later by the same child which served your administrator will happen to gain the same rights.

The obvious fix is:

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

1.4.3 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.

To solve this particular `/o` modifier problem refer to Compiled Regular Expressions.

1.5 Script's name space

Scripts under `Apache::Registry` do not run in package `main`, they run in a unique name space based on the requested URI. For example, if your URI is `/perl/test.pl` the package will be called `Apache::ROOT::perl::test_2epl`.

1.6 @INC and mod_perl

The basic Perl `@INC` behaviour is explained in section use(), require(), do(), %INC and @INC Explained.

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 `PerlTaintCheck` mode turned on.

- In the startup file directly alter the `@INC`. For example

```
startup.pl
----------
use lib qw(/home/httpd/perl /home/httpd/mymodules);
1;
```

and load the startup file from the configuration file by:

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

1.7 Reloading Modules and Required Files

You might want to read the "use(), require(), do(), %INC and @INC Explained" before you proceed with this section.

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 mod_perl-enabled server to recognize changes in library modules? Well, there are a couple of techniques:

1.7.1 Restarting the server

The simplest approach is to restart the server each time you apply some change to your code. See Server Restarting techniques.

After restarting the server about 100 times, you will tire of it and you will look for other solutions.

1.7.2 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, 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.

1.7.3 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 two ways to do that.

1.7.3.1 Register Modules Implicitly

The first way is to 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.

1.7.3.2 Register Modules Explicitly

The second way is to 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.

1.7.3.3 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.

1.7.3.4 Caveats

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

Also if you have modules loaded from directories which are not in `@INC`, `Apache::Reload` will fail to find the files, due the fact that `@INC` is reset to its original value even if it gets temporary modified in the script. The solution is to extend `@INC` at the server startup to include directories you load the files from which aren't in `@INC`.

For example, if you have a script which loads *MyTest.pm* from */home/stas/myproject*:

```
use lib qw(/home/stas/myproject);
require MyTest;
```

`Apache::Reload` won't find this file, unless you alter `@INC` in *startup.pl* (or *httpd.conf*):

```
startup.pl
----------
use lib qw(/home/stas/myproject);
```

and restart the server. Now the problem is solved.

1.7.3.5 Availability

This module is available from CPAN.

1.7.4 Configuration Files: Writing, Dynamically Updating and Reloading

Checking all the modules in `%INC` on every request can add a large overhead to server response times, and you certainly would not want the `Apache::StatINC` module to be enabled in your production site's configuration. But sometimes you want a configuration file reloaded when it is updated, without restarting

the server.

This is an especially important feature if for example you have a person that is allowed to modify some of the tool configuration, but for security reasons it's undesirable for him to telnet to the server to restart it.

1.7.4.1 Writing Configuration Files

Since we are talking about configuration files, I would like to show you some good and bad approaches to configuration file writing.

If you have a configuration file of just a few variables, it doesn't really matter how you do it. But generally this is not the case. Configuration files tend to grow as a project grows. It's very relevant to projects that generate HTML files, since they tend to demand many easily configurable parameters, like headers, footers, colors and so on.

So let's start with the approach that is most often taken by CGI scripts writers. All configuration variables are defined in a separate file.

For example:

```
$cgi_dir = "/home/httpd/perl";
$cgi_url = "/perl";
$docs_dir = "/home/httpd/docs";
$docs_url = "/";
$img_dir = "/home/httpd/docs/images";
$img_url = "/images";
... many more config params here ...
$color_hint   = "#777777";
$color_warn   = "#990066";
$color_normal = "#000000";
```

The `use strict;` pragma demands that all the variables be declared. When we want to use these variables in a mod_perl script we must declare them with `use vars` in the script. (Under Perl v5.6.0 `our()` has replaced `use vars`.)

So we start the script with:

```
use strict;
use vars qw($cgi_dir $cgi_url $docs_dir $docs_url
            ... many more config params here ....
            $color_hint  $color_warn $color_normal
           );
```

It is a nightmare to maintain such a script, especially if not all the features have been coded yet. You have to keep adding and removing variable names. But that's not a big deal.

Since we want our code clean, we start the configuration file with `use strict;` as well, so we have to list the variables with `use vars` pragma here as well. A second list of variables to maintain.

If you have many scripts, you may get collisions between configuration files. One of the best solutions is to declare packages, with unique names of course. For example for our configuration file we might declare the following package name:

```
package My::Config;
```

The moment you add a package declaration and think that you are done, you realize that the nightmare has just begun. When you have declared the package, you cannot just require() the file and use the variables, since they now belong to a different package. So you have either to modify all your scripts to use a fully qualified notation like `$My::Config::cgi_url` instead of just `$cgi_url` or to import the needed variables into any script that is going to use them.

Since you don't want to do the extra typing to make the variables fully qualified, you'd go for importing approach. But your configuration package has to export them first. That means that you have to list all the variables again and now you have to keep at least three variable lists updated when you make some changes in the naming of the configuration variables. And that's when you have only one script that uses the configuration file, in the general case you have many of them. So now our example configuration file looks like this:

```
package My::Config;
use strict;

BEGIN {
  use Exporter ();

  @My::HTML::ISA       = qw(Exporter);
  @My::HTML::EXPORT    = qw();
  @My::HTML::EXPORT_OK = qw($cgi_dir $cgi_url $docs_dir $docs_url
                            ... many more config params here ....
                            $color_hint $color_warn $color_normal);
}

use vars qw($cgi_dir $cgi_url $docs_dir $docs_url
            ... many more config params here ....
            $color_hint  $color_warn $color_normal
           );

$cgi_dir = "/home/httpd/perl";
$cgi_url = "/perl";
$docs_dir = "/home/httpd/docs";
$docs_url = "/";
$img_dir = "/home/httpd/docs/images";
$img_url = "/images";
... many more config params here ...
$color_hint   = "#777777";
$color_warn   = "#990066";
$color_normal = "#000000";
```

And in the code:

```
use strict;
use My::Config qw($cgi_dir $cgi_url $docs_dir $docs_url
                  ... many more config params here ....
                  $color_hint  $color_warn $color_normal
                 );
use vars       qw($cgi_dir $cgi_url $docs_dir $docs_url
                  ... many more config params here ....
                  $color_hint  $color_warn $color_normal
                 );
```

This approach is especially bad in the context of mod_perl, since exported variables add a memory overhead. The more variables exported the more memory you use. If we multiply this overhead by the number of servers we are going to run, we get a pretty big number which could be used to run a few more servers instead.

As a matter of fact things aren't so bad. You can group your variables, and call the groups by special names called tags, which can later be used as arguments to the import() or use() calls. You are probably familiar with:

```
use CGI qw(:standard :html);
```

We can implement this quite easily, with the help of export_ok_tags() from `Exporter`. For example:

```
BEGIN {
  use Exporter ();
  use vars qw( @ISA @EXPORT @EXPORT_OK %EXPORT_TAGS);
  @ISA         = qw(Exporter);
  @EXPORT      = qw();
  @EXPORT_OK   = qw();

  %EXPORT_TAGS = (
    vars => [qw($fname $lname)],
    subs => [qw(reread_conf untaint_path)],
  );
  Exporter::export_ok_tags('vars');
  Exporter::export_ok_tags('subs');
}
```

You export subroutines exactly like variables, since what's actually being exported is a symbol. The definition of these subroutines is not shown here.

Notice that we didn't use export_tags(), as it exports the variables automatically without the user asking for them in first place, which is considered bad style. If a module automatically exports variables with export_tags() you can stop this by not exporting at all:

```
use My::Config ();
```

In your code you can now write:

```
use My::Config qw(:subs :vars);
```

Groups of group tags:

The `:all` tag from `CGI.pm` is a group tag of all other groups. It will require a little more effort to implement, but you can always save time by looking at the solution in `CGI.pm`'s code. It's just a matter of a little code to expand all the groups recursively.

After going through the pain of maintaining a list of variables in a big project with a huge configuration file (more than 100 variables) and many files actually using them, I came up with a much simpler solution: keeping all the variables in a single hash, which is built from references to other anonymous scalars, arrays and hashes.

Now my configuration file looks like this:

```
package My::Config;
use strict;

BEGIN {
  use Exporter ();

  @My::Config::ISA       = qw(Exporter);
  @My::Config::EXPORT    = qw();
  @My::Config::EXPORT_OK = qw(%c);
}

use vars qw(%c);

%c = (
   dir => {
       cgi  => "/home/httpd/perl",
       docs => "/home/httpd/docs",
       img  => "/home/httpd/docs/images",
   },
   url => {
       cgi  => "/perl",
       docs => "/",
       img  => "/images",
   },
   color => {
       hint   => "#777777",
       warn   => "#990066",
       normal => "#000000",
   },
  );
```

Good perl style suggests keeping a comma at the end of lists. That's because additional items tend to be added to the end of the list. If you keep that last comma in place, you don't have to remember to add one when you add a new item.

So now the script looks like this:

```
use strict;
use My::Config qw(%c);
use vars       qw(%c)
print "Content-type: text/plain\r\n\r\n";
print "My url docs root: $c{url}{docs}\n";
```

Do you see the difference? The whole mess has gone, there is only one variable to worry about.

There is one small downside to taking this approach: auto-vivification. For example, if we wrote `$c{url}{doc}` by mistake, perl would silently create this element for us with the value *undef*. When we `use strict;` Perl will tell us about any misspelling of this kind for a simple scalar, but this check is not performed for hash elements. This puts the onus of responsibility back on us since we must take greater care. A possible solution to this is to use pseudo-hashes, but they are still considered experimental so we won't cover them here.

The benefits of the hash approach are significant and we can make do even better. I would like to get rid of the `Exporter` stuff completely. I remove all the exporting code so my config file now looks like:

```
package My::Config;
use strict;
use vars qw(%c);

%c = (
   dir => {
       cgi  => "/home/httpd/perl",
       docs => "/home/httpd/docs",
       img  => "/home/httpd/docs/images",
   },
   url => {
       cgi  => "/perl",
       docs => "/",
       img  => "/images",
   },
   color => {
       hint   => "#777777",
       warn   => "#990066",
       normal => "#000000",
   },
  );
```

And the code:

```
use strict;
use My::Config ();
print "Content-type: text/plain\r\n\r\n";
print "My url docs root: $My::Config::c{url}{docs}\n";
```

Since we still want to save lots of typing, and since now we need to use a fully qualified notation like `$My::Config::c{url}{docs}`, let's use the magical Perl aliasing feature. I'll modify the code to be:

```
use strict;
use My::Config ();
use vars qw(%c);
*c = \%My::Config::c;
print "Content-type: text/plain\r\n\r\n";
print "My url docs root: $c{url}{docs}\n";
```

I have aliased the `*c` glob with `\%My::Config::c`, a reference to a hash. From now on, `%My::Config::c` and `%c` are the same hash and you can read from or modify either of them.

Just one last little point. Sometimes you see a lot of redundancy in the configuration variables, for example:

```
$cgi_dir  = "/home/httpd/perl";
$docs_dir = "/home/httpd/docs";
$img_dir  = "/home/httpd/docs/images";
```

Now if you want to move the base path `"/home/httpd"` into a new place, it demands lots of typing. Of course the solution is:

```
$base     = "/home/httpd";
$cgi_dir  = "$base/perl";
$docs_dir = "$base/docs";
$img_dir  = "$docs_dir/images";
```

You cannot do the same trick with a hash, since you cannot refer to its values before the definition is finished. So this wouldn't work:

```
%c =
  (
   base => "/home/httpd",
   dir => {
       cgi  => "$c{base}/perl",
       docs => "$c{base}/docs",
       img  => "$c{base}{docs}/images",
   },
  );
```

But nothing stops us from adding additional variables, which are lexically scoped with my(). The following code is correct.

```
my $base = "/home/httpd";
%c =
  (
   dir => {
       cgi  => "$base/perl",
       docs => "$base/docs",
       img  => "$base/docs/images",
   },
  );
```

You have just learned how to make configuration files easily maintainable, and how to save memory by avoiding the export of variables into a script's namespace.

1.7.4.2 Reloading Configuration Files

First, lets look at a simple case, when we just have to look after a simple configuration file like the one below. Imagine a script that tells you who is the patch pumpkin of the current Perl release.

Sidenote: *Pumpkin* A humorous term for the token (notional or real) that gives its possessor (the "pumpking" or the "pumpkineer") exclusive access to something, e.g. applying patches to a master copy of some source (for which the token is called the "patch pumpkin").

```
use CGI ();
use strict;

my $fname = "Larry";
my $lname = "Wall";
my $q = CGI->new;

print $q->header(-type=>'text/html');
print $q->p("$fname $lname holds the patch pumpkin" .
            "for this Perl release.");
```

The script has a hardcoded value for the name. It's very simple: initialize the CGI object, print the proper HTTP header and tell the world who is the current patch pumpkin.

When the patch pumpkin changes we don't want to modify the script. Therefore, we put the `$fname` and `$lname` variables into a configuration file.

```
$fname = "Gurusamy";
$lname = "Sarathy";
1;
```

Please note that there is no package declaration in the above file, so the code will be evaluated in the caller's package or in the `main::` package if none was declared. This means that the variables `$fname` and `$lname` will override (or initialize if they weren't yet) the variables with the same names in the caller's namespace. This works for global variables only--you cannot update variables defined lexically (with my()) using this technique.

You have started the server and everything is working properly. After a while you decide to modify the configuration. How do you let your running server know that the configuration was modified without restarting it? Remember we are in production and server restarting can be quite expensive for us. One of the simplest solutions is to poll the file's modification time by calling stat() before the script starts to do real work. If we see that the file was updated, we force a reconfiguration of the variables located in this file. We will call the function that reloads the configuration reread_conf() and have it accept a single argument, which is the relative path to the configuration file.

`Apache::Registry` calls a chdir() to the script's directory before it starts the script's execution. So if your CGI script is invoked under the `Apache::Registry` handler you can put the configuration file in the same directory as the script. Alternatively you can put the file in a directory below that and use a path relative to the script directory. You have to make sure that the file will be found, somehow. Be aware that do() searches the libraries in the directories in `@INC`.

```
use vars qw(%MODIFIED);
sub reread_conf{
    my $file = shift;
    return unless defined $file;
    return unless -e $file and -r _;
    my $mod = -M _;
    unless (exists $MODIFIED{$file} and $MODIFIED{$file} == $mod) {
        my $result;
        unless ($result = do $file) {
            warn "couldn't parse $file: $@" if $@;
            warn "couldn't do $file: $!"    unless defined $result;
            warn "couldn't run $file"       unless         $result;
        }
        $MODIFIED{$file} = $mod; # Update the MODIFICATION times
    }
} # end of reread_conf
```

Notice that we use the `==` comparison operator when checking file's modification timestamp, because all we want to know whether the file was changed or not.

When the require(), use() and do() operators successfully return, the file that was passed as an argument is inserted into `%INC` (the key is the name of the file and the value the path to it). Specifically, when Perl sees require() or use() in the code, it first tests `%INC` to see whether the file is already there and thus loaded. If the test returns true, Perl saves the overhead of code re-reading and re-compiling; however calling do() will (re)load regardless.

You generally don't notice with plain perl scripts, but in mod_perl it's used all the time; after the first request served by a process all the files loaded by require() stay in memory. If the file is preloaded at server startup, even the first request doesn't have the loading overhead.

We use do() to reload the code in this file and not require() because while do() behaves almost identically to require(), it reloads the file unconditionally. 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.

The configuration file can be broken if someone has incorrectly modified it. We don't want the whole service that uses that file to be broken, just because of that. We trap the possible failure to do() the file and ignore the changes, by the resetting the modification time. If do() fails to load the file it might be a good idea to send an email to the system administrator about the problem.

Notice however, that since do() updates `%INC` like require() does, if you are using `Apache::StatINC` it will attempt to reload this file before the reread_conf() call. So if the file wouldn't compile, the request will be aborted. `Apache::StatINC` shouldn't be used in production (because it slows things down by stat()'ing all the files listed in `%INC`) so this shouldn't be a problem.

Note that we assume that the entire purpose of this function is to reload the configuration if it was changed. This is fail-safe, because if something goes wrong we just return without modifying the server configuration. The script should not be used to initialize the variables on its first invocation. To do that, you would need to replace each occurrence of return() and warn() with die(). If you do that, take a look at the section "Redirecting Errors to the Client instead of error_log".

I used the above approach when I had a huge configuration file that was loaded only at server startup, and another little configuration file that included only a few variables that could be updated by hand or through the web interface. Those variables were initialized in the main configuration file. If the webmaster breaks the syntax of this dynamic file while updating it by hand, it won't affect the main (write-protected) configuration file and so stop the proper execution of the programs. Soon we will see a simple web interface which allows us to modify the configuration file without actually breaking it.

A sample script using the presented subroutine would be:

```
use vars qw(%MODIFIED $fname $lname);
use CGI ();
use strict;

my $q = CGI->new;
print $q->header(-type=>'text/plain');
my $config_file = "./config.pl";
reread_conf($config_file);
print $q->p("$fname $lname holds the patch pumpkin" .
            "for this Perl release.");

sub reread_conf{
    my $file = shift;
    return unless defined $file;
    return unless -e $file and -r _;
    my $mod = -M _;
    unless ($MODIFIED{$file} and $MODIFIED{$file} == $mod) {
        my $result;
        unless ($result = do $file) {
            warn "couldn't parse $file: $@" if $@;
            warn "couldn't do $file: $!"    unless defined $result;
            warn "couldn't run $file"       unless $result;
        }
        $MODIFIED{$file} = $mod; # Update the MODIFICATION times
    }
} # end of reread_conf
```

Remember that you should be using `(stat $file)[9]` instead of `-M $file` if you are modifying the `$^T` variable. In some of my scripts, I reset `$^T` to the time of the script invocation with `"$^T = time()"`. That way I can perform `-M` and the similar (`-A`, `-C`) file status tests relative to the script invocation time, and not the time the process was started.

If your configuration file is more sophisticated and it declares a package and exports variables, the above code will work just as well. Even if you think that you will have to import() variables again, when do() recompiles the script the originally imported variables get updated with the values from the reloaded code.

1.7.4.3 Dynamically updating configuration files

The CGI script below allows a system administrator to dynamically update a configuration file through the web interface. Combining this with the code we have just seen to reload the modified files, you get a system which is dynamically reconfigurable without needing to restart the server. Configuration can be performed from any machine having just a web interface (a simple browser connected to the Internet).

Let's say you have a configuration file like this:

```
package MainConfig;

use strict;
use vars qw(%c);

%c = (
      name     => "Larry Wall",
      release  => "5.000",
      comments => "Adding more ways to do the same thing :)",

      other    => "More config values",

      hash     => { foo => "ouch",
                    bar => "geez",
                  },

      array    => [qw( a b c)],

     );
```

You want to make the variables `name`, `release` and `comments` dynamically configurable. You want to have a web interface with an input form that allows you to modify these variables. Once modified you want to update the configuration file and propagate the changes to all the currently running processes. Quite a simple task.

Let's look at the main stages of the implementation. Create a form with preset current values of the variables. Let the administrator modify it and submit the changes. Validate the submitted information (numeric fields should carry numbers, literals--words, etc). Update the configuration file. Update the modified value in the memory of the current process. Present the form as before but with updated fields if any.

The only part that seems to be complicated to implement is a configuration file update, for a couple of reasons. If updating the file breaks it, the whole service won't work. If the file is very big and includes comments and complex data structures, parsing the file can be quite a challenge.

So let's simplify the task. If all we want is to update a few variables, why don't we create a tiny configuration file with just those variables? It can be modified through the web interface and overwritten each time there is something to be changed. This way we don't have to parse the file before updating it. If the main configuration file is changed we don't care, we don't depend on it any more.

The dynamically updated variables are duplicated, they will be in the main file and in the dynamic file. We do this to simplify maintenance. When a new release is installed the dynamic configuration file won't exist at all. It will be created only after the first update. As we just saw, the only change in the main code is to add a snippet to load this file if it exists and was changed.

This additional code must be executed after the main configuration file has been loaded. That way the updated variables will override the default values in the main file.

META: extend on the comments:

```
# remember to run this code in taint mode

use strict;
use vars qw($q %c $dynamic_config_file %vars_to_change %validation_rules);

use CGI ();

use lib qw(.);
use MainConfig ();
*c = \%MainConfig::c;

$dynamic_config_file = "./config.pl";

# load the dynamic configuration file if it exists, and override the
# default values from the main configuration file
do $dynamic_config_file if -e $dynamic_config_file and -r _;

# fields that can be changed and their titles
%vars_to_change =
  (
   'name'     => "Patch Pumpkin's Name",
   'release'  => "Current Perl Release",
   'comments' => "Release Comments",
  );

%validation_rules =
  (
   'name'     => sub { $_[0] =~ /^[\w\s\.]+$/;   },
   'release'  => sub { $_[0] =~ /^\d+\.[\d_]+$/; },
   'comments' => sub { 1;                        },
  );

$q = CGI->new;
print $q->header(-type=>'text/html'),
      $q->start_html();

my %updates = ();

# We always rewrite the dynamic config file, so we want all the
# vars to be passed, but to save time we will only do checking

# of vars that were changed.  The rest will be retrieved from
# the 'prev_foo' values.
foreach (keys %vars_to_change) {
  # copy var so we can modify it
  my $new_val = $q->param($_) || '';

  # strip a possible ^M char (DOS/WIN)
  $new_val =~ s/\cM//g;

  # push to hash if was changed
  $updates{$_} = $new_val
    if defined $q->param("prev_".$_)
       and $new_val ne $q->param("prev_".$_);
}
```

```
  # Note that we cannot trust the previous values of the variables
  # since they were presented to the user as hidden form variables,
  # and the user can mangle those. We don't care: it cannot do any
  # damage, as we verify each variable by rules which we define.

  # Process if there is something to process. Will be not called if
  # it's invoked a first time to display the form or when the form
  # was submitted but the values weren't modified (we know that by
  # comparing with the previous values of the variables, which are
  # the hidden fields in the form)

  # process and update the values if valid
  process_change_config(%updates) if %updates;

  # print the update form
  conf_modification_form();

  # update the config file but first validate that the values are correct ones
  ##########################
  sub process_change_config{
    my %updates = @_;

      # we will list here all the malformatted vars
    my %malformatted = ();

    print $q->b("Trying to validate these values<BR>");
    foreach (keys %updates) {
      print "<DT><B>$_</B> => <PRE>$updates{$_}</PRE>";

      # now we have to handle each var to be changed very carefully
      # since this file goes immediately into production!
      $malformatted{$_} = delete $updates{$_}
        unless $validation_rules{$_}->($updates{$_});

    } # end of foreach (keys %updates)

    # print warnings if there are any invalid changes
    print $q->hr,
      $q->p($q->b(qq{Warning! These variables were changed
           but found malformed, thus the original
           values will be preserved.})
     ),
      join(",<BR>",
     map { $q->b($vars_to_change{$_}) . " : $malformatted{$_}\n"
         } keys %malformatted)

        if %malformatted;

    # Now complete the vars that weren't changed from the
    # $q->param('prev_var') values
    map { $updates{$_} = $q->param('prev_'.$_) unless exists $updates{$_}
        } keys %vars_to_change;

    # Now we have all the data that should be written into the dynamic
    # config file
```

```
    # escape single quotes "'" while creating a file
  my $content = join "\n",
    map { $updates{$_} =~ s/(['\\])/\\$1/g;
    '$c{' . $_ . "}  =  '" . $updates{$_} . "';\n"
  } keys %updates;

    # now add '1;' to make require() happy
  $content .= "\n1;";

    # keep the dummy result in $res so it won't complain
  eval {my $res = $content};
  if ($@) {
    print qq{Warning! Something went wrong with config file
       generation!<P> The error was : <BR><PRE>$@</PRE>};
    return;
  }

  print $q->hr;

    # overwrite the dynamic config file
  use Symbol ();
  my $fh = Symbol::gensym();
  open $fh, ">$dynamic_config_file.bak"
    or die "Can't open $dynamic_config_file.bak for writing :$! \n";
  flock $fh,2; # exclusive lock
  seek $fh,0,0;   # rewind to the start
  truncate $fh, 0; # the file might shrink!
     print $fh $content;
  close $fh;

    # OK, now we make a real file
  rename "$dynamic_config_file.bak",$dynamic_config_file
    or die "Failed to rename: $!";

    # rerun it to update variables in the current process! Note that
    # it won't update the variables in other processes. Special
    # code that watches the timestamps on the config file will do this
    # work for each process. Since the next invocation will update the
    # configuration anyway, why do we need to load it here? The reason
    # is simple: we are going to fill the form's input fields with
    # the updated data.
  do $dynamic_config_file;

} # end sub process_change_config

############################
sub conf_modification_form{

  print $q->center($q->h3("Update Form"));

  print $q->hr,
    $q->p(qq{This form allows you to dynamically update the current
       configuration. You don\'t need to restart the server in
       order for changes to take an effect}
         );

    # set the previous settings in the form's hidden fields, so we
```

```
    # know whether we have to do some changes or not
  map {$q->param("prev_$_",$c{$_}) } keys %vars_to_change;

    # rows for the table, go into the form
  my @configs = ();

    # prepare one textfield entries
  push @configs,
    map {
      $q->td(
       $q->b("$vars_to_change{$_}:"),
      ),
      $q->td(

       $q->textfield(-name      => $_,
             -default   => $c{$_},
             -override  => 1,
             -size      => 20,
             -maxlength => 50,
            )
      ),
    } qw(name release);

    # prepare multiline textarea entries
  push @configs,
    map {
      $q->td(
       $q->b("$vars_to_change{$_}:"),
      ),
      $q->td(
       $q->textarea(-name    => $_,
            -default => $c{$_},
            -override  => 1,
            -rows    => 10,
            -columns => 50,
            -wrap    => "HARD",
            )
      ),
    } qw(comments);

  print $q->startform('POST',$q->url),"\n",
  $q->center($q->table(map {$q->Tr($_),"\n",} @configs),
         $q->submit('','Update!'),"\n",
        ),
  map ({$q->hidden("prev_".$_, $q->param("prev_".$_))."\n" }
       keys %vars_to_change), # hidden previous values
  $q->br,"\n",
  $q->endform,"\n",
  $q->hr,"\n",
  $q->end_html;

} # end sub conf_modification_form
```

Once updated the script generates a file like:

```
$c{release}  =  '5.6';

$c{name}  =  'Gurusamy Sarathy';

$c{comments}  =  'Perl rules the world!';

1;
```

1.7.5 Reloading handlers

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

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

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

1.8 Name collisions with Modules and libs

This section requires an in-depth understanding of use(), require(), do(), %INC and @INC .

To make things clear before we go into details: each child process has its own `%INC` hash which is used to store information about its compiled modules. The keys of the hash are the names of the modules and files passed as arguments to require() and use(). The values are the full or relative paths to these modules and files.

Suppose we have `my-lib.pl` and `MyModule.pm` both located at `/home/httpd/perl/my/`.

- `/home/httpd/perl/my/` is in `@INC` at server startup.

  ```
  require "my-lib.pl";
  use MyModule.pm;
  print $INC{"my-lib.pl"},"\n";
  print $INC{"MyModule.pm"},"\n";
  ```

  prints:

  ```
  /home/httpd/perl/my/my-lib.pl
  /home/httpd/perl/my/MyModule.pm
  ```

  Adding `use lib`:

  ```
  use lib qw(.);
  require "my-lib.pl";
  use MyModule.pm;
  print $INC{"my-lib.pl"},"\n";
  print $INC{"MyModule.pm"},"\n";
  ```

  prints:

```
my-lib.pl
MyModule.pm
```

- `/home/httpd/perl/my/` isn't in `@INC` at server startup.

  ```
  require "my-lib.pl";
  use MyModule.pm;
  print $INC{"my-lib.pl"},"\n";
  print $INC{"MyModule.pm"},"\n";
  ```

  wouldn't work, since perl cannot find the modules.

  Adding `use lib`:

  ```
  use lib qw(.);
  require "my-lib.pl";
  use MyModule.pm;
  print $INC{"my-lib.pl"},"\n";
  print $INC{"MyModule.pm"},"\n";
  ```

  prints:

  ```
  my-lib.pl
  MyModule.pm
  ```

Let's look at three scripts with faults related to name space. For the following discussion we will consider just one individual child process.

- **Scenario 1**

  First, You can't have two identical module names running on the same server! Only the first one found in a use() or require() statement will be compiled into the package, the request for the other module will be skipped, since the server will think that it's already compiled. This is a direct result of using `%INC`, which has keys equal to the names of the modules. Two identical names will refer to the same key in the hash. (Refer to the section 'Looking inside the server' to find out how you can know what is loaded and where.)

  So if you have two different `Foo` modules in two different directories and two scripts `script1.pl` and `script2.pl`, placed like this:

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

  Where some sample code could be:

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

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

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

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

Both scripts call `use Foo;`. Only the first one called will know about `Foo`. When you call the second script it will not know about `Foo` at all--it's like you've forgotten to write `use Foo;`. Run the server in single server mode to detect this kind of bug immediately.

You will see the following in the error_log file:

```
Undefined subroutine
&Apache::ROOT::perl::tool2::tool2_2epl::foo called at
/home/httpd/perl/tool2/tool2.pl line 4.
```

- **Scenario 2**

  If the files do not declare a package, the above is true for libraries (i.e. *my-lib.pl"*) you require() as well:

  Suppose that you have a directory structure like this:

```
./tool1/config.pl
./tool1/tool1.pl
./tool2/config.pl
./tool2/tool2.pl
```

  and both scripts contain:

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

  while *./tool1/config.pl* can be something like this:

```
$foo = 0;
1;
```

and *./tool2/config.pl*:

```
$foo = 1;
1;
```

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

- **Scenario 3**

  It is interesting that the following scenario will fail too!

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

  where `tool1.pl` and `tool2.pl` both require() the *same* `config.pl`.

There are three solutions for this:

- **Solution 1**

  The first two faulty scenarios can be solved by placing your library modules in a subdirectory structure so that they have different path prefixes. The file system layout will be something like:

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

  And modify the scripts:

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

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

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

  And each script contains respectively:

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

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

This solution isn't good, since while it might work for you now, if you add another script that wants to use the same module or `config.pl` file, it would fail as we saw in the third scenario.

Let's see some better solutions.

- **Solution 2**

  Another option is to use a full path to the script, so it will be used as a key in `%INC`;

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

  This solution solves the problem of the first two scenarios. I was surprised that it worked for the third scenario as well!

  With this solution you lose some portability. If you move the tool around in the file system you will have to change the base directory or write some additional script that will automatically update the hardcoded path after it was moved. Of course you will have to remember to invoke it.

- **Solution 3**

  Make sure you read all of this solution.

  Declare a package name in the required files! It should be unique in relation to the rest of the package names you use. `%INC` will then use the unique package name for the key. It's a good idea to use at least two-level package names for your private modules, e.g. `MyProject::Carp` and not `Carp`, since the latter will collide with an existing standard package. Even though a package may not exist in the standard distribution now, a package may come along in a later distribution which collides with a name you've chosen. Using a two part package name will help avoid this problem.

  Even a better approach is to use three level naming, like `CompanyName::Project-Name::Module`, which is most unlikely to have conflicts with later Perl releases. Foresee problems like this and save yourself future trouble.

  What are the implications of package declaration?

  Without package declarations, it is very convenient to use() or require() files because all the variables and subroutines are part of the `main::` package. Any of them can be used as if they are part of the main script. With package declarations things are more awkward. You have to use the `Package::function()` method to call a subroutine from `Package` and to access a global variable `$foo` inside the same package you have to write `$Package::foo`.

  Lexically defined variables, those declared with my() inside `Package` will be inaccessible from outside the package.

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

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

You can export both subroutines and global variables. Note however that this method has the disadvantage of consuming more memory for the current process.

See `perldoc Exporter` for information about exporting other variables and symbols.

This completely covers the third scenario. When you use different module names in package declarations, as explained above, you cover the first two as well.

- **A Hack**

  The following solution should be used only as a short term bandaid. You can force reloading of the modules by either fiddling with `%INC` or replacing use() and require() calls with do().

  If you delete the module entry from the `%INC` hash, before calling require() or use() the module will be loaded and compiled again. For example:

```
./project/runA.pl
-----------------
BEGIN {
    delete $INC{"MyConfig.pm"};
}
use lib qw(.);
use MyConfig;
print "Content-type: text/plain\n\n";
print "Script A\n";
print "Inside project: ", project_name();
```

  Apply the same fix to *runB.pl*.

  Another alternative is to force module reload via do():

```
./project/runA.pl
-----------------
use lib qw(.);
do "MyConfig.pm";
print "Content-type: text/plain\n\n";
print "Script B\n";
print "Inside project: ", project_name();
```

  Apply the same fix to *runB.pl*.

  If you needed to import() something from the loaded module, call the import() method explicitly. For example if you had:

```
use MyConfig qw(foo bar);
```

  now the code will look as:

```
do "MyConfig.pm";
MyConfig->import(qw(foo bar));
```

Both presented solutions are ineffective, since the modules in question will be reloaded on each request, slowing down the response times. Therefore use these only when a very quick fix is needed and provide one of the more robust solutions discussed in the previous sections.

See also the `perlmodlib` and `perlmod` manpages.

From the above discussion it should be clear that you cannot run development and production versions of the tools using the same apache server! You have to run a separate server for each. They can be on the same machine, but the servers will use different ports.

1.9 More package name related issues

If you have the following:

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

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

1.10 __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 error:

```
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.

1.11 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. See the "Forking or Executing subprocesses from mod_perl" section to learn about implications of forking.

Also read about Apache::SubProcess for overridden system() and exec() implementations that work with mod_perl.

1.12 Using format() and write()

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

Otherwise you could use sprintf() to replace format(): `##.##` becomes `%2.2f` and `####.##` becomes `%4.2f`.

Pad all strings with (" " x 80) before using, and set their length with: %.25s for a max 25 char string. Or prefix the string with (" " x 80) for right-justifying.

Another alternative is to use the `Text::Reform` module.

1.13 Terminating requests and processes, the exit() and child_terminate() functions

Perl's `exit()` built-in function (all versions prior to 5.6) 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. Starting from Perl version 5.6 mod_perl will override exit() behind the scenes, using `CORE::GLOBAL::`, a new *magical* package.

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:

```
use constant IS_MODPERL => $ENV{MOD_PERL};
use subs qw(exit);
# Select the correct exit function
*exit = IS_MODPERL ? \&Apache::exit : sub { CORE::exit };
```

Now the correct `exit()` is always chosen, whether the script is running under mod_perl, ordinary CGI or from the shell. Notice that since we are using the constant pragma, there is no runtime overhead to select one of the code references, since `IS_MODPERL` constant is folded, that block is optimized away at compile time outside of mod_perl.

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.

1.14 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 interperter 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 interpeter 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(); }
```

1.15 Return Codes

`Apache::Registry` normally assumes a return code of `OK` (`200`). If you want to send another return code, use `$r->status()`:

```
use Apache::Constants qw(NOT_FOUND);
$r->status(NOT_FOUND);
```

Of course if you do that, you don't have to call `$r->send_http_header()` (assuming that you have `PerlSendHeader Off`).

1.16 Testing the Code from the Shell

Your CGI scripts will **not** yet run from the command line unless you use `CGI::Switch` or `CGI.pm` and have Perl 5.004 or later. They must not make any direct calls to Apache's Perl API methods.

1.17 I/O is different

If you are using Perl 5.004 or later, most CGI scripts can run under mod_perl untouched.

If you're using 5.003, Perl's built-in `read()` and `print()` functions do not work as they do under CGI. If you're using `CGI.pm`, use `$query->print` instead of plain ol' `print()`.

1.18 STDIN, STDOUT and STDERR streams

In mod_perl both `STDIN` and `STDOUT` are tied to the socket the request came from. Because the C level STDOUT is not hooked up to the client, you can re-open the STDOUT filehandler using tie(). For example if you want to dup an STDOUT filehandler and for the code to work with mod_perl and without it, the following example will do:

```
use constant IS_MODPERL => $ENV{MOD_PERL};
if (IS_MODPERL) {
  tie *OUT, 'Apache';
} else {
  open (OUT, ">-");
}
```

Note that `OUT` was picked just as an example -- there is nothing special about it. If you are looking to redirect the STDOUT stream into a scalar, see the Redirecting STDOUT into a String section.

`STDERR` is tied to the file defined by the `ErrorLog` directive.

1.19 Redirecting STDOUT into a Scalar

Sometimes you have a situation where a black box functions prints the output to `STDOUT` and you want to get this output into a scalar. This is just as valid under mod_perl, where you want the `STDOUT` to be tied to the `Apache` object. So that's where the `IO::String` package comes to help. You can re-tie() the STDOUT (or any other filehandler to a string) by doing a simple select() on the `IO::String` object and at the end to re-tie() the STDOUT back to its original stream:

```
my $str;
my $str_fh = IO::String->new($str);
my $old_fh = select($str_fh);

# some function that prints to currently selected file handler.
print_stuff()

# reset default fh to previous value
select($old_fh) if defined $old_fh;
```

1.20 Apache::print() and CORE::print()

Under mod_perl `CORE::print()` will redirect its data to `Apache::print()` since the STDOUT filehandle is tied to the *Apache* module. This allows us to run CGI scripts unmodified under `Apache::Registry` by chaining the output of one content handler to the input of the other handler.

`Apache::print()` behaves mostly like the built-in *print()* function. In addition it sets a timeout so that if the client connection is broken the handler won't wait forever trying to print data downstream to the client.

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

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

If you still want to print the reference you can always call:

```
$r->print(\\$foo);
```

or by forcing it into a scalar context:

```
print(scalar($foo));
```

1.21 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. See "Variables globally, lexically scoped and fully qualified" for more information.

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.

Here is an example with Perl hash variables, which store the iteration state in the hash variable and that state persists between requests unless explicitly reset. Consider the following registry script:

```
#file:hash_iteration.pl
#----------------------
our %hash;
%hash = map {$_ => 1 } 'a'..'c' unless %hash;

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

for (my ($k, $v) = each %hash) {
    print "$k $v\n";
    last;
}
```

That script prints different values on the first 3 invocations and prints nothing on the 4th, and then repeats the loop. (when you run with httpd -X). There are 3 hash key/value pairs in the global variable `%hash`.

In order to get the iteration state to its initial state at the beginning of each request, you need to reset the iterator as explained in the manpage for the `each()` operator. So adding:

```
keys %hash;
```

before using `%hash` solves the problem for the current example.

1.22 Generating correct HTTP Headers

A HTTP response header consists of at least two fields. HTTP response and MIME type header `Content-type`:

```
HTTP/1.0 200 OK
Content-Type: text/plain
```

After adding one more new line, you can start printing the content. A more complete response includes the date timestamp and server type, for example:

```
HTTP/1.0 200 OK
Date: Tue, 28 Dec 1999 18:47:58 GMT
Server: Apache/1.3.10-dev (Unix) mod_perl/1.21_01-dev
Content-Type: text/plain
```

To notify that the server was configured with KeepAlive Off, you need to tell the client that the connection was closed, with:

```
Connection: close
```

There can be other headers as well, like caching control and others specified by the HTTP protocol. You can code the response header with a single print():

```
  print qq{HTTP/1.1 200 OK
Date: Tue, 28 Dec 1999 18:49:41 GMT
Server: Apache/1.3.10-dev (Unix) mod_perl/1.21_01-dev
Connection: close
Content-type: text/plain

  };
```

or with a *"here"* style print:

```
  print <<EOT;
HTTP/1.1 200 OK
Date: Tue, 28 Dec 1999 18:49:41 GMT
Server: Apache/1.3.10-dev (Unix) mod_perl/1.21_01-dev
Connection: close
Content-type: text/plain

EOT
```

Notice the double new line at the end. But you have to prepare a timestamp string (`Apache::Util::ht_time()` does just this) and to know what server you are running under. You needed to send only the response MIME type (`Content-type`) under mod_cgi, so why would you want to do this manually under mod_perl?

Actually sometimes you do want to set some headers manually, but not every time. So mod_perl gives you the default set of headers, just like in the example above. And if you want to override or add more headers you can do that as well. Let's see how to do that.

When writing your own handlers and scripts with the Perl Apache API the proper way to send the HTTP header is with the send_http_header() method. If you need to add or override methods you can use the header_out() method:

```
$r->header_out("Server" => "Apache Next Generation 10.0");
$r->header_out("Date" => "Tue, 28 Dec 1999 18:49:41 GMT");
```

When you have prepared all the headers you send them with:

```
$r->send_http_header;
```

Some headers have special aliases:

```
$r->content_type('text/plain');
```

is the same as:

```
$r->header_out("Content-type" => "text/plain");
```

A typical handler looks like this:

```
$r->content_type('text/plain');
$r->send_http_header;
return OK if $r->header_only;
```

If the client issues an HTTP `HEAD` request rather than the usual `GET`, to be compliant with the HTTP protocol we should not send the document body, but only the HTTP header. When Apache receives a `HEAD` request, *header_only()* returns *true*. If we see that this has happened, we return from the handler immediately with an `OK` status code.

Generally, you don't need the explicit content type setting, since Apache does this for you, by looking up the MIME type of the request and by matching the extension of the URI in the MIME tables (from the *mime.types* file). So if the request URI is */welcome.html*, the `text/html` content-type will be picked. However for CGI scripts or URIs that cannot be mapped by a known extension, you should set the appropriate type by using content_type() method.

The situation is a little bit different with `Apache::Registry` and similar handlers. If you take a basic CGI script like this:

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

it wouldn't work, because the HTTP header will not be sent out. By default, mod_perl does not send any headers itself. You may wish to change this by adding

```
PerlSendHeader On
```

in the `Apache::Registry` `<Location>` section of your configuration. Now, the response line and common headers will be sent as they are by mod_cgi. Just as with mod_cgi, `PerlSendHeader` will not send the MIME type and a terminating double newline. Your script must send that itself, e.g.:

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

According to HTTP specs, you should send "\cM\cJ", "\015\012" or "\0x0D\0x0A" string. The "\r\n" is the way to do that on UNIX and MS-DOS/Windows machines. However, on a Mac "\r\n" eq "\012\015", exactly the other way around.

Note, that in most UNIX CGI scripts, developers use a simpler "\n\n" and not "\r\n\r\n". There are occasions where sending "\n" without "\r" can cause problems, make it a habit to always send "\r\n" every time.

If you use an OS which uses the EBCDIC as character set (e.g. BS2000-Posix), you should use this method to send the Content-type header:

```
shift->send_http_header('text/html');
```

The `PerlSendHeader On` directive tells mod_perl to intercept anything that looks like a header line (such as `Content-Type: text/plain`) and automatically turn it into a correctly formatted HTTP/1.0 header, the same way it happens with CGI scripts running under mod_cgi. This allows you to keep your CGI scripts unmodified.

You can use `$ENV{PERL_SEND_HEADER}` to find out whether `PerlSendHeader` is `On` or `Off`. You use it in your module like this:

```
if($ENV{PERL_SEND_HEADER}) {
    print "Content-type: text/html\r\n\r\n";
}
else {
    my $r = Apache->request;
    $r->content_type('text/html');
    $r->send_http_header;
}
```

Note that you can always use the code in the else part of the above example, no matter whether the `PerlSendHeader` directive is On or Off.

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

There is no free lunch--you get the mod_cgi behavior at the expense of the small but finite overhead of parsing the text that is sent. Note that mod_perl makes the assumption that individual headers are not split across print statements.

The `Apache::print()` routine has to gather up the headers that your script outputs, in order to pass them to `$r->send_http_header`. This happens in `src/modules/perl/Apache.xs` (`print`) and `Apache/Apache.pm` (`send_cgi_header`). There is a shortcut in there, namely the assumption that each print statement contains one or more complete headers. If for example you generate a `Set-Cookie` header by multiple `print()` statements, like this:

```
print "Content-type: text/plain\n";
print "Set-Cookie: iscookietext\; ";
print "expires=Wednesday, 09-Nov-1999 00:00:00 GMT\; ";
print "path=\/\; ";
print "domain=\.mmyserver.com\; ";
print "\r\n\r\n";
print "hello";
```

Your generated `Set-Cookie` header is split over a number of print() statements and gets lost. The above example wouldn't work! Try this instead:

```
my $cookie = "Set-Cookie: iscookietext\; ";
$cookie .= "expires=Wednesday, 09-Nov-1999 00:00:00 GMT\; ";
$cookie .= "path=\/\; ";
$cookie .= "domain=\.mmyserver.com\; ";
print "Content-type: text/plain\n",
print "$cookie\r\n\r\n";
print "hello";
```

Of course using a special purpose cookie generator modules, like `Apache::Cookie`, `CGI::Cookie` etc is an even cleaner solution.

Sometimes when you call a script you see an ugly "Content-Type: text/html" displayed at the top of the page, and of course the rest of the HTML code won't be rendered correctly by the browser. As you have seen above, this generally happens when your code has already sent the header so you see the duplicate header rendered into the browser's page. This might happen when you call the `CGI.pm` `$q->header` method or mod_perl's `$r->send_http_header`.

If you have a complicated application where the header might be generated from many different places, depending on the calling logic, you might want to write a special subroutine that sends a header, and keeps track of whether the header has been already sent. Of course you can use a global variable to flag that the header has already been sent:

```
use strict;
use vars qw{$header_printed};
$header_printed = 0;

print_header("text/plain");
print "It worked!\n";
print_header("text/plain");

sub print_header {
  my $type = shift || "text/html";
  unless  ($header_printed) {
    $header_printed = 1;
    my $r = Apache->request;
    $r->content_type($type);
    $r->send_http_header;
  }
}
```

`$header_printed` is the variable that flags whether the header was sent or not and it gets initialized to false (0) at the beginning of each code invocation. Note that the second invocation of print_header() within the same code, will do nothing, since `$header_printed` will become true after print_header() will be executed for the first time.

A solution that is a little bit more memory friendly is to use a fully qualified variable instead:

```
use strict;
$main::header_printed = 0;

print_header("text/plain");
print "It worked!\n";
print_header("text/plain");

sub print_header {
  my $type = shift || "text/html";
  unless  ($main::header_printed) {
    $main::header_printed = 1;
    my $r = Apache->request;
```

```
    $r->content_type($type);
    $r->send_http_header;
  }
}
```

We just removed the global variable predeclaration, which allowed us to use `$header_printed` under `"use strict"` and replaced `$header_printed` with `$main::header_printed`;

You may become tempted to use a more elegant Perl solution--the nested subroutine effect which seems to be a natural approach to take here. Unfortunately it will not work. If the process was starting fresh for each script or handler, like with plain mod_cgi scripts, it would work just fine:

```
use strict;

print_header("text/plain");
print "It worked!\n";
print_header("text/plain");

{
  my $header_printed = 0;
  sub print_header {
    my $type = shift || "text/html";
    unless  ($header_printed) {
      $header_printed = 1;
      my $r = Apache->request;
      $r->content_type($type);
      $r->send_http_header;
    }
  }
}
```

In this code `$header_printed` is declared as lexically scoped (with my()) outside the subroutine print_header() and modified inside of it. Curly braces define the block which limits the scope of the lexically variable.

This means that once print_header() sets it to 1, it will stay 1 as long as the code is running. So all subsequent calls to this subroutine will just return without doing a thing. This would serve our purpose, but unfortunately it will work only for the first time the script is invoked within a process. When the script is executed for the second or subsequent times and is served by the same process--the header will not be printed anymore, since print_header() will remember that the value of `$header_printed` is equal to 1--it won't be reinitialized, since the subroutine won't be recompiled.

Why can't we use a lexical without hitting the nested subroutine effect? Because when we've discussed Apache::Registry secrets we have seen that the code is wrapped in a `handler` routine, effectively turning any subroutines within the file a script resides in into nested subroutines. Hence we are forced to use a global in this situation.

Let's make our smart method more elaborate with respect to the `PerlSendHeader` directive, so that it always does the right thing. It's especially important if you write an application that you are going to distribute, hopefully under one of the Open Source or GPL licenses.

You can continue to improve this subroutine even further to handle additional headers, such as cookies.

See also Correct Headers--A quick guide for mod_perl users

1.23 NPH (Non Parsed Headers) scripts

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

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

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

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

1.24 BEGIN blocks

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

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

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

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

- Only once, if pulled in by the parent process via

  `Apache::RegistryLoader`.
- Once per-child process if not pulled in by the parent process.

- An additional time, once per child process, each time the script file changes on disk.
- An additional time, in the parent process on each restart if pulled in by the parent process via `Apache::RegistryLoader` and `PerlFreshRestart` is `On`.

Make sure you read Evil things might happen when using PerlFreshRestart.

1.25 END blocks

As the `perlmod` manpage explains, an `END` subroutine is executed as late as possible, that is, when the interpreter exits. In the mod_perl environment, the interpreter does not exit until the server shuts down. However, mod_perl does make a special case for `Apache::Registry` scripts.

Normally, `END` blocks are executed by Perl during its `perl_run()` function. This is called once each time the Perl program is executed, i.e. under mod_cgi, once per invocation of the CGI script. However, mod_perl only calls `perl_run()` once, during server startup. Any `END` blocks encountered during main server startup, i.e. those pulled in by `PerlRequire`, `PerlModule` and the startup file, are suspended.

Except during the cleanup phase, any `END` blocks encountered during compilation of `Apache::Registry` scripts (including those defined in the packages `use()`'d by the script), including subsequent invocations when the script is cached in memory, are called after the script has completed.

All other `END` blocks encountered during other `Perl*Handler` call-backs, e.g. `PerlChildInitHandler`, will be suspended while the process is running and called during `child_exit()` when the process is shutting down. Module authors might wish to use `$r->register_cleanup()` as an alternative to `END` blocks if this behavior is not desirable. `$r->register_cleanup()` is called at the CleanUp processing phase of each request and thus can be used to emulate plain perl's `END{}` block behavior.

The last paragraph is very important for handling the case of 'User Pressed the Stop Button'.

If you only want something to run once in the parent on shutdown or restart you can use `$r->register_cleanup()` in the *startup.pl*.

```
#PerlRequire startup.pl
warn "parent pid is $$\n";
Apache->server->register_cleanup
  (sub { warn "server cleanup in $$\n"});
```

This is usually useful when some server wide cleanup should be performed when the server is stopped or restarted.

1.26 CHECK And INIT Blocks

These blocks run when compilation is complete, but before the program starts. `CHECK` can mean "checkpoint" or "double-check" or even just "stop". `INIT` stands for "initialization". The difference is subtle; `CHECK` blocks are run just after the compilation ends, `INIT` just before the runtime begins. (Hence the `-c` command-line flag to perl runs `CHECK` blocks but not `INIT` blocks.)

Perl only calls these blocks during perl_parse(), which mod_perl calls once at startup time. Therefore `CHECK` and `INIT` blocks don't work for the same reason these don't:

```
% perl -e 'eval qq(CHECK { print "ok\n" })'
% perl -e 'eval qq(INIT  { print "ok\n" })'
```

1.27 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.

1.27.1 Warnings

There are three ways to enable warnings:

- **Globally to all Processes**

  Setting:

  ```
  PerlWarn On
  ```

  in `httpd.conf` will turn warnings `On` in any script.

  You can then fine tune your code, turning warnings `Off` and `On` by using the `warnings` pragma in your scripts (or by setting the `$^W` variable, if you prefer to be compatible with older, pre-5.6, perls).

- **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 with `no warnings;` and `use warnings;` as noted above.

- **Locally to a block**

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

  ```
  {
    use warnings;
    # some code
  }
  # back to the previous mode here
  ```

This turns it `Off`:

```
{
  no warnings;
  # some code
}
# back to the previous mode here
```

This turns `Off` only the warnings from the listed categories : (warnings categories are explicited in the `perldiag` manpage.)

```
{
  no warnings qw(uninitialized unopened);
  # some code
}
# back to the previous mode here
```

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

```
use warnings;
```

at the beginning of the file.

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.

1.27.2 Taint Mode

Perl's `-T` switch enables *Taint* mode. (META: Link to security chapter). 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. Also read the *re* pragma's manpage.)

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.

1.27.3 Other switches

Finally, if you still need to set additional perl startup flags such as `-d` and `-D`, you can use an environment variable `PERL5OPT`. 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.

[META: verify]

See also Apache::PerlRun.

1.28 The strict pragma

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

```
use strict;
```

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

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

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

1.29 Passing ENV variables to CGI

To pass an environment variable from *httpd.conf*, add to it:

```
PerlSetEnv key val
PerlPassEnv key
```

e.g.:

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

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

%ENV is only set up for CGI emulation. If you are using the API, you should use $r->subprocess_env, $r->notes or $r->pnotes for passing data around between handlers. %ENV is slow because it must update the underlying C environment table. It also insecure since its use exposes the data on systems which allow users to see the environment with ps.

In any case, %ENV and the tables used by those methods are all cleared after the request is served.

The Perl %ENV is cleared during startup, but the C environment is left intact. With a combo of forking 'env' and <Perl> sections you can do even do wildcards matching. For example, this passes all environment variables that begin with the letter H:

```
<Perl>
  local $ENV{PATH} = '/usr/bin';
  local $_;

  for ('env') {
      next unless /^(H.*)=/;
      push @PassEnv, $1;
  }
</Perl>
```

See also PerlSetupEnv which can enable/disable environment variables settings.

1.30 -M and other time() file tests under mod_perl

Under mod_perl, files that have been created after the server's (child) startup are reported as having a negative age with -M (-C -A) test. This is obvious if you remember that you will get the negative result if the server was started before the file was created. It's normal behavior with perl.

If you want to have -M report the time relative to the current request, you should reset the $^T variable just as with any other perl script. Add:

```
local $^T = time;
```

at the beginning of the script.

Another even simpler solution would be to specify a fixup handler, which will be executed before each script is run:

```
sub Apache::PerlBaseTime::handler {
  $^T = shift->request_time;
  return Apache::Constants::DECLINED;
}
```

and then in the *httpd.conf*:

```
PerlFixupHandler Apache::PerlBaseTime
```

This technique is better performance-wise as it skips the time() system call, and uses the already available time of the request has been started at via `$r->request_time` method.

1.31 Apache and syslog

When native syslog support is enabled, the stderr stream will be redirected to `/dev/null`!

It has nothing to do with mod_perl (plain Apache does the same). Doug wrote the Apache::LogSTDERR module to work around this.

1.32 File tests operators

Remember that with mod_perl you might get negative times when you use file test operators like `-M` -- last modification time, `-A` -- last access time, `-C` -- last inode-change time, and others. `-M` returns the difference in time between the modification time of the file and the time the script was started. Because the `^T` variable is not reset on each script invocation, and is equal to the time when the process was forked, you might want to perform:

```
$^T = time;
```

at the beginning of your scripts to simulate the regular perl script behaviour of file tests.

META: Above is near duplicate of "-M and other time() file tests under mod_perl" make a link instead

1.33 Filehandlers and locks leakages

META: duplication at debug.pod: =head3 Safe Resource Locking

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

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

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

```
close IN;
```

If you forget to `close()`, you might get file descriptor leakage and (if you `flock()ed` on this file descriptor) also unlock problems.

Even if you do call close(), if for some reason the interpreter was stopped before the `close()` call, the leakage will still happen. See for example Handling the 'User pressed Stop button' case. After a long run without restarting Apache your machine might run out of file descriptors, and worse, files might be left locked and unusable.

What can you do? Use `IO::File` (and the other `IO::*` modules). This allows you to assign the file handler to variable which can be `my()` (lexically) scoped. When this variable goes out of scope the file or other file system entity will be properly closed (and unlocked if it was locked). Lexically scoped variables will always go out of scope at the end of the script's invocation even if it was aborted in the middle. If the variable was defined inside some internal block, it will go out of scope at the end of the block. For example:

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

As I have just mentioned, you don't have to create a special block for this purpose. A script in a file is effectively written in a block with the same scope as the file, so you can simply write:

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

Using a `{ BLOCK }`) makes sure is that the file is closed the moment that the end of the block is reached.

An even faster and lighter technique is to use `Symbol.pm`:

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

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

Under perl 5.6.0 we can do this instead:

```
open my $fh, $filename or die $! ;
```

1.34 Code has been changed, but it seems the script is running the old code

Files pulled in via `use` or `require` statements are not automatically reloaded when they change on disk. See Reloading Modules and Required Files for more information.

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

You still can win from using mod_perl.

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

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

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

See Apache::PerlRun--a closer look

Another "bad", but workable method is to set `MaxRequestsPerChild` to 1, which will force each child to exit after serving only one request. You will get the preloaded modules, etc., but the script will be compiled for each request, then be thrown away. This isn't good for "high-traffic" sites, as the parent server will need to fork a new child each time one is killed. You can fiddle with `MaxStartServers` and `MinSpareServers`, so that the parent pre-spawns more servers than actually required and the killed one will immediately be replaced with a fresh one. Probably that's not what you want.

1.36 Apache::PerlRun--a closer look

`Apache::PerlRun` gives you the benefit of preloaded Perl and its modules. This module's handler emulates the CGI environment, allowing programmers to write scripts that run under CGI or mod_perl without any change. Unlike `Apache::Registry`, the `Apache::PerlRun` handler does not cache the script inside a subroutine. Scripts will be "compiled" on each request. After the script has run, its name space is flushed of all variables and subroutines. Still, you don't have the overhead of loading the Perl interpreter and the compilation time of the standard modules. If your script is very light, but uses lots of standard modules, you will see no difference between `Apache::PerlRun` and `Apache::Registry`!.

Be aware though, that if you use packages that use internal variables that have circular references, they will be not flushed!!! `Apache::PerlRun` only flushes your script's name space, which does not include any other required packages' name spaces. If there's a reference to a `my()` scoped variable that's keeping it from being destroyed after leaving the eval scope (of `Apache::PerlRun`), that cleanup might not be taken care of until the server is shutdown and `perl_destruct()` is run, which always happens after running command line scripts. Consider this example:

```
package Foo;
sub new { bless {} }
sub DESTROY {
  warn "Foo->DESTROY\n";
}

eval <<'EOF';
package my_script;
my $self = Foo->new;
#$self->{circle} = $self;
```

```
EOF

print $@ if $@;
print "Done with script\n";
```

When executed as a plain script you'll see:

```
Foo->DESTROY
Done with script
```

Then, uncomment the line where `$self` makes a circular reference, and you'll see:

```
Done with script
Foo->DESTROY
```

If you run this example with the circular reference enabled under mod_perl you won't see `Foo->DESTROY` until server shutdown, or until your module properly takes care of things. Note that the warn() call logs its messages to the *error_log* file, so you should expect the output there and not together with STDOUT.

1.37 Sharing variables between processes

META: to be completed

- Global variables initialized at server startup, through the Perl startup file, can be shared between processes, until modified by some of the processes. e.g. when you write:

  ```
  $My::debug = 1;
  ```

  all processes will read the same value. If one of the processes changes that value to `0`, it will still be equal to `1` for any other process, but not for the one which actually made the change. When a process modifies a shared variable, it becomes the process' private copy.

- `IPC::Shareable` can be used to share variables between children.

- libmm

- other methods?

1.38 Preventing Apache::Constants Stringification

In mod_perl, you are going to use a certain number of constants in your code, mainly exported from `Apache::Constants`. However, in some cases, Perl will not understand that the constant you're trying to call is really a constant, but interprets it as a string. This is the case with the hash notation `=>`, which automatically stringifies the key.

For example:

```
$r->custom_response(FORBIDDEN => "File size exceeds quota.");
```

This will not set a custom response for `FORBIDDEN`, but for the string `"FORBIDDEN"`, which clearly isn't what is expected. You'll get an error like this:

```
[Tue Apr 23 19:46:14 2002] null: Argument "FORBIDDEN" isn't numeric
                           in subroutine entry at ...
```

Therefore, you can avoid this by not using the hash notation for things that don't require it.

```
$r->custom_response(FORBIDDEN,  "File size exceeds quota.");
```

There are other workarounds, which you should avoid using unless you really have to use hash notation:

```
my %hash = (
    FORBIDDEN()    => 'this is forbidden',
    +AUTH_REQUIRED => "You aren't authorized to enter!",
);
```

Another important note is that you should be using the correct constants defined here, and not direct HTTP codes. For example:

```
sub handler {
    return 200;
}
```

Is not correct. The correct use is:

```
use Apache::Constants qw(OK);

sub handler {
    return OK;
}
```

Also remember that `OK != HTTP_OK`.

1.39 Transitioning from Apache::Registry to Apache handlers

Even if you are a CGI script die-hard at some point you might want to move a few or all your scripts to Apache Perl handlers. Actually this is an easy task, since we saw already what `Apache::Registry` makes our scripts appear to Apache to be Perl handlers.

When you no longer need backward mod_cgi compatibility you can benefit from the Perl libraries working only under mod_perl. We will see why in a moment.

Let's see an example. We will start with a mod_cgi compatible CGI script running under `Apache::Registry`, transpose it into a Perl content handler and then convert it to use `Apache::Request` and `Apache::Cookie`.

1.39.1 Starting with mod_cgi Compatible Script

This is the original script's code we are going to work with:

```
cookie_script.pl
----------------
use strict;
use CGI;
use CGI::Cookie;
use vars qw($q $switch $status $sessionID);

init();
print_header();
print_status();

### <-- subroutines --> ###

# the init code
############
sub init{
  $q = new CGI;

  $switch = $q->param("switch") ? 1 : 0;

    # try to retrieve the session ID
            # fetch existing cookies
  my %cookies = CGI::Cookie->fetch;
  $sessionID = exists $cookies{'sessionID'}
     ? $cookies{'sessionID'}->value : '';

    # 0 = not running, 1 = running
  $status = $sessionID ? 1 : 0;

    # switch status if asked to
  $status = ($status+1) % 2 if $switch;

  if ($status){
      # preserve sessionID if exists or create a new one
    $sessionID ||= generate_sessionID() if $status;
  } else {
      # delete the sessionID
    $sessionID = '';
  }

} # end of sub init

#################
sub print_header{
    # prepare a cookie
  my $c = CGI::Cookie->new
    ('-name'    => 'sessionID',
     '-value'   => $sessionID,
     '-expires' => '+1h');

  print $q->header
    (-type   => 'text/html',
```

```
      -cookie => $c);

} # end of sub print_header


# print the current Session status and a form to toggle the status
#################
sub print_status{

  print qq{<HTML><HEAD><TITLE>Cookie</TITLE></HEAD><BODY>};

    # print status
  print "<B>Status:</B> ",
        $status
          ? "Session is running with ID: $sessionID"
          : "No session is running";


    # change status form
  my $button_label = $status ? "Stop" : "Start";
  print qq{<HR>
       <FORM>
         <INPUT TYPE=SUBMIT NAME=switch VALUE=" $button_label ">
       </FORM>
         };

  print qq{</BODY></HTML>};

} # end of sub print_status

# A dummy ID generator
# Replace with a real session ID generator
########################
sub generate_sessionID {
  return scalar localtime;
} # end of sub generate_sessionID
```

The code is very simple. It creates a session if you've pressed the *'Start'* button or deletes it if you've pressed the *'Stop'* button. The session is stored and retrieved using the cookies technique.

Note that we have split the obviously simple and short code into three logical units, by putting the code into three subroutines. init() to initialize global variables and parse incoming data, print_header() to print the HTTP headers including the cookie header, and finally print_status() to generate the output. Later we will see that this logical separation will allow us an easy conversion to Perl content handler code.

We have used global variables for a few variables since we didn't want to pass them from function to function. In a big project you should be very restrictive about what variables should be allowed to be global, if any at all. In any case, the init() subroutine makes sure all these variables are re-initialized for each code reinvocation.

Note that we have used a very simple generate_sessionID() function that returns a date string (i.e. Wed Apr 12 15:02:23 2000) as a session ID. You want to replace this one with code which generates a unique session every time it was called. And it should be secure, i.e. users will not be able to forge one and do nasty things.

1.39.2 Converting into Perl Content Handler

Now let's convert this script into a content handler. There are two parts to this task; the first one is to configure Apache to run the new code as a Perl handler, the second one is to modify the code itself.

First we add the following snippet to *httpd.conf*:

```
PerlModule Test::Cookie
<Location /test/cookie>
  SetHandler perl-script
  PerlHandler Test::Cookie
</Location>
```

After we restart the server, when there is a request whose URI starts with */test/cookie*, Apache will execute the `Test::Cookie::handler()` subroutine as a content handler. We made sure to preload the `Test::Cookie` module at server start-up, with the `PerlModule` directive.

Now we are going to modify the script itself. We copy the content to the file *Cookie.pm* and place it into one of the directories listed in `@INC`. For example if */home/httpd/perl* is a part of `@INC` and since we want to call this package `Test::Cookie`, we can put *Cookie.pm* into the */home/httpd/perl/Test/* directory.

So this is the new code. Notice that all the subroutines were left unmodified from the original script, so to make the differences clear we do not repeat them here.

```
Test/Cookie.pm
--------------
package Test::Cookie;
use Apache::Constants qw(:common);

use strict;
use CGI;
use CGI::Cookie;
use vars qw($q $switch $status $sessionID);

sub handler{
  my $r = shift;
  Apache->request($r);

  init();
  print_header();
  print_status();

  return OK;
}

### <-- subroutines --> ###
# all subroutines as before

1;
```

As you see there are two lines added to the beginning of the code:

```
package Test::Cookie;
use Apache::Constants qw(:common);
```

The first one declares the package name and the second one imports some symbols commonly used in Perl handlers to return status codes.

```
use strict;
use CGI;
use CGI::Cookie;
use vars qw($q $switch $status $sessionID);
```

This code is left unchanged just as before.

```
sub handler{
  my $r = shift;
  Apache->request($r);

  init();
  print_header();
  print_status();

  return OK;
}
```

Each content handler (and any other handler) should begin with a subroutine called handler(). This subroutine is called when a request's URI starts with */test/cookie* as per our configuration. Of course you can choose a different name, for example execute(), but then you must explicitly use it in the configuration directives in the following way:

```
PerlModule Test::Cookie
<Location /test/cookie>
  SetHandler perl-script
  PerlHandler Test::Cookie::execute
</Location>
```

But we will use the default name, handler().

The handler() subroutine is just like any other subroutine, but generally it has the following structure:

```
sub handler{
  my $r = shift;

  # the code

  # status (OK, DECLINED or else)
  return OK;
}
```

First we get the request object by shifting it from `@_` and assigning it to the `$r` variable.

Second we write the code that does the processing of the request.

Third we return the status of the execution. There are many possible statuses, the most commonly used are `OK` and `DECLINED`, which tell the server whether they have completed the request phase that the handler was assigned to do or not. If not, another handler must complete the processing. `Apache::Constants` imports these two and other some commonly used status codes.

So in our example all we had to do was to wrap the three calls:

```
    init();
    print_header();
    print_status();
```

inside:

```
  sub handler{
    my $r = shift;
    Apache->request($r);

    return OK;
  }
```

There is one line we didn't discuss:

```
    Apache->request($r);
```

Since we use `CGI.pm`, it relies on the fact that `$r` was set in the `Apache` module. `Apache::Registry` did that behind the scenes. Since we don't use `Apache::Registry` here, we have to do that ourselves.

The one last thing we should do is to add `1;` at the end of the module, just like with any Perl module, so `PerlModule` will not fail when it tries to load `Test::Cookie`.

So to summarize, we took the original script's code and added the following eight lines:

```
  package Test::Cookie;
  use Apache::Constants qw(:common);

  sub handler{
    my $r = shift;
    Apache->request($r);

    return OK;
  }
  1;
```

and now we have a fully fledged Perl Content Handler.

1.39.3 Converting to use Apache Perl Modules

So now we have a complete PerlHandler, let's convert it to use Apache Perl modules. This breaks the backward compatibility, but gives us better performance, mainly because the internals of many of these Perl modules are implemented in C, therefore we should get a significant improvement in speed. The section "TMTOWTDI: Convenience and Performance" compares the three approaches.

What we are going to do is to replace `CGI.pm` and `CGI::Cookie` with `Apache::Request` and `Apache::Cookie` respectively. The two modules are written in C with the XS interface to Perl, which makes code much faster if it utilizes any of these modules a lot. `Apache::Request` uses an API similar to the one `CGI` uses, the same goes for `Apache::Cookie` and `CGI::Cookie`. This allows an easy porting process. Basically we just replace:

```
use CGI;
$q = new CGI;
```

with:

```
use Apache::Request ();
my $q = Apache::Request->new($r);
```

and

```
use CGI::Cookie ();
my $cookie = CGI::Cookie->new(...)
```

with

```
use Apache::Cookie ();
my $cookie = Apache::Cookie->new($r, ...);
```

This is the new code for `Test::Cookie2`:

```
Test/Cookie2.pm
--------------
package Test::Cookie2;
use Apache::Constants qw(:common);

use strict;
use Apache::Request;
use Apache::Cookie ();
use vars qw($r $q $switch $status $sessionID);

sub handler{
  $r = shift;

  init();
  print_header();
  print_status();

  return OK;
}

### <-- subroutines --> ###

# the init code
############
sub init{

  $q = Apache::Request->new($r);
  $switch = $q->param("switch") ? 1 : 0;
```

```
      # fetch existing cookies
  my %cookies = Apache::Cookie->fetch;
    # try to retrieve the session ID
  $sessionID = exists $cookies{'sessionID'}
     ? $cookies{'sessionID'}->value : '';

    # 0 = not running, 1 = running
  $status = $sessionID ? 1 : 0;

    # switch status if asked to
  $status = ($status+1) % 2 if $switch;


  if ($status){
      # preserve sessionID if exists or create a new one
    $sessionID ||= generate_sessionID() if $status;
  } else {
      # delete the sessionID
    $sessionID = '';
  }


} # end of sub init


#################
sub print_header{
    # prepare a cookie
  my $c = Apache::Cookie->new
    ($r,
     -name    => 'sessionID',
     -value   => $sessionID,
     -expires => '+1h');

    # Add a Set-Cookie header to the outgoing headers table
  $c->bake;

  $r->send_http_header('text/html');

} # end of sub print_header


# print the current Session status and a form to toggle the status
#################
sub print_status{

  print qq{<HTML><HEAD><TITLE>Cookie</TITLE></HEAD><BODY>};

    # print status
  print "<B>Status:</B> ",
        $status
          ? "Session is running with ID: $sessionID"

          : "No session is running";


    # change status form
```

```
    my $button_label = $status ? "Stop" : "Start";
    print qq{<HR>
         <FORM>
           <INPUT TYPE=SUBMIT NAME=switch VALUE=" $button_label ">
         </FORM>
           };

    print qq{</BODY></HTML>};

  } # end of sub print_status

  # replace with a real session ID generator
  ########################
  sub generate_sessionID {
    return scalar localtime;
  }

  1;
```

The only other changes are in the print_header() function, where instead of passing the cookie code to the `CGI`'s header() to return a proper HTTP header:

```
    print $q->header
      (-type   => 'text/html',
       -cookie => $c);
```

we do it in two stages.

```
    $c->bake;
```

Adds a `Set-Cookie` header to the outgoing headers table, and:

```
    $r->send_http_header('text/html');
```

sends out the header itself. We have also eliminated:

```
    Apache->request($r);
```

since we don't rely on `CGI.pm` any more and in this case we don't need it.

The rest of the code is unchanged.

Of course we add the following snippet to *httpd.conf*:

```
  PerlModule Test::Cookie2
  <Location /test/cookie2>
    SetHandler perl-script
    PerlHandler Test::Cookie2
  </Location>
```

So now the magic URI that will trigger the above code execution will be the one starting with */test/cookie2* . We save the code in the file */home/httpd/perl/Test/Cookie2.pm* since we have called this package `Test::Cookie2`.

1.39.4 Conclusion

If your took care to write the original plain CGI script's code in a clean and modular way, you can see that the transition is a very simple one and doesn't take a lot of effort. Almost no code was modified.

1.40 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Stas Bekman <stas (at) stason.org>

1.41 Authors

- Stas Bekman <stas (at) stason.org>

Only the major authors are listed above. For contributors see the Changes file.

Table of Contents: