

# General Documentation

Here you can find documentation concerning mod\_perl in general, but also not strictly mod\_perl related information that is still very useful for working with mod\_perl. Most of the information here applies to mod\_perl 1.0 and 2.0.

Last modified Sat Dec 6 12:16:23 2003 GMT

## **Part I: Perl**

### ▶ 1. Perl Reference

This document was born because some users are reluctant to learn Perl, prior to jumping into mod\_perl. I will try to cover some of the most frequent pure Perl questions being asked at the list.

## **Part II: Packaging and Testing**

### ▶ 2. Preparing mod\_perl modules for CPAN

This document provides information for CPAN modules developers whose modules require mod\_perl.

### ▶ 3. Running and Developing Tests with the Apache::Test Framework

The title is self-explanatory :)

## **Part III: HTTP**

### ▶ 4. Issuing Correct HTTP Headers

To make caching of dynamic documents possible, which can give you a considerable performance gain, setting a number of HTTP headers is of a vital importance. This document explains which headers you need to pay attention to, and how to work with them.

## **Part IV: Server Administration**

### ▶ 5. mod\_perl for ISPs. mod\_perl and Virtual Hosts

mod\_perl hosting by ISPs: fantasy or reality? This section covers some topics that might be of interest to users looking for ISPs to host their mod\_perl-based website, and ISPs looking for a way to provide such services.

### ▶ 6. Choosing an Operating System and Hardware

Before you use the techniques documented on this site to tune servers and write code you need to consider the demands which will be placed on the hardware and the operating system. There is no point in investing a lot of time and money in configuration and coding only to find that your server's performance is poor because you did not choose a suitable platform in the first place.

### ▶ 7. Controlling and Monitoring the Server

Covers techniques to restart mod\_perl enabled Apache, SUID scripts, monitoring, and other maintenance chores, as well as some specific setups.

## **Part V: mod\_perl Advocacy**

### ▶ 8. mod\_perl Advocacy

Having a hard time getting mod\_perl into your organization? We have collected some arguments you can use to convince your boss why the organization wants mod\_perl.

### ▶ 9. Popular Perl Complaints and Myths

This document tries to explain the myths about Perl and overturn the FUD certain bodies try to spread.

## **Part VI: OS Specific Documentation**

- ▶ 10. OS-specific Info  
Documents concerning OS-specific issues.

# **1 Perl Reference**

## 1.1 Description

This document was born because some users are reluctant to learn Perl, prior to jumping into `mod_perl`. I will try to cover some of the most frequent pure Perl questions being asked at the list.

Before you decide to skip this chapter make sure you know all the information provided here. The rest of the Guide assumes that you have read this chapter and understood it.

## 1.2 `perldoc`'s Rarely Known But Very Useful Options

First of all, I want to stress that you cannot become a Perl hacker without knowing how to read Perl documentation and search through it. Books are good, but an easily accessible and searchable Perl reference at your fingertips is a great time saver. It always has the up-to-date information for the version of perl you're using.

Of course you can use online Perl documentation at the Web. The two major sites are <http://www.perldoc.com> and <http://theoryx5.uwinnipeg.ca/CPAN/perl/>.

The `perldoc` utility provides you with access to the documentation installed on your system. To find out what Perl manpages are available execute:

```
% perldoc perl
```

To find what functions perl has, execute:

```
% perldoc perlfunc
```

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

```
% perldoc -f open
```

Note: In perl5.005\_03 and earlier, there is a bug in this and the `-q` options of `perldoc`. It won't call `pod2man`, but will display the section in POD format instead. Despite this bug it's still readable and very useful.

The Perl FAQ (*perlfaq* manpage) is in several sections. To search through the sections for `open` you would execute:

```
% perldoc -q open
```

This will show you all the matching Question and Answer sections, still in POD format.

To read the *perldoc* manpage you would execute:

```
% perldoc perldoc
```

## 1.3 Tracing Warnings Reports

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

Here is an example:

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

use strict;

correct();
incorrect();

sub correct{
    print_value("Perl");
}

sub incorrect{
    print_value();
}

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

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

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

we get the warning:

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

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

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

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

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

```

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

```

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

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

```

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

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

correct();
incorrect();

sub correct{
    print_value("Perl");
}

sub incorrect{
    print_value();
}

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

```

Now when we execute it, we see:

```

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

```

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

```

script => incorrect() => print_value()

```

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

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

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

Getting the trace helps a lot.

## 1.4 Variables Globally, Lexically Scoped And Fully Qualified

META: this material is new and requires polishing so read with care.

You will hear a lot about namespaces, symbol tables and lexical scoping in Perl discussions, but little of it will make any sense without a few key facts:

### 1.4.1 Symbols, Symbol Tables and Packages; Typeglobs

There are two important types of symbol: package global and lexical. We will talk about lexical symbols later, for now we will talk only about package global symbols, which we will refer to simply as *global symbols*.

The names of pieces of your code (subroutine names) and the names of your global variables are symbols. Global symbols reside in one symbol table or another. The code itself and the data do not; the symbols are the names of pointers which point (indirectly) to the memory areas which contain the code and data. (Note for C/C++ programmers: we use the term 'pointer' in a general sense of one piece of data referring to another piece of data not in a specific sense as used in C or C++.)

There is one symbol table for each package, (which is why *global symbols* are really *package global symbols*).

You are always working in one package or another.

Like in C, where the first function you write must be called `main()`, the first statement of your first Perl script is in package `main::` which is the default package. Unless you say otherwise by using the `package` statement, your symbols are all in package `main::`. You should be aware straight away that files and packages are *not related*. You can have any number of packages in a single file; and a single package can be in one file or spread over many files. However it is very common to have a single package in a single file. To declare a package you write:



```
package mypackagename;
```

From the following line you are in package `mypackagename` and any symbols you declare reside in that package. When you create a symbol (variable, subroutine etc.) Perl uses the name of the package in which you are currently working as a prefix to create the fully qualified name of the symbol.

When you create a symbol, Perl creates a symbol table entry for that symbol in the current package's symbol table (by default `main::`). Each symbol table entry is called a *typeglob*. Each typeglob can hold information on a scalar, an array, a hash, a subroutine (code), a filehandle, a directory handle and a format, each of which all have the same name. So you see now that there are two indirections for a global variable: the symbol, (the thing's name), points to its typeglob and the typeglob for the thing's type (scalar, array, etc.) points to the data. If we had a scalar and an array with the same name their name would point to the same typeglob, but for each type of data the typeglob points to somewhere different and so the scalar's data and the array's data are completely separate and independent, they just happen to have the same name.

Most of the time, only one part of a typeglob is used (yes, it's a bit wasteful). You will by now know that you distinguish between them by using what the authors of the Camel book call a *funny character*. So if we have a scalar called `'line'` we would refer to it in code as `$line`, and if we had an array of the same name, that would be written, `@line`. Both would point to the same typeglob (which would be called `*line`), but because of the *funny character* (also known as *decoration*) perl won't confuse the two. Of course we might confuse ourselves, so some programmers don't ever use the same name for more than one type of variable.

Every global symbol is in some package's symbol table. To refer to a global symbol we could write the *fully qualified* name, e.g. `$main::line`. If we are in the same package as the symbol we can omit the package name, e.g. `$line` (unless you use the `strict` pragma and then you will have to predeclare the variable using the `vars` pragma). We can also omit the package name if we have imported the symbol into our current package's namespace. If we want to refer to a symbol that is in another package and which we haven't imported we must use the fully qualified name, e.g. `$otherpkg::box`.

Most of the time you do not need to use the fully qualified symbol name because most of the time you will refer to package variables from within the package. This is very like C++ class variables. You can work entirely within package `main::` and never even know you are using a package, nor that the symbols have package names. In a way, this is a pity because you may fail to learn about packages and they are extremely useful.

The exception is when you *import* the variable from another package. This creates an alias for the variable in the *current* package, so that you can access it without using the fully qualified name.

Whilst global variables are useful for sharing data and are necessary in some contexts it is usually wisest to minimize their use and use *lexical variables*, discussed next, instead.

Note that when you create a variable, the low-level business of allocating memory to store the information is handled automatically by Perl. The interpreter keeps track of the chunks of memory to which the pointers are pointing and takes care of undefining variables. When all references to a variable have ceased to exist then the perl garbage collector is free to take back the memory used ready for recycling. However perl almost never returns back memory it has already used to the operating system during the lifetime of the

process.

### 1.4.1.1 Lexical Variables and Symbols

The symbols for lexical variables (i.e. those declared using the keyword `my`) are the only symbols which do *not* live in a symbol table. Because of this, they are not available from outside the block in which they are declared. There is no typeglob associated with a lexical variable and a lexical variable can refer only to a scalar, an array, a hash or a code reference. (Since perl-5.6 it can also refer to a file glob).

If you need access to the data from outside the package then you can return it from a subroutine, or you can create a global variable (i.e. one which has a package prefix) which points or refers to it and return that. The pointer or reference must be global so that you can refer to it by a fully qualified name. But just like in C try to avoid having global variables. Using OO methods generally solves this problem, by providing methods to get and set the desired value within the object that can be lexically scoped inside the package and passed by reference.

The phrase "lexical variable" is a bit of a misnomer, we are really talking about "lexical symbols". The data can be referenced by a global symbol too, and in such cases when the lexical symbol goes out of scope the data will still be accessible through the global symbol. This is perfectly legitimate and cannot be compared to the terrible mistake of taking a pointer to an automatic C variable and returning it from a function--when the pointer is dereferenced there will be a segmentation fault. (Note for C/C++ programmers: having a function return a pointer to an auto variable is a disaster in C or C++; the perl equivalent, returning a reference to a lexical variable created in a function is normal and useful.)

- `my( )` vs. `use vars:`

With `use vars()`, you are making an entry in the symbol table, and you are telling the compiler that you are going to be referencing that entry without an explicit package name.

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

- `my( )` vs. `local( ):`

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

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

### 1.4.2 Additional reading references

For more information see: Using global variables and sharing them between modules/packages and an article by Mark-Jason Dominus about how Perl handles variables and namespaces, and the difference between `use vars()` and `my()` - <http://www.plover.com/~mjd/perl/FAQs/Namespaces.html> .

## 1.5 my() Scoped Variable in Nested Subroutines

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

### 1.5.1 The Poison

Let's look at this code:

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

use strict;

sub print_power_of_2 {
    my $x = shift;

    sub power_of_2 {
        return $x ** 2;
    }

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

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

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

```
% ./nested.pl

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

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

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

And run it:

```
% ./nested.pl

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

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

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

and the result is:

```
% ./nested.pl

3^2 = 9
5^2 = 9
```

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

## 1.5.2 The Diagnosis

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

```
% ./nested.pl

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

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

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

use diagnostics;
use strict;
```

And execute it:

```
% ./nested.pl

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

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

When the inner subroutine is called, it will probably see the value of
the outer subroutine's variable as it was before and during the
```

*\*first\** call to the outer subroutine; in this case, after the first call to the outer subroutine is complete, the inner and outer subroutines will no longer share a common value for the variable. In other words, the variable will no longer be shared.

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

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

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

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

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

### 1.5.3 The Remedy

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

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

Let's rewrite the code to use this technique:

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

use strict;

sub print_power_of_2 {
    my $x = shift;

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

```

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

print_power_of_2(5);
print_power_of_2(6);

```

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

Let's verify:

```

% ./anonymous.pl

5^2 = 25
6^2 = 36

```

So we can see that the problem is solved.

## 1.6 Understanding Closures -- the Easy Way

In Perl, a closure is just a subroutine that refers to one or more lexical variables declared outside the subroutine itself and must therefore create a distinct clone of the environment on the way out.

And both named subroutines and anonymous subroutines can be closures.

Here's how to tell if a subroutine is a closure or not:

```

for (1..5) {
    push @a, sub { "hi there" };
}
for (1..5) {
    {
        my $b;
        push @b, sub { $b."hi there" };
    }
}
print "anon normal:\n", join "\t\n", @a, "\n";
print "anon closure:\n", join "\t\n", @b, "\n";

```

which generates:

```

anon normal:
CODE(0x80568e4)
CODE(0x80568e4)
CODE(0x80568e4)
CODE(0x80568e4)
CODE(0x80568e4)

anon closure:
CODE(0x804b4c0)

```

```
CODE(0x8056b54)
CODE(0x8056bb4)
CODE(0x80594d8)
CODE(0x8059538)
```

Note how each code reference from the non-closure is identical, but the closure form must generate distinct coderefs to point at the distinct instances of the closure.

And now the same with named subroutines:

```
for (1..5) {
    sub a { "hi there" };
    push @a, \&a;
}
for (1..5) {
    {
        my $b;
        sub b { $b."hi there" };
        push @b, \&b;
    }
}
print "normal:\n", join "\t\n", @a, "\n";
print "closure:\n", join "\t\n", @b, "\n";
```

which generates:

```
anon normal:
CODE(0x80568c0)
CODE(0x80568c0)
CODE(0x80568c0)
CODE(0x80568c0)
CODE(0x80568c0)

anon closure:
CODE(0x8056998)
CODE(0x8056998)
CODE(0x8056998)
CODE(0x8056998)
CODE(0x8056998)
```

We can see that both versions has generated the same code reference. For the subroutine *a* it's easy, since it doesn't include any lexical variables defined outside it in the same lexical scope.

As for the subroutine *b*, it's indeed a closure, but Perl won't recompile it since it's a named subroutine (see the *perlsub* manpage). It's something that we don't want to happen in our code unless we want it for this special effect, similar to *static* variables in C.

This is the underpinnings of that famous "*won't stay shared*" message. A *my* variable in a named subroutine context is generating identical code references and therefore it ignores any future changes to the lexical variables outside of it.

## ***1.6.1 Mike Guy's Explanation of the Inner Subroutine Behavior***

From: mjtg@cus.cam.ac.uk (M.J.T. Guy)  
Newsgroups: comp.lang.perl.misc  
Subject: Re: Lexical scope and embedded subroutines.  
Date: 6 Jan 1998 18:22:39 GMT  
Message-ID: <68tspf\$9f0\$1@lyra.csx.cam.ac.uk>

In article <68sc4k\$3p2\$1@brokaw.wa.com>, Aaron Harsh <ajh@rtk.com> wrote:

> Before I read this thread (and perlsub to get the details) I would  
> have assumed the original code was fine.  
>  
> This behavior brings up the following questions:  
> o Is Perl's behavior some sort of speed optimization?

No, but see below.

> o Did the Perl gods just decide that scheme-like behavior was less  
> important than the pseduo-static variables described in perlsub?

This subject has been kicked about at some length on perl5-porters. The current behaviour was chosen as the best of a bad job. In the context of Perl, it's not obvious what "scheme-like behavior" means. So it isn't an option. See below for details.

> o Does anyone else find Perl's behavior counter-intuitive?

\*Everyone\* finds it counterintuitive. The fact that it only generates a warning rather than a hard error is part of the Perl Gods policy of hurling thunderbolts at those so irreverent as not to use -w.

> o Did programming in scheme destroy my ability to judge a decent  
> language  
> feature?

You're still interested in Perl, so it can't have rotted your brain completely.

> o Have I misremembered how scheme handles these situations?

Probably not.

> o Do Perl programmers really care how much Perl acts like scheme?

Some do.

> o Should I have stopped this message two or three questions ago?

Yes.

The problem to be solved can be stated as

"When a subroutine refers to a variable which is instantiated more than once (i.e. the variable is declared in a for loop, or in a



subroutine), which instance of that variable should be used?"

The basic problem is that Perl isn't Scheme (or Pascal or any of the other comparators that have been used).

In almost all lexically scoped languages (i.e. those in the Algol60 tradition), named subroutines are also lexically scoped. So the scope of the subroutine is necessarily contained in the scope of any external variable referred to inside the subroutine. So there's an obvious answer to the "which instance?" problem.

But in Perl, named subroutines are globally scoped. (But in some future Perl, you'll be able to write

```
my sub lex { ... }
```

to get lexical scoping.) So the solution adopted by other languages can't be used.

The next suggestion most people come up with is "Why not use the most recently instantiated variable?". This Does The Right Thing in many cases, but fails when recursion or other complications are involved.

Consider:

```
sub outer {
    inner();
    outer();
    my $trouble;
    inner();
    sub inner { $trouble };
    outer();
    inner();
}
```

Which instance of \$trouble is to be used for each call of inner()? And why?

The consensus was that an incomplete solution was unacceptable, so the simple rule "Use the first instance" was adopted instead.

And it is more efficient than possible alternative rules. But that's not why it was done.

Mike Guy

## 1.7 When You Cannot Get Rid of The Inner Subroutine

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

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

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

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

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

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

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

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

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

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

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

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

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

We have just written a partial prototype of the daemon we wanted. The only outstanding problem is how to pass the path to the script to the daemon. This detail is left as an exercise for the reader.

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

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

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

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

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

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

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

### ***1.7.1 Remedies for Inner Subroutines***

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

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

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

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

use strict;
use warnings;

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

```

}

sub run{

    my $counter = 0;

    increment_counter();
    increment_counter();

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

} # end of sub run

```

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

```

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

```

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

```

% ./multirun.pl

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

```

Apparently, the `$counter` variable is not reinitialized on each execution of `run()`, it retains its value from the previous execution, and `increment_counter()` increments that. Actually that is not quite what happens. On each execution of `run()` a new `$counter` variable is initialized to zero but `increment_counter()` remains bound to the `$counter` variable from the first call to `run()`.

The simplest of the work-rounds is to use package-scoped variables. These can be declared using `our` or, on older versions of Perl, the `vars` pragma. Note that whereas using `my` declaration also implicitly initializes variables to undefined the `our` declaration does not, and so you will probably need to add explicit initialisation for variables that lacked it.

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

use strict;
use warnings;

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

sub run {

    our $counter = 0;

    increment_counter();
    increment_counter();

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

} # end of sub run
```

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

```
% ./multirun1.pl

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

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

In the above example we know `$counter` is just a simple small scalar. In the general case variables could reference external resource handles or large data structures. In that situation the fact that the variable would not be released immediately when `run()` completes could be a problem. To avoid this you can put `local` in front of the `our` declaration of all variables other than simple scalars. This has the effect of restoring the variable to its previous value (usually undefined) upon exit from the current scope. As a side-effect `local` also initializes the variables to `undef`. So, if you recall that thing I said about adding

explicit initialization when you replace `my` by `our`, well, you can forget it again if you replace `my` with `local` `our`.

Be warned that `local` will not release circular data structures. If the original CGI script relied upon process termination to clean up after it then it will leak memory as a registry script.

A variant of the package variable approach is not to declare your variables, but instead to use explicit package qualifiers. This has the advantage on old versions of Perl that there is no need to load the `vars` module, but it adds a significant typing overhead. Another downside is that you become dependant on the "used only once" warning to detect typos in variable names. The explicit package name approach is not really suitable for registry scripts because it pollutes the `main::` namespace rather than staying properly within the namespace that has been allocated. Finally, note that the overhead of loading the `vars` module only has to be paid once per Perl interpreter.

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

use strict;

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

sub run {

    $main::counter = 0;

    increment_counter();
    increment_counter();

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

} # end of sub run
```

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

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

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

```

use strict;
use warnings;

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

sub run {

    my $counter = 0;

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

    sub increment_counter{
        my $counter = shift;

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

        return $counter;
    }

} # end of sub run

```

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

```

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

use strict;
use warnings;

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

sub run {

    my $counter = 0;

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

    sub increment_counter{
        my $r_counter = shift;

        $$r_counter++;
    }
}

```

```

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

} # end of sub run

```

Here is yet another and more obscure reference usage. We modify the value of `$counter` inside the subroutine by using the fact that variables in `@_` are aliases for the actual scalar parameters. Thus if you called a function with two arguments, those would be stored in `$_[0]` and `$_[1]`. In particular, if an element `$_[0]` is updated, the corresponding argument is updated (or an error occurs if it is not updatable as would be the case of calling the function with a literal, e.g. *increment\_counter(5)*).

```

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

use strict;
use warnings;

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

sub run {

    my $counter = 0;

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

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

} # end of sub run

```

The approach given above should be properly documented of course.

Here is a solution that avoids the problem entirely by splitting the code into two files; the first is really just a wrapper and loader, the second file contains the heart of the code. This second file must go into a directory in your `@INC`. Some people like to put the library in the same directory as the script but this assumes that the current working directory will be equal to the directory where the script is located and also that `@INC` will contain `'.'`, neither of which are assumptions you should expect to hold in all cases.

Note that the name chosen for the library must be unique throughout the entire server and indeed every server on which you may ever install the script. This solution is probably more trouble than it is worth - it is only included because it was mentioned in previous versions of this guide.



```

multirun6.pl
-----
#!/usr/bin/perl

use strict;
use warnings;

require 'multirun6-lib.pl';

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

```

Separate file:

```

multirun6-lib.pl
-----
use strict;
use warnings;

my $counter;

sub run {
    $counter = 0;

    increment_counter();
    increment_counter();
}

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

1 ;

```

An alternative version of the above, that mitigates some of the disadvantages, is to use a Perl5-style Exporter module rather than a Perl4-style library. The global uniqueness requirement still applies to the module name, but at least this is a problem Perl programmers should already be familiar with when creating modules.

```

multirun7.pl
-----
#!/usr/bin/perl

use strict;
use warnings;
use My::Multirun7;

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

```

Separate file:

```
My/Multirun7.pm
-----
package My::Multirun7;
use strict;
use warnings;
use base qw( Exporter );
our @EXPORT = qw( run );

my $counter;

sub run {
    $counter = 0;

    increment_counter();
    increment_counter();
}

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

1 ;
```

Now you have at least five workarounds to choose from (not counting numbers 2 and 6).

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

## 1.8 use(), require(), do(), %INC and @INC Explained

### 1.8.1 The @INC array

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

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

### 1.8.2 The %INC hash

%INC is another special Perl variable that is used to cache the names of the files and the modules that were successfully loaded and compiled by use(), require() or do() statements. Before attempting to load a file or a module with use() or require(), Perl checks whether it's already in the %INC hash. If it's there, the loading and therefore the compilation are not performed at all. Otherwise the file is loaded into memory and an attempt is made to compile it. do() does unconditional loading--no lookup in the %INC hash is

made.

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

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

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

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

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

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

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

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

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

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

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

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

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

test.pm => test.pm
```

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

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

test.pm => test.pm
```

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

### 1.8.2 The %INC hash

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

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

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

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

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

The code:

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

can be replaced with the more elegant:

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

Which is almost equivalent to our `BEGIN` block and is the recommended approach.

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

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

For the sake of completeness, I'll present this module anyway.

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

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

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

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

```
% /tmp/load.pl

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

This is just like use `lib` except that no hard coded path is required.

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

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

This has a slight overhead because it will load from disk and recompile the `FindBin` module on each request. So it may not be worth it.

## 1.8.3 Modules, Libraries and Program Files

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

### ● Libraries

These are files which contain Perl subroutines and other code.

When these are used to break up a large program into manageable chunks they don't generally include a package declaration; when they are used as subroutine libraries they often do have a package declaration.

Their last statement returns true, a simple `1;` statement ensures that.

They can be named in any way desired, but generally their extension is *.pl*.

Examples:

```
config.pl
-----
# No package so defaults to main::
$dir = "/home/httpd/cgi-bin";
$cgi = "/cgi-bin";
1;

mysubs.pl
-----
# No package so defaults to main::
sub print_header{
    print "Content-type: text/plain\r\n\r\n";
}
1;
```

```

web.pl
-----
package web ;
# Call like this: web::print_with_class('loud', "Don't shout!");
sub print_with_class{
    my( $class, $text ) = @_ ;
    print qq{<span class="$class">$text</span>};
}
1;

```

## ● Modules

A file which contains perl subroutines and other code.

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

Modules are generally used either as function libraries (which *.pl* files are still but less commonly used for), or as object libraries where a module is used to define a class and its methods.

Its last statement returns true.

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

Example:

```

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

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

```

## ● Program Files

Many Perl programs exist as a single file. Under Linux and other Unix-like operating systems the file often has no suffix since the operating system can determine that it is a perl script from the first line (shebang line) or if it's Apache that executes the code, there is a variety of ways to tell how and when the file should be executed. Under Windows a suffix is normally used, for example *.pl* or *.plx*.

The program file will normally `require()` any libraries and `use()` any modules it requires for execution.

It will contain Perl code but won't usually have any package names.

Its last statement may return anything or nothing.

## 1.8.4 *require()*

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

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

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

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

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

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

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

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

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

This is equal to:

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

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

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

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

When we execute the program:

### 1.8.5 use()

```
% ./require.pl

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

Hello
```

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

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

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

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

% ./require1.pl

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

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

For more information refer to the perlfunc manpage.

## 1.8.5 use()

use(), just like require(), loads and compiles files containing Perl code, but it works with modules only and is executed at compile time.

The only way to pass a module to load is by its module name and not its filename. If the module is located in *MyCode.pm*, the correct way to use() it is:

```
use MyCode
```

and not:

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

use() translates the passed argument into a file name replacing :: with the operating system's path separator (normally /) and appending .pm at the end. So *My::Module* becomes *My/Module.pm*.



use() is exactly equivalent to:

```
BEGIN { require Module; Module->import(LIST); }
```

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

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

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

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

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

The second pair exports only the tags passed as arguments.

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

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

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

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

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

## 1.8.6 do()

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

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

## 1.9 Using Global Variables and Sharing Them Between Modules/Packages

It helps when you code your application in a structured way, using the perl packages, but as you probably know once you start using packages it's much harder to share the variables between the various packagings. A configuration package comes to mind as a good example of the package that will want its variables to be accessible from the other modules.

Of course using the Object Oriented (OO) programming is the best way to provide an access to variables through the access methods. But if you are not yet ready for OO techniques you can still benefit from using the techniques we are going to talk about.

### *1.9.1 Making Variables Global*

When you first wrote `$x` in your code you created a (package) global variable. It is visible everywhere in your program, although if used in a package other than the package in which it was declared (`main::` by default), it must be referred to with its fully qualified name, unless you have imported this variable with `import()`. This will work only if you do not use `strict` pragma; but you *have* to use this pragma if you want to run your scripts under `mod_perl`. Read The strict pragma to find out why.

### *1.9.2 Making Variables Global With strict Pragma On*

First you use :

```
use strict;
```

Then you use:

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

This declares the named variables as package globals in the current package. They may be referred to within the same file and package with their unqualified names; and in different files/packages with their fully qualified names.

With perl5.6 you can use the `our` operator instead:

```
our($scalar, %hash, @array);
```

If you want to share package global variables between packages, here is what you can do.

### *1.9.3 Using Exporter.pm to Share Global Variables*

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

```

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

My::HTML::printmyheader();

```

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

```

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

BEGIN {
    use Exporter ();

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

use vars qw($q);

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

```

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

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

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

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

Then you add the same `Exporter` code that we used in `My::HTML`, into `My::Doc`, so that it also exports `$q`.

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

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

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

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

My::HTML::printmyheader();

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

BEGIN {
    use Exporter ();

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

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

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

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

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

BEGIN {
    use Exporter ();

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

use vars qw($q);

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

```
    print $q->hl($title);
}
1;
```

## 1.9.4 Using the Perl Aliasing Feature to Share Global Variables

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

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

    array_var  => [qw(foo bar)],

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

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

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

    # Now you can access the variables from the My::Config
print ${scalar_var};
print ${array_var}[0];
print ${hash_var}{foo};
```

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

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

```
my *c = \%My::Config::c; # ERROR!
```

Which is an error. But you can write:

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

For more information about aliasing, refer to the Camel book, second edition, pages 51-52.

## 1.9.5 Using Non-Hardcoded Configuration Module Names

You have just seen how to use a configuration module for configuration centralization and an easy access to the information stored in this module. However, there is somewhat of a chicken-and-egg problem--how to let your other modules know the name of this file? Hardcoding the name is brittle--if you have only a single project it should be fine, but if you have more projects which use different configurations and you will want to reuse their code you will have to find all instances of the hardcoded name and replace it.

Another solution could be to have the same name for a configuration module, like `My::Config` but putting a different copy of it into different locations. But this won't work under `mod_perl` because of the namespace collision. You cannot load different modules which uses the same name, only the first one will be loaded.

Luckily, there is another solution which allows us to stay flexible. `PerlSetVar` comes to rescue. Just like with environment variables, you can set server's global Perl variables which can be retrieved from any module and script. Those statements are placed into the *httpd.conf* file. For example

```
PerlSetVar FooBaseDir      /home/httpd/foo
PerlSetVar FooConfigModule Foo::Config
```

Now we require() the file where the above configuration will be used.

```
PerlRequire /home/httpd/perl/startup.pl
```

In the *startup.pl* we might have the following code:

```
# retrieve the configuration module path
use Apache;
my $s = Apache->server;
my $base_dir = $s->dir_config('FooBaseDir') || '';
my $config_module = $s->dir_config('FooConfigModule') || '';
die "FooBaseDir and FooConfigModule aren't set in httpd.conf"
    unless $base_dir and $config_module;

# build the real path to the config module
my $path = "$base_dir/$config_module";
$path =~ s|:|/|;
$path .= ".pm";
```

```
# we have something like "/home/httpd/foo/Foo/Config.pm"

# now we can pull in the configuration module
require $path;
```

Now we know the module name and it's loaded, so for example if we need to use some variables stored in this module to open a database connection, we will do:

```
Apache::DBI->connect_on_init
( "DBI:mysql:${$config_module.'::DB_NAME'}:${$config_module.'::SERVER'}",
  ${$config_module.'::USER'},
  ${$config_module.'::USER_PASSWD'},
  {
    PrintError => 1, # warn() on errors
    RaiseError => 0, # don't die on error
    AutoCommit => 1, # commit executes immediately
  }
);
```

Where variable like:

```
${$config_module.'::USER'}
```

In our example are really:

```
$Foo::Config::USER
```

If you want to access these variable from within your code at the run time, instead accessing to the server object `$c`, use the request object `$r`:

```
my $r = shift;
my $base_dir      = $r->dir_config('FooBaseDir')      || '';
my $config_module = $r->dir_config('FooConfigModule') || '';
```

## 1.10 The Scope of the Special Perl Variables

Special Perl variables like `$|` (buffering), `$^T` (script's start time), `$^W` (warnings mode), `$/` (input record separator), `$\` (output record separator) and many more are all true global variables; they do not belong to any particular package (not even `main::`) and are universally available. This means that if you change them, you change them anywhere across the entire program; furthermore you cannot scope them with `my()`. However you can `local()`ise them which means that any changes you apply will only last until the end of the enclosing scope. In the `mod_perl` situation where the child server doesn't usually exit, if in one of your scripts you modify a global variable it will be changed for the rest of the process' life and will affect all the scripts executed by the same process. Therefore localizing these variables is highly recommended, I'd say mandatory.

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

## 1.11 Compiled Regular Expressions

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

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

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

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

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

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

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

Note that if you call a subroutine after you've set a global variable but within the enclosing block, the global variable will be visible with its new value inside the subroutine.

## 1.11 Compiled Regular Expressions

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

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

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

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



There are two solutions to this problem:

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

The above code fragment would be rewritten as:

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

Just saying:

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

means that we recompile the regex for every element in the list even though the regex doesn't change.

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

The above code fragment becomes:

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

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

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

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

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

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

Another approach:

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

Maybe with a helper routine, it's easier to remember. Here is one slightly modified from Jeffery Friedl's example in his book "*Mastering Regular Expressions*".

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

Example usage:

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

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

And of course you can use the qr() operator which makes the code even more efficient:

```
my $pat = '^foo$';
my $re = qr($pat);
foreach( @list ) {
    print if /$re/o;
}
```

The qr() operator compiles the pattern for each request and then use the compiled version in the actual match.

## 1.12 Exception Handling for mod\_perl

Here are some guidelines for clean(er) exception handling in mod\_perl, although the technique presented can be applied to all of your Perl programming.

The reasoning behind this document is the current broken status of `$SIG{__DIE__}` in the perl core - see both the perl5-porters and the mod\_perl mailing list archives for details on this discussion. (It's broken in at least Perl v5.6.0 and probably in later versions as well). In short summary, `$SIG{__DIE__}` is a little bit too global, and catches exceptions even when you want to catch them yourself, using an `eval{}` block.

### 1.12.1 Trapping Exceptions in Perl

To trap an exception in Perl we use the `eval{}` construct. Many people initially make the mistake that this is the same as the `eval EXPR` construct, which compiles and executes code at run time, but that's not the case. `eval{}` compiles at compile time, just like the rest of your code, and has next to zero run-time penalty. For the hardcore C programmers among you, it uses the `setjmp/longjmp` POSIX routines internally, just like C++ exceptions.

When in an eval block, if the code being executed `die()`'s for any reason, an exception is thrown. This exception can be caught by examining the `$@` variable immediately after the eval block; if `$@` is true then an exception occurred and `$@` contains the exception in the form of a string. The full construct looks like this:

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

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

Perl's exception handling is similar to that of other languages, though it may not seem so at first sight:

Perl	Other language
<pre>eval {     # execute here     # raise our own exception:     die "Oops" if /error/;     # execute more } ; if(\$@) {     # handle exceptions     if( \$@ =~ /Fail/ ) {         print "Failed\n" ;     }     elsif( \$@ =~ /Oops/ ) {         # Pass it up the chain         die if \$@ =~ /Oops/;     } }</pre>	<pre>try {     // execute here     // raise our own exception:     if(error==1){throw Exception.Oops;}     // execute more } catch {     switch( Exception.id ) {         Fail : fprintf( stderr, "Failed\n" ) ;                break ;          Oops : throw Exception ;     } }</pre>

```

    }
    else {
        # handle all other
        # exceptions here
    }
    // If we got here all is OK or handled
}
else { # optional
    # all is well
}
# all is well or has been handled

```

## 1.12.2 *Alternative Exception Handling Techniques*

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

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

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

```

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

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

```

The code unfortunately prints out:

```
In eval
handler
```

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

```

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

```

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

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

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

`$^S` isn't totally reliable in certain Perl versions. e.g. 5.005\_03 and 5.6.1 both do the wrong thing with it in certain situations. Instead, you can use the `caller()` function to figure out if we are called in the `eval()` context:

```
$SIG{__DIE__} = sub {
    my $in_eval = 0;
    for(my $stack = 1; my $sub = (CORE::caller($stack))[3]; $stack++) {
        $in_eval = 1 if $sub =~ /^\(eval\)/;
    }
    my_die_handler(@_) unless $in_eval;
};
```

The other problem with `$SIG{__DIE__}` also relates to its global nature. Because you might have more than one application running under `mod_perl`, you can't be sure which has set a `$SIG{__DIE__}` handler when and for what. This can become extremely confusing when you start scaling up from a set of simple registry scripts that might rely on `CGI::Carp` for global exception handling (which uses `$SIG{__DIE__}` to trap exceptions) to having many applications installed with a variety of exception handling mechanisms in place.

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

### 1.12.3 Better Exception Handling

The `eval{ }` construct in itself is a fairly weak way to handle exceptions as strings. There's no way to pass more information in your exception, so you have to handle your exception in more than one place - at the location the error occurred, in order to construct a sensible error message, and again in your exception handler to de-construct that string into something meaningful (unless of course all you want your exception handler to do is dump the error to the browser). The other problem is that you have no way of automatically detecting where the exception occurred using `eval{ }` construct. In a `$SIG{__DIE__}` block you always have the use of the `caller()` function to detect where the error occurred. But we can fix that...

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

[As a footnote here, the only thing that is really missing here from Java exceptions is a guaranteed Finally clause, although its possible to get about 98.62% of the way towards providing that using `eval { }.`]

### 1.12.3.1 A Little Housekeeping

First though, before we delve into the details, a little housekeeping is in order. Most, if not all, `mod_perl` programs consist of a main routine that is entered, and then dispatches itself to a routine depending on the parameters passed and/or the form values. In a normal C program this is your `main()` function, in a `mod_perl` handler this is your `handler()` function/method. The exception to this rule seems to be `Apache::Registry` scripts, although the techniques described here can be easily adapted.

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

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

eval {
    dispatch();
};
if ($@) {
    # handle exception
}

sub dispatch {
    ...
}
```

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

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

    eval {
        dispatch($r);
    };
    if ($@) {
        # handle exception
    }
}

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

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

### 1.12.3.2 An Exception Class

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

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

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

1;
```

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

```
use My::Exception;

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

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

## 1.12.4 Catching Uncaught Exceptions

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

### 1.12.4.1 Using `$SIG{__DIE__}`

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

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

All this does is catch your exception and re-throw it. It's not as dangerous as we stated earlier that `$SIG{__DIE__}` can be, because we're actually re-throwing the exception, rather than catching it and stopping there. Even though `$SIG{__DIE__}` is a global handler, because we are simply re-throwing the exception we can let other applications outside of our control simply catch the exception and not worry about it.

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

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

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

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

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

We can now let other code happily continue. Note that there is a bug in Perl 5.6 which may affect people here: Stringification does not occur when an object is operated on by a regular expression (via the `=~` operator). A work around is to explicitly stringify using qq double quotes, however that doesn't help the poor soul who is using other applications. This bug has been fixed in later versions of Perl.

### 1.12.4.2 Overriding the Core `die()` Function

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



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

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

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

sub die (@); # prototype to match CORE::die

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

sub die (@) {
    if (!ref($_[0])) {
        CORE::die My::Exception->UnCaught(text => join('', @_));
    }
    CORE::die $_[0]; # only use first element because its an object
}
```

That wasn't so bad, was it? We're relying on `Exporter`'s `export()` function to do the hard work for us, exporting the `die()` function into the `CORE::GLOBAL` namespace. If we don't want to overload `die()` everywhere this can still be an extremely useful technique. By just using `Exporter`'s default `import()` method we can export our new `die()` method into any package of our choosing. This allows us to short-cut the long calling convention and simply `die()` with a string, and let the system handle the actual construction into an object for us.

Along with the above overloaded stringification, we now have a complete exception system (well, mostly complete. Exception die-hards would argue that there's no "finally" clause, and no exception stack, but that's another topic for another time).

## 1.12.5 A Single UnCaught Exception Class

Until the Perl core gets its own base exception class (which will likely happen for Perl 6, but not sooner), it is vitally important that you decide upon a single base exception class for all of the applications that you install on your server, and a single exception handling technique. The problem comes when you have multiple applications all doing exception handling and all expecting a certain type of "UnCaught" exception class. Witness the following application:

```
package Foo;

eval {
    # do something
}
if ($@) {
    if ($@->isa('Foo::Exception::Bar')) {
        # handle "Bar" exception
    }
}
```

### 1.12.6 Some Uses

```
    elsif ($@->isa('Foo::Exception::UnCaught')) {  
        # handle uncaught exceptions  
    }  
}
```

All will work well until someone installs application "TrapMe" on the same machine, which installs its own UnCaught exception handler, overloading CORE::GLOBAL::die or installing a \$SIG{\_\_DIE\_\_} handler. This is actually a case where using \$SIG{\_\_DIE\_\_} might actually be preferable, because you can change your handler() routine to look like this:

```
sub handler {  
    my $r = shift;  
  
    local $SIG{__DIE__};  
    Foo::Exception->Init(); # sets $SIG{__DIE__}  
  
    eval {  
        dispatch($r);  
    };  
    if ($@) {  
        # handle exception  
    }  
}  
  
sub dispatch {  
    my $r = shift;  
    ...  
}
```

In this case the very nature of \$SIG{\_\_DIE\_\_} being a lexical variable has helped us, something we couldn't fix with overloading CORE::GLOBAL::die. However there is still a gotcha. If someone has overloaded die() in one of the applications installed on your mod\_perl machine, you get the same problems still. So in short: Watch out, and check the source code of anything you install to make sure it follows your exception handling technique, or just uses die() with strings.

## 1.12.6 Some Uses

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

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

Returns a 204 error code (HTTP\_NO\_CONTENT), which is caught at my top level exception handler:

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

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

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

## 1.12.7 Conclusions

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

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

## 1.12.8 The `My::Exception` class in its entirety

```
package My::Exception;

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

sub die (@);

sub import {
    my $pkg = shift;
    # allow "use My::Exception 'die';" to mean import locally only
    $pkg->export('CORE::GLOBAL', 'die') unless @_;
    Exporter::import($pkg, @_);
}

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

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

sub AUTOLOAD {
    no strict 'refs', 'subs';
```

```

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

1;

```

## 1.12.9 Other Implementations

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

- **Error.pm**

Graham Barr's excellent OO styled "try, throw, catch" module (from CPAN). This should be considered your best option for structured exception handling because it is well known and well supported and used by a lot of other applications.

- **Exception::Class and Devel::StackTrace**

by Dave Rolsky both available from CPAN of course.

`Exception::Class` is a bit cleaner than the `AUTOLOAD` method from above as it can catch typos in exception class names, whereas the method above will automatically create a new class for you. In addition, it lets you create actual class hierarchies for your exceptions, which can be useful if you want to create exception classes that provide extra methods or data. For example, an exception class for database errors could provide a method for returning the SQL and bound parameters in use at the time of the error.

- **Try.pm**

Tony Olekshy's. Adds an unwind stack and some other interesting features. Not on the CPAN. Available at <http://www.avrasoft.com/perl/rfc/try-1136.zip>

## 1.13 Customized \_\_DIE\_\_ handler

As we saw in the previous sections it's a bad idea to do:

```
require Carp;
$SIG{__DIE__} = \&Carp::confess;
```

since it breaks the error propogations within eval {} blocks,. But starting from perl 5.6.x you can use another solution to trace errors. For example you get an error:

```
"exit" is not exported by the GLOB(0x88414cc) module at (eval 397) line 1
```

and you have no clue where it comes from, you can override the exit() function and plug the tracer inside:

```
require Carp;
use subs qw(CORE::GLOBAL::die);
*CORE::GLOBAL::die = sub {
    if ($_[0] =~ /"exit" is not exported/){
        local *CORE::GLOBAL::die = sub { CORE::die($_) };
        Carp::confess($_); # Carp uses die() internally!
    } else {
        CORE::die($_); # could write &CORE::die to forward @_
    }
};
```

Now we can test that it works properly without breaking the eval {} blocks error propagation:

```
eval { foo(); }; warn $@ if $@;
print "\n";
eval { poo(); }; warn $@ if $@;

sub foo{ bar(); }
sub bar{ die qq{"exit" is not exported}}

sub poo{ tar(); }
sub tar{ die "normal exit"}
```

prints:

```
$ perl -w test
Subroutine die redefined at test line 5.
"exit" is not exported at test line 6
    main::__ANON__('"exit" is not exported') called at test line 17
    main::bar() called at test line 16
    main::foo() called at test line 12
    eval {...} called at test line 12

normal exit at test line 5.
```

the 'local' in:

## 1.14 Maintainers

```
local *CORE::GLOBAL::die = sub { CORE::die($_) };
```

is important, so you won't lose the overloaded `CORE::GLOBAL::die`.

## 1.14 Maintainers

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

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

## 1.15 Authors

- Stas Bekman <stas (at) stason.org>
- Matt Sergeant <matt (at) sergeant.org>

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

## **2 Preparing mod\_perl modules for CPAN**

## 2.1 Description

This document provides information for CPAN modules developers whose modules require `mod_perl`.

## 2.2 Defining Makefile.PL Prerequisites that Require `mod_perl`

If there are any prerequisites that need to be defined in *Makefile.PL*, but require a `mod_perl` environment to successfully get loaded, the following workaround can be used. The following example will specify two prerequisites: `CGI.pm` and `Apache::DBI`, the latter can be loaded only under `mod_perl` whereas the former can be loaded from the command line.

```
file:Makefile.PL
-----
use ExtUtils::MakeMaker;

# set prerequisites
my %prereq = (
    'CGI' => 2.71,
);

# Manually test whether Apache::DBI is installed and add it to the
# PREREQ_PM if it's not installed, so CPAN.pm will automatically fetch
# it. If Apache::DBI is already installed it will fail to get loaded by
# MakeMaker because it requires the mod_perl environment to load.
eval { require Apache::DBI };
if ($@ && $@ !~ /Can't locate object method/) {
    $prereq{'Apache::DBI'} = 0.87;
}

WriteMakefile(
    NAME          => 'Apache::SuperDuper',
    VERSION_FROM  => 'SuperDuper.pm',
    PREREQ_PM     => \%prereq,
    # ... the rest
);
```

Notice that *Can't locate object method* is a part of the error generated when `Apache::DBI` is installed but is attempted to be loaded outside of `mod_perl`, e.g. at the command line, which is the case with *Makefile.PL*.

## 2.3 Writing the Test Suite

The `Apache::Test` framework provides an easy way to test modules which require `mod_perl` (or `Apache` in general), be it 1.0 or 2.0 generation. Here is the complete guide to the `Apache::Test` framework.



## 2.4 Maintainers

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

Stas Bekman <stas (at) stason.org>

## 2.5 Authors

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

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

## **3 Running and Developing Tests with the Apache::Test Framework**

## 3.1 Description

The title is self-explanatory :)

The `Apache::Test` framework was designed for creating test suits for products running on Apache httpd webserver (not necessarily `mod_perl`). Originally designed for the `mod_perl` Apache module, it was extended to be used for any Apache module.

This chapter is talking about the `Apache::Test` framework, and in particular explains how to:

1. **run existing tests**
2. **setup a testing environment for a new project**
3. **develop new tests**

## 3.2 Basics of Perl Modules Testing

The tests themselves are written in Perl. The framework provides an extensive functionality which makes the tests writing a simple and therefore enjoyable process.

If you have ever written or looked at the tests most Perl modules come with, `Apache::Test` uses the same concept. The script `t/TEST` is running all the files ending with `.t` it finds in the `t/` directory. When executed a typical test prints the following:

```
1..3      # going to run 3 tests
ok 1      # the first  test has passed
ok 2      # the second test has passed
not ok 3  # the third  test has failed
```

Every `ok` or `not ok` is followed by the number which tells which sub-test has succeeded or failed.

`t/TEST` uses the `Test::Harness` module which intercepts the `STDOUT` stream, parses it and at the end of the tests print the results of the tests running: how many tests and sub-tests were run, how many succeeded, skipped or failed.

Some tests may be skipped by printing:

```
1..0 # all tests in this file are going to be skipped.
```

Usually a test may be skipped when some feature is optional and/or prerequisites are not installed on the system, but this is not critical for the usefulness of the test. Once you test that you cannot proceed with the tests and it's not a must pass test, you just skip it.

By default `print()` statements in the test script are filtered out by `Test::Harness`. if you want the test to print what it does (if you decide to debug some test) use `-verbose` option. So for example if your test does this:

### 3.3 Prerequisites

```
print "# testing : feature foo\n";
print "# expected: $expected\n";
print "# received: $received\n";
ok $expected eq $received;
```

in the normal mode, you won't see any of these prints. But if you run the test with *t/TEST -verbose*, you will see something like this:

```
# testing : feature foo
# expected: 2
# received: 2
ok 2
```

When you develop the test you should always put the debug statements there, and once the test works for you do not comment out or delete these debug statements. This is because if some user reports a failure in some test, you can ask him to run the failing test in the verbose mode and send you back the report. It'll be much easier to understand what the problem is if you get these debug printings from the user.

In the section Writing Tests several helper functions which make the tests writing easier are discussed.

For more details about the `Test::Harness` module please refer to its manpage. Also see the `Test` manpage about Perl's test suite.

## 3.3 Prerequisites

In order to use `Apache::Test` it has to be installed first.

Install `Apache::Test` using the familiar procedure:

```
% cd Apache-Test
% perl Makefile.PL
% make && make test && make install
```

If you install `mod_perl 2.0`, you get `Apache::Test` installed as well.

## 3.4 Running Tests

It's much easier to copy-cat things, than creating from scratch. It's much easier to develop tests, when you have some existing system that you can test, see how it works and build your own testing environment in a similar fashion. Therefore let's first look at how the existing test environments work.

You can look at the `modperl-2.0's` or `httpd-test's` (*perl-framework*) testing environments which both use `Apache::Test` for their test suites.

### 3.4.1 Testing Options

Run:

```
% t/TEST -help
```

to get the list of options you can use during testing. Most options are covered further in this document.

### 3.4.2 Basic Testing

Running tests is just like for any CPAN Perl module; first we generate the *Makefile* file and build everything with *make*:

```
% perl Makefile.PL [options]
% make
```

Now we can do the testing. You can run the tests in two ways. The first one is usual:

```
% make test
```

but it adds quite an overhead, since it has to check that everything is up to date (the usual *make* source change control). Therefore you have to run it only once after *make* and for re-running the tests it's faster to run the tests directly via:

```
% t/TEST
```

When *make test* or *t/TEST* are run, all tests found in the *t* directory (files ending with *.t* are recognized as tests) will be run.

### 3.4.3 Individual Testing

To run a single test, simply specify it at the command line. For example to run the test file *t/protocol/echo.t*, execute:

```
% t/TEST protocol/echo
```

Notice that you don't have to add the *t/* prefix and *.t* extension for the test filenames if you specify them explicitly, but you can have these as well. Therefore the following are all valid commands:

```
% t/TEST    protocol/echo.t
% t/TEST t/protocol/echo
% t/TEST t/protocol/echo.t
```

The server will be stopped if it was already running and a new one will be started before running the *t/protocol/echo.t* test. At the end of the test the server will be shut down.

When you run specific tests you may want to run them in the verbose mode, and depending on how the test was written, you may get more debug information under this mode. This mode is turned on with *-verbose* option:

#### 3.4.4 Repetitive Testing

```
% t/TEST -verbose protocol/echo
```

You can run groups of tests at once. This command:

```
% ./t/TEST modules protocol/echo
```

will run all the tests in *t/modules/* directory, followed by *t/protocol/echo.t* test.

### 3.4.4 Repetitive Testing

By default when you run the test without *-run-tests* option, the server will be started before the testing and stopped at the end. If during a debugging process you need to re-run tests without a need to restart the server, you can start the server once:

```
% t/TEST -start-httpd
```

and then run the test(s) with *-run-tests* option many times:

```
% t/TEST -run-tests
```

without waiting for the server to restart.

When you are done with tests, stop the server with:

```
% t/TEST -stop-httpd
```

When the server is started you can modify *.t* files and rerun the tests without restarting the server. However if you modify response handlers, you must restart the server for changes to take an effect. However the changes are done in the perl code only, it's possible to orrange for Apache::Test to handle the code reload without restarting the server.

The *-start-httpd* option always stops the server first if any is running.

Normally when *t/TEST* is run without specifying the tests to run, the tests will be sorted alphabetically. If tests are explicitly passed as arguments to *t/TEST* they will be run in a specified order.

### 3.4.5 Parallel Testing

Sometimes you need to run more than one Apache::Test framework instances at the same time. In this case you have to use different ports for each instance. You can specify explicitly which port to use, using the *-port* configuration option. For example to run the server on port 34343:

```
% t/TEST -start-httpd -port=34343
```

or by setting an environment variable *APACHE\_PORT* to the desired value before starting the server.

Specifying the port explicitly may not be the most convenient option if you happen to run many instances of the Apache::Test framework. The *-port=select* option comes to help. This option will automatically pick for the next available port. For example if you run:

```
% t/TEST -start-httpd -port=select
```

and there is already one server from a different test suite which uses the default port 8529, the new server will try to use a higher port.

There is one problem that remains to be resolved though. It's possible that two or more servers running `-port=select` will still decide to use the same port, because when the server is configured it only tests whether the port is available but doesn't call `bind()` immediately. Therefore there is a race condition here, which needs to be resolved. Currently the workaround is to start the instances of the `Apache::Test` framework with a slight delay between each other. Depending on the speed of your machine, 4-5 seconds can be a good choice. That's approximately the time it takes to configure and start the server on a quite slow machine.

### 3.4.6 Verbose Mode

In case something goes wrong you should run the tests in the verbose mode:

```
% t/TEST -verbose
```

In this case the test may print useful information, like what values it expects and what values it receives, given that the test is written to report these. In the silent mode (without `-verbose`) these printouts are filtered out by `Test::Harness`. When running in the *verbose* mode usually it's a good idea to run only problematic tests to minimize the size of the generated output.

When debugging problems it helps to keep the *error\_log* file open in another console, and see the debug output in the real time via `tail(1)`:

```
% tail -f t/logs/error_log
```

Of course this file gets created only when the server starts, so you cannot run `tail(1)` on it before the server starts. Every time `t/TEST -clean` is run, *t/logs/error\_log* gets deleted, therefore you have to run the `tail(1)` command again, when the server is started.

### 3.4.7 Colored Trace Mode

If your terminal supports colored text you may want to set the environment variable `APACHE_TEST_COLOR` to 1 to enable the colored tracing when running in the non-batch mode, which makes it easier to tell the reported errors and warnings, from the rest of the notifications.

### 3.4.8 Controlling the Apache::Test's Signal to Noise Ratio

In addition to controlling the verbosity of the test scripts, you can control the amount of information printed by the `Apache::Test` framework itself. Similar to Apache's log levels, `Apache::Test` uses these levels for controlling its signal to noise ratio:

```
emerg alert crit error warning notice info debug
```

where *emerg* is the for the most important messages and *debug* for the least important ones.

Currently the default level is *info*, therefore any messages which fall into the *info* category and above (*notice*, *warning*, etc). If for example you want to see the *debug* messages you can change the default level using *-trace* option:

```
% t/TEST -trace=debug ...
```

or if you want to get only *warning* messages and above, use:

```
% t/TEST -trace=warning ...
```

## 3.4.9 Stress Testing

### 3.4.9.1 The Problem

When we try to test a stateless machine (i.e. all tests are independent), running all tests once ensures that all tested things properly work. However when a state machine is tested (i.e. where a run of one test may influence another test) it's not enough to run all the tests once to know that the tested features actually work. It's quite possible that if the same tests are run in a different order and/or repeated a few times, some tests may fail. This usually happens when some tests don't restore the system under test to its pristine state at the end of the run, which may influence other tests which rely on the fact that they start on pristine state, when in fact it's not true anymore. In fact it's possible that a single test may fail when run twice or three times in a sequence.

### 3.4.9.2 The Solution

To reduce the possibility of such dependency errors, it's important to run random testing repeated many times with many different pseudo-random engine initialization seeds. Of course if no failures get spotted that doesn't mean that there are no tests inter-dependencies, unless all possible combinations were run (exhaustive approach). Therefore it's possible that some problems may still be seen in production, but this testing greatly minimizes such a possibility.

The `Apache::Test` framework provides a few options useful for stress testing.

- **-times**

You can run the tests N times by using the *-times* option. For example to run all the tests 3 times specify:

```
% t/TEST -times=3
```

- **-order**

It's possible that certain tests aren't cleaning up after themselves and modify the state of the server, which may influence other tests. But since normally all the tests are run in the same order, the potential problem may not be discovered until the code is used in production, where the real world testing



hits the problem. Therefore in order to try to detect as many problems as possible during the testing process, it's may be useful to run tests in different orders.

This if of course mosly useful in conjunction with *-times=N* option.

Assuming that we have tests a, b and c:

- **-order=rotate**

rotate the tests: a, b, c, a, b, c

- **-order=repeat**

repeat the tests: a, a, b, b, c, c

- **-order=random**

run in the random order, e.g.: a, c, c, b, a, b

In this mode the seed picked by `srand()` is printed to `STDOUT`, so it then can be used to rerun the tests in exactly the same order (remember to log the output).

- **-order=SEED**

used to initialize the pseudo-random algorithm, which allows to reproduce the same sequence of tests. For example if we run:

```
% t/TEST -order=random -times=5
```

and the seed 234559 is used, we can repeat the same order of tests, by running:

```
% t/TEST -order=234559 -times=5
```

Alternatively, the environment variable `APACHE_TEST_SEED` can be set to the value of a seed when *-order=random* is used. e.g. under `bash(1)`:

```
% APACHE_TEST_SEED=234559 t/TEST -order=random -times=5
```

or with any shell program if you have the `env(1)` utility:

```
$ env APACHE_TEST_SEED=234559 t/TEST -order=random -times=5
```

### 3.4.9.3 Resolving Sequence Problems

When this kind of testing is used and a failure is detected there are two problems:

1. First is to be able to reproduce the problem so if we think we fixed it, we could verify the fix. This one is easy, just remember the sequence of tests run till the failed test and rerun the same sequence once again after the problem has been fixed.

2. Second is to be able to understand the cause of the problem. If during the random test the failure has happened after running 400 tests, how can we possibly know which previously running tests has caused to the failure of the test 401. Chances are that most of the tests were clean and don't have inter-dependency problem. Therefore it'd be very helpful if we could reduce the long sequence to a minimum. Preferably 1 or 2 tests. That's when we can try to understand the cause of the detected problem.

### 3.4.9.4 Apache::TestSmoke Solution

Apache::TestSmoke attempts to solve both problems. When it's run, at the end of each iteration it reports the minimal sequence of tests causing a failure. This doesn't always succeed, but works in many cases.

You should create a small script to drive Apache::TestSmoke, usually *t/SMOKE.PL*. If you don't have it already, create it:

```
#file:t/SMOKE.PL
#-----
#!perl

use strict;
use warnings FATAL => 'all';

use FindBin;
use lib "$FindBin::Bin/../../Apache-Test/lib";
use lib "$FindBin::Bin/../../lib";

use Apache::TestSmoke ();

Apache::TestSmoke->new(@ARGV)->run;
```

Usually *Makefile.PL* converts it into *t/SMOKE* while adjusting the perl path, but you can create *t/SMOKE* in first place as well.

*t/SMOKE* performs the following operations:

1. Runs the tests randomly until the first failure is detected. Or non-randomly if the option *-order* is set to *repeat* or *rotate*.
2. Then it tries to reduce that sequence of tests to a minimum, and this sequence still causes to the same failure.
3. It reports all the successful reductions as it goes to STDOUT and report file of the format: *smoke-report-<date>.txt*.

In addition the systems build parameters are logged into the report file, so the detected problems could be reproduced.

4. Goto 1 and run again using a new random seed, which potentially should detect different failures.

Currently for each reduction path, the following reduction algorithms are applied:

1. Binary search: first try the upper half then the lower.
2. Random window: randomize the left item, then the right item and return the items between these two points.

You can get the usage information by executing:

```
% t/SMOKE -help
```

By default you don't need to supply any arguments to run it, simply execute:

```
% t/SMOKE
```

If you want to work on certain tests you can specify them in the same way you do with *t/TEST*:

```
% t/SMOKE foo/bar foo/tar
```

If you already have a sequence of tests that you want to reduce (perhaps because a previous run of the smoke testing didn't reduce the sequence enough to be able to diagnose the problem), you can request to do just that:

```
% t/SMOKE -order=rotate -times=1 foo/bar foo/tar
```

*-order=rotate* is used just to override the default *-order=random*, since in this case we want to preserve the order. We also specify *-times=1* for the same reason (override the default which is 50).

You can override the number of `srand()` iterations to perform (read: how many times to randomize the sequence), the number of times to repeat the tests (the default is 10) and the path to the file to use for reports:

```
% t/SMOKE -times=5 -iterations=20 -report=../myreport.txt
```

Finally, any other options passed will be forwarded to *t/TEST* as is.

### 3.4.10 RunTime Configuration Overriding

After the server is configured during `make test` or with *t/TEST -config*, it's possible to explicitly override certain configuration parameters. The override-able parameters are listed when executing:

```
% t/TEST -help
```

Probably the most useful parameters are:

- **-preamble**

configuration directives to add at the beginning of *httpd.conf*. For example to turn the tracing on:

```
% t/TEST -preamble "PerlTrace all"
```

- **-postamble**

configuration directives to add at the end of *httpd.conf*. For example to load a certain Perl module:

```
% t/TEST -postamble "PerlModule MyDebugMode"
```

- **-user**

run as user *nobody*:

```
% t/TEST -user nobody
```

- **-port**

run on a different port:

```
% t/TEST -port 8799
```

- **-servername**

run on a different server:

```
% t/TEST -servername test.example.com
```

- **-httpd**

configure an httpd other than the default (that apxs figures out):

```
% t/TEST -httpd ~/httpd-2.0/httpd
```

- **-apxs**

switch to another apxs:

```
% t/TEST -apxs ~/httpd-2.0-prefork/bin/apxs
```

For a complete list of override-able configuration parameters see the output of `t/TEST -help`.

### ***3.4.11 Request Generation and Response Options***

We have mentioned already the most useful run-time options. Here are some other options that you may find useful during testing.

- **-ping**

Ping the server to see whether it runs

```
% t/TEST -ping
```

Ping the server and wait until the server starts, report waiting time.

```
% t/TEST -ping=block
```

This can be useful in conjunction with *-run-tests* option during debugging:

```
% t/TEST -ping=block -run-tests
```

normally, *-run-tests* will immediately quit if it detects that the server is not running, but with *-ping=block* in effect, it'll wait indefinitely for the server to start up.

- **-head**

Issue a HEAD request. For example to request */server-info*:

```
% t/TEST -head /server-info
```

- **-get**

Request the body of a certain URL via GET.

```
% t/TEST -get /server-info
```

If no URL is specified */* is used.

Also you can issue a GET request but to get only headers as a response (e.g. useful to just check *Content-length*)

```
% t/TEST -head -get /server-info
```

GET URL with authentication credentials:

```
% t/TEST -get /server-info -username dougm -password domination
```

(please keep the password secret!)

- **-post**

Generate a POST request.

Read content to POST from string:

```
% t/TEST -post /TestApache__post -content 'name=dougm&company=covalent'
```

Read content to POST from STDIN:

```
% t/TEST -post /TestApache__post -content - < foo.txt
```

Generate a content body of 1024 bytes in length:

```
% t/TEST -post /TestApache__post -content x1024
```

The same but print only the response headers, e.g. useful to just check Content-length:

```
% t/TEST -post -head /TestApache__post -content x1024
```

- **-header**

Add headers to (-get|-post|-head) request:

```
% t/TEST -get -header X-Test=10 -header X-Host=example.com /server-info
```

- **-ssl**

Run all tests through mod\_ssl:

```
% t/TEST -ssl
```

- **-http11**

Run all tests with HTTP/1.1 (KeepAlive) requests:

```
% t/TEST -http11
```

- **-proxy**

Run all tests through mod\_proxy:

```
% t/TEST -proxy
```

- 

The debugging options *-debug* and *-breakpoint* are covered in the Debugging Tests section.

For a complete list of available switches see the output of `t/TEST -help`.

#### 3.4.12 Batch Mode

When running in the batch mode and redirecting STDOUT, this state is automatically detected and the *no color* mode is turned on, under which the program generates a minimal output to make the log files useful. If this doesn't work and you still get all the mess printed during the interactive run, set the `APACHE_TEST_NO_COLOR=1` environment variable.

## 3.5 Setting Up Testing Environment

We will assume that you setup your testing environment even before you have started coding the project, which is a very smart thing to do. Of course it'll take you more time upfront, but it'll will save you a lot of time during the project developing and debugging stages. The extreme programming methodology says that tests should be written before starting the code development.

### 3.5.1 Basic Testing Environment

So the first thing is to create a package and all the helper files, so later on we can distribute it on CPAN. We are going to develop an `Apache::Amazing` module as an example.

```
% h2xs -AXn Apache::Amazing
Writing Apache/Amazing/Amazing.pm
Writing Apache/Amazing/Makefile.PL
Writing Apache/Amazing/README
Writing Apache/Amazing/test.pl
Writing Apache/Amazing/Changes
Writing Apache/Amazing/MANIFEST
```

`h2xs` is a nifty utility that gets installed together with Perl and helps us to create some of the files we will need later.

However we are going to use a little bit different files layout, therefore we are going to move things around a bit.

We want our module to live in the *Apache-Amazing* directory, so we do:

```
% mv Apache/Amazing Apache-Amazing
% rmdir Apache
```

From now on the *Apache-Amazing* directory is our working directory.

```
% cd Apache-Amazing
```

We don't need the *test.pl*. as we are going to create a whole testing environment:

```
% rm test.pl
```

We want our package to reside under the *lib* directory, so later we will be able to do live testing, without rerunning make every time we change the code:

```
% mkdir lib
% mkdir lib/Apache
% mv Amazing.pm lib/Apache
```

Now we adjust the *lib/Apache/Amazing.pm* to look like this:

```
#file:lib/Apache/Amazing.pm
#-----
package Apache::Amazing;

use strict;
use warnings;

use Apache::RequestRec ();
use Apache::RequestIO ();

$Apache::Amazing::VERSION = '0.01';
```

### 3.5.1 Basic Testing Environment

```
use Apache::Const -compile => 'OK';

sub handler {
    my $r = shift;
    $r->content_type('text/plain');
    $r->print("Amazing!");
    return Apache::OK;
}
1;
__END__
... pod documentation goes here...
```

The only thing it does is setting the *text/plain* header and responding with *"Amazing!"*.

Next adjust or create the *Makefile.PL* file:

```
#file:Makefile.PL
#-----
require 5.6.1;

use ExtUtils::MakeMaker;

use lib qw(..../bilib/lib lib );

use Apache::TestMM qw(test clean); #enable 'make test'

# prerequisites
my %require =
(
    "Apache::Test" => "", # any version will do
);
my @scripts = qw(t/TEST);

# accept the configs from command line
Apache::TestMM::filter_args();
Apache::TestMM::generate_script('t/TEST');

WriteMakefile(
    NAME          => 'Apache::Amazing',
    VERSION_FROM  => 'lib/Apache/Amazing.pm',
    PREREQ_PM     => \%require,
    clean         => {
        FILES => "@{ clean_files() }",
    },
    ($] >= 5.005 ?
        (ABSTRACT_FROM => 'lib/Apache/Amazing.pm',
         AUTHOR         => 'Stas Bekman <stas (at) stason.org>',
         ) : ( )
    ),
);

sub clean_files {
    return [@scripts];
}
```



Apache::TestMM will do a lot of thing for us, such as building a complete Makefile with proper *'test'* and *'clean'* targets, automatically converting *.PL* and *conf/\*.in* files and more.

As you see we specify a prerequisites hash with *Apache::Test* in it, so if the package gets distributed on CPAN, CPAN.pm shell will know to fetch and install this required package.

Next we create the test suite, which will reside in the *t* directory:

```
% mkdir t
```

First we create *t/TEST.PL* which will be automatically converted into *t/TEST* during *perl Makefile.PL* stage:

```
#file:t/TEST.PL
#-----
#!perl

use strict;
use warnings FATAL => 'all';

use lib qw(lib);

use Apache::TestRunPerl ();

Apache::TestRunPerl->new->run(@ARGV);
```

Assuming that *Apache::Test* is already installed on your system and Perl can find it. If not you should tell Perl where to find it. For example you could add:

```
use lib qw(Apache-Test/lib);
```

to *t/TEST.PL*, if *Apache::Test* is located in a parallel directory.

As you can see we didn't write the real path to the Perl executable, but *#!perl*. When *t/TEST* is created the correct path will be placed there automatically.

Next we need to prepare extra Apache configuration bits, which will reside in *t/conf*:

```
% mkdir t/conf
```

We create the *t/conf/extra.conf.in* file which will be automatically converted into *t/conf/extra.conf* before the server starts. If the file has any placeholders like *@documentroot@*, these will be replaced with the real values specific for the used server. In our case we put the following configuration bits into this file:

```
#file:t/conf/extra.conf.in
#-----
# this file will be Include-d by @ServerRoot@/httpd.conf

# where Apache::Amazing can be found
PerlSwitches -I@ServerRoot@/../lib
# preload the module
PerlModule Apache::Amazing
```

### 3.5.1 Basic Testing Environment

```
<Location /test/amazing>
    SetHandler modperl
    PerlResponseHandler Apache::Amazing
</Location>
```

As you can see we just add a simple `<Location>` container and tell Apache that the namespace `/test/amazing` should be handled by `Apache::Amazing` module running as a `mod_perl` handler. Notice that:

```
SetHandler modperl
```

is `mod_perl 2.0` configuration, if you are running under `mod_perl 1.0` use:

```
SetHandler perl-script
```

which also works for `mod_perl 2.0`.

Now we can create a simple test:

```
#file:t/basic.t
#-----
use strict;
use warnings FATAL => 'all';

use Apache::Amazing;
use Apache::Test;
use Apache::TestUtil;
use Apache::TestRequest 'GET_BODY';

plan tests => 2;

ok 1; # simple load test

my $url = '/test/amazing';
my $data = GET_BODY $url;

ok t_cmp(
    "Amazing!",
    $data,
    "basic test",
);
```

Now create the *README* file.

```
% touch README
```

Don't forget to put in the relevant information about your module, or arrange for `ExtUtils::Maker::WriteMakefile()` to do this for you with:

```
#file:Makefile.PL
#-----
WriteMakefile(
    #...
    dist => {
        PREOP => 'pod2text lib/Apache/Amazing.pm > ${DISTVNAME}/README',
    },
    #...
);
```

in this case *README* will be created from the documentation POD sections in *lib/Apache/Amazing.pm*, but the file has to exist for *make dist* to succeed.

and finally we adjust or create the *MANIFEST* file, so we can prepare a complete distribution. Therefore we list all the files that should enter the distribution including the *MANIFEST* file itself:

```
#file:MANIFEST
#-----
lib/Apache/Amazing.pm
t/TEST.PL
t/basic.t
t/conf/extra.conf.in
Makefile.PL
Changes
README
MANIFEST
```

That's it. Now we can build the package. But we need to know the location of the *apxs* utility from the installed *httpd* server. We pass its path as an option to *Makefile.PL*:

```
% perl Makefile.PL -apxs ~/httpd/prefork/bin/apxs
% make
% make test

basic.....ok
All tests successful.
Files=1, Tests=2, 1 wallclock secs ( 0.52 cusr + 0.02 csys = 0.54 CPU)
```

To install the package run:

```
% make install
```

Now we are ready to distribute the package on CPAN:

```
% make dist
```

will create the package which can be immediately uploaded to CPAN. In this example the generated source package with all the required files will be called: *Apache-Amazing-0.01.tar.gz*.

The only thing that we haven't done and hope that you will do is to write the POD sections for the *Apache::Amazing* module, explaining how amazingly it works and how amazingly it can be deployed by other users.

## 3.5.2 Extending Configuration Setup

Sometimes you need to add extra *httpd.conf* configuration and perl startup specific to your project that uses `Apache::Test`. This can be accomplished by creating the desired files with an extension *.in* in the *t/conf/* directory and running:

```
panic% t/TEST -config
```

which for each file with the extension *.in* will create a new file, without this extension, convert any template placeholders into real values and link it from the main *httpd.conf*. The latter happens only if the file have the following extensions:

- **.conf.in**

will add to *t/conf/httpd.conf*:

```
Include foo.conf
```

- **.pl.in**

will add to *t/conf/httpd.conf*:

```
PerlRequire foo.pl
```

- **other**

other files with *.in* extension will be processed as well, but not linked from *httpd.conf*.

Files whose name matches the following pattern:

```
/\..last\.(conf|pl).in$/
```

will be included very last in *httpd.conf*.

As mentioned before the converted files are created, any special token in them are getting replaced with the appropriate values. For example the token `@ServerRoot@` will be replaced with the value defined by the `ServerRoot` directive, so you can write a file that does the following:

```
#file:my-extra.conf.in
#-----
PerlSwitches -I@ServerRoot@/../lib
```

and assuming that the *ServerRoot* is `~/modperl-2.0/t/`, when *my-extra.conf* will be created, it'll look like:

```
#file:my-extra.conf
#-----
PerlSwitches -I~/modperl-2.0/t/../lib
```

The valid tokens are defined in `%Apache::TestConfig::Usage` and also can be seen in the output of `t/TEST -help`'s *configuration options* section. The tokens are case insensitive.

### 3.5.3 Special Configuration Files

Some of the files in the *t/conf* directory have a special meaning, since the `Apache::Test` framework uses them for the minimal configuration setup. But they can be overridden:

- if the file *t/conf/httpd.conf.in* exists, it will be used instead of the default template (in *Apache/Test-Config.pm*).
- if the file *t/conf/extra.conf.in* exists, it will be used to generate *t/conf/extra.conf* with `@variable@` substitutions.
- if the file *t/conf/extra.conf* exists, it will be included by *httpd.conf*.
- if the file *t/conf/modperl\_extra.pl* exists, it will be included by *httpd.conf* as a `mod_perl` file (`PerlRequire`).

### 3.5.4 Inheriting from System-wide httpd.conf

`Apache::Test` tries to find a global *httpd.conf* file and inherit its configuration when autogenerating *t/conf/httpd.conf*. For example it picks `LoadModule` directives.

It's possible to explicitly specify which file to inherit from using the `-httpd_conf` option. For example during the build:

```
% perl Makefile.PL -httpd_conf /path/to/httpd.conf
```

or during the configuration:

```
% t/TEST -conf -httpd_conf /path/to/httpd.conf
```

Certain projects need to have a control of what gets inherited. For example if your global *httpd.conf* includes a directive:

```
LoadModule apreq_module "/home/joe/apache2/modules/mod_apreq.so"
```

And you want to run the test suite for `Apache::Request 2.0`, inheriting the above directive will load the pre-installed *mod\_apreq.so* and not the newly built one, which is wrong. In such cases it's possible to tell the test suite which modules shouldn't be inherited. In our example `Apache-Request` has the following code in *t/TEST.PL*:

```
use base 'Apache::TestRun';
$Apache::TestTrace::Level = 'debug';
main:-->new->run(@ARGV);

sub pre_configure {
    my $self = shift;
    # Don't load an installed mod_apreq
    Apache::TestConfig::autoconfig_skip_module_add('mod_apreq.c');
}
```

it subclasses `Apache::TestRun` and overrides the `pre_configure` method, which excludes the module `mod_apreq.c` from the list of inherited modules (notice that the extension is `.c`).

## 3.6 Apache::Test Framework's Architecture

In the previous section we have written a basic test, which doesn't do much. In the following sections we will explain how to write more elaborate tests.

When you write the test for Apache, unless you want to test some static resource, like fetching a file, usually you have to write a response handler and the corresponding test that will generate a request which will exercise this response handler and verify that the response is as expected. From now we may call these two parts as client and server parts of the test, or request and response parts of the test.

In some cases the response part of the test runs the test inside itself, so all it requires from the request part of the test, is to generate the request and print out a complete response without doing anything else. In such cases `Apache::Test` can auto-generate the client part of the test for you.

### 3.6.1 Developing Response-only Part of a Test

If you write only a response part of the test, `Apache::Test` will automatically generate the corresponding test part that will generate the response. In this case your test should print `'ok 1'`, `'not ok 2'` responses as usual tests do. The autogenerated request part will receive the response and print them out automatically completing the `Test::Harness` expectations.

The corresponding request part of the test is named just like the response part, using the following translation:

```
$response_test =~ s|t/[^/]+/Test([^/]+)/(.*)\.pm$|t/L$1\E/$2.t|;
```

so for example `t/response/TestApache/write.pm` becomes: `t/apache/write.t`.

If we look at the autogenerated test `t/apache/write.t`, we can see that it starts with the warning that it has been autogenerated, so you won't attempt to change it. Then you can see the trace of the calls that generated this test, in case you want to figure out how the test was generated. And finally the test loads the `Apache::TestRequest` module, imports the `GET` shortcut and prints the response's body if it was successful. Otherwise it dies to flag the problem with the server side. The latter is done because there is nothing on the client side, that tells the testing framework that things went wrong. Without it the test will be skipped, and that's not what we want.

```
use Apache::TestRequest 'GET_BODY_ASSERT';
print GET_BODY_ASSERT "/TestApache__write";
```

As you can see the request URI is autogenerated from the response test name:

```
$response_test =~ s|.*\/([^\/]+)\/(.*)\.pm$|/$1__$2|;
```

So *t/response/TestApache/write.pm* becomes: */TestApache\_\_write*.

Now a simple response test may look like this:

```
#file:t/response/TestApache/write.pm
#-----
package TestApache::write;

use strict;
use warnings FATAL => 'all';

use constant BUFSIZ => 512; #small for testing
use Apache::Const -compile => 'OK';

sub handler {
    my $r = shift;
    $r->content_type('text/plain');

    $r->write("1..2\n");
    $r->write("ok 1")
    $r->write("not ok 2")

    Apache::OK;
}
1;
```

[F] `Apache::Const` is `mod_perl 2.0`'s package, if you test under 1.0, use the `Apache::Constants` module instead [F].

The configuration part for this test will be autogenerated by the `Apache::Test` framework and added to the autogenerated file *t/conf/httpd.conf* when `make test` or `t/TEST -configure` is run. In our case the following configuration section will be added:

```
<Location /TestApache__write>
    SetHandler modperl
    PerlResponseHandler TestApache::write
</Location>
```

You should remember to run:

```
% t/TEST -configure
```

so the configuration file will be re-generated when new tests are added.

Also notice that if you manually add configuration the `<Location>` path can't include `' : '` characters in the first segment, due to Apache security protection on WinFU platforms. So please make sure that you don't create entries like:

```
<Location /Foo::bar/>
```

You can include `' : '` characters in the further segments, so this is OK:

```
<Location /tests/Foo::bar/>
```

Of course if your code is not intended to run on WinFU you can ignore this detail.

### 3.6.2 *Developing Response and Request Parts of a Test*

But in most cases you want to write a two parts test where the client (request) parts generates various requests and tests the responses.

It's possible that the client part tests a static file or some other feature that doesn't require a dynamic response. In this case, only the request part of the test should be written.

If you need to write the complete test, with two parts, you proceed just like in the previous section, but now you write the client part of the test by yourself. It's quite easy, all you have to do is to generate requests and check the response. So a typical test will look like this:

```
#file:t/apache/cool.t
#-----
use strict;
use warnings FATAL => 'all';

use Apache::Test;
use Apache::TestUtil;
use Apache::TestRequest 'GET_BODY';

plan tests => 1; # plan one test.

Apache::TestRequest::module('default');

my $config = Apache::Test::config();
my $hostport = Apache::TestRequest::hostport($config) || '';
t_debug("connecting to $hostport");

my $received = GET_BODY "/TestApache__cool";
my $expected = "COOL";

ok t_cmp(
    $expected,
    $received,
    "testing TestApache::cool",
);
```

See the `Apache::TestUtil` manpage for more info on the `t_cmp()` function (e.g. it works with regexs as well).

And the corresponding response part:

```
#file:t/response/TestApache/cool.pm
#-----
package TestApache::cool;

use strict;
use warnings FATAL => 'all';
```



```

use Apache::Const -compile => 'OK';

sub handler {
    my $r = shift;
    $r->content_type('text/plain');

    $r->write("COOL");

    Apache::OK;
}
1;

```

Again, remember to run *t/TEST -clean* before running the new test so the configuration will be created for it.

As you can see the test generates a request to */TestApache\_\_cool*, and expects it to return *"COOL"*. If we run the test:

```
% ./t/TEST t/apache/cool
```

We see:

```

apache/cool....ok
All tests successful.
Files=1, Tests=1, 1 wallclock secs ( 0.52 cusr + 0.02 csys = 0.54 CPU)

```

But if we run it in the debug (verbose) mode, we can actually see what we are testing, what was expected and what was received:

```

apache/cool....1..1
# connecting to localhost:8529
# testing : testing TestApache::cool
# expected: COOL
# received: COOL
ok 1
ok
All tests successful.
Files=1, Tests=1, 1 wallclock secs ( 0.49 cusr + 0.03 csys = 0.52 CPU)

```

So in case in our simple test we have received something different from *COOL* or nothing at all, we can immediately see what's the problem.

The name of the request part of the test is very important. If *Apache::Test* cannot find the corresponding test for the response part it'll automatically generate one and in this case it's probably not what you want. Therefore when you choose the filename for the test, make sure to pick the same *Apache::Test* will pick. So if the response part is named: *t/response/TestApache/cool.pm* the request part should be named *t/apache/cool.t*. See the regular expression that does that in the previous section.

### 3.6.3 *Developing Test Response Handlers in C*

If you need to exercise some C API and you don't have a Perl glue for it, you can still use `Apache::Test` for the testing. It allows you to write response handlers in C and makes it easy to integrate these with other Perl tests and use Perl for request part which will exercise the C module.

The C modules look just like standard Apache C modules, with a couple of differences to:

- **a**  
help them fit into the test suite
- **b**  
allow them to compile nicely with Apache 1.x or 2.x.

The *httpd-test* ASF project is a good example to look at. The C modules are located under: *httpd-test/perl-framework/c-modules/*. Look at *c-modules/echo\_post/echo\_post.c* for a nice simple example. `mod_echo_post` simply echos data that is POSTed to it.

The differences between various tests may be summarized as follows:

- If the first line is:

```
#define HTTPD_TEST_REQUIRE_APACHE 1
```

or

```
#define HTTPD_TEST_REQUIRE_APACHE 2
```

then the test will be skipped unless the version matches. If a module is compatible with the version of Apache used then it will be automatically compiled by *t/TEST* with `-DAPACHE1` or `-DAPACHE2` so you can conditionally compile it to suit different httpd versions.

- If there is a section bounded by:

```
#if CONFIG_FOR_HTTPD_TEST
...
#endif
```

in the *.c* file then that section will be inserted verbatim into *t/conf/httpd.conf* by *t/TEST*.

There is a certain amount of magic which hopefully allows most modules to be compiled for Apache 1.3 or Apache 2.0 without any conditional stuff. Replace XXX with the module name, for example `echo_post` or `random_chunk`:

- You should:

```
#include "apache_httpd_test.h"
```

which should be preceded by an:

```
#define APACHE_HTTPD_TEST_HANDLER XXX_handler
```

*apache\_httpd\_test.h* pulls in a lot of required includes and defines some constants and types that are not defined for Apache 1.3.

- The handler function should be:

```
static int XXX_handler(request_rec *r);
```

- At the end of the file should be an:

```
APACHE_HTTPD_TEST_MODULE(XXX)
```

where XXX is the same as that in APACHE\_HTTPD\_TEST\_HANDLER. This will generate the hooks and stuff.

### 3.6.4 Request and Response Methods

If you have LWP (libwww-perl) installed its `LWP::UserAgent` serves as an user agent in tests, otherwise `Apache::TestClient` tries to emulate partial LWP functionality. So most of the LWP documentation applies here, but the `Apache::Test` framework provides shortcuts that hide many details, making the test writing a simple and swift task. Before using these shortcuts `Apache::TestRequest` should be loaded, and its `import()` method will fetch the shortcuts into the caller namespace:

```
use Apache::TestRequest;
```

Request generation methods issue a request and return a response object (`HTTP::Response` if LWP is available). They are documented in the `HTTP::Request::Common` manpage. The following methods are available:

- **GET**

Issues the GET request. For example, issue a request and retrieve the response content:

```
$url = "$location?foo=1&bar=2";
$res = GET $url;
$str = $res->content;
```

To set request headers, supply them after the `$url`, e.g.:

```
$res = GET $url, 'Content-type' => 'text/html';
```

- **HEAD**

Issues the HEAD request. For example issue a request and check that the response's *Content-type* is *text/plain*:

```
$url = "$location?foo=1&bar=2";
$res = HEAD $url;
ok $res->content_type() eq 'text/plain';
```

- **POST**

Issues the POST request. For example:

```
$content = 'PARAM=%33';
$res = POST $location, content => $content;
```

The second argument to POST can be a reference to an array or a hash with key/value pairs to emulate HTML <form> POSTing.

- **PUT**

Issues the PUT request.

- **OPTIONS**

META: ???

These are two special methods added by the `Apache::Test` framework:

- **UPLOAD**

This special method allows to upload a file or a string which will look as an uploaded file to the server. To upload a file use:

```
UPLOAD $location, filename => $filename;
```

You can add extra request headers as well:

```
UPLOAD $location, filename => $filename, 'X-Header-Test' => 'Test';
```

To upload a string as a file, use:

```
UPLOAD $location, content => 'some data';
```

- **UPLOAD\_BODY**

Retrieves the content from the response resulted from doing UPLOAD. It's equal to:

```
my $body = UPLOAD(@_)->content;
```

For example, this code retrieves the content of the response resulted from file upload request:

```
my $str = UPLOAD_BODY $location, filename => $filename;
```

Once the response object is returned, various response object methods can be applied to it. Probably the most useful ones are:

```
$content = $res->content;
```

to retrieve the content of the response and:

```
$content_type = $res->header('Content-type');
```

to retrieve specific headers.

Refer to the `HTTP::Response` manpage for a complete reference of these and other methods.

A few response retrieval shortcuts can be used to retrieve the wanted parts of the response. To apply these simply add the shortcut name to one of the request shortcuts listed earlier. For example instead of retrieving the content part of the response via:

```
$res = GET $url;  
$str = $res->content;
```

simply use:

```
$str = GET_BODY $url;
```

- **RC**

returns the *response code*, equivalent to:

```
$res->code;
```

For example to test whether some URL is bogus:

```
use Apache::Const 'NOT_FOUND';  
ok GET_RC('/bogus_url') == NOT_FOUND;
```

You usually need to import and use `Apache::Const` constants for the response code comparisons, rather than using codes' corresponding numerical values directly. You can import groups of code as well. For example:

```
use Apache::Const ':common';
```

Refer to the `Apache::Const` manpage for a complete reference. Also you may need to use `APR` and `mod_perl` constants, which reside in `APR::Const` and `ModPerl::Const` modules respectively.

- **OK**

tests whether the response was successful, equivalent to:

```
$res->is_success;
```

For example:

```
ok GET_OK '/foo';
```

- **STR**

returns the response (both, headers and body) as a string and is equivalent to:

```
$res->as_string;
```

Mostly useful for debugging, for example:

```
use Apache::TestUtil;
t_debug POST_STR '/test.pl', content => 'foo';
```

- **HEAD**

returns the headers part of the response as a multi-line string.

For example, this code dumps all the response headers:

```
use Apache::TestUtil;
t_debug GET_HEAD '/index.html';
```

- **BODY**

returns the response body and is equivalent to:

```
$res->content;
```

For example, this code validates that the response's body is the one that was expected:

```
use Apache::TestUtil;
ok GET_BODY('/index.html') eq $expect;
```

- **BODY\_ASSERT**

Same as the BODY shortcut, but will assert if the request has failed. So for example if the test's output is generated on the server side, the client side may only need to print out what the server has sent and we want it to report that the test has failed if the request has failed:

```
use Apache::TestUtil;
print GET_BODY_ASSERT "/foo"
```

## 3.6.5 *Other Request Generation helpers*

META: these methods need documentation

Request part:

```
Apache::TestRequest::scheme('http'); #force http for t/TEST -ssl
Apache::TestRequest::module($module);
my $config = Apache::Test::config();
my $hostport = Apache::TestRequest::hostport($config);
```

Getting the request object? `Apache::TestRequest::user_agent()`

### 3.6.6 Starting Multiple Servers

By default the `Apache::Test` framework sets up only a single server to test against.

In some cases you need to have more than one server. If this is the situation, you have to override the `maxclients` configuration directive, whose default is 1. Usually this is done in `t/TEST.PL` by subclassing the parent test run class and overriding the `new_test_config()` method. For example if the parent class is `Apache::TestRunPerl`, you can change your `t/TEST.PL` to be:

```
use strict;
use warnings FATAL => 'all';

use lib "../lib"; # test against the source lib for easier dev
use lib map {"../blib/$_", "../../blib/$_" } qw(lib arch);

use Apache::TestRunPerl ();

package MyTest;

our @ISA = qw(Apache::TestRunPerl);

# subclass new_test_config to add some config vars which will be
# replaced in generated httpd.conf
sub new_test_config {
    my $self = shift;

    $self->{conf_opts}->{maxclients} = 2;

    return $self->SUPER::new_test_config;
}

MyTest->new->run(@ARGV);
```

### 3.6.7 Multiple User Agents

By default the `Apache::Test` framework uses a single user agent which talks to the server (this is the LWP user agent, if you have LWP installed). You almost never use this agent directly in the tests, but via various wrappers. However if you need a second user agent you can clone these. For example:

```
my $ua2 = Apache::TestRequest::user_agent()->clone;
```

### 3.6.8 Hitting the Same Interpreter (Server Thread/Process Instance)

When a single instance of the server thread/process is running, all the tests go through the same server. However if the `Apache::Test` framework was configured to to run a few instances, two subsequent sub-tests may not hit the same server instance. In certain tests (e.g. testing the closure effect or the BEGIN blocks) it's important to make sure that a sequence of sub-tests are run against the same server instance. The `Apache::Test` framework supports this internally.

Here is an example from `ModPerl::Registry` closure tests. Using the counter closure problem under `ModPerl::Registry`:

```
#file:cgi-bin/closure.pl
#-----
#!perl -w
print "Content-type: text/plain\r\n\r\n";

# this is a closure (when compiled inside handler()):
my $counter = 0;
counter();

sub counter {
    #warn "$$";
    print ++$counter;
}
```

If this script get invoked twice in a row and we make sure that it gets executed by the same server instance, the first time it'll return 1 and the second time 2. So here is the gist of the request part that makes sure that its two subsequent requests hit the same server instance:

```
#file:closure.t
#-----
...
my $url = "/same_interp/cgi-bin/closure.pl";
my $same_interp = Apache::TestRequest::same_interp_tie($url);

# should be no closure effect, always returns 1
my $first = req($same_interp, $url);
my $second = req($same_interp, $url);
ok t_cmp(
    1,
    $first && $second && ($second - $first),
    "the closure problem is there",
);
sub req {
    my($same_interp, $url) = @_;
    my $res = Apache::TestRequest::same_interp_do($same_interp,
                                                    \&GET, $url);
    return $res ? $res->content : undef;
}
```

In this test we generate two requests to *cgi-bin/closure.pl* and expect the returned value to increment for each new request, because of the closure problem generated by `ModPerl::Registry`. Since we don't know whether some other test has called this script already, we simply check whether the subtraction of the two subsequent requests' outputs gives a value of 1.

The test starts by requesting the server to tie a single instance to all requests made with a certain identifier. This is done using the `same_interp_tie()` function which returns a unique server instance's identifier. From now on any requests made through `same_interp_do()` and supplying this identifier as the first argument will be served by the same server instance. The second argument to `same_interp_do()` is the method to use for generating the request and the third is the URL to use. Extra arguments can be supplied if needed by the request generation method (e.g. headers).



This technique works for testing purposes where we know that we have just a few server instances. What happens internally is when `same_interp_tie()` is called the server instance that served it returns its unique UUID, so when we want to hit the same server instance in subsequent requests we generate the same request until we learn that we are being served by the server instance that we want. This magic is done by using a fixup handler which returns OK only if it sees that its unique id matches. As you understand this technique would be very inefficient in production with many server instances.

## 3.7 Writing Tests

All the communications between tests and `Test::Harness` which executes them is done via STDOUT. I.e. whatever tests want to report they do by printing something to STDOUT. If a test wants to print some debug comment it should do it starting on a separate line, and each debug line should start with `#`. The `t_debug()` function from the `Apache::TestUtil` package should be used for that purpose.

### 3.7.1 Defining How Many Sub-Tests Are to Be Run

Before sub-tests of a certain test can be run it has to declare how many sub-tests it is going to run. In some cases the test may decide to skip some of its sub-tests or not to run any at all. Therefore the first thing the test has to print is:

```
1..M\n
```

where `M` is a positive integer. So if the test plans to run 5 sub-tests it should do:

```
print "1..5\n";
```

In `Apache::Test` this is done as follows:

```
use Apache::Test;
plan tests => 5;
```

### 3.7.2 Skipping a Whole Test

Sometimes when the test cannot be run, because certain prerequisites are missing. To tell `Test::Harness` that the whole test is to be skipped do:

```
print "1..0 # skipped because of foo is missing\n";
```

The optional comment after `# skipped` will be used as a reason for test's skipping. Under `Apache::Test` the optional last argument to the `plan()` function can be used to define prerequisites and skip the test:

```
use Apache::Test;
plan tests => 5, $test_skipping_prerequisites;
```

This last argument can be:

- **a SCALAR**

the test is skipped if the scalar has a false value. For example:

```
plan tests => 5, 0;
```

But this won't hint the reason for skipping therefore it's better to use `have()`:

```
plan tests => 5,
  have 'LWP',
    { "not Win32" => sub { $^O eq 'MSWin32' } };
```

- **an ARRAY reference**

`have_module()` is called for each value in this array. The test is skipped if `have_module()` returns false (which happens when at least one C or Perl module from the list cannot be found). For example:

```
plan tests => 5, [qw(mod_index mod_mime)];
```

- **a CODE reference**

the tests will be skipped if the function returns a false value. For example:

```
plan tests => 5, \&have_lwp;
```

the test will be skipped if LWP is not available

There is a number of useful functions whose return value can be used as a last argument for `plan()`:

- **have\_module()**

`have_module()` tests for presense of Perl modules or C modules *mod\_\**. It accepts a list of modules or a reference to the list. If at least one of the modules is not found it returns a false value, otherwise it returns a true value. For example:

```
plan tests => 5, have_module qw(Chatbot::Eliza CGI mod_proxy);
```

will skip the whole test unless both Perl modules `Chatbot::Eliza` and `CGI` and the C module *mod\_proxy.c* are available.

- **have\_min\_module\_version()**

Used to require a minimum version of a module

For example:

```
plan tests => 5, have_min_module_version(CGI => 2.81);
```

requires `CGI.pm` version 2.81 or higher.

Currently works only for perl modules.

- **have()**

have() called as a last argument of plan() can impose multiple requirements at once.

have()'s arguments can include scalars, which are passed to have\_module(), and hash references. If hash references are used, the keys, are strings, containing a reason for a failure to satisfy this particular entry, the values are the condition, which are satisfaction if they return true. If the value is a scalar it's used as is. If the value is a code reference, it gets executed at the time of check and its return value is used to check the condition. If the condition check fails, the provided (in a key) reason is used to tell user why the test was skipped.

For example:

```
plan tests => 5,
    have 'LWP',
        { "perl >= 5.8.0 is required" => ($] >= 5.008) },
        { "not Win32"                => sub { $^O eq 'MSWin32' },
          "foo is disabled"          => \&is_foo_enabled,
        },
    'cgid';
```

In this example, we require the presense of the LWP Perl module, mod\_cgid, that we run under perl >= 5.7.3 on Win32, and that is\_foo\_enabled returns true. If any of the requirements from this list fail, the test will be skipped and each failed requiremnt will print a reason for its failure.

- **have\_perl()**

have\_perl('foo') checks whether the value of \$Config{foo} or \$Config{usefoo} is equal to 'define'. For example:

```
plan tests => 2, have_perl 'ithreads';
```

if Perl wasn't compiled with -Duseithreads the condition will be false and the test will be skipped.

Also it checks for Perl extensions. For example:

```
plan tests => 5, have_perl 'iolayers';
```

tests whether PerlIO is available.

- **have\_min\_perl\_version()**

Used to require a minimum version of Perl.

For example:

```
plan tests => 5, have_min_perl_version("5.008001");
```

requires Perl 5.8.1 or higher.

- **have\_threads()**

have\_threads checks whether whether threads are supported by both Apache and Perl.

```
plan tests => 2, have_threads;
```

- **under\_construction()**

this is just a shortcut to skip the test while printing:

```
"skipped: this test is under construction";
```

For example:

```
plan tests => 2, under_construction;
```

- **have\_lwp()**

Tests whether the Perl module LWP is installed.

- **have\_http11()**

Tries to tell LWP that sub-tests need to be run under HTTP 1.1 protocol. Fails if the installed version of LWP is not capable of doing that.

- **have\_cgi()**

tests whether mod\_cgi or mod\_cgid is available.

- **have\_apache()**

tests for a specific generation of httpd. For example:

```
plan tests => 2, have_apache 2;
```

will skip the test if not run under the 2nd Apache generation (httpd-2.x.xx).

```
plan tests => 2, have_apache 1;
```

will skip the test if not run under the 1st Apache generation (apache-1.3.xx).

- **have\_min\_apache\_version**

Used to require a minimum version of Apache. For example:

```
plan tests => 5, have_min_apache_version("2.0.40");
```

requires Apache 2.0.40 or higher.

- **have\_apache\_version**

Used to require a specific version of Apache.

For example:

```
plan tests => 5, have_apache_version("2.0.40");
```

requires Apache 2.0.40.

### 3.7.3 *Skipping Numerous Tests*

Just like you can tell `Apache::Test` to run only specific tests, you can tell it to run all but a few tests.

If all files in a directory *t/foo* should be skipped, create:

```
#file:t/foo/all.t
#-----
print "1..0\n";
```

Alternatively you can specify which tests should be skipped from a single file *t/SKIP*. This file includes a list of tests to be skipped. You can include comments starting with `#` and you can use the `*` wildcard for multiply files matching.

For example if in `mod_perl 2.0` test suite we create the following file:

```
#file:t/SKIP
#-----
# skip all files in protocol
protocol

# skip basic cgi test
modules/cgi.t

# skip all filter/input_* files
filter/input*.t
```

In our example the first pattern specifies the directory name *protocol*, since we want to skip all tests in it. But since the skipping is done based on matching the skip patterns from *t/SKIP* against a list of potential tests to be run, some other tests may be skipped as well if they match the pattern. Therefore it's safer to use a pattern like this:

```
protocol/*.t
```

The second pattern skips a single test *modules/cgi.t*. Note that you shouldn't specify the leading *t/*. The *.t* extension is optional, so you can tell:

### 3.7.4 Reporting a Success or a Failure of Sub-tests

```
# skip basic cgi test
modules/cgi
```

The last pattern tells `Apache::Test` to skip all the tests starting with *filter/input*.

### 3.7.4 Reporting a Success or a Failure of Sub-tests

After printing the number of planned sub-tests, and assuming that the test is not skipped, the tests is running its sub-tests and each sub-test is expected to report its success or failure by printing *ok* or *not ok* respectively followed by its sequential number and a new line. For example:

```
print "ok 1\n";
print "not ok 2\n";
print "ok 3\n";
```

In `Apache::Test` this is done using the `ok()` function which prints *ok* if its argument is a true value, otherwise it prints *not ok*. In addition it keeps track of how many times it was called, and every time it prints an incremental number, therefore you can move sub-tests around without needing to remember to adjust sub-test's sequential number, since now you don't need them at all. For example this test snippet:

```
use Apache::Test;
use Apache::TestUtil;
plan tests => 3;
ok "success";
t_debug("expecting to fail next test");
ok "";
ok 0;
```

will print:

```
1..3
ok 1
# expecting to fail next test
not ok 2
not ok 3
```

Most of the sub-tests perform one of the following things:

- test whether some variable is defined:

```
ok defined $object;
```

- test whether some variable is a true value:

```
ok $value;
```

or a false value:

```
ok !$value;
```

- test whether a received from somewhere value is equal to an expected value:

```
$expected = "a good value";
$received = get_value();
ok defined $received && $received eq $expected;
```

### 3.7.5 Skipping Sub-tests

If the standard output line contains the substring *# Skip* (with variations in spacing and case) after *ok* or *ok NUMBER*, it is counted as a skipped test. `Test::Harness` reports the text after *# Skip\S\*\s+* as a reason for skipping. So you can count a sub-test as a skipped as follows:

```
print "ok 3 # Skip for some reason\n";
```

or using the `Apache::Test`'s `skip()` function which works similarly to `ok()`:

```
skip $should_skip, $test_me;
```

so if `$should_skip` is true, the test will be reported as skipped. The second argument is the one that's sent to `ok()`, so if `$should_skip` is true, a normal `ok()` sub-test is run. The following example represent four possible outcomes of using the `skip()` function:

```
skip_subtest_1.t
-----
use Apache::Test;
plan tests => 4;

my $ok      = 1;
my $not_ok  = 0;

my $should_skip = "foo is missing";
skip $should_skip, $ok;
skip $should_skip, $not_ok;

$should_skip = '';
skip $should_skip, $ok;
skip $should_skip, $not_ok;
```

now we run the test:

```
% ./t/TEST -run-tests -verbose skip_subtest_1
skip_subtest_1....1..4
ok 1 # skip foo is missing
ok 2 # skip foo is missing
ok 3
not ok 4
# Failed test 4 in skip_subtest_1.t at line 13
Failed 1/1 test scripts, 0.00% okay. 1/4 subtests failed, 75.00% okay.
```

As you can see since `$should_skip` had a true value, the first two sub-tests were explicitly skipped (using `$should_skip` as a reason), so the second argument to `skip` didn't matter. In the last two sub-tests `$should_skip` had a false value therefore the second argument was passed to the `ok()` function. Basically the following code:

### 3.7.6 Running only Selected Sub-tests

```
$should_skip = '';
skip $should_skip, $ok;
skip $should_skip, $not_ok;
```

is equivalent to:

```
ok $ok;
ok $not_ok;
```

However if you want to use `t_cmp()` or some other function call in the arguments to `ok()` that won't quite work since the function will be always called no matter whether the first argument will evaluate to a true or a false value. For example, if you had a function:

```
ok t_cmp($expected, $received, $comment);
```

and now you want to run this sub-test if module `HTTP::Date` is available, changing it to:

```
my $should_skip = eval { require HTTP::Date } ? "" : "missing HTTP::Date";
skip $should_skip, t_cmp($expected, $received, $comment);
```

will still run `t_cmp()` even if `HTTP::Date` is not available. Therefore it's probably better to code it in this way:

```
if (eval {require HTTP::Date}) {
    ok t_cmp($expected, $received, $comment);
}
else {
    skip "Skip HTTP::Date not found";
}
```

### 3.7.6 Running only Selected Sub-tests

`Apache::Test` also allows to write tests in such a way that only selected sub-tests will be run. The test simply needs to switch from using `ok()` to `sok()`. Where the argument to `sok()` is a CODE reference or a BLOCK whose return value will be passed to `ok()`. If sub-tests are specified on the command line only those will be run/passed to `ok()`, the rest will be skipped. If no sub-tests are specified, `sok()` works just like `ok()`. For example, you can write this test:

```
#file:skip_subtest_2.t
#-----
use Apache::Test;
plan tests => 4;
sok {1};
sok {0};
sok sub {'true'};
sok sub {''};
```

and then ask to run only sub-tests 1 and 3 and to skip the rest.



```
% ./t/TEST -verbose skip_subtest_2 1 3
skip_subtest_2....1..4
ok 1
ok 2 # skip skipping this subtest
ok 3
ok 4 # skip skipping this subtest
ok, 2/4 skipped: skipping this subtest
All tests successful, 2 subtests skipped.
```

Only the sub-tests 1 and 3 get executed.

A range of sub-tests to run can be given using the Perl's range operand:

```
% ./t/TEST -verbose skip_subtest_2 2..4
skip_subtest_2....1..4
ok 1 # skip asking this subtest
not ok 2
# Failed test 2
ok 3
not ok 4
# Failed test 4
Failed 1/1 test scripts, 0.00% okay. 2/4 subtests failed, 50.00% okay.
```

In this run, only the first sub-test gets executed.

### 3.7.7 *Todo Sub-tests*

In a safe fashion to skipping specific sub-tests, it's possible to declare some sub-tests as *todo*. This distinction is useful when we know that some sub-test is failing but for some reason we want to flag it as a todo sub-test and not as a broken test. `Test::Harness` recognizes *todo* sub-tests if the standard output line contains the substring `# TODO` after `not ok` or `not ok NUMBER` and is counted as a todo sub-test. The text afterwards is the explanation of the thing that has to be done before this sub-test will succeed. For example:

```
print "not ok 42 # TODO not implemented\n";
```

In `Apache::Test` this can be done with passing a reference to a list of sub-tests numbers that should be marked as *todo* sub-test:

```
plan tests => 7, todo => [3, 6];
```

In this example sub-tests 3 and 6 will be marked as *todo* sub-tests.

### 3.7.8 *Making it Easy to Debug*

Ideally we want all the tests to pass, reporting minimum noise or none at all. But when some sub-tests fail we want to know the reason for their failure. If you are a developer you can dive into the code and easily find out what's the problem, but when you have a user who has a problem with the test suite it'll make his and your life much easier if you make it easy for the user to report you the exact problem.

Usually this is done by printing the comment of what the sub-test does, what is the expected value and what's the received value. This is a good example of debug friendly sub-test:

```
#file:debug_comments.t
#-----
use Apache::Test;
use Apache::TestUtil;
plan tests => 1;

t_debug("testing feature foo");
$expected = "a good value";
$received = "a bad value";
t_debug("expected: $expected");
t_debug("received: $received");
ok defined $received && $received eq $expected;
```

If in this example `$received` gets assigned a *bad value* string, the test will print the following:

```
% t/TEST debug_comments
debug_comments....FAILED test 1
```

No debug help here, since in a non-verbose mode the debug comments aren't printed. If we run the same test using the verbose mode, enabled with `-verbose`:

```
% t/TEST -verbose debug_comments
debug_comments....1..1
# testing feature foo
# expected: a good value
# received: a bad value
not ok 1
```

we can see exactly what's the problem, by visual expecting of the expected and received values.

It's true that adding a few print statements for each sub tests is cumbersome, and adds a lot of noise, when you could just tell:

```
ok "a good value" eq "a bad value";
```

but no fear, `Apache::TestUtil` comes to help. The function `t_cmp()` does all the work for you:

```
use Apache::Test;
use Apache::TestUtil;
ok t_cmp(
    "a good value",
    "a bad value",
    "testing feature foo");
```

`t_cmp()` will handle undef'ined values as well, so you can do:

```
my $expected;
ok t_cmp(undef, $expected, "should be undef");
```

Finally you can use `t_cmp()` for regex comparisons. This feature is mostly useful when there may be more than one valid expected value, which can be described with regex. For example this can be useful to inspect the value of `$@` when `eval()` is expected to fail:

```
eval {foo();}
if ($@) {
    ok t_cmp(qr/^expecting foo/, $@, "func eval");
}
```

which is the same as:

```
eval {foo();}
if ($@) {
    t_debug("func eval");
    ok $@ =~ /^expecting foo/ ? 1 : 0;
}
```

### 3.7.9 Tie-ing STDOUT to a Response Handler Object

It's possible to run the sub-tests in the response handler, and simply return them as a response to the client which in turn will print them out. Unfortunately in this case you cannot use `ok()` and other functions, since they print and don't return the results, therefore you have to do it manually. For example:

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

    $r->print("1..2\n");
    $r->print("ok 1\n");
    $r->print("not ok 2\n");

    return Apache::OK;
}
```

now the client should print the response to STDOUT for `Test::Harness` processing.

If the response handler is configured as:

```
SetHandler perl-script
```

STDOUT is already tied to the request object `$r`. Therefore you can now rewrite the handler as:

```
use Apache::Test;
sub handler {
    my $r = shift;

    Apache::Test::test_pm_refresh();
    plan tests => 2;
    ok "true";
    ok "";

    return Apache::OK;
}
```

However to be on the safe side you also have to call `Apache::Test::test_pm_refresh()` allowing `plan()` and friends to be called more than once per-process.

Under different settings `STDOUT` is not tied to the request object. If the first argument to `plan()` is an object, such as an `Apache::RequestRec` object, `STDOUT` will be tied to it. The `Test.pm` global state will also be refreshed by calling `Apache::Test::test_pm_refresh`. For example:

```
use Apache::Test;
sub handler {
    my $r = shift;

    plan $r, tests => 2;
    ok "true";
    ok "";

    return Apache::OK;
}
```

Yet another alternative to handling the test framework printing inside response handler is to use `Apache::TestToString` class.

The `Apache::TestToString` class is used to capture `Test.pm` output into a string. Example:

```
use Apache::Test;
sub handler {
    my $r = shift;

    Apache::TestToString->start;

    plan tests => 2;
    ok "true";
    ok "";

    my $output = Apache::TestToString->finish;
    $r->print($output);

    return Apache::OK;
}
```

In this example `Apache::TestToString` intercepts and buffers all the output from `Test.pm` and can be retrieved with its `finish()` method. Which then can be printed to the client in one shot. Internally it calls `Apache::Test::test_pm_refresh()` to make sure `plan()`, `ok()` and other functions() will work correctly more than one test is running under the same interpreter.

### 3.7.10 *Helper Functions*

`Apache::TestUtil` provides other helper functions, useful for writing tests, not mentioned in this tutorial:

```
t_cmp()
t_debug()
t_append_file()
t_write_file()
```

```

t_open_file()
t_mkdir()
t_rmtree()
t_is_equal()
t_write_perl_script()
t_write_shell_script()
t_chown()
t_server_log_error_is_expected()
t_server_log_warn_is_expected()
t_client_log_error_is_expected()
t_client_log_warn_is_expected()

```

See the `Apache::TestUtil` manpage for more information.

### 3.7.11 Auto Configuration

If the test is comprised only from the request part, you have to manually configure the targets you are going to use. This is usually done in `t/conf/extra.conf.in`.

If your tests are comprised from the request and response parts, `Apache::Test` automatically adds the configuration section for each response handler it finds. For example for the response handler:

```

package TestResponse::nice;
... some code
1;

```

it will put into `t/conf/httpd.conf`:

```

<Location /TestResponse__nice>
    SetHandler modperl
    PerlResponseHandler TestResponse::nice
</Location>

```

If you want to add some extra configuration directives, use the `__DATA__` section, as in this example:

```

package TestResponse::nice;
... some code
1;
__DATA__
PerlSetVar Foo Bar

```

These directives will be wrapped into the `<Location>` section and placed into `t/conf/httpd.conf`:

```

<Location /TestResponse__nice>
    SetHandler modperl
    PerlResponseHandler TestResponse::nice
    PerlSetVar Foo Bar
</Location>

```

This autoconfiguration feature was added to:

- simplify (less lines) test configuration.
- ensure unique namespace for <Location ...>'s.
- force <Location ...> names to be consistent.
- prevent clashes within main configuration.

### 3.7.11.1 Forcing Configuration Sections into the Top Level

If some directives are supposed to go to the base configuration, i.e. not to be automatically wrapped into <Location> block, you should use a special <Base>..  
</Base> block:

```
__DATA__
<Base>
    PerlSetVar Config ServerConfig
</Base>
PerlSetVar Config LocalConfig
```

Now the autogenerated section will look like this:

```
PerlSetVar Config ServerConfig
<Location /TestResponse__nice>
    SetHandler modperl
    PerlResponseHandler TestResponse::nice
    PerlSetVar Config LocalConfig
</Location>
```

As you can see the <Base>..  
</Base> block has gone. As you can imagine this block was added to support our virtue of laziness, since most tests don't need to add directives to the base configuration and we want to keep the configuration sections in tests to a minimum and let Perl do the rest of the job for us.

### 3.7.11.2 Bypassing Auto-Configuration

In more complicated cases, usually when virtual hosts containers are involved, the auto-configuration might stand in a way and you will simply want to bypass it. If that's the case, put the configuration inside the <NoAutoConfig>..  
</NoAutoConfig> container. For example:

```
<NoAutoConfig>
    <VirtualHost TestPreConnection::note>
        PerlPreConnectionHandler TestPreConnection::note

        <Location /TestPreConnection__note>
            SetHandler modperl
            PerlResponseHandler TestPreConnection::note::response
        </Location>
    </VirtualHost>
</NoAutoConfig>
```

Notice, that the internal sections will be still parsed, tokens @var@ will be substituted and Virtual-Host sections will be rewritten with an automatically assigned port number and ServerName.

### 3.7.11.3 Virtual Hosts

Apache::Test automatically assigns an unused port for the virtual host configuration. Just make sure that you use the package name in the place where you usually specify a *hostname:port* value. For example for the following package:

```
#file:MyApacheTest/Foo.pm
#-----
package MyApacheTest::Foo;
...
1;
__END__
<VirtualHost MyApacheTest::Foo>
    <Location /test_foo>
        ....
    </Location>
</VirtualHost>
```

After running:

```
% t/TEST -conf
```

Check the auto-generated *t/conf/httpd.conf* and you will find what port was assigned. Of course it can change when more tests which require a special virtual host are used.

Now in the request script, you can figure out what port that virtual host was assigned, using the package name. For example:

```
#file:test_foo.t
#-----
use Apache::TestRequest;

my $module = "MyApacheTest::Foo";
my $config = Apache::Test::config();
Apache::TestRequest::module($module);
my $hostport = Apache::TestRequest::hostport($config);

print GET_BODY_ASSERT "http://$hostport/test_foo";
```

### 3.7.11.4 Running Pre-Configuration Code

Sometimes you need to setup things for the test. This usually includes creating directories and files, and populating the latter with some data, which will be used at request time. Instead of performing that operation in the client script every time a test is run, it's usually better to do it once when the server is configured. If you wish to run such a code, all you have to do is to add a special subroutine `APACHE_TEST_CONFIGURE` in the response package (assuming that that response package exists). When server is configured (`t/TEST -conf`) it scans all the response packages for that subroutine and if found runs it.

`APACHE_TEST_CONFIGURE` accepts two arguments: the package name of the file this subroutine is defined in and the `Apache::TestConfig` configuration object.

Here is an example of a package that uses such a subroutine:

```
package TestDirective::perlmodule;

use strict;
use warnings FATAL => 'all';

use Apache::Test ();

use Apache::RequestRec ();
use Apache::RequestIO ();
use File::Spec::Functions qw(catfile);

use Apache::Const -compile => 'OK';

sub handler {
    my $r = shift;

    $r->content_type('text/plain');
    $r->puts($ApacheTest::PerlModuleTest::MAGIC || '');

    Apache::OK;
}

sub APACHE_TEST_CONFIGURE {
    my ($class, $self) = @_;

    my $vars = $self->{vars};
    my $target_dir = catfile $vars->{documentroot}, 'testdirective';

    my $magic = __PACKAGE__;
    my $content = <<EOF;
package ApacheTest::PerlModuleTest;
\ApacheTest::PerlModuleTest::MAGIC = '$magic';
1;
EOF
    my $file = catfile $target_dir,
        'perlmodule-vh', 'ApacheTest', 'PerlModuleTest.pm';
    $self->writefile($file, $content, 1);
}
1;
```

In this example's function a directory is created. Then a file with some perl code as a content is created.

### 3.7.11.5 Controlling the Configuration Order

Sometimes it's important in which order the configuration section of each response package is inserted. `Apache::Test` controls the insertion order using a special token `APACHE_TEST_CONFIG_ORDER`. To decide on the configuration insertion order, `Apache::Test` scans all response packages and tries to match the following pattern:

```
/APACHE_TEST_CONFIG_ORDER\s+([+-]?[d+])/
```



So you can assign any integer number (positive or negative). If the match fails, it's assumed that the token's value is 0. Next a simple numerical search is performed and those configuration sections with lower token value are inserted first.

It's not specified how sections with the same token value are ordered. This usually depends on the order the files were read from the disk, which may vary from machine to machine and shouldn't be relied upon.

As already mentioned by default all configuration sections have a token whose value is 0, meaning that their ordering is unimportant. Now if you want to make sure that some section is inserted first, assign to it a negative number, e.g.:

```
# APACHE_TEST_CONFIG_ORDER -150
```

Now if a new test is added and it has to be the first, add to this new test a token with a negative value whose absolute value is higher than -150, e.g.:

```
# APACHE_TEST_CONFIG_ORDER -151
```

or

```
# APACHE_TEST_CONFIG_ORDER -500
```

Decide how big the gaps should be by thinking ahead. This is similar to the Basic language line numbering ;) In any case, you can always adjust other tests' token if you need to squeeze a number between two consequent integers.

If on the other hand you want to ensure that some test is configured last, use the highest positive number, e.g.:

```
# APACHE_TEST_CONFIG_ORDER 100
```

If some other test needs to be configured just before the one we just inserted, assign a token with a lower value, e.g.:

```
# APACHE_TEST_CONFIG_ORDER 99
```

### ***3.7.12 Threaded versus Non-threaded Perl Test's Compatibility***

Since the tests are supposed to run properly under non-threaded and threaded perl, you have to worry to enclose the threaded perl specific configuration bits in:

```
<IfDefine PERL_USEITHREADS>
... configuration bits
</IfDefine>
```

Apache::Test will start the server with -DPERL\_USEITHREADS if the Perl is ithreaded.

For example `PerlOptions +Parent` is valid only for the threaded perl, therefore you have to write:

```
<IfDefine PERL_USEITHREADS>
    # a new interpreter pool
    PerlOptions +Parent
</IfDefine>
```

Just like the configuration, the test's code has to work for both versions as well. Therefore you should wrap the code specific to the threaded perl into:

```
if (have_perl 'ithreads'){
    # ithread specific code
}
```

which is essentially does a lookup in `$Config{useithreads}`.

### ***3.7.13 Retrieving the Server Configuration Data***

The server configuration data can be retrieved and used in the tests via the configuration object:

```
use Apache::Test;
my $cfg = Apache::Test::config();
```

#### **3.7.13.1 Module Magic Number**

The following code retrieves the major and minor MMN numbers.

```
my $cfg = Apache::Test::config();
my $info = $cfg->{httpd_info};

my $major = $info->{MODULE_MAGIC_NUMBER_MAJOR};
my $minor = $info->{MODULE_MAGIC_NUMBER_MINOR};

print "major=$major, minor=$minor\n";
```

For example for MMN 20011218:0, this code prints:

```
major=20011218, minor=0
```

## **3.8 Debugging Tests**

Sometimes your tests won't run properly or even worse will segfault. There are cases where it's possible to debug broken tests with simple print statements but usually it's very time consuming and ineffective. Therefore it's a good idea to get yourself familiar with Perl and C debuggers, and this knowledge will save you a lot of time and grief in a long run.

### ***3.8.1 Under C debugger***

`mod_perl-2.0` provides built in 'make test' debug facility. So in case you get a core dump during make test, or just for fun, run in one shell:

```
% t/TEST -debug
```

in another shell:

```
% t/TEST -run-tests
```

then the *-debug* shell will have a (gdb) prompt, type where for stacktrace:

```
(gdb) where
```

You can change the default debugger by supplying the name of the debugger as an argument to *-debug*. E.g. to run the server under ddd:

```
% ./t/TEST -debug=ddd
```

META: list supported debuggers

If you debug mod\_perl internals you can set the breakpoints using the *-breakpoint* option, which can be repeated as many times as needed. When you set at least one breakpoint, the server will start running till it meets the *ap\_run\_pre\_config* breakpoint. At this point we can set the breakpoint for the mod\_perl code, something we cannot do earlier if mod\_perl was built as DSO. For example:

```
% ./t/TEST -debug -breakpoint=modperl_cmd_switches \
    -breakpoint=modperl_cmd_options
```

will set the *modperl\_cmd\_switches* and *modperl\_cmd\_options* breakpoints and run the debugger.

If you want to tell the debugger to jump to the start of the mod\_perl code you may run:

```
% ./t/TEST -debug -breakpoint=modperl_hook_init
```

In fact *-breakpoint* automatically turns on the debug mode, so you can run:

```
% ./t/TEST -breakpoint=modperl_hook_init
```

## 3.8.2 Under Perl debugger

When the Perl code misbehaves it's the best to run it under the Perl debugger. Normally started as:

```
% perl -debug program.pl
```

the flow control gets passed to the Perl debugger, which allows you to run the program in single steps and examine its states and variables after every executed statement. Of course you can set up breakpoints and watches to skip irrelevant code sections and watch after certain variables. The *perldebug* and the *perldebug-tut* manpages are covering the Perl debugger in fine details.

The Apache::Test framework extends the Perl debugger and plugs in LWP's debug features, so you can debug the requests. Let's take test *apache/read* from mod\_perl 2.0 and present the features as we go:

META: to be completed

run .t test under the perl debugger

```
% t/TEST -debug perl t/modules/access.t
```

run .t test under the perl debugger (nonstop mode, output to t/logs/perldb.out)

```
% t/TEST -debug perl=nostop t/modules/access.t
```

turn on -v and LWP trace (1 is the default) mode in Apache::TestRequest

```
% t/TEST -debug lwp t/modules/access.t
```

turn on -v and LWP trace mode (level 2) in Apache::TestRequest

```
% t/TEST -debug lwp=2 t/modules/access.t
```

### 3.8.3 Tracing

To get Start the server under strace(1):

```
% t/TEST -debug strace
```

The output goes to *t/logs/strace.log*.

Now in a second terminal run:

```
% t/TEST -run-tests
```

Beware that *t/logs/strace.log* is going to be very big.

META: can we provide strace(1) opts if we want to see only certain syscalls?

## 3.9 Using Apache::Test to Speed up Project Development

When developing a project, as the code is written or modified it is desirable to test it at the same time. If you write tests as you code, or even before you code, Apache::Test can speed up the modify-test code development cycle. The idea is to start the server once and then run the tests without restarting it, and make the server reload the modified modules behind the scenes. This of course works only if you modify plain perl modules. If you develop XS/C components, you have no choice but to restart the server before you want to test the modified code.

First of all, your perl modules need to reside under the *lib* directory, the same way they reside in *blib/lib*. In the section Basic Testing Environment we've already arranged for that. If *Amazing.pm* resides in the top-level directory, it's not possible to perform `'require Apache::Amazing'`. Only after running `make`, the file will be moved to *blib/lib/Apache/Amazing.pm*, which is when we can load it. But you don't want to run `make` every time you change the file. It's both annoying and error-prone, since at times you'd do some change, try to verify it and it will appear to be wrong, and you will try to understand why,

whereas in reality you just forgot to run `make` and the server was testing against the old unmodified version in `blib/lib`. Of you course if you always run `make test` it'll always do the right thing, but it's not the most effecient way to undertake when you want to test a specific test and you do it every few seconds.

The following scenario will make you a much happier Perl developer.

First, we need to instruct `Apache::Test` to modify `@INC`, which we could do in `t/conf/modperl_extra.pl` or `t/conf/extra.conf.in`, but the problem is that you may not want to keep that change in the released package. There is a better way, if the environment variable `APACHE_TEST_LIVE_DEV` is set to a true value, `Apache::Test` will automatically add the `lib/` directory if it exists. Executing:

```
% APACHE_TEST_LIVE_DEV=1 t/TEST -configure
```

will add code to add `/path/to/Apache-Amazing/lib` to `@INC` in `t/conf/modperl_inc.pl`. This technique is convenient since you don't need to modify your code to include that directory.

Second, we need to configure `mod_perl` to use `Apache::Reload` to automatically reload the module when it's changed, by adding following configuration directives to `t/conf/extra.conf.in`:

```
PerlModule Apache::Reload
PerlInitHandler Apache::Reload
PerlSetVar ReloadAll Off
PerlSetVar ReloadModules "Apache::Amazing"
```

(For more information about `Apache::Reload`, depending on the used `mod_perl` generation, refer to the `mod_perl` 1.0 documentation or the `Apache::Reload` manpage for `mod_perl` 2.0.)

now we execute:

```
% APACHE_TEST_LIVE_DEV=1 t/TEST -configure
```

which will generate `t/conf/extra.conf` and start the server:

```
% t/TEST -start
```

from now on, we can modify `Apache/Amazing.pm` and repeatedly run:

```
% t/TEST -run basic
```

without restarting the server.

## 3.10 Writing Tests Methodology

META: to be completed

### ***3.10.1 When Tests Should Be Written***

- **A New feature is Added**

Every time a new feature is added new tests should be added to cover the new feature.

- **A Bug is Reported**

Every time a bug gets reported, before you even attempt to fix the bug, write a test that exposes the bug. This will make much easier for you to test whether your fix actually fixes the bug.

Now fix the bug and make sure that test passes ok.

It's possible that a few tests can be written to expose the same bug. Write them all -- the more tests you have the less chances are that there is a bug in your code.

If the person reporting the bug is a programmer you may try to ask her to write the test for you. But usually if the report includes a simple code that reproduces the bug, it should probably be easy to convert this code into a test.

## **3.11 Other Webserver Regression Testing Frameworks**

- **Puffin**

Puffin is a web application regression testing system. It allows you to test any web application from end to end based application as if it were a "black box" accepting inputs and returning outputs.

It's available from <http://puffin.sourceforge.net/>

## **3.12 References**

- **extreme programming methodology**

Extreme Programming: A Gentle Introduction: <http://www.extremeprogramming.org/>.

Extreme Programming: <http://www.xprogramming.com/>.

See also other sites linked from these URLs.

## **3.13 Maintainers**

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

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

## 3.14 Authors

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

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

## **4 Issuing Correct HTTP Headers**



## 4.1 Description

To make caching of dynamic documents possible, which can give you a considerable performance gain, setting a number of HTTP headers is of a vital importance. This document explains which headers you need to pay attention to, and how to work with them.

As there is always more than one way to do it, I'm tempted to believe one must be the best. Hardly ever am I right.

## 4.2 The Origin of this Chapter

This chapter has been contributed to the documentation by Andreas Koenig. You will find the references and other related info at the bottom of this page. It was previously distributed from CPAN, but this documentation is now its official resting-place.

If you have any questions regarding this specific document only, please refer to Andreas, since he is the guru on this subject. On any other matter please contact the `mod_perl` mailing list.

## 4.3 Why Headers

Dynamic Content is dynamic, after all, so why would anybody care about HTTP headers? Header composition is a task often neglected in the CGI world. Because pages are generated dynamically, you might expect that pages without a `Last-Modified` header are fine, and that an `If-Modified-Since` header in the browser's request can be ignored. This *laissez-faire* principle gets in the way when you try to establish a server that is entirely driven by dynamic components and the number of hits is significant.

If the number of hits is not significant, don't bother to read this document.

If the number of hits is significant, you might want to consider what cache-friendliness means (you may also want to read [4]) and how you can cooperate with caches to increase the performance of your site. Especially if you use Squid in accelerator mode (helpful hints for Squid, see [1]), you will have a strong motivation to cooperate with it. This document may help you to do it correctly.

## 4.4 Which Headers

The HTTP standard (v 1.1 is specified in [3], v 1.0 in [2]) describes lots of headers. In this document, we only discuss those headers which are most relevant to caching.

I have grouped the headers into three groups: date headers, content headers, and the special Vary header.

## 4.4.1 *Date Related Headers*

### 4.4.1.1 Date

Section 14.18 of the HTTP standard deals with the circumstances under which you must or must not send a `Date` header. For almost everything a normal `mod_perl` user is doing, a `Date` header needs to be generated. But the `mod_perl` programmer doesn't have to worry about this header since the Apache server guarantees that this header is sent.

In `http_protocol.c` the `Date` header is set according to `$r->request_time`. A `mod_perl` script can read, but not change, `$r->request_time`.

### 4.4.1.2 Last-Modified

Section 14.29 of the HTTP standard deals with this. The `Last-Modified` header is mostly used as a so-called weak validator. Here are two sentences from the HTTP specs:

```
A validator that does not always change when the resource
changes is a "weak validator."
```

```
One can think of a strong validator as one that changes
whenever the bits of an entity changes, while a weak value
changes whenever the meaning of an entity changes.
```

This tells us that we should consider the semantics of the page we are generating and not the date when we are running. The question is, when did the **meaning** of this page change last time? Let's imagine the document in question is a text-to-gif renderer that takes as input a font to use, background and foreground colours, and a string to render. Although the actual image is created on-the-fly, the semantics of the page are determined when the script was last changed, right?

Actually, a few more things are relevant: the semantics also change a little when you update one of the fonts that may be used or when you update your `ImageMagick` or equivalent program. It's something you should consider, if you want to get it right.

If you have a page which comprises several components, you should ask all the components when they changed their semantic behaviour last time. Then pick the oldest of those times.

`mod_perl` offers you two convenient methods to deal with this header: `update_mtime()` and `set_last_modified()`. These methods and several others are unavailable in the normal `mod_perl` environment but are silently imported when you use `Apache::File`. Refer to the `Apache::File` manpage for more info.

`update_mtime()` takes a UNIX time as its argument and sets Apache's request structure `finfo.st_mtime` to this value. It does so only when the argument is greater than a previously stored `finfo.st_mtime`.

`set_last_modified()` sets the outgoing header `Last-Modified` to the string that corresponds to the stored `finfo.st_mtime`. By passing a UNIX time to `set_last_modified()`, `mod_perl` calls `update_mtime()` with this argument first.

```

use Apache::File;
use Date::Parse;
# Date::Parse parses RCS format, Apache::Util::parsedate doesn't
$Mtime ||=
    Date::Parse::str2time(substr q$Date: 2002/07/31 14:41:49 $, 6);
$r->set_last_modified($Mtime);

```

### 4.4.1.3 Expires and Cache-Control

Section 14.21 of the HTTP standard deals with the Expires header. The purpose of the Expires header is to determine a point in time after which the document should be considered out of date (stale). Don't confuse this with the very different meaning of the Last-Modified header. The Expires header is useful to avoid unnecessary validation from now on until the document expires and it helps the recipients to clean up their stored documents. A sentence from the HTTP standard:

The presence of an Expires field does not imply that the original resource will change or cease to exist at, before, or after that time.

So think before you set up a time when you believe a resource should be regarded as stale. Most of the time I can determine an expected lifetime from "now", that is the time of the request. I would not recommend hardcoding the date of Expiry, because when you forget that you did it, and the date arrives, you will serve "already expired" documents that cannot be cached at all by anybody. If you believe a resource will never expire, read this quote from the HTTP specs:

To mark a response as "never expires," an origin server sends an Expires date approximately one year from the time the response is sent. HTTP/1.1 servers SHOULD NOT send Expires dates more than one year in the future.

Now the code for the mod\_perl programmer who wants to expire a document half a year from now:

```

$r->header_out('Expires',
    HTTP::Date::time2str(time + 180*24*60*60));

```

A very handy alternative to this computation is available in HTTP 1.1, the cache control mechanism. Instead of setting the Expires header you can specify a delta value in a Cache-Control header. You can do that by executing just:

```

$r->header_out('Cache-Control', "max-age=" . 180*24*60*60);

```

which is, of course much cheaper than the first example because perl computes the value only once at compile time and optimizes it into a constant.

As this alternative is only available in HTTP 1.1 and old cache servers may not understand this header, it is advisable to send both headers. In this case the Cache-Control header takes precedence, so the Expires header is ignored on HTTP 1.1 compliant servers. Or you could go with an if/else clause:

```

if ($r->protocol =~ /(\d\.\d)/ && $1 >= 1.1){
    $r->header_out('Cache-Control', "max-age=" . 180*24*60*60);
} else {
    $r->header_out('Expires',
                  HTTP::Date::time2str(time + 180*24*60*60));
}

```

If you restart your Apache server regularly, I'd save the Expires header in a global variable. Oh, well, this is probably over-engineered now.

To avoid caching altogether call:

```
$r->no_cache(1);
```

which sets the headers:

```

Pragma: no-cache
Cache-control: no-cache

```

which should work in major browsers.

Don't set Expires with `$r->header_out` if you use `$r->no_cache`, because `header_out()` takes precedence. The problem that remains is that there are broken browsers which ignore Expires headers.

## 4.4.2 Content Related Headers

### 4.4.2.1 Content-Type

You are most probably familiar with Content-Type. Sections 3.7, 7.2.1 and 14.17 of the HTTP specs cover the details. `mod_perl` has the `content_type()` method to deal with this header, for example:

```
$r->content_type("image/png");
```

Content-Type *should* be included in all messages according to the specs, and Apache will generate one if you don't. It will be whatever is specified in the relevant `DefaultType` configuration directive or `text/plain` if none is active.

### 4.4.2.2 Content-Length

According to section 14.13 of the HTTP specifications, the Content-Length header is the number of octets in the body of a message. If it can be determined prior to sending, it can be very useful for several reasons to include it. The most important reason why it is good to include it is that keepalive requests only work with responses that contain a Content-Length header. In `mod_perl` you can say

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

If you use `Apache::File`, you get the additional `set_content_length()` method for the `Apache` class which is a bit more efficient than the above. You can then say:

```
$r->set_content_length($length);
```

The Content-Length header can have an important impact on caches by invalidating cache entries as the following extract from the specification explains:

```
The response to a HEAD request MAY be cacheable in the sense that
the information contained in the response MAY be used to update a
previously cached entity from that resource.  If the new field values
indicate that the cached entity differs from the current entity (as
would be indicated by a change in Content-Length, Content-MD5, ETag
or Last-Modified), then the cache MUST treat the cache entry as
stale.
```

So be careful never to send a wrong Content-Length, either in a GET or in a HEAD request.

### 4.4.2.3 Entity Tags

An Entity Tag is a validator which can be used instead of, or in addition to, the Last-Modified header. An entity tag is a quoted string which can be used to identify different versions of a particular resource. An entity tag can be added to the response headers like so:

```
$r->header_out("ETag", "\"$VERSION\"");
```

Note: mod\_perl offers the `Apache::set_etag()` method if you have loaded `Apache::File`. It is strongly recommended that you *do not* use this method unless you know what you are doing. `set_etag()` is expecting to be used in conjunction with a static request for a file on disk that has been stat()ed in the course of the current request. It is inappropriate and "dangerous" to use it for dynamic content.

By sending an entity tag you promise the recipient that you will not send the same ETag for the same resource again unless the content is *'equal'* to what you are sending now (see below for what equality means).

The pros and cons of using entity tags are discussed in section 13.3 of the HTTP specs. For us mod\_perl programmers that discussion can be summed up as follows:

There are strong and weak validators. Strong validators change whenever a single bit changes in the response. Weak validators change when the meaning of the response changes. Strong validators are needed for caches to allow for sub-range requests. Weak validators allow a more efficient caching of equivalent objects. Algorithms like MD5 or SHA are good strong validators, but what we usually want, when we want to take advantage of caching, is a good weak validator.

A Last-Modified time, when used as a validator in a request, can be strong or weak, depending on a couple of rules. Please refer to section 13.3.3 of the HTTP standard to understand these rules. This is mostly relevant for range requests as this citation of section 14.27 explains:

```
If the client has no entity tag for an entity, but does have a
Last-Modified date, it MAY use that date in a If-Range header.
```

But it is not limited to range requests. Section 13.3.1 succinctly states that:

The Last-Modified entity-header field value is often used as a cache validator.

The fact that a Last-Modified date may be used as a strong validator can be pretty disturbing if we are in fact changing our output slightly without changing the semantics of the output. To prevent these kinds of misunderstanding between us and the cache servers in the response chain, we can send a weak validator in an ETag header. This is possible because the specs say:

If a client wishes to perform a sub-range retrieval on a value for which it has only a Last-Modified time and no opaque validator, it MAY do this only if the Last-Modified time is strong in the sense described here.

In other words: by sending them an ETag that is marked as weak we prevent them from using the Last-Modified header as a strong validator.

An ETag value is marked as a weak validator by preceding the string W/ to the quoted string, otherwise it is strong. In perl this would mean something like this:

```
$r->header_out('ETag', "W/\ "$VERSION\ " ");
```

Consider carefully which string you choose to act as a validator. You are on your own with this decision because...

... only the service author knows the semantics of a resource well enough to select an appropriate cache validation mechanism, and the specification of any validator comparison function more complex than byte-equality would open up a can of worms. Thus, comparisons of any other headers (except Last-Modified, for compatibility with HTTP/1.0) are never used for purposes of validating a cache entry.

If you are composing a message from multiple components, it may be necessary to combine some kind of version information for all these components into a single string.

If you are producing relatively large documents, or content that does not change frequently, you most likely will prefer a strong entity tag, thus giving caches a chance to transfer the document in chunks. (Anybody in the mood to add a chapter about ranges to this document?)

### 4.4.3 Content Negotiation

Content negotiation is a particularly wonderful feature that was introduced with HTTP 1.1. Unfortunately it is not yet widely supported. Probably the most popular usage scenario of content negotiation is language negotiation. A user specifies in the browser preferences the languages they understand and how well they understand them. The browser includes these settings in an Accept-Language header when it sends the request to the server and the server then chooses from several available representations of the document the one that best fits the user's preferences. Content negotiation is not limited to language. Citing the specs:

HTTP/1.1 includes the following request-header fields for enabling server-driven negotiation through description of user agent capabilities and user preferences: Accept (section 14.1), Accept-Charset (section 14.2), Accept-Encoding (section 14.3), Accept-Language (section 14.4), and User-Agent (section 14.43). However, an origin server is not limited to these dimensions and MAY vary the response based on any aspect of the request, including information outside the request-header fields or within extension header fields not defined by this specification.

### 4.4.3.1 Vary

In order to signal to the recipient that content negotiation has been used to determine the best available representation for a given request, the server must include a Vary header. This tells the recipient which request headers have been used to determine it. So an answer may be generated like this:

```
$r->header_out('Vary', join ", ",
               qw(accept accept-language accept-encoding user-agent));
```

The header of a very cool page may greet the user with something like

```
Hallo Kraut, Dein NutScrape versteht zwar PNG aber leider
kein GZIP.
```

but it has the side effect of being expensive for a caching proxy. As of this writing, Squid (version 2.1PATCH2) does not cache resources that come with a Vary header at all. So unless you find a clever workaround, you won't enjoy your Squid accelerator for these documents :-)

## 4.5 Requests

Section 13.11 of the specifications states that the only two cacheable methods are GET and HEAD.

### 4.5.1 HEAD

Among the above recommended headers, the date-related ones (Date, Last-Modified, and Expires/Cache-Control) are usually easy to produce and thus should be computed for HEAD requests just the same as for GET requests.

The Content-Type and Content-Length headers should be exactly the same as would be supplied to the corresponding GET request. But as it can be expensive to compute them, they can just as well be omitted, since there is nothing in the specs that forces you to compute them.

What is important for the mod\_perl programmer is that the response to a HEAD request *must not* contain a message-body. The code in your mod\_perl handler might look like this:

```
# compute the headers that are easy to compute
if ( $r->header_only ){ # currently equivalent to $r->method eq "HEAD"
    $r->send_http_header;
    return OK;
}
```

If you are running a Squid accelerator, it will be able to handle the whole HEAD request for you, but under some circumstances it may not be allowed to do so.

## 4.5.2 POST

The response to a POST request is not cacheable due to an underspecification in the HTTP standards. Section 13.4 does not forbid caching of responses to POST requests but no other part of the HTTP standard explains how caching of POST requests could be implemented, so we are in a vacuum here and all existing caching servers therefore refuse to implement caching of POST requests. This may change if somebody does the groundwork of defining the semantics for cache operations on POST. Note that some browsers with their more aggressive caching do implement caching of POST requests.

Note: If you are running a Squid accelerator, you should be aware that it accelerates outgoing traffic, but does not bundle incoming traffic. If you have long POST requests, Squid doesn't buy you anything. So always consider using a GET instead of a POST if possible.

## 4.5.3 GET

A normal GET is what we usually write our mod\_perl programs for. Nothing special about it. We send our headers followed by the body.

But there is a certain case that needs a workaround to achieve better cacheability. We need to deal with the "?" in the rel\_path part of the requested URI. Section 13.9 specifies that

```
... caches MUST NOT treat responses to such URIs as fresh unless
the server provides an explicit expiration time. This specifically
means that responses from HTTP/1.0 servers for such URIs SHOULD NOT
be taken from a cache.
```

You're tempted to believe that if we are using HTTP 1.1 and send an explicit expiration time we're on the safe side? Unfortunately reality is a little bit different. It has been a bad habit for quite a long time to misconfigure cache servers such that they treat all GET requests containing a question mark as uncacheable. People even used to mark everything as uncacheable that contained the string `cgi-bin`.

To work around this bug in the HEAD requests, I have stopped calling my CGI directories `cgi-bin` and I have written the following handler that lets me work with CGI-like query strings without rewriting the software (such as `Apache::Request` and `CGI.pm`) that deals with them.

```
sub handler {
    my($r) = @_;
    my $uri = $r->uri;
    if ( my($u1,$u2) = $uri =~ / ^ ([^?]+?) ; ([^?]* ) $ /x ) {
        $r->uri($u1);
        $r->args($u2);
    } elsif ( my($u1,$u2) = $uri =~ m/^(.?)%3[Bb](.*)$/ ) {
        # protect against old proxies that escape volens nolens
        # (see HTTP standard section 5.1.2)
        $r->uri($u1);
        $u2 =~ s/%3B/;/gi;
        $u2 =~ s/%26/;/gi; # &
```



```

    $u2 =~ s/%3D/=gi;
    $r->args($u2);
}
DECLINED;
}

```

This handler must be installed as a `PerlPostReadRequestHandler`.

The handler takes any request that contains one or more semicolons but *no* question mark such that the first semicolon is interpreted as a question mark and everything after that as the query string. You can now exchange the request:

```
http://example.com/query?BGCOLOR=blue;FGCOLOR=red
```

with:

```
http://example.com/query;BGCOLOR=blue;FGCOLOR=red
```

Thus it allows the co-existence of queries from ordinary forms that are being processed by a browser and predefined requests for the same resource. It has one minor bug: Apache doesn't allow percent-escaped slashes in such a query string. So instead of:

```
http://example.com/query;BGCOLOR=blue;FGCOLOR=red;FONT=%2Ffont%2Fbla
```

you have to use:

```
http://example.com/query;BGCOLOR=blue;FGCOLOR=red;FONT=/font/bla
```

## 4.5.4 Conditional GET

A rather challenging request mod\_perl programmers can get is the conditional GET, which typically means a request with an `If-Modified-Since` header. The HTTP specifications have this to say:

```

The semantics of the GET method change to a "conditional GET"
if the request message includes an If-Modified-Since,
If-Unmodified-Since, If-Match, If-None-Match, or If-Range
header field. A conditional GET method requests that the
entity be transferred only under the circumstances described
by the conditional header field(s). The conditional GET method
is intended to reduce unnecessary network usage by allowing
cached entities to be refreshed without requiring multiple
requests or transferring data already held by the client.

```

So how can we reduce the unnecessary network usage in such a case? mod\_perl makes it easy for you by offering Apache's `meets_conditions()`. You have to set up your `Last-Modified` (and possibly `ETag`) header before calling this method. If the return value of this method is anything other than `OK`, you should return that value from your handler and you're done. Apache handles the rest for you. The following example is taken from [5]:

```

if((my $rc = $r->meets_conditions) != OK) {
    return $rc;
}
#else ... go and send the response body ...

```

If you have a Squid accelerator running, it will often handle the conditionals for you and you can enjoy its extremely fast responses for such requests by reading the *access.log*. Just grep for `TCP_IMS_HIT/304`. But as with a HEAD request there are circumstances under which it may not be allowed to do so. That is why the origin server (which is the server you're programming) needs to handle conditional GETs as well even if a Squid accelerator is running.

## 4.6 Avoiding Dealing with Headers

There is another approach to dynamic content that is possible with `mod_perl`. This approach is appropriate if the content changes relatively infrequently, if you expect lots of requests to retrieve the same content before it changes again and if it is much cheaper to test whether the content needs refreshing than it is to refresh it.

In this case a `PerlFixupHandler` can be installed for the relevant location. It tests whether the content is up to date. If so, it returns `DECLINED` and lets the Apache core serve the content from a file. Otherwise, it regenerates the content into the file, updates the `$r->finfo` status and again returns `DECLINED` so that Apache serves the updated file. Updating `$r->finfo` can be achieved by calling

```
$r->filename($file); # force update of finfo
```

even if this seems redundant because the filename is already equal to `$file`. Setting the filename has the side effect of doing a `stat()` on the file. This is important because otherwise Apache would use the out of date `finfo` when generating the response header.

## 4.7 References

### 4.7.1 [1]

Stas Bekman: `mod_perl` Guide

### 4.7.2 [2]

T. Berners-Lee et al.: Hypertext Transfer Protocol -- HTTP/1.0, RFC 1945.

### 4.7.3 [3]

R. Fielding et al.: Hypertext Transfer Protocol -- HTTP/1.1, RFC 2616.

#### 4.7.4 [4]

Martin Hamilton: Cachebusting - cause and prevention, draft-hamilton-cachebusting-01. Also available online at <http://vancouver-webpages.com/CacheNow/>

#### 4.7.5 [5]

Lincoln Stein, Doug MacEachern: Writing Apache Modules with Perl and C, O'Reilly, 1-56592-567-X. Selected chapters available online at <http://www.modperl.com/>.

### 4.8 Other resources

- Prevent the browser from Caching a page <http://www.pacificnet.net/~johnr/meta.html>

This page is an explanation of using the Meta tag to prevent caching, by browser or proxy, of an individual page wherein the page in question has data that may be of a sensitive nature as in a "form page for submittal" and the creator of the page wants to make sure that the page does not get submitted twice. Please notice that some of the information on this page is a little bit outdated, but it's still a good resource for those who cannot generate their own HTTP headers.

- Web Caching and Content Delivery Resources <http://www.web-caching.com/>

### 4.9 Maintainers

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

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

### 4.10 Authors

- Andreas Koenig <andreas.koenig (at) anima.de>

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

## **5 mod\_perl for ISPs. mod\_perl and Virtual Hosts**

## 5.1 Description

mod\_perl hosting by ISPs: fantasy or reality? This section covers some topics that might be of interest to users looking for ISPs to host their mod\_perl-based website, and ISPs looking for a way to provide such services.

Today, it is a reality: there are a number of ISPs hosting mod\_perl, although the number of these is not as big as we would have liked it to be. To see a list of ISPs that can provide mod\_perl hosting, see [ISPs supporting mod\\_perl](#).

Note: At this moment this document talks about mod\_perl 1.0. mod\_perl 2.0 coupled with the perchild mpm (<http://httpd.apache.org/docs-2.0/mod/perchild.html>) will allow different users run mod\_perl handlers under different uid/gid. This solves the problem of secure co-existing of more than one mod\_perl user on the same httpd server.

## 5.2 ISPs providing mod\_perl services - a fantasy or a reality

- You installed mod\_perl on your box at home, and you fell in love with it. So now you want to convert your CGI scripts (which currently are running on your favorite ISPs machine) to run under mod\_perl. Then you discover that your ISP has never heard of mod\_perl, or he refuses to install it for you.
- You are an old sailor in the ISP business, you have seen it all, you know how many ISPs are out there and you know that the sales margins are too low to keep you happy. You are looking for some new service almost no one else provides, to attract more clients to become your users and hopefully to have a bigger slice of the action than your competitors.

If you are a user asking for a mod\_perl service or an ISP considering to provide this service, this section should make things clear for both of you.

An ISP has three choices:

1. ISPs probably cannot let users run scripts under mod\_perl on the main server. There are many reasons for this:

Scripts might leak memory, due to sloppy programming. There will not be enough memory to run as many servers as required, and clients will be not satisfied with the service because it will be slower.

The question of file permissions is a very important issue: any user who is allowed to write and run a CGI script can at least read (if not write) any other files that belong to the same user and/or group the web server is running as. Note that it's impossible to run suEXEC and cgiwrap extensions under mod\_perl 1.0.

Another issue is the security of the database connections. If you use `Apache::DBI`, by hacking the `Apache::DBI` code you can pick a connection from the pool of cached connections even if it was opened by someone else and your scripts are running on the same web server.

Yet another security issue is a potential compromise of the systems via user's code running on the web servers. One of the possible solutions here is to use chroot(1) or jail(8) mechanisms which allow to run subsystems isolated from the main system. So if a subsystem gets compromised the whole system is still safe.

There are many more things to be aware of so at this time you have to say *No*.

Of course as an ISP you can run mod\_perl internally, without allowing your users to map their scripts so that they will run under mod\_perl. If as a part of your service you provide scripts such as guest books, counters etc. which are not available for user modification, you can still have these scripts running very fast.

2. But, hey why can't I let my users run their own servers, so I can wash my hands of them and don't have to worry about how dirty and sloppy their code is (assuming that the users are running their servers under their own usernames, to prevent them from stealing code and data from each other).

This option is fine as long as you are not concerned about your new systems resource requirements. If you have even very limited experience with mod\_perl, you know that mod\_perl enabled Apache servers while freeing up your CPU and allowing you to run scripts very much faster, have huge memory demands (5-20 times that of plain Apache).

The size depends on the code length, the sloppiness of the programming, possible memory leaks the code might have and all that multiplied by the number of children each server spawns. A very simple example: a server, serving an average number of scripts, demanding 10Mb of memory which spawns 10 children, already raises your memory requirements by 100Mb (the real requirement is actually much smaller if your OS allows code sharing between processes and programmers exploit these features in their code). Now multiply the average required size by the number of server users you intend to have and you will get the total memory requirement.

Since ISPs never say *No*, you'd better take the inverse approach - think of the largest memory size you can afford then divide it by one user's requirements as I have shown in this example, and you will know how many mod\_perl users you can afford :)

But you cannot tell how much memory your users may use? Their requirements from a single server can be very modest, but do you know how many servers they will run? After all, they have full control of *httpd.conf* - and it has to be this way, since this is essential for the user running mod\_perl.

All this rumbling about memory leads to a single question: is it possible to prevent users from using more than X memory? Or another variation of the question: assuming you have as much memory as you want, can you charge users for their average memory usage?

If the answer to either of the above questions is *Yes*, you are all set and your clients will prize your name for letting them run mod\_perl! There are tools to restrict resource usage (see for example the man pages for `ulimit(3)`, `getrlimit(2)`, `setrlimit(2)` and `sysconf(3)`, the last three have the corresponding Perl modules: `BSD::Resource` and `Apache::Resource`).

[ReaderMETA]: If you have experience with other resource limiting techniques please share it with us. Thank you!

If you have chosen this option, you have to provide your client with:

- Shutdown and startup scripts installed together with the rest of your daemon startup scripts (e.g. `/etc/rc.d` directory), so that when you reboot your machine the user's server will be correctly shutdown and will be back online the moment your system starts up. Also make sure to start each server under the username the server belongs to, or you are going to be in big trouble!
- Proxy services (in forward or httpd accelerator mode) for the user's virtual host. Since the user will have to run their server on an unprivileged port ( $>1024$ ), you will have to forward all requests from `user.given.virtual.hostname:80` (which is `user.given.virtual.hostname` without the default port 80) to `your.machine.ip:port_assigned_to_user`. You will also have to tell the users to code their scripts so that any self referencing URLs are of the form `user.given.virtual.hostname`.

Letting the user run a mod\_perl server immediately adds a requirement for the user to be able to restart and configure their own server. Only root can bind to port 80, this is why your users have to use port numbers greater than 1024.

Another solution would be to use a setuid startup script, but think twice before you go with it, since if users can modify the scripts they will get a root access. For more information refer to the section "SUID Start-up Scripts".

- Another problem you will have to solve is how to assign ports between users. Since users can pick any port above 1024 to run their server, you will have to lay down some rules here so that multiple servers do not conflict.

A simple example will demonstrate the importance of this problem: I am a malicious user or I am just a rival of some fellow who runs his server on your ISP. All I need to do is to find out what port my rival's server is listening to (e.g. using `netstat(8)`) and configure my own server to listen on the same port. Although I am unable to bind to this port, imagine what will happen when you reboot your system and my startup script happens to be run before my rival's one! I get the port first, now all requests will be redirected to my server. I'll leave to your imagination what nasty things might happen then.

Of course the ugly things will quickly be revealed, but not before the damage has been done.

Luckily there are special tools that can ensure that users that aren't authorized to bind to certain ports (above 1024) won't be able to do so. One such a tool is called `cbs` and its documentation can be found at <http://www.epita.fr/~flav/cbs/doc/html>.

Basically you can preassign each user a port, without them having to worry about finding a free one, as well as enforce `MaxClients` and similar values by implementing the following scenario:

For each user have two configuration files, the main file, *httpd.conf* (non-writable by user) and the user's file, *username.httpd.conf* where they can specify their own configuration parameters and override the ones defined in *httpd.conf*. Here is what the main configuration file looks like:

```
httpd.conf
-----
# Global/default settings, the user may override some of these
...
...
# Included so that user can set his own configuration
Include username.httpd.conf

# User-specific settings which will override any potentially
# dangerous configuration directives in username.httpd.conf
...
...

username.httpd.conf
-----
# Settings that your user would like to add/override,
# like <Location> and PerlModule directives, etc.
```

Apache reads the global/default settings first. Then it reads the *Include*'d *username.httpd.conf* file with whatever settings the user has chosen, and finally it reads the user-specific settings that we don't want the user to override, such as the port number. Even if the user changes the port number in his *username.httpd.conf* file, Apache reads our settings last, so they take precedence. Note that you can use Perl sections to make the configuration much easier.

3. A much better, but costly solution is *co-location*. Let the user hook his (or your) stand-alone machine into your network, and forget about this user. Of course either the user or you will have to undertake all the system administration chores and it will cost your client more money.

Who are the people who seek `mod_perl` support? They are people who run serious projects/businesses. Money is not usually an obstacle. They can afford a stand alone box, thus achieving their goal of autonomy whilst keeping their ISP happy.

## 5.2.1 Virtual Servers Technologies

As we have just seen one of the obstacles of using `mod_perl` in ISP environments, is the problem of isolating customers using the same machine from each other. A number of virtual servers (don't confuse with virtual hosts) technologies (both commercial and Open Source) exist today. Here are some of them:

- **The User-mode Linux Kernel**

<http://user-mode-linux.sourceforge.net/>

User-Mode Linux is a safe, secure way of running Linux versions and Linux processes. Run buggy software, experiment with new Linux kernels or distributions, and poke around in the internals of Linux, all without risking your main Linux setup.



User-Mode Linux gives you a virtual machine that may have more hardware and software virtual resources than your actual, physical computer. Disk storage for the virtual machine is entirely contained inside a single file on your physical machine. You can assign your virtual machine only the hardware access you want it to have. With properly limited access, nothing you do on the virtual machine can change or damage your real computer, or its software.

So if you want to completely protect one user from another and yourself from your users this might be yet another alternative to the solutions suggested at the beginning of this chapter.

- **VMWare Technology**

Allows running a few instances of the same or different OSs on the same machine. This technology comes in two flavors:

Open source: <http://savannah.nongnu.org/projects/plex86/>

Commercial: <http://www.vmware.com/>

So you may want to run a separate OS for each of your clients

- **freeVSD Technology**

freeVSD (<http://www.freevds.org>), an open source project sponsored by Idaya Ltd. The software enables ISPs to securely partition their physical servers into many *virtual servers*, each capable of running popular hosting applications such as Apache, Sendmail and MySQL.

- **S/390 IBM server**

Quoting from: <http://www.s390.ibm.com/linux/vif/>

"The S/390 Virtual Image Facility enables you to run tens to hundreds of Linux server images on a single S/390 server. It is ideally suited for those who want to move Linux and/or UNIX workloads deployed on multiple servers onto a single S/390 server, while maintaining the same number of distinct server images. This provides centralized management and operation of the multiple image environment, reducing complexity, easing administration and lowering costs."

In two words, this a great solution to huge ISPs, as it allows you to run hundreds of mod\_perl servers while having only one box to maintain. The drawback is the price :)

Check out this scalable mailing list thread for more details from those who know: <http://archive.develooper.com/scalable@arctic.org/msg00235.html>

## 5.3 Virtual Hosts in the guide

If you are about to use *Virtual Hosts* you might want to read these sections:

Apache Configuration in Perl

Easing the Chores of Configuring Virtual Hosts with mod\_macro

Is There a Way to Provide a Different startup.pl File for Each Individual Virtual Host

Is There a Way to Modify @INC on a Per-Virtual-Host or Per-Location Basis.

A Script From One Virtual Host Calls a Script with the Same Path From the Other Virtual Host

## 5.4 Maintainers

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

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

## 5.5 Authors

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

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

## **6 Choosing an Operating System and Hardware**

## 6.1 Description

Before you use the techniques documented on this site to tune servers and write code you need to consider the demands which will be placed on the hardware and the operating system. There is no point in investing a lot of time and money in configuration and coding only to find that your server's performance is poor because you did not choose a suitable platform in the first place.

While the tips below could apply to many web servers, they are aimed primarily at administrators of mod\_perl enabled Apache server.

Because hardware platforms and operating systems are developing rapidly (even while you are reading this document), this discussion must be in general terms.

## 6.2 Choosing an Operating System

First let's talk about Operating Systems (OSs).

Most of the time I prefer to use Linux or something from the \*BSD family. Although I am personally a Linux devotee, I do not want to start yet another OS war.

I will try to talk about what characteristics and features you should be looking for to support an Apache/mod\_perl server, then when you know what you want from your OS, you can go out and find it. Visit the Web sites of the operating systems you are interested in. You can gauge user's opinions by searching the relevant discussions in newsgroups and mailing list archives. Deja - <http://deja.com> and eGroups - <http://egroups.com> are good examples. I will leave this fan research to the reader.

### 6.2.1 *Stability and Robustness*

Probably the most important features in an OS are stability and robustness. You are in an Internet business. You do not keep normal 9am to 5pm working hours like many conventional businesses you know. You are open 24 hours a day. You cannot afford to be off-line, for your customers will go shop at another service like yours (unless you have a monopoly :). If the OS of your choice crashes every day, first do a little investigation. There might be a simple reason which you can find and fix. There are OSs which won't work unless you reboot them twice a day. You don't want to use the OS of this kind, no matter how good the OS' vendor sales department. Do not follow flushy advertisements, follow developers advices instead.

Generally, people who have used the OS for some time can tell you a lot about its stability. Ask them. Try to find people who are doing similar things to what you are planning to do, they may even be using the same software. There are often compatibility issues to resolve. You may need to become familiar with patching and compiling your OS. It's easy.

## ***6.2.2 Memory Management***

You want an OS with a good memory management, some OSs are well known as memory hogs. The same code can use twice as much memory on one OS compared to another. If the size of the `mod_perl` process is 10Mb and you have tens of these running, it definitely adds up!

## ***6.2.3 Memory Leaks***

Some OSs and/or their libraries (e.g. C runtime libraries) suffer from memory leaks. A leak is when some process requests a chunk of memory for temporary storage, but then does not subsequently release it. The chunk of memory is not then available for any purpose until the process which requested it dies. We cannot afford such leaks. A single `mod_perl` process sometimes serves thousands of requests before it terminates. So if a leak occurs on every request, the memory demands could become huge. Of course our code can be the cause of the memory leaks as well (check out the `Apache::Leak` module on CPAN). Certainly, we can reduce the number of requests to be served over the process' life, but that can degrade performance.

## ***6.2.4 Sharing Memory***

We want an OS with good memory sharing capabilities. As we have seen, if we preload the modules and scripts at server startup, they are shared between the spawned children (at least for a part of a process' life - memory pages can become "dirty" and cease to be shared). This feature can reduce memory consumption a lot!

## ***6.2.5 Cost and Support***

If we are in a big business we probably do not mind paying another \$1000 for some fancy OS with bundled support. But if our resources are low, we will look for cheaper and free OSs. Free does not mean bad, it can be quite the opposite. Free OSs can have the best support we can find. Some do. It is very easy to understand - most of the people are not rich and will try to use a cheaper or free OS first if it does the work for them. Since it really fits their needs, many people keep using it and eventually know it well enough to be able to provide support for others in trouble. Why would they do this for free? One reason is for the spirit of the first days of the Internet, when there was no commercial Internet and people helped each other, because someone helped them in first place. I was there, I was touched by that spirit and I am keen to keep that spirit alive.

But, let's get back to our world. We are living in material world, and our bosses pay us to keep the systems running. So if you feel that you cannot provide the support yourself and you do not trust the available free resources, you must pay for an OS backed by a company, and blame them for any problem. Your boss wants to be able to sue someone if the project has a problem caused by the external product that is being used in the project. If you buy a product and the company selling it claims support, you have someone to sue or at least to put the blame on.

If we go with Open Source and it fails we do not have someone to sue... wrong--in the last years many companies have realized how good the Open Source products are and started to provide an official support for these products. So your boss cannot just dismiss your suggestion of using an Open Source Operating System. You can get a paid support just like with any other commercial OS vendor.

Also remember that the less money you spend on OS and Software, the more you will be able to spend on faster and stronger hardware.

### ***6.2.6 Discontinued Products***

The OSs in this hazard group tend to be developed by a single company or organization.

You might find yourself in a position where you have invested a lot of time and money into developing some proprietary software that is bundled with the OS you chose (say writing a `mod_perl` handler which takes advantage of some proprietary features of the OS and which will not run on any other OS). Things are under control, the performance is great and you sing with happiness on your way to work. Then, one day, the company which supplies your beloved OS goes bankrupt (not unlikely nowadays), or they produce a newer incompatible version and they will not support the old one (happens all the time). You are stuck with their early masterpiece, no support and no source code! What are you going to do? Invest more money into porting the software to another OS...

Everyone can be hit by this mini-disaster so it is better to check the background of the company when making your choice. Even so you never know what will happen tomorrow - in 1980, a company called Tektronix did something similar to one of the Guide reviewers with its microprocessor development system. The guy just had to buy another system. He didn't buy it from Tektronix, of course. The second system never really worked very well and the firm he bought it from went bust before they ever got around to fixing it. So in 1982 he wrote his own microprocessor development system software. It didn't take long, it works fine, and he's still using it 18 years later.

Free and Open Source OSs are probably less susceptible to this kind of problem. Development is usually distributed between many companies and developers, so if a person who developed a really important part of the kernel lost interest in continuing, someone else will pick the falling flag and carry on. Of course if tomorrow some better project shows up, developers might migrate there and finally drop the development: but in practice people are often given support on older versions and helped to migrate to current versions. Development tends to be more incremental than revolutionary, so upgrades are less traumatic, and there is usually plenty of notice of the forthcoming changes so that you have time to plan for them.

Of course with the Open Source OSs you can have the source! So you can always have a go yourself, but do not under-estimate the amounts of work involved. There are many, many man-years of work in an OS.

### ***6.2.7 OS Releases***

Actively developed OSs generally try to keep pace with the latest technology developments, and continually optimize the kernel and other parts of the OS to become better and faster. Nowadays, Internet and networking in general are the hottest topics for system developers. Sometimes a simple OS upgrade to the latest stable version can save you an expensive hardware upgrade. Also, remember that when you buy new hardware, chances are that the latest software will make the most of it.

If a new product supports an old one by virtue of backwards compatibility with previous products of the same family, you might not reap all the benefits of the new product's features. Perhaps you get almost the same functionality for much less money if you were to buy an older model of the same product.

## 6.3 Choosing Hardware

Sometimes the most expensive machine is not the one which provides the best performance. Your demands on the platform hardware are based on many aspects and affect many components. Let's discuss some of them.

In the discussion we use terms that may be unfamiliar to some readers:

- Cluster - a group of machines connected together to perform one big or many small computational tasks in a reasonable time. Clustering can also be used to provide 'fail-over' where if one machine fails its processes are transferred to another without interruption of service. And you may be able to take one of the machines down for maintenance (or an upgrade) and keep your service running - the main server will simply not dispatch the requests to the machine that was taken down.
- Load balancing - users are given the name of one of your machines but perhaps it cannot stand the heavy load. You can use a clustering approach to distribute the load over a number of machines. The central server, which users access initially when they type the name of your service, works as a dispatcher. It just redirects requests to other machines. Sometimes the central server also collects the results and returns them to the users. You can get the advantages of clustering too.

There are many load balancing techniques. (See High-Availability Linux Project for more info.)

- NIC - Network Interface Card. A hardware component that allows to connect your machine to the network. It performs packets sending and receiving, newer cards can encrypt and decrypt packets and perform digital signing and verifying of the such. These are coming in different speeds categories varying from 10Mbps to 10Gbps and faster. The most used type of the NIC card is the one that implements the Ethernet networking protocol.
- RAM - Random Access Memory. It's the memory that you have in your computer. (Comes in units of 8Mb, 16Mb, 64Mb, 256Mb, etc.)
- RAID - Redundant Array of Inexpensive Disks.

An array of physical disks, usually treated by the operating system as one single disk, and often forced to appear that way by the hardware. The reason for using RAID is often simply to achieve a high data transfer rate, but it may also be to get adequate disk capacity or high reliability. Redundancy means that the system is capable of continued operation even if a disk fails. There are various types of RAID array and several different approaches to implementing them. Some systems provide protection against failure of more than one drive and some ('hot-swappable') systems allow a drive to be replaced without even stopping the OS. See for example the Linux 'HOWTO' documents Disk-HOWTO, Module-HOWTO and Parallel-Processing-HOWTO.

### ***6.3.1 Machine Strength Demands According to Expected Site Traffic***

If you are building a fan site and you want to amaze your friends with a mod\_perl guest book, any old 486 machine could do it. If you are in a serious business, it is very important to build a scalable server. If your service is successful and becomes popular, the traffic could double every few days, and you should be ready to add more resources to keep up with the demand. While we can define the webserver scalability more precisely, the important thing is to make sure that you can add more power to your webserver(s) without investing much additional money in software development (you will need a little software effort to connect your servers, if you add more of them). This means that you should choose hardware and OSs that can talk to other machines and become a part of a cluster.

On the other hand if you prepare for a lot of traffic and buy a monster to do the work for you, what happens if your service doesn't prove to be as successful as you thought it would be? Then you've spent too much money, and meanwhile faster processors and other hardware components have been released, so you lose.

Wisdom and prophecy, that's all it takes :)

#### **6.3.1.1 Single Strong Machine vs Many Weaker Machines**

Let's start with a claim that a four years old processor is still very powerful and can be put to a good use. Now let's say that for a given amount of money you can probably buy either one new very strong machine or about ten older but very cheap machines. I claim that with ten old machines connected into a cluster and by deploying load balancing you will be able to serve about five times more requests than with one single new machine.

Why is that? Because generally the performance improvement on a new machine is marginal while the price is much higher. Ten machines will do faster disk I/O than one single machine, even if the new disk is quite a bit faster. Yes, you have more administration overhead, but there is a chance you will have it anyway, for in a short time the new machine you have just bought might not stand the load. Then you will have to purchase more equipment and think about how to implement load balancing and web server file system distribution anyway.

Why I'm so convinced? Look at the busiest services on the Internet: search engines, web-email servers and the like -- most of them use a clustering approach. You may not always notice it, because they hide the real implementation behind proxy servers.

### ***6.3.2 Internet Connection***

You have the best hardware you can get, but the service is still crawling. Make sure you have a fast Internet connection. Not as fast as your ISP claims it to be, but fast as it should be. The ISP might have a very good connection to the Internet, but put many clients on the same line. If these are heavy clients, your traffic will have to share the same line and your throughput will suffer. Think about a dedicated connection and make sure it is truly dedicated. Don't trust the ISP, check it!



The idea of having a connection to **The Internet** is a little misleading. Many Web hosting and co-location companies have large amounts of bandwidth, but still have poor connectivity. The public exchanges, such as MAE-East and MAE-West, frequently become overloaded, yet many ISPs depend on these exchanges.

Private peering means that providers can exchange traffic much quicker.

Also, if your Web site is of global interest, check that the ISP has good global connectivity. If the Web site is going to be visited mostly by people in a certain country or region, your server should probably be located there.

Bad connectivity can directly influence your machine's performance. Here is a story one of the developers told on the `mod_perl` mailing list:

```
What relationship has 10% packet loss on one upstream provider got
to do with machine memory ?
```

```
Yes.. a lot. For a nightmare week, the box was located downstream of
a provider who was struggling with some serious bandwidth problems
of his own... people were connecting to the site via this link, and
packet loss was such that retransmits and tcp stalls were keeping
httpd heavies around for much longer than normal.. instead of
blasting out the data at high or even modem speeds, they would be
stuck at 1k/sec or stalled out... people would press stop and
refresh, httpds would take 300 seconds to timeout on writes to
no-one.. it was a nightmare. Those problems didn't go away till I
moved the box to a place closer to some decent backbones.
```

```
Note that with a proxy, this only keeps a lightweight httpd tied up,
assuming the page is small enough to fit in the buffers. If you are
a busy internet site you always have some slow clients. This is a
difficult thing to simulate in benchmark testing, though.
```

### 6.3.3 I/O Performance

If your service is I/O bound (does a lot of read/write operations to disk) you need a very fast disk, especially if the you need a relational database, which are the main I/O stream creators. So you should not spend the money on Video card and monitor! A cheap card and a 14" monochrome monitor are perfectly adequate for a Web server, you will probably access it by `telnet` or `ssh` most of the time. Look for disks with the best price/performance ratio. Of course, ask around and avoid disks that have a reputation for headcrashes and other disasters.

You must think about RAID or similar systems if you have an enormous data set to serve (what is an enormous data set nowadays? Gigabytes, Terabytes?) or you expect a really big web traffic.

Ok, you have a fast disk, what's next? You need a fast disk controller. There may be one embedded on your computer's motherboard. If the controller is not fast enough you should buy a faster one. Don't forget that it may be necessary to disable the original controller.

### 6.3.4 Memory

Memory should be well tested. Many memory test programs are practically useless. Running a busy system for a few weeks without ever shutting it down is a pretty good memory test. If you increase the amount of RAM on a well-tested box, use well-tested RAM.

How much RAM do you need? Nowadays, the chances are that you will hear: "Memory is cheap, the more you buy the better". But how much is enough? The answer is pretty straightforward: *you do not want your machine to swap*. When the CPU needs to write something into memory, but memory is already full, it takes the least frequently used memory pages and swaps them out to disk. This means you have to bear the time penalty of writing the data to disk. If another process then references some of the data which happens to be on one of the pages that has just been swapped out, the CPU swaps it back in again, probably swapping out some other data that will be needed very shortly by some other process. Carried to the extreme, the CPU and disk start to *thrash* hopelessly in circles, without getting any real work done. The less RAM there is, the more often this scenario arises. Worse, you can exhaust swap space as well, and then your troubles really start...

How do you make a decision? You know the highest rate at which your server expects to serve pages and how long it takes on average to serve one. Now you can calculate how many server processes you need. If you know the maximum size your servers can grow to, you know how much memory you need. If your OS supports memory sharing, you can make best use of this feature by preloading the modules and scripts at server startup, and so you will need less memory than you have calculated.

Do not forget that other essential system processes need memory as well, so you should plan not only for the Web server, but also take into account the other players. Remember that requests can be queued, so you can afford to let your client wait for a few moments until a server is available to serve it. Most of the time your server will not have the maximum load, but you should be ready to bear the peaks. You need to reserve at least 20% of free memory for peak situations. Many sites have crashed a few moments after a big scoop about them was posted and an unexpected number of requests suddenly came in. (This is called the Slashdot effect, which was born at <http://slashdot.org> ). If you are about to announce something cool, be aware of the possible consequences.

### 6.3.5 CPU

Make sure that the CPU is operating within its specifications. Many boxes are shipped with incorrect settings for CPU clock speed, power supply voltage etc. Sometimes a cooling fan is not fitted. It may be ineffective because a cable assembly fouls the fan blades. Like faulty RAM, an overheating processor can cause all kinds of strange and unpredictable things to happen. Some CPUs are known to have bugs which can be serious in certain circumstances. Try not to get one of them.

### 6.3.6 Bottlenecks

You might use the most expensive components, but still get bad performance. Why? Let me introduce an annoying word: bottleneck.

A machine is an aggregate of many components. Almost any one of them may become a bottleneck.

If you have a fast processor but a small amount of RAM, the RAM will probably be the bottleneck. The processor will be under-utilized, usually it will be waiting for the kernel to swap the memory pages in and out, because memory is too small to hold the busiest pages.

If you have a lot of memory, a fast processor, a fast disk, but a slow disk controller, the disk controller will be the bottleneck. The performance will still be bad, and you will have wasted money.

Use a fast NIC that does not create a bottleneck. They are cheap. If the NIC is slow, the whole service is slow. This is a most important component, since web servers are much more often network-bound than they are disk-bound!

### 6.3.6.1 Solving Hardware Requirement Conflicts

It may happen that the combination of software components which you find yourself using gives rise to conflicting requirements for the optimization of tuning parameters. If you can separate the components onto different machines you may find that this approach (a kind of clustering) solves the problem, at much less cost than buying faster hardware, because you can tune the machines individually to suit the tasks they should perform.

For example if you need to run a relational database engine and mod\_perl server, it can be wise to put the two on different machines, since while RDBMS need a very fast disk, mod\_perl processes need lots of memory. So by placing the two on different machines it's easy to optimize each machine at separate and satisfy the each software components requirements in the best way.

### 6.3.7 Conclusion

To use your money optimally you have to understand the hardware very well, so you will know what to pick. Otherwise, you should hire a knowledgeable hardware consultant and employ them on a regular basis, since your needs will probably change as time goes by and your hardware will likewise be forced to adapt as well.

## 6.4 Maintainers

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

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

## 6.5 Authors

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

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

## **7 Controlling and Monitoring the Server**

## 7.1 Description

Covers techniques to restart mod\_perl enabled Apache, SUID scripts, monitoring, and other maintenance chores, as well as some specific setups.

## 7.2 Restarting Techniques

All of these techniques require that you know the server process id (PID). The easiest way to find the PID is to look it up in the *httpd.pid* file. It's easy to discover where to look, by looking in the *httpd.conf* file. Open the file and locate the entry *PidFile*. Here is the line from one of my own *httpd.conf* files:

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

As you see, with my configuration the file is */usr/local/var/httpd\_perl/run/httpd.pid*.

Another way is to use the *ps* and *grep* utilities. Assuming that the binary is called *httpd\_perl*, we would do:

```
% ps auxc | grep httpd_perl
```

or maybe:

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

This will produce a list of all the *httpd\_perl* (parent and children) processes. You are looking for the parent process. If you run your server as root, you will easily locate it since it belongs to root. If you run the server as some other user (when you don't have root access, the processes will belong to that user unless defined differently in *httpd.conf*). It's still easy to find which is the parent--usually it's the process with the smallest PID.

You will see several *httpd* processes running on your system, but you should never need to send signals to any of them except the parent, whose pid is in the *PidFile*. There are three signals that you can send to the parent: *SIGTERM*, *SIGHUP*, and *SIGUSR1*.

Some folks prefer to specify signals using numerical values, rather than using symbols. If you are looking for these, check out your *kill(1)* man page. My page points to */usr/include/linux/signal.h*, the relevant entries are:

```
#define SIGHUP      1      /* hangup, generated when terminal disconnects */
#define SIGKILL     9      /* last resort */
#define SIGTERM     15     /* software termination signal */
#define SIGUSR1     30     /* user defined signal 1 */
```

Note that to send these signals from the command line the *SIG* prefix must be omitted and under some operating systems they will need to be preceded by a minus sign, e.g. *kill -15* or *kill -TERM* followed by the PID.

## 7.3 Server Stopping and Restarting

We will concentrate here on the implications of sending TERM, HUP, and USR1 signals (as arguments to kill(1)) to a mod\_perl enabled server. See <http://www.apache.org/docs/stopping.html> for documentation on the implications of sending these signals to a plain Apache server.

- **TERM Signal: Stop Now**

Sending the TERM signal to the parent causes it to immediately attempt to kill off all its children. Any requests in progress are terminated, and no further requests are served. This process may take quite a few seconds to complete. To stop a child, the parent sends it a SIGHUP signal. If that fails it sends another. If that fails it sends the SIGTERM signal, and as a last resort it sends the SIGKILL signal. For each failed attempt to kill a child it makes an entry in the *error\_log*.

When all the child processes were terminated, the parent itself exits and any open log files are closed. This is when all the accumulated END blocks, apart from the ones located in scripts running under Apache::Registry or Apache::PerlRun handlers. In the latter case, END blocks are executed after each request is served.

- **HUP Signal: Restart Now**

Sending the HUP signal to the parent causes it to kill off its children as if the TERM signal had been sent, i.e. any requests in progress are terminated; but the parent does not exit. Instead, the parent re-reads its configuration files, spawns a new set of child processes and continues to serve requests. It is almost equivalent to stopping and then restarting the server.

If the configuration files contain errors when restart is signaled, the parent will exit, so it is important to check the configuration files for errors before issuing a restart. How to perform the check will be covered shortly;

Sometimes using this approach to restart mod\_perl enabled Apache may cause the processes memory incremental growth after each restart. This happens when Perl code loaded in memory is not completely torn down, leading to a memory leak.

- **USR1 Signal: Gracefully Restart Now**

The USR1 signal causes the parent process to advise the children to exit after serving their current requests, or to exit immediately if they're not serving a request. The parent re-reads its configuration files and re-opens its log files. As each child dies off the parent replaces it with a child from the new generation (the new children use the new configuration) and it begins serving new requests immediately.

The only difference between USR1 and HUP is that USR1 allows the children to complete any current requests prior to killing them off and there is no interruption in the services compared to the killing with HUP signal, where it might take a few seconds for a restart to get completed and there is no real service at this time.

By default, if a server is restarted (using `kill -USR1 `cat logs/httpd.pid`` or with the HUP signal), Perl scripts and modules are not reloaded. To reload `PerlRequires`, `PerlModules`, other `use()`'d modules and flush the `Apache::Registry` cache, use this directive in `httpd.conf`:

```
PerlFreshRestart On
```

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

## 7.4 Speeding up the Apache Termination and Restart

We've already mentioned that restart or termination can sometimes take quite a long time, (e.g. tens of seconds), for a `mod_perl` server. The reason for that is a call to the `perl_destruct()` Perl API function during the child exit phase. This will cause proper execution of `END` blocks found during server startup and will invoke the `DESTROY` method on global objects which are still alive.

It is also possible that this operation may take a long time to finish, causing a long delay during a restart. Sometimes this will be followed by a series of messages appearing in the server *error\_log* file, warning that certain child processes did not exit as expected. This happens when after a few attempts advising the child process to quit, the child is still in the middle of `perl_destruct()`, and a lethal `KILL` signal is sent, aborting any operation the child has happened to execute and *brutally* killing it.

If your code does not contain any `END` blocks or `DESTROY` methods which need to be run during child server shutdown, or may have these, but it's insignificant to execute them, this destruction can be avoided by setting the `PERL_DESTRUCT_LEVEL` environment variable to `-1`. For example add this setting to the `httpd.conf` file:

```
PerlSetEnv PERL_DESTRUCT_LEVEL -1
```

What constitutes a significant cleanup? Any change of state outside of the current process that would not be handled by the operating system itself. So committing database transactions and removing the lock on some resource are significant operations, but closing an ordinary file isn't.

## 7.5 Using `apachectl` to Control the Server

The Apache distribution comes with a script to control the server. It's called `apachectl` and it is installed into the same location as the `httpd` executable. We will assume for the sake of our examples that it's in `/usr/local/sbin/httpd_perl/apachectl`:

To start `httpd_perl`:

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

To stop `httpd_perl`:

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

To restart `httpd_perl` (if it is running, send `SIGHUP`; if it is not already running just start it):

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

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

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

To do a configuration test:

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

Replace `httpd_perl` with `httpd_docs` in the above calls to control the `httpd_docs` server.

There are other options for `apachectl`, use the `help` option to see them all.

It's important to remember that `apachectl` uses the PID file, which is specified by the `PIDFILE` directive in `httpd.conf`. If you delete the PID file by hand while the server is running, `apachectl` will be unable to stop or restart the server.

## 7.6 Safe Code Updates on a Live Production Server

You have prepared a new version of code, uploaded it into a production server, restarted it and it doesn't work. What could be worse than that? You also cannot go back, because you have overwritten the good working code.

It's quite easy to prevent it, just don't overwrite the previous working files!

Personally I do all updates on the live server with the following sequence. Assume that the server root directory is `/home/httpd/perl/rel`. When I'm about to update the files I create a new directory `/home/httpd/perl/beta`, copy the old files from `/home/httpd/perl/rel` and update it with the new files. Then I do some last sanity checks (check file permissions are `[read+executable]`, and run `perl -c` on the new modules to make sure there no errors in them). When I think I'm ready I do:

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

Let me explain what this does.

Firstly, note that I put all the commands on one line, separated by `&&`, and only then press the Enter key. As I am working remotely, this ensures that if I suddenly lose my connection (sadly this happens sometimes) I won't leave the server down if only the `stop` command squeezed in. `&&` also ensures that if any command fails, the rest won't be executed. I am using aliases (which I have already defined) to make the typing easier:



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

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

Taking the line apart piece by piece:

```
mv rel old &&
```

back up the working directory to *old*

```
mv beta rel &&
```

put the new one in its place

```
stop &&
```

stop the server

```
sleep 3 &&
```

give it a few seconds to shut down (it might take even longer)

```
restart &&
```

restart the server

```
err
```

view of the tail of the *error\_log* file in order to see that everything is OK

`apachectl` generates the status messages a little too early (e.g. when you issue `apachectl stop` it says the server has been stopped, while in fact it's still running) so don't rely on it, rely on the *error\_log* file instead.

Also notice that I use `restart` and not just `start`. I do this because of Apache's potentially long stopping times (it depends on what you do with it of course!). If you use `start` and Apache hasn't yet released the port it's listening to, the start would fail and *error\_log* would tell you that the port is in use, e.g.:

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

But if you use `restart`, it will wait for the server to quit and then will cleanly restart it.

Now what happens if the new modules are broken? First of all, I see immediately an indication of the problems reported in the *error\_log* file, which I `tail -f` immediately after a restart command. If there's a problem, I just put everything back as it was before:

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

Usually everything will be fine, and I have had only about 10 seconds of downtime, which is pretty good!

## 7.7 An Intentional Disabling of Live Scripts

What happens if you really must take down the server or disable the scripts? This situation might happen when you need to do some maintenance work on your database server. If you have to take your database down then any scripts that use it will fail.

If you do nothing, the user will see either the grey `An Error has happened` message or perhaps a customized error message if you have added code to trap and customize the errors. See [Redirecting Errors to the Client](#) instead of to the `error_log` for the latter case.

A much friendlier approach is to confess to your users that you are doing some maintenance work and plead for patience, promising (keep the promise!) that the service will become fully functional in X minutes. There are a few ways to do this:

The first doesn't require messing with the server. It works when you have to disable a script running under `Apache::Registry` and relies on the fact that it checks whether the file was modified before using the cached version. Obviously it won't work under other handlers because these serve the compiled version of the code and don't check to see if there was a change in the code on the disk.

So if you want to disable an `Apache::Registry` script, prepare a little script like this:

```
/home/http/perl/maintenance.pl
-----
#!/usr/bin/perl -Tw

use strict;
use CGI;
my $q = new CGI;
print $q->header, $q->p(
  "Sorry, the service is temporarily down for maintenance.
  It will be back in ten to fifteen minutes.
  Please, bear with us.
  Thank you!");
```

So if you now have to disable a script for example `/home/http/perl/chat.pl`, just do this:

```
% mv /home/http/perl/chat.pl /home/http/perl/chat.pl.orig
% ln -s /home/http/perl/maintenance.pl /home/http/perl/chat.pl
```

Of course your server configuration should allow symbolic links for this trick to work. Make sure you have the directive

```
Options FollowSymLinks
```

in the <Location> or <Directory> section of your *httpd.conf*.

When you're done, it's easy to restore the previous setup. Just do this:

```
% mv /home/http/perl/chat.pl.orig /home/http/perl/chat.pl
```

which overwrites the symbolic link.

Now make sure that the script will have the current timestamp:

```
% touch /home/http/perl/chat.pl
```

Apache will automatically detect the change and will use the moved script instead.

The second approach is to change the server configuration and configure a whole directory to be handled by a `My::Maintenance` handler (which you must write). For example if you write something like this:

```
My/Maintenance.pm
-----
package My::Maintenance;
use strict;
use Apache::Constants qw(:common);
sub handler {
    my $r = shift;
    print $r->send_http_header("text/plain");
    print qq{
        We apologize, but this service is temporarily stopped for
        maintenance. It will be back in ten to fifteen minutes.
        Please, bear with us. Thank you!
    };
    return OK;
}
1;
```

and put it in a directory that is in the server's @INC, to disable all the scripts in Location `/perl` you would replace:

```
<Location /perl>
    SetHandler perl-script
    PerlHandler My::Handler
    [snip]
</Location>
```

with

```
<Location /perl>
    SetHandler perl-script
    PerlHandler My::Maintenance
    [snip]
</Location>
```

Now restart the server. Your users will be happy to go and read <http://slashdot.org> for ten minutes, knowing that you are working on a much better version of the service.

If you need to disable a location handled by some module, the second approach would work just as well.

## 7.8 SUID Start-up Scripts

If you want to allow a few people in your team to start and stop the server you will have to give them the root password, which is not a good thing to do. The less people know the password, the less problems are likely to be encountered. But there is an easy solution for this problem available on UNIX platforms. It's called a *setuid* executable.

### 7.8.1 Introduction to SUID Executables

The *setuid* executable has a *setuid* permissions bit set. This sets the process's effective user ID to that of the file upon execution. You perform this setting with the following command:

```
% chmod u+s filename
```

You probably have used *setuid* executables before without even knowing about it. For example when you change your password you execute the *passwd* utility, which among other things modifies the */etc/passwd* file. In order to change this file you need root permissions, the *passwd* utility has the *setuid* bit set, therefore when you execute this utility, its effective ID is the same of the root user ID.

You should avoid using *setuid* executables as a general practice. The less *setuid* executables you have the less likely that someone will find a way to break into your system, by exploiting some bug you didn't know about.

When the executable is *setuid* to root, you have to make sure that it doesn't have the group and world read and write permissions. If we take a look at the *passwd* utility we will see:

```
% ls -l /usr/bin/passwd
-r-s--x--x 1 root root 12244 Feb 8 00:20 /usr/bin/passwd
```

You achieve this with the following command:

```
% chmod 4511 filename
```

The first digit (4) stands for *setuid* bit, the second digit (5) is a compound of read (4) and executable (1) permissions for the user, and the third and the fourth digits are setting the executable permissions for the group and the world.

### 7.8.2 Apache Startup SUID Script's Security

In our case, we want to allow *setuid* access only to a specific group of users, who all belong to the same group. For the sake of our example we will use the group named *apache*. It's important that users who aren't root or who don't belong to the *apache* group will not be able to execute this script. Therefore we perform the following commands:

```
% chgrp apache apachectl
% chmod 4510 apachectl
```

The execution order is important. If you swap the command execution order you will lose the setuid bit.

Now if we look at the file we see:

```
% ls -l apachectl
-r-s--x--- 1 root apache 32 May 13 21:52 apachectl
```

Now we are all set... Almost...

When you start Apache, Apache and Perl modules are being loaded, code can be executed. Since all this happens with root effective ID, any code executed as if the root user was doing that. You should be very careful because while you didn't give anyone the root password, all the users in the *apache* group have an indirect root access. Which means that if Apache loads some module or executes some code that is writable by some of these users, users can plant code that will allow them to gain a shell access to root account and become a real root.

Of course if you don't trust your team you shouldn't use this solution in first place. You can try to check that all the files Apache loads aren't writable by anyone but root, but there are too many of them, especially in the *mod\_perl* case, where many Perl modules are loaded at the server startup.

By the way, don't let all this setuid stuff to confuse you -- when the parent process is loaded, the children processes are spawned as non-root processes. This section has presented a way to allow non-root users to start the server as root user, the rest is exactly the same as if you were executing the script as root in first place.

### 7.8.3 Sample Apache Startup SUID Script

Now if you are still with us, here is an example of the setuid Apache startup script.

Note the line marked *WORKAROUND*, which fixes an obscure error when starting *mod\_perl* enabled Apache by setting the real UID to the effective UID. Without this workaround, a mismatch between the real and the effective UID causes Perl to croak on the *-e* switch.

Note that you must be using a version of Perl that recognizes and emulates the suid bits in order for this to work. This script will do different things depending on whether it is named *start\_httpd*, *stop\_httpd* or *restart\_httpd*. You can use symbolic links for this purpose.

```
suid_apache_ctl
-----
#!/usr/bin/perl -T

# These constants will need to be adjusted.
$PID_FILE = '/home/www/logs/httpd.pid';
$HTTPD = '/home/www/httpd -d /home/www';

# These prevent taint warnings while running suid
$ENV{PATH}='/bin:/usr/bin';
$ENV{IFS}=' ';
```

```

# This sets the real to the effective ID, and prevents
# an obscure error when starting apache/mod_perl
$< = $>; # WORKAROUND
$( = $) = 0; # set the group to root too

# Do different things depending on our name
($name) = $0 =~ m|([^\s]+)|;

if ($name eq 'start_httpd') {
    system $HTTPD and die "Unable to start HTTP";
    print "HTTP started.\n";
    exit 0;
}

# extract the process id and confirm that it is numeric
$pid = `cat $PID_FILE`;
$pid =~ /(\d+)/ or die "PID $pid not numeric";
$pid = $1;

if ($name eq 'stop_httpd') {
    kill 'TERM', $pid or die "Unable to signal HTTP";
    print "HTTP stopped.\n";
    exit 0;
}

if ($name eq 'restart_httpd') {
    kill 'HUP', $pid or die "Unable to signal HTTP";
    print "HTTP restarted.\n";

    exit 0;
}

die "Script must be named start_httpd, stop_httpd, or restart_httpd.\n";

```

## 7.9 Preparing for Machine Reboot

When you run your own development box, it's okay to start the webserver by hand when you need to. On a production system it is possible that the machine the server is running on will have to be rebooted. When the reboot is completed, who is going to remember to start the server? It's easy to forget this task, and what happens if you aren't around when the machine is rebooted?

After the server installation is complete, it's important not to forget that you need to put a script to perform the server startup and shutdown into the standard system location, for example */etc/rc.d* under RedHat Linux, or */etc/init.d/apache* under Debian Slink Linux.

This is the directory which contains scripts to start and stop all the other daemons. The directory and file names vary from one Operating System (OS) to another, and even between different distributions of the same OS.

Generally the simplest solution is to copy the `apachectl` script to your startup directory or create a symbolic link from the startup directory to the `apachectl` script. You will find `apachectl` in the same directory as the `httpd` executable after Apache installation. If you have more than one Apache server you will need a separate script for each one, and of course you will have to rename them so that they can co-exist in the same directories.

For example on a RedHat Linux machine with two servers, I have the following setup:

```
/etc/rc.d/init.d/httpd_docs
/etc/rc.d/init.d/httpd_perl
/etc/rc.d/rc3.d/S91httpd_docs -> ../init.d/httpd_docs
/etc/rc.d/rc3.d/S91httpd_perl -> ../init.d/httpd_perl
/etc/rc.d/rc6.d/K16httpd_docs -> ../init.d/httpd_docs
/etc/rc.d/rc6.d/K16httpd_perl -> ../init.d/httpd_perl
```

The scripts themselves reside in the `/etc/rc.d/init.d` directory. There are symbolic links to these scripts in other directories. The names are the same as the script names but they have numerical prefixes, which are used for executing the scripts in a particular order: the lower numbers are executed earlier.

When the system starts (level 3) we want the Apache to be started when almost all of the services are running already, therefore I've used `S91`. For example if the `mod_perl` enabled Apache issues a `connect_on_init()` the SQL server should be started before Apache.

When the system shuts down (level 6), Apache should be stopped as one of the first processes, therefore I've used `K16`. Again if the server does some cleanup processing during the shutdown event and requires third party services to be running (e.g. SQL server) it should be stopped before these services.

Notice that it's normal for more than one symbolic link to have the same sequence number.

Under RedHat Linux and similar systems, when a machine is booted and its runlevel set to 3 (multiuser + network), Linux goes into `/etc/rc.d/rc3.d/` and executes the scripts the symbolic links point to with the `start` argument. When it sees `S91httpd_perl`, it executes:

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

When the machine is shut down, the scripts are executed through links from the `/etc/rc.d/rc6.d/` directory. This time the scripts are called with the `stop` argument, like this:

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

Most systems have GUI utilities to automate the creation of symbolic links. For example RedHat Linux includes the `control-panel` utility, which amongst other things includes the `RunLevel Manager`. (which can be invoked directly as either `ntsysv(8)` or `tksysv(8)`). This will help you to create the proper symbolic links. Of course before you use it, you should put `apachectl` or similar scripts into the `init.d` or equivalent directory. Or you can have a symbolic link to some other location instead.

The simplest approach is to use the `chkconfig(8)` utility which adds and removes the services for you. The following example shows how to add an `httpd_perl` startup script to the system.

First move or copy the file into the directory */etc/rc.d/init.d*:

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

Now open the script in your favorite editor and add the following lines after the main header of the script:

```
# Comments to support chkconfig on RedHat Linux
# chkconfig: 2345 91 16
# description: mod_perl enabled Apache Server
```

So now the beginning of the script looks like:

```
#!/bin/sh
#
# Apache control script designed to allow an easy command line
# interface to controlling Apache.  Written by Marc Slemko,
# 1997/08/23

# Comments to support chkconfig on RedHat Linux
# chkconfig: 2345 91 16
# description: mod_perl enabled Apache Server

#
# The exit codes returned are:
# ...
```

Adjust the line:

```
# chkconfig: 2345 91 16
```

to your needs. The above setting says to says that the script should be started in levels 2, 3, 4, and 5, that its start priority should be 91, and that its stop priority should be 16.

Now all you have to do is to ask `chkconfig` to configure the startup scripts. Before we do that let's look at what we have:

```
% find /etc/rc.d | grep httpd_perl

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

Which means that we only have the startup script itself. Now we execute:

```
% chkconfig --add httpd_perl
```

and see what has changed:



```
% find /etc/rc.d | grep httpd_perl

/etc/rc.d/init.d/httpd_perl
/etc/rc.d/rc0.d/K16httpd_perl
/etc/rc.d/rc1.d/K16httpd_perl
/etc/rc.d/rc2.d/S91httpd_perl
/etc/rc.d/rc3.d/S91httpd_perl
/etc/rc.d/rc4.d/S91httpd_perl
/etc/rc.d/rc5.d/S91httpd_perl
/etc/rc.d/rc6.d/K16httpd_perl
```

As you can see `chkconfig` created all the symbolic links for us, using the startup and shutdown priorities as specified in the line:

```
# chkconfig: 2345 91 16
```

If for some reason you want to remove the service from the startup scripts, all you have to do is to tell `chkconfig` to remove the links:

```
% chkconfig --del httpd_perl
```

Now if we look at the files under the directory `/etc/rc.d/` we see again only the script itself.

```
% find /etc/rc.d | grep httpd_perl

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

Of course you may keep the startup script in any other directory as long as you can link to it. For example if you want to keep this file with all the Apache binaries in `/usr/local/apache/bin`, all you have to do is to provide a symbolic link to this file:

```
% ln -s /usr/local/apache/bin/apachectl /etc/rc.d/init.d/httpd_perl
```

and then:

```
% chkconfig --add httpd_perl
```

Note that in case of using symlinks the link name in `/etc/rc.d/init.d` is what matters and not the name of the script the link points to.

## 7.10 Monitoring the Server. A watchdog.

With `mod_perl` many things can happen to your server. It is possible that the server might die when you are not around. As with any other critical service you need to run some kind of watchdog.

One simple solution is to use a slightly modified `apachectl` script, which I've named *apache.watchdog*. Call it from the crontab every 30 minutes -- or even every minute -- to make sure the server is up all the time.

The crontab entry for 30 minutes intervals:

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

The script:

```
#!/bin/sh

# this script is a watchdog to see whether the server is online
# It tries to restart the server, and if it's
# down it sends an email alert to admin

# admin's email
EMAIL=webmaster@example.com

# the path to your PID file
PIDFILE=/usr/local/var/httpd_perl/run/httpd.pid

# the path to your httpd binary, including options if necessary
HTTPD=/usr/local/sbin/httpd_perl/httpd_perl

# check for pidfile
if [ -f $PIDFILE ] ; then
    PID=`cat $PIDFILE`

    if kill -0 $PID; then
        STATUS="httpd (pid $PID) running"
        RUNNING=1
    else
        STATUS="httpd (pid $PID?) not running"
        RUNNING=0
    fi
else
    STATUS="httpd (no pid file) not running"
    RUNNING=0
fi

if [ $RUNNING -eq 0 ]; then
    echo "$0 $ARG: httpd not running, trying to start"
    if $HTTPD ; then
        echo "$0 $ARG: httpd started"
        mail $EMAIL -s "$0 $ARG: httpd started" > /dev/null 2>&1
    else
        echo "$0 $ARG: httpd could not be started"
        mail $EMAIL -s \
            "$0 $ARG: httpd could not be started" > /dev/null 2>&1
    fi
fi
```

Another approach, probably even more practical, is to use the cool LWP Perl package to test the server by trying to fetch some document (script) served by the server. Why is it more practical? Because while the server can be up as a process, it can be stuck and not working. Failing to get the document will trigger restart, and "probably" the problem will go away.

Like before we set a cronjob to call this script every few minutes to fetch some very light script. The best thing of course is to call it every minute. Why so often? If your server starts to spin and trash your disk space with multiple error messages filling the *error\_log*, in five minutes you might run out of free disk space which might bring your system to its knees. Chances are that no other child will be able to serve requests, since the system will be too busy writing to the *error\_log* file. Think big--if you are running a heavy service (which is very fast since you are running under mod\_perl) adding one more request every minute will not be felt by the server at all.

So we end up with a crontab entry like this:

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

And the watchdog itself:

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

# untaint
$ENV{'PATH'} = '/bin:/usr/bin';
delete @ENV{'IFS', 'CDPATH', 'ENV', 'BASH_ENV'};

use strict;
use diagnostics;
use URI::URL;
use LWP::MediaTypes qw(media_suffix);

my $VERSION = '0.01';
use vars qw($ua $proxy);
$proxy = '';

require LWP::UserAgent;
use HTTP::Status;

##### Config #####
my $test_script_url = 'http://www.example.com:81/perl/test.pl';
my $monitor_email = 'root@localhost';
my $restart_command = '/usr/local/sbin/httpd_perl/apachectl restart';
my $mail_program = '/usr/lib/sendmail -t -n';
#####

$ua = new LWP::UserAgent;
$ua->agent("$0/watchdog " . $ua->agent);
# Uncomment the proxy if you access a machine from behind a firewall
# $proxy = "http://www-proxy.com";
$ua->proxy('http', $proxy) if $proxy;

# If it returns '1' it means we are alive
exit 1 if checkurl($test_script_url);

# Houston, we have a problem.
# The server seems to be down, try to restart it.
my $status = system $restart_command;

my $message = ($status == 0)
    ? "Server was down and successfully restarted!"
```

```

        : "Server is down. Can't restart.";

my $subject = ($status == 0)
    ? "Attention! Webserver restarted"
    : "Attention! Webserver is down. can't restart";

# email the monitoring person
my $to = $monitor_email;
my $from = $monitor_email;
send_mail($from,$to,$subject,$message);

# input:  URL to check
# output: 1 for success, 0 for failure
#####
sub checkurl{
    my ($url) = @_;

    # Fetch document
    my $res = $ua->request(HTTP::Request->new(GET => $url));

    # Check the result status
    return 1 if is_success($res->code);

    # failed
    return 0;
} # end of sub checkurl

# send email about the problem
#####
sub send_mail{
    my($from,$to,$subject,$messagebody) = @_;

    open MAIL, "|$mail_program"
        or die "Can't open a pipe to a $mail_program :$!\n";

    print MAIL <<__END_OF_MAIL__;
To: $to
From: $from
Subject: $subject

$messagebody

__END_OF_MAIL__

    close MAIL;
}

```

## 7.11 Running a Server in Single Process Mode

Often while developing new code, you will want to run the server in single process mode. Sometimes it works Sometimes it does Not and Names collisions with Modules and libs. Running in single process mode inhibits the server from "daemonizing", and this allows you to run it under the control of a debugger more easily.

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

When you use the `-X` switch the server will run in the foreground of the shell, so you can kill it with *Ctrl-C*.

Note that in `-X` (single-process) mode the server will run very slowly when fetching images.

Note for Netscape users:

If you use Netscape while your server is running in single-process mode, HTTP's `KeepAlive` feature gets in the way. Netscape tries to open multiple connections and keep them open. Because there is only one server process listening, each connection has to time out before the next succeeds. Turn off `KeepAlive` in *httpd.conf* to avoid this effect while developing. If you use the image size parameters, Netscape will be able to render the page without the images so you can press the browser's *STOP* button after a few seconds.

In addition you should know that when running with `-X` you will not see the control messages that the parent server normally writes to the *error\_log* ("*server started*", "*server stopped*" etc). Since `httpd -X` causes the server to handle all requests itself, without forking any children, there is no controlling parent to write the status messages.

## 7.12 Starting a Personal Server for Each Developer

If you are the only developer working on the specific `server:port` you have no problems, since you have complete control over the server. However, often you will have a group of developers who need to develop `mod_perl` scripts and modules concurrently. This means that each developer will want to have control over the server - to kill it, to run it in single server mode, to restart it, etc., as well as having control over the location of the log files, configuration settings like `MaxClients`, and so on.

You *can* work around this problem by preparing a few *httpd.conf* files and forcing each developer to use

```
httpd_perl -f /path/to/httpd.conf
```

but I approach it in a different way. I use the `-Dparameter` startup option of the server. I call my version of the server

```
% http_perl -Dstas
```

In *httpd.conf* I write:

```
# Personal development Server for stas
# stas uses the server running on port 8000
<IfDefine stas>
Port 8000
PidFile /usr/local/var/httpd_perl/run/httpd.pid.stas
ErrorLog /usr/local/var/httpd_perl/logs/error_log.stas
Timeout 300
KeepAlive On
MinSpareServers 2
MaxSpareServers 2
```

```

StartServers 1
MaxClients 3
MaxRequestsPerChild 15
</IfDefine>

# Personal development Server for userfoo
# userfoo uses the server running on port 8001
<IfDefine userfoo>
Port 8001
PidFile /usr/local/var/httpd_perl/run/httpd.pid.userfoo
ErrorLog /usr/local/var/httpd_perl/logs/error_log.userfoo
Timeout 300
KeepAlive Off
MinSpareServers 1
MaxSpareServers 2
StartServers 1
MaxClients 5
MaxRequestsPerChild 0
</IfDefine>

```

With this technique we have achieved full control over start/stop, number of children, a separate error log file, and port selection for each server. This saves Stas from getting called every few minutes by Eric: "Stas, I'm going to restart the server".

In the above technique, you need to discover the PID of your parent `httpd_perl` process, which is written in `/usr/local/var/httpd_perl/run/httpd.pid.stas` (and the same for the user *eric*). To make things even easier we change the *apachectl* script to do the work for us. We make a copy for each developer called **apachectl.username** and we change two lines in each script:

```

PIDFILE=/usr/local/var/httpd_perl/run/httpd.pid.username
HTTPD='/usr/local/sbin/httpd_perl/httpd_perl -Dusername'

```

So for the user *stas* we prepare a startup script called *apachectl.stas* and we change these two lines in the standard *apachectl* script as it comes unmodified from Apache distribution.

```

PIDFILE=/usr/local/var/httpd_perl/run/httpd.pid.stas
HTTPD='/usr/local/sbin/httpd_perl/httpd_perl -Dstas'

```

So now when user *stas* wants to stop the server he will execute:

```
apachectl.stas stop
```

And to start:

```
apachectl.stas start
```

Certainly the rest of the *apachectl* arguments apply as before.

You might think about having only one *apachectl* and know who is calling by checking the UID, but since you have to be root to start the server it is not possible, unless you make the `setuid` bit on this script, as we've explained in the beginning of this chapter. If you do so, you can have a single *apachectl* script for all developers, after you modify it to automatically find out the UID of the user, who executes the script and set the right paths.

The last thing is to provide developers with an option to run in single process mode by:

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

In addition to making life easier, we decided to use relative links everywhere in the static documents, including the calls to CGIs. You may ask how using relative links will get to the right server port. It's very simple, we use `mod_rewrite`.

To use `mod_rewrite` you have to configure your *httpd\_docs* server with `--enable-module=rewrite` and recompile, or use DSO and load the module in *httpd.conf*. In the *httpd.conf* of our *httpd\_docs* server we have the following code:

```
RewriteEngine on

# stas's server
# port = 8000
RewriteCond %{REQUEST_URI} ^/(perl|cgi-perl)
RewriteCond %{REMOTE_ADDR} 123.34.45.56
RewriteRule ^(.*)          http://example.com:8000/$1 [P,L]

# eric's server
# port = 8001
RewriteCond %{REQUEST_URI} ^/(perl|cgi-perl)
RewriteCond %{REMOTE_ADDR} 123.34.45.57
RewriteRule ^(.*)          http://example.com:8001/$1 [P,L]

# all the rest
RewriteCond %{REQUEST_URI} ^/(perl|cgi-perl)
RewriteRule ^(.*)          http://example.com:81/$1 [P]
```

The IP addresses are the addresses of the developer desktop machines (where they are running their web browsers). So if an html file includes a relative URI */perl/test.pl* or even *http://www.example.com/perl/test.pl*, clicking on the link will be internally proxied to *http://www.example.com:8000/perl/test.pl* if the click has been made at the user *stas*'s desktop machine, or to *http://www.example.com:8001/perl/test.pl* for a request generated from the user *eric*'s machine, per our above URI rewrite example.

Another possibility is to use `REMOTE_USER` variable if all the developers are forced to authenticate themselves before they can access the server. If you do, you will have to change the `RewriteRules` to match `REMOTE_USER` in the above example.

We wish to stress again, that the above setup will work only with relative URIs in the HTML code. If you choose to generate full URIs including non-80 port the requests originated from this HTML code will bypass the light server listening to the default port 80, and go directly to the *server:port* of the full URI.

## 7.13 Wrapper to Emulate the Server Perl Environment

Often you will start off debugging your script by running it from your favorite shell program. Sometimes you encounter a very weird situation when the script runs from the shell but dies when processed as a CGI script by a web-server. The real problem often lies in the difference between the environment variables

that is used by your web-server and the ones used by your shell program.

For example you may have a set of non-standard Perl directories, used for local Perl modules. You have to tell the Perl interpreter where these directories are. If you don't want to modify `@INC` in all scripts and modules, you can use a `PERL5LIB` environment variable, to tell Perl where the directories are. But then you might forget to alter the `mod_perl` startup script to correct `@INC` there as well. And if you forget this, you can be quite puzzled why the scripts are running from the shell program, but not from the web.

Of course the *error\_log* will help as well to find out what the problem is, but there can be other obscure cases, where you do something different at the shell program and your scripts refuse to run under the web-server.

Another example is when you have more than one version of Perl installed. You might call the first version of the Perl executable in the first script's line (the shebang line), but to have the web-server compiled with another Perl version. Since `mod_perl` ignores the path to the Perl executable at the first line of the script, you can get quite confused the code won't do the same when processed as request, compared to be executed from the command line. It will take a while before you realize that you test the scripts from the shell program using the *wrong* Perl version.

The best debugging approach is to write a wrapper that emulates the exact environment of the server, first deleting environment variables like `PERL5LIB` and then calling the same perl binary that it is being used by the server. Next, set the environment identical to the server's by copying the Perl run directives from the server startup and configuration files or even *require()*'ing the startup file, if it doesn't include `Apache::modules` stuff, unavailable under shell. This will also allow you to remove completely the first line of the script, since `mod_perl` doesn't need it anyway and the wrapper knows how to call the script.

Here is an example of such a script. Note that we force the use of `-Tw` when we call the real script. Since when debugging we want to make sure that the code is working when the taint mode is on, and we want to see all the warnings, to help Perl help us have a better code.

We have also added the ability to pass parameters, which will not happen when you will issue a request to script, but it can be helpful at times.

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

# This is a wrapper example

# It simulates the web server environment by setting @INC and other
# stuff, so what will run under this wrapper will run under Web and
# vice versa.

#
# Usage: wrap.pl some_cgi.pl
#
BEGIN {
    # we want to make a complete emulation, so we must reset all the
    # paths and add the standard Perl libs
    @INC =
        qw(/usr/lib/perl5/5.00503/i386-linux
           /usr/lib/perl5/5.00503
           /usr/lib/perl5/site_perl/5.005/i386-linux
```



```

    /usr/lib/perl5/site_perl/5.005
    .
  );
}

use strict;
use File::Basename;

# process the passed params
my $cgi = shift || '';
my $params = (@ARGV) ? join(" ", @ARGV) : '';

die "Usage:\n\t$0 some_cgi.pl\n" unless $cgi;

# Set the environment
my $PERL5LIB = join ":", @INC;

# if the path includes the directory
# we extract it and chdir there
if (index($cgi, '/') >= 0) {
    my $dirname = dirname($cgi);
    chdir $dirname or die "Can't chdir to $dirname: $! \n";

    $cgi =~ m|$dirname/(.*)|;
    $cgi = $1;
}

# run the cgi from the script's directory
# Note that we set Warning and Taint modes ON!!!
system qq{/usr/bin/perl -I$PERL5LIB -Tw $cgi $params};

```

## 7.14 Server Maintenance Chores

It's not enough to have your server and service up and running. You have to maintain the server even when everything seems to be fine. This includes security auditing, keeping an eye on the size of remaining unused disk space, available RAM, the load of the system, etc.

If you forget about these chores one day (sooner or later) your system will crash either because it has run out of free disk space, all the available CPU has been used and system has started heavily to swap or someone has broken in. Unfortunately the scope of this guide is not covering the latter, since it will take more than one book to profoundly cover this issue, but the rest of the thing are quite easy to prevent if you follow our advices.

Certainly, your particular system might have maintenance chores that aren't covered here, but at least you will be alerted that these chores are real and should be taken care of.

### 7.14.1 *Handling Log Files*

There are two issues to solve with log files. First they should be rotated and compressed on the constant basis, since they tend to use big parts of the disk space over time. Second these should be monitored for possible sudden explosive growth rates, when something goes astray in your code running at the mod\_perl server and the process starts to log thousands of error messages in second without stopping, until all the

disk space is used, and the server cannot work anymore.

### 7.14.1.1 Log Rotation

The first issue is solved by having a process run by crontab at certain times (usually off hours, if this term is still valid in the Internet era) and rotate the logs. The log rotation includes the current log file renaming, server restart (which creates a fresh new log file), and renamed file compression and/or moving it on a different disk.

For example if we want to rotate the *access\_log* file we could do:

```
% mv access_log access_log.renamed
% apachectl restart
% sleep 5; # allow all children to complete requests and logging
           # now it's safe to use access_log.renamed
% mv access_log.renamed /some/directory/on/another/disk
```

This is the script that we run from the crontab to rotate the log files:

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

# This script does log rotation. Called from crontab.

use strict;
$ENV{PATH}='/bin:/usr/bin';

### configuration
my @logfiles = qw(access_log error_log);
umask 0;
my $server = "httpd_perl";
my $logs_dir = "/usr/local/var/$server/logs";
my $restart_command = "/usr/local/sbin/$server/apachectl restart";
my $gzip_exec = "/usr/bin/gzip";

my ($sec,$min,$hour,$mday,$mon,$year) = localtime(time);
my $time = sprintf "%0.4d.%0.2d.%0.2d-%0.2d.%0.2d.%0.2d",
    $year+1900,++$mon,$mday,$hour,$min,$sec;
$^I = ".$time";

# rename log files
chdir $logs_dir;
@ARGV = @logfiles;
while (<>) {
    close ARGV;
}

# now restart the server so the logs will be restarted
system $restart_command;

# allow all children to complete requests and logging
sleep 5;
```

```
# compress log files
foreach (@logfiles) {
    system "$gzip_exec $_.$time";
}
```

Note: Setting `$^I` sets the in-place edit flag to a dot followed by the time. We copy the names of the logfiles into `@ARGV`, and open each in turn and immediately close them without doing any changes; but because the in-place edit flag is set they are effectively renamed.

As you see the rotated files will include the date and the time in their filenames.

Here is a more generic set of scripts for log rotation. Cron job fires off setuid script called log-roller that looks like this:

```
#!/usr/bin/perl -Tw
use strict;
use File::Basename;

$ENV{PATH} = "/usr/ucb:/bin:/usr/bin";

my $ROOT = "/WWW/apache"; # names are relative to this
my $CONF = "$ROOT/conf/httpd.conf"; # master conf
my $MIDNIGHT = "MIDNIGHT"; # name of program in each logdir

my ($user_id, $group_id, $pidfile); # will be set during parse of conf
die "not running as root" if $>;

chdir $ROOT or die "Cannot chdir $ROOT: $!";

my %midnights;
open CONF, "<$CONF" or die "Cannot open $CONF: $!";
while (<CONF>) {
    if (/^User (\w+)/i) {
        $user_id = getpwnam($1);
        next;
    }
    if (/^Group (\w+)/i) {
        $group_id = getgrnam($1);
        next;
    }
    if (/^PidFile (.*)/i) {
        $pidfile = $1;
        next;
    }
    next unless /^ErrorLog (.*)/i;
    my $midnight = (dirname $1)."/$MIDNIGHT";
    next unless -x $midnight;
    $midnights{$midnight}++;
}
close CONF;

die "missing User definition" unless defined $user_id;
die "missing Group definition" unless defined $group_id;
die "missing PidFile definition" unless defined $pidfile;
```

```

open PID, $pidfile or die "Cannot open $pidfile: $!";
<PID> =~ /(\d+)/;
my $httpd_pid = $1;
close PID;
die "missing pid definition" unless defined $httpd_pid and $httpd_pid;
kill 0, $httpd_pid or die "cannot find pid $httpd_pid: $!";

for (sort keys %midnights) {
    defined(my $pid = fork) or die "cannot fork: $!";
    if ($pid) {
        ## parent:
        waitpid $pid, 0;
    } else {
        my $dir = dirname $_;
        ($($, $)) = ($group_id, $group_id);
        ($(<, $>)) = ($user_id, $user_id);
        chdir $dir or die "cannot chdir $dir: $!";
        exec ".$MIDNIGHT";
        die "cannot exec $MIDNIGHT: $!";
    }
}

kill 1, $httpd_pid or die "Cannot SIGHUP $httpd_pid: $!";

```

And then individual MIDNIGHT scripts can look like this:

```

#!/usr/bin/perl -Tw
use strict;

die "bad guy" unless getpwuid($<) =~ /^(root|nobody)$/;
my @LOGFILES = qw(access_log error_log);
umask 0;
$I = ".$time;
@ARGV = @LOGFILES;
while (<>) {
    close ARGV;
}

```

Can you spot the security holes? Take your time... This code shouldn't be used in hostile situations.

### 7.14.1.2 Non-Scheduled Emergency Log Rotation

As we have mentioned before, there are times when the web server goes wild and starts to log lots of messages to the *error\_log* file non-stop. If no one monitors this, it is possible that in a few minutes all the free disk spaces will be filled and no process will be able to work normally. When this happens, the I/O the faulty server causes is so heavy that its sibling processes cannot serve requests.

Generally this is not the case, but a few people have reported to encounter this problem. If you are one of these people, you should run the monitoring program that checks the log file size and if it notices that the file has grown too large, it should attempt to restart the server and probably trim the log file.

When we have used a quite old `mod_perl` version, sometimes we have had bursts of an error *Callback called exit* showing up in our *error\_log*. The file could grow to 300 Mbytes in a few minutes.

We will show you is an example of the script that should be executed from the crontab, to handle the situations like this. The cron job should run every few minutes or even every minute, since if you experience this problem you know that log files fills up very fast. The example script will rotate when the *error\_log* will grow over 100K. Note that this script is useful when you have the normal scheduled log rotation facility working, remember that this one is an emergency solver and not to be used for routine log rotation.

```
emergency_rotate.sh
-----
#!/bin/sh
S=`ls -s /usr/local/apache/logs/error_log | awk '{print $1}'`
if [ "$S" -gt 100000 ] ; then
    mv /usr/local/apache/logs/error_log /usr/local/apache/logs/error_log.old
    /etc/rc.d/init.d/httpd restart
    date | /bin/mail -s "error_log $S kB on inx" admin@example.com
fi
```

Of course you could write a more advanced script, using the timestamps and other whistles. This example comes to illustrate how to solve the problem in question.

Another solution is to use an out of box tools that are written for this purpose. The `daemontools` package (<ftp://koobera.math.uic.edu/www/daemontools.html>) includes a utility called `multilog`. This utility saves stdin stream to one or more log files. It optionally timestamps each line and, for each log, includes or excludes lines matching specified patterns. It automatically rotates logs to limit the amount of disk space used. If the disk fills up, it pauses and tries again, without losing any data.

The obvious caveat is that it doesn't restart the server, so while it tries to solve the log file handling problem it doesn't handle the originator of the problem. But since the I/O of the log writing process Apache process will be quite heavy, the rest of the servers will work very slowly if at all, and a normal watchdog should detect this abnormal situation and restart the Apache server.

## 7.15 Swapping Prevention

Before I delve into swapping process details, let's refresh our knowledge of memory components and memory management

The computer memory is called RAM, which stands for Random Access Memory. Reading and writing to RAM is, by a few orders, faster than doing the same operations on a hard disk, the former uses non-movable memory cells, while the latter uses rotating magnetic media.

On most operating systems swap memory is used as an extension for RAM and not as a duplication of it. So if your OS is one of those, if you have 128MB of RAM and 256MB swap partition, you have a total of 384MB of memory available. You should never count the extra memory when you decide on the maximum number of processes to be run, and I will show why in the moment.

The swapping memory can be built of a number of hard disk partitions and swap files formatted to be used as swap memory. When you need more swap memory you can always extend it on demand as long as you have some free disk space (for more information see the *mkswap* and *swapon* manpages).

System memory is quantified in units called memory pages. Usually the size of a memory page is between 1KB and 8KB. So if you have 256MB of RAM installed on your machine and the page size is 4KB your system has 64,000 main memory pages to work with and these pages are fast. If you have 256MB swap partition the system can use yet another 64,000 memory pages, but they are much slower.

When the system is started all memory pages are available for use by the programs (processes).

Unless the program is really small, the process running this program uses only a few segments of the program, each segment mapped onto its own memory page. Therefore only a few memory pages are required to be loaded into the memory.

When the process needs an additional program's segment to be loaded into the memory, it asks the system whether the page containing this segment is already loaded in the memory. If the page is not found--an event known as a *page fault* occurs, which requires the system to allocate a free memory page, go to the disk, read and load the requested program's segment into the allocated memory page.

If a process needs to bring a new page into physical memory and there are no free physical pages available, the operating system must make room for this page by discarding another page from physical memory.

If the page to be discarded from physical memory came from an image or data file and has not been written to then the page does not need to be saved. Instead it can be discarded and if the process needs that page again it can be brought back into memory from the image or data file.

However, if the page has been modified, the operating system must preserve the contents of that page so that it can be accessed at a later time. This type of page is known as a *dirty page* and when it is removed from memory it is saved in a special sort of file called the swap file. This process is referred to as a *swapping out*.

Accesses to the swap file are very long relative to the speed of the processor and physical memory and the operating system must juggle the need to write pages to disk with the need to retain them in memory to be used again.

In order to improve the swapping out process, to decrease the possibility that the page that has just been swapped out, will be needed at the next moment, the LRU (least recently used) or a similar algorithm is used.

To summarize the two swapping scenarios, read-only pages discarding incurs no overhead in contrast with the discarding scenario of the data pages that have been written to, since in the latter case the pages have to be written to a swap partition located on the slow disk. Therefore your machine's overall performance will be much better if there will be less memory pages that can become dirty.

But the problem is, Perl is a language with no strong data types, which means that both the program code and the program data are seen as a data pages by OS since both mapped to the same memory pages. Therefore a big chunk of your Perl code becomes dirty when its variables are modified and when the pages need to be discarded they have to be written to the swap partition.

This leads us to two important conclusions about swapping and Perl.

- Running your system when there is no free main memory available hinders performance, because processes memory pages should be discarded and then reread from disk again and again.
- Since a majority of the running code is a Perl code, in addition to the overhead of reading the previously discarded pages in, the overhead of saving the dirty pages to the swap partition is occurring.

When the system has to swap memory pages in and out, the system slows down, not serving the processes as fast as before. This leads to an accumulation of processes waiting for their turn to run, which further causes processing demands to go up, which in turn slows down the system even more as more memory is required. This ever worsening spiral will lead the machine to halt, unless the resource demand suddenly drops down and allows the processes to catch up with their tasks and go back to normal memory usage.

In addition it's important to know that for a better performance, most programs, particularly programs written in Perl, on most modern OSs don't return memory pages while they are running. If some of the memory gets freed it's reused when needed by the process, without creating the additional overhead of asking the system to allocate new memory pages. That's why you will observe that Perl programs grow in size as they run and almost never shrink.

When the process quits it returns its memory pages to the pool of freely available pages for other processes to use.

This scenario is certainly educating, and it should be now obvious that your system that runs the web server should never swap. It's absolutely normal for your desktop to start swapping. You will see it immediately since things will slow down and sometimes the system will freeze for a short periods. But as I've just mentioned, you can stop starting new programs and can quit some, thus allowing the system to catch up with the load and come back to use the RAM.

In the case of the web server you have much less control since it's users who load your machine by issuing requests to your server. Therefore you should configure the server, so that the maximum number of possible processes will be small enough using the `MaxClients` directive (For the technique for choosing the right `MaxClients` refer to the section 'Choosing MaxClients'). This will ensure that at peak hours the system won't swap. Remember that swap space is an emergency pool, not a resource to be used routinely. If you are low on memory and you badly need it, buy it or reduce the number of processes to prevent swapping.

However sometimes, due to the faulty code, some process might start spinning in an unconstrained loop, consuming all the available RAM and starting to heavily use swap memory. In such a situation it helps when you have a big emergency pool (i.e. lots of swap memory). But you have to resolve this problem as soon as possible since this pool won't last for a long time. In the meanwhile the `Apache::Resource` module can be handy.

For swapping monitoring techniques see the section 'Apache::VMonitor -- Visual System and Apache Server Monitor'.

## 7.16 Preventing mod\_perl Processes From Going Wild

Sometimes people report that they had a problem with their code running under mod\_perl that has caused all the RAM or all the disk to be used. The following tips should help you prevent these problems, before if at all they hit you.

### 7.16.1 All RAM Consumed

Sometimes calling an undefined subroutine in a module can cause a tight loop that consumes all the available memory. Here is a way to catch such errors. Define an `UNIVERSAL::AUTOLOAD` subroutine in your *startup.pl*, or in a `<Perl></Perl>` section in your *httpd.conf* file:

```
sub UNIVERSAL::AUTOLOAD {
    my $class = shift;
    warn "$class can't \${UNIVERSAL::AUTOLOAD}=${UNIVERSAL::AUTOLOAD}!\n";
}
```

You can either put it in your *startup.pl*, or in a `<Perl></Perl>` section in your *httpd.conf* file. I do the latter. Putting it in all your mod\_perl modules would be redundant (and might give you compile-time errors).

This will produce a nice error in *error\_log*, giving the line number of the call and the name of the undefined subroutine.

## 7.17 Maintainers

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

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

## 7.18 Authors

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

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



## **8 mod\_perl Advocacy**

## 8.1 Description

Having a hard time getting mod\_perl into your organization? We have collected some arguments you can use to convince your boss why the organization wants mod\_perl.

You can contact the mod\_perl advocacy list if you have any more questions, or good arguments you have used (any success-stories are also welcome to the docs-dev list).

Also see Popular Perl Complaints and Myths.

## 8.2 Thoughts about scalability and flexibility

Your need for scalability and flexibility depends on what you need from your web site. If you only want a simple guest book or database gateway with no feature headroom, you can get away with any EASY\_AND\_FAST\_TO\_DEVELOP\_TOOL (Exchange, MS IIS, Lotus Notes, etc).

Experience shows that you will soon want more functionality, at which point you'll discover the limitations of these "easy" tools. Gradually, your boss will ask for increasing functionality and at some point you'll realize that the tool lacks flexibility and/or scalability. Then your boss will either buy another EASY\_AND\_FAST\_TO\_DEVELOP\_WITH\_TOOLS and repeat the process (with different unforeseen problems), or you'll start investing time in learning how to use a powerful, flexible tool to make the long-term development cycle easier.

If you and your company are serious about delivering flexible Internet functionality, do your homework. Then urge your boss to invest a little extra time and resources in choosing the right tool for the job. The extra quality and manageability of your site along with your ability to deliver new and improved functionality of high quality and in good time will prove the superiority of using solid flexible tools.

## 8.3 The boss, the developer and advocacy

Each developer has a boss who participates in the decision-making process. Remember that the boss considers input from sales people, developers, the media and associates before handing down large decisions. Of course, results count! A sales brochure makes very little impact compared to a working demonstration, and demonstrations of company-specific and developer-specific results count for a lot!

Personally, when I discovered mod\_perl I did a lot of testing and coding at home and at work. Once I had a working heavy application, I came to my boss with two URLs - one for the plain CGI server and the other for the mod\_perl-enabled server. It took about 30 secs for my boss to say: 'Go with it'. Of course since then I have had to provide all the support for other developers, which is why I took time to learn it in first place (and why this guide was created!).

Chances are that if you've done your homework, learnt the tools and can deliver results, you'll have a successful project. If you convince your boss to try a tool that you don't know very well, your results may suffer. If your boss follows your development process closely and sees that your progress is much worse than expected, you might be told to "forget it" and mod\_perl might not get a second chance.

Advocacy is a great thing for the open-source software movement, but it's best done quietly until you have confidence that you can show productivity. If you can demonstrate to your boss a heavy CGI which is running much faster under mod\_perl, that may be a strong argument for further evaluation. Your company may even sponsor a portion of your learning process.

Learn the technology by working on sample projects. Learn how to support yourself and learn how to get support from the community; then advocate your ideas to your boss. Then you'll have the knowledge; your company will have the benefit; and mod\_perl will have the reputation it deserves.

## 8.4 A summary of perl/CGI discussion at slashdot.org

Well, there was a nice discussion of merits of Perl in CGI world. I took the time to summarize this thread, so here is what I've got:

Perl Domination in CGI Programming? <http://slashdot.org/askslashdot/99/10/20/1246241.shtml>

- Perl is cool and fun to code with.
- Perl is very fast to develop with.
- Perl is even faster to develop with if you know what CPAN is. :)
- Math intensive code and other stuff which is faster in C/C++, can be plugged into Perl with XS/SWIG and may be used transparently by Perl programmers.
- Most CGI applications do text processing, at which Perl excels
- Forking and loading (unless the code is shared) of C/C++ CGI programs produces an overhead.
- Except for Intranets, bandwidth is usually a bigger bottleneck than Perl performance, although this might change in the future.
- For database driven applications, the database itself is a bottleneck. Lots of posts talk about latency vs throughput.
- mod\_perl, FastCGI, Velocigen and PerlEx all give good performance gains over plain mod\_cgi.
- Other light alternatives to Perl and its derivatives which have been mentioned: PHP, Python.
- There were almost no voices from users of M\$ and similar technologies, I guess that's because they don't read <http://slashdot.org> :)
- Many said that in many people's minds: 'CGI' eq 'Perl'

## 8.5 Maintainers

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

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

## 8.6 Authors

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

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

## **9 Popular Perl Complaints and Myths**

## 9.1 Description

This document tries to explain the myths about Perl and overturn the FUD certain bodies try to spread.

## 9.2 Abbreviations

- **M** = Misconception or Myth
- **R** = Response

### 9.2.1 *Interpreted vs. Compiled*

- **M:**

Each dynamic perl page hit needs to load the Perl interpreter and compile the script, then run it each time a dynamic web page is hit. This dramatically decreases performance as well as makes Perl an unscalable model since so much overhead is required to search each page.

- **R:**

This myth was true years ago before the advent of `mod_perl`. `mod_perl` loads the interpreter once into memory and never needs to load it again. Each perl program is only compiled once. The compiled version is then kept into memory and used each time the program is run. In this way there is no extra overhead when hitting a `mod_perl` page.

#### 9.2.1.1 Interpreted vs. Compiled (More Gory Details)

- **R:**

Compiled code always has the potential to be faster than interpreted code. Ultimately, all interpreted code needs to eventually be converted to native instructions at some point, and this is invariably has to be done by a compiled application.

That said, an interpreted language CAN be faster than a comparable native application in certain situations, given certain, common programming practices. For example, the allocation and de-allocation of memory can be a relatively expensive process in a tightly scoped compiled language, whereas interpreted languages typically use garbage collectors which don't need to do expensive deallocation in a tight loop, instead waiting until additional memory is absolutely necessary, or for a less computationally intensive period. Of course, using a garbage collector in C would eliminate this edge in this situation, but where using garbage collectors in C is uncommon, Perl and most other interpreted languages have built-in garbage collectors.

It is also important to point out that few people use the full potential of their modern CPU with a single application. Modern CPUs are not only more than fast enough to run interpreted code, many processors include instruction sets designed to increase the performance of interpreted code.

### ***9.2.2 Perl is overly memory intensive making it unscalable***

- **M:**

Each child process needs the Perl interpreter and all code in memory. Even with mod\_perl httpd processes tend to be overly large, slowing performance, and requiring much more hardware.

- **R:**

In mod\_perl the interpreter is loaded into the parent process and shared between the children. Also, when scripts are loaded into the parent and the parent forks a child httpd process, that child shares those scripts with the parent. So while the child may take 6MB of memory, 5MB of that might be shared meaning it only really uses 1MB per child. Even 5 MB of memory per child is not uncommon for most web applications on other languages.

Also, most modern operating systems support the concept of shared libraries. Perl can be compiled as a shared library, enabling the bulk of the perl interpreter to be shared between processes. Some executable formats on some platforms (I believe ELF is one such format) are able to share entire executable TEXT segments between unrelated processes.

#### **9.2.2.1 More Tuning Advice:**

- Stas Bekman's Performance Guide

### ***9.2.3 Not enough support, or tools to develop with Perl. (Myth)***

- **R:**

Of all web applications and languages, Perl arguable has the most support and tools. **CPAN** is a central repository of Perl modules which are freely downloadable and usually well supported. There are literally thousands of modules which make building web apps in Perl much easier. There are also countless mailing lists of extremely responsive Perl experts who usually respond to questions within an hour. There are also a number of Perl development environments to make building Perl Web applications easier. Just to name a few, there is Apache::ASP, Mason, embPerl, ePerl, etc...

### ***9.2.4 If Perl scales so well, how come no large sites use it? (myth)***

- **R:**

Actually, many large sites DO use Perl for the bulk of their web applications. Here are some, just as an example: **e-Toys**, **CitySearch**, **Internet Movie Database**( <http://imdb.com> ), **Value Click** ( <http://valueclick.com> ), **Paramount Digital Entertainment**, **CMP** ( <http://cmpnet.com> ), **HotBot Mail/HotBot Homepages**, and **DejaNews** to name a few. Even **Microsoft** has taken interest in Perl via <http://www.activestate.com/>.

### ***9.2.5 Perl even with mod\_perl, is always slower than C.***

- **R:**

The Perl engine is written in C. There is no point arguing that Perl is faster than C because anything written in Perl could obviously be re-written in C. The same holds true for arguing that C is faster than assembly. There are two issues to consider here. First of all, many times a web application written in Perl **CAN be faster** than C thanks to the low level optimizations in the Perl compiler. In other words, its easier to write poorly written C than well written Perl. Secondly its important to weigh all factors when choosing a language to build a web application in. Time to market is often one of the highest priorities in creating a web application. Development in Perl can often be twice as fast as in C. This is mostly due to the differences in the language themselves as well as the wealth of free examples and modules which speed development significantly. Perl's speedy development time can be a huge competitive advantage.

### ***9.2.6 Java does away with the need for Perl.***

- **M:**

Perl had its place in the past, but now there's Java and Java will kill Perl.

- **R:**

Java and Perl are actually more complimentary languages than competitive. Its widely accepted that server side Java solutions such as JServ, JSP and JRUN, are far slower than mod\_perl solutions (see next myth). Even so, Java is often used as the front end for server side Perl applications. Unlike Perl, with Java you can create advanced client side applications. Combined with the strength of server side Perl these client side Java applications can be made very powerful.

### ***9.2.7 Perl can't create advanced client side applications***

- **R:**

True. There are some client side Perl solutions like PerlScript in MSIE 5.0, but all client side Perl requires the user to have the Perl interpreter on their local machine. Most users do not have a Perl interpreter on their local machine. Most Perl programmers who need to create an advanced client side application use Java as their client side programming language and Perl as the server side solution.

### ***9.2.8 ASP makes Perl obsolete as a web programming language.***

- **M:**

With Perl you have to write individual programs for each set of pages. With ASP you can write simple code directly within HTML pages. ASP is the Perl killer.



- **R:**

There are many solutions which allow you to embed Perl in web pages just like ASP. In fact, you can actually use Perl IN ASP pages with PerlScript. Other solutions include: Mason, Apache::ASP, ePerl, embPerl and XPP. Also, Microsoft and ActiveState have worked very hard to make Perl run equally well on NT as Unix. You can even create COM modules in Perl that can be used from within ASP pages. Some other advantages Perl has over ASP: mod\_perl is usually much faster than ASP, Perl has much more example code and full programs which are freely downloadable, and Perl is cross platform, able to run on Solaris, Linux, SCO, Digital Unix, Unix V, AIX, OS2, VMS MacOS, Win95-98 and NT to name a few.

Also, Benchmarks show that embedded Perl solutions outperform ASP/VB on IIS by several orders of magnitude. Perl is a much easier language for some to learn, especially those with a background in C or C++.

## 9.3 Credits

Thanks to the mod\_perl list for all of the good information and criticism. I'd especially like to thank,

- Stas Bekman <stas@stason.org>
- Thornton Prime <thornton@cnation.com>
- Chip Turner <chip@ns.zfx.com>
- Clinton <clint@drtech.co.uk>
- Joshua Chamas <joshua@chamas.com>
- John Edstrom <edstrom@Poopsie.hmsc.orst.edu>
- Rasmus Lerdorf <rasmus@lerdorf.on.ca>
- Nedim Cholich <nedim@comstar.net>
- Mike Perry <<http://www.icorp.net/icorp/feedback.htm> >
- Finally, I'd like to thank Robert Santos <robert@cnation.com>, CyberNation's lead Business Development guy for inspiring this document.

## 9.4 Maintainers

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

- Contact the mod\_perl docs list

## 9.5 Authors

- Adam Pisoni <adam@cnaion.com>

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

## Table of Contents:

<b>General Documentation</b>	1
<b>Perl Reference</b>	4
1 Perl Reference	4
1.1 Description	5
1.2 perldoc's Rarely Known But Very Useful Options	5
1.3 Tracing Warnings Reports	6
1.4 Variables Globally, Lexically Scoped And Fully Qualified	8
1.4.1 Symbols, Symbol Tables and Packages; Typeglobs	8
1.4.1.1 Lexical Variables and Symbols	10
1.4.2 Additional reading references	11
1.5 my() Scoped Variable in Nested Subroutines	11
1.5.1 The Poison	11
1.5.2 The Diagnosis	12
1.5.3 The Remedy	13
1.6 Understanding Closures -- the Easy Way	14
1.6.1 Mike Guy's Explanation of the Inner Subroutine Behavior	16
1.7 When You Cannot Get Rid of The Inner Subroutine	17
1.7.1 Remedies for Inner Subroutines	19
1.8 use(), require(), do(), %INC and @INC Explained	26
1.8.1 The @INC array	26
1.8.2 The %INC hash	26
1.8.3 Modules, Libraries and Program Files	29
1.8.4 require()	31
1.8.5 use()	32
1.8.6 do()	33
1.9 Using Global Variables and Sharing Them Between Modules/Packages	34
1.9.1 Making Variables Global	34
1.9.2 Making Variables Global With strict Pragma On	34
1.9.3 Using Exporter.pm to Share Global Variables	34
1.9.4 Using the Perl Aliasing Feature to Share Global Variables	37
1.9.5 Using Non-Hardcoded Configuration Module Names	38
1.10 The Scope of the Special Perl Variables	39
1.11 Compiled Regular Expressions	40
1.12 Exception Handling for mod_perl	42
1.12.1 Trapping Exceptions in Perl	43
1.12.2 Alternative Exception Handling Techniques	44
1.12.3 Better Exception Handling	45
1.12.3.1 A Little Housekeeping	46
1.12.3.2 An Exception Class	47
1.12.4 Catching Uncaught Exceptions	47
1.12.4.1 Using \$SIG{__DIE__}	48
1.12.4.2 Overriding the Core die() Function	48
1.12.5 A Single UnCaught Exception Class	49
1.12.6 Some Uses	50

## Table of Contents:

1.12.7	Conclusions . . . . .	51
1.12.8	The My::Exception class in its entirety . . . . .	51
1.12.9	Other Implementations . . . . .	52
1.13	Customized __DIE__ handler . . . . .	53
1.14	Maintainers . . . . .	54
1.15	Authors . . . . .	54
	<b>Preparing mod_perl modules for CPAN</b> . . . . .	55
2	Preparing mod_perl modules for CPAN . . . . .	55
2.1	Description . . . . .	56
2.2	Defining Makefile.PL Prerequisites that Require mod_perl . . . . .	56
2.3	Writing the Test Suite . . . . .	56
2.4	Maintainers . . . . .	57
2.5	Authors . . . . .	57
	<b>Running and Developing Tests with the Apache::Test Framework</b> . . . . .	58
3	Running and Developing Tests with the Apache::Test Framework . . . . .	58
3.1	Description . . . . .	59
3.2	Basics of Perl Modules Testing . . . . .	59
3.3	Prerequisites . . . . .	60
3.4	Running Tests . . . . .	60
3.4.1	Testing Options . . . . .	61
3.4.2	Basic Testing . . . . .	61
3.4.3	Individual Testing . . . . .	61
3.4.4	Repetitive Testing . . . . .	62
3.4.5	Parallel Testing . . . . .	62
3.4.6	Verbose Mode . . . . .	63
3.4.7	Colored Trace Mode . . . . .	63
3.4.8	Controlling the Apache::Test's Signal to Noise Ratio . . . . .	63
3.4.9	Stress Testing . . . . .	64
3.4.9.1	The Problem . . . . .	64
3.4.9.2	The Solution . . . . .	64
3.4.9.3	Resolving Sequence Problems . . . . .	65
3.4.9.4	Apache::TestSmoke Solution . . . . .	66
3.4.10	RunTime Configuration Overriding . . . . .	67
3.4.11	Request Generation and Response Options . . . . .	68
3.4.12	Batch Mode . . . . .	70
3.5	Setting Up Testing Environment . . . . .	70
3.5.1	Basic Testing Environment . . . . .	71
3.5.2	Extending Configuration Setup . . . . .	76
3.5.3	Special Configuration Files . . . . .	77
3.5.4	Inheriting from System-wide httpd.conf . . . . .	77
3.6	Apache::Test Framework's Architecture . . . . .	78
3.6.1	Developing Response-only Part of a Test . . . . .	78
3.6.2	Developing Response and Request Parts of a Test . . . . .	80
3.6.3	Developing Test Response Handlers in C . . . . .	82
3.6.4	Request and Response Methods . . . . .	83
3.6.5	Other Request Generation helpers . . . . .	86
3.6.6	Starting Multiple Servers . . . . .	87

3.6.7 Multiple User Agents . . . . .	87
3.6.8 Hitting the Same Interpreter (Server Thread/Process Instance) . . . . .	87
3.7 Writing Tests . . . . .	89
3.7.1 Defining How Many Sub-Tests Are to Be Run . . . . .	89
3.7.2 Skipping a Whole Test . . . . .	89
3.7.3 Skipping Numerous Tests . . . . .	93
3.7.4 Reporting a Success or a Failure of Sub-tests . . . . .	94
3.7.5 Skipping Sub-tests . . . . .	95
3.7.6 Running only Selected Sub-tests . . . . .	96
3.7.7 Todo Sub-tests . . . . .	97
3.7.8 Making it Easy to Debug . . . . .	97
3.7.9 Tie-ing STDOUT to a Response Handler Object . . . . .	99
3.7.10 Helper Functions . . . . .	100
3.7.11 Auto Configuration . . . . .	101
3.7.11.1 Forcing Configuration Sections into the Top Level . . . . .	102
3.7.11.2 Bypassing Auto-Configuration . . . . .	102
3.7.11.3 Virtual Hosts . . . . .	103
3.7.11.4 Running Pre-Configuration Code . . . . .	103
3.7.11.5 Controlling the Configuration Order . . . . .	104
3.7.12 Threaded versus Non-threaded Perl Test's Compatibility . . . . .	105
3.7.13 Retrieving the Server Configuration Data . . . . .	106
3.7.13.1 Module Magic Number . . . . .	106
3.8 Debugging Tests . . . . .	106
3.8.1 Under C debugger . . . . .	106
3.8.2 Under Perl debugger . . . . .	107
3.8.3 Tracing . . . . .	108
3.9 Using Apache::Test to Speed up Project Development . . . . .	108
3.10 Writing Tests Methodology . . . . .	109
3.10.1 When Tests Should Be Written . . . . .	110
3.11 Other Webserver Regression Testing Frameworks . . . . .	110
3.12 References . . . . .	110
3.13 Maintainers . . . . .	110
3.14 Authors . . . . .	111
<b>Issuing Correct HTTP Headers . . . . .</b>	<b>112</b>
4 Issuing Correct HTTP Headers . . . . .	112
4.1 Description . . . . .	113
4.2 The Origin of this Chapter . . . . .	113
4.3 Why Headers . . . . .	113
4.4 Which Headers . . . . .	113
4.4.1 Date Related Headers . . . . .	114
4.4.1.1 Date . . . . .	114
4.4.1.2 Last-Modified . . . . .	114
4.4.1.3 Expires and Cache-Control . . . . .	115
4.4.2 Content Related Headers . . . . .	116
4.4.2.1 Content-Type . . . . .	116
4.4.2.2 Content-Length . . . . .	116
4.4.2.3 Entity Tags . . . . .	117

Table of Contents:

4.4.3 Content Negotiation . . . . .	118
4.4.3.1 Vary . . . . .	119
4.5 Requests . . . . .	119
4.5.1 HEAD . . . . .	119
4.5.2 POST . . . . .	120
4.5.3 GET . . . . .	120
4.5.4 Conditional GET . . . . .	121
4.6 Avoiding Dealing with Headers . . . . .	122
4.7 References . . . . .	122
4.7.1 [1] . . . . .	122
4.7.2 [2] . . . . .	122
4.7.3 [3] . . . . .	122
4.7.4 [4] . . . . .	123
4.7.5 [5] . . . . .	123
4.8 Other resources . . . . .	123
4.9 Maintainers . . . . .	123
4.10 Authors . . . . .	123
<b>mod_perl for ISPs. mod_perl and Virtual Hosts</b> . . . . .	124
5 mod_perl for ISPs. mod_perl and Virtual Hosts . . . . .	124
5.1 Description . . . . .	125
5.2 ISPs providing mod_perl services - a fantasy or a reality . . . . .	125
5.2.1 Virtual Servers Technologies . . . . .	128
5.3 Virtual Hosts in the guide . . . . .	129
5.4 Maintainers . . . . .	130
5.5 Authors . . . . .	130
<b>Choosing an Operating System and Hardware</b> . . . . .	131
6 Choosing an Operating System and Hardware . . . . .	131
6.1 Description . . . . .	132
6.2 Choosing an Operating System . . . . .	132
6.2.1 Stability and Robustness . . . . .	132
6.2.2 Memory Management . . . . .	133
6.2.3 Memory Leaks . . . . .	133
6.2.4 Sharing Memory . . . . .	133
6.2.5 Cost and Support . . . . .	133
6.2.6 Discontinued Products . . . . .	134
6.2.7 OS Releases . . . . .	134
6.3 Choosing Hardware . . . . .	135
6.3.1 Machine Strength Demands According to Expected Site Traffic . . . . .	136
6.3.1.1 Single Strong Machine vs Many Weaker Machines . . . . .	136
6.3.2 Internet Connection . . . . .	136
6.3.3 I/O Performance . . . . .	137
6.3.4 Memory . . . . .	138
6.3.5 CPU . . . . .	138
6.3.6 Bottlenecks . . . . .	138
6.3.6.1 Solving Hardware Requirement Conflicts . . . . .	139
6.3.7 Conclusion . . . . .	139
6.4 Maintainers . . . . .	139

6.5 Authors . . . . .	139
<b>Controlling and Monitoring the Server . . . . .</b>	<b>140</b>
7 Controlling and Monitoring the Server . . . . .	140
7.1 Description . . . . .	141
7.2 Restarting Techniques . . . . .	141
7.3 Server Stopping and Restarting . . . . .	142
7.4 Speeding up the Apache Termination and Restart . . . . .	143
7.5 Using apachectl to Control the Server . . . . .	143
7.6 Safe Code Updates on a Live Production Server . . . . .	144
7.7 An Intentional Disabling of Live Scripts . . . . .	146
7.8 SUID Start-up Scripts . . . . .	148
7.8.1 Introduction to SUID Executables . . . . .	148
7.8.2 Apache Startup SUID Script's Security . . . . .	148
7.8.3 Sample Apache Startup SUID Script . . . . .	149
7.9 Preparing for Machine Reboot . . . . .	150
7.10 Monitoring the Server. A watchdog. . . . .	153
7.11 Running a Server in Single Process Mode . . . . .	156
7.12 Starting a Personal Server for Each Developer . . . . .	157
7.13 Wrapper to Emulate the Server Perl Environment . . . . .	159
7.14 Server Maintenance Chores . . . . .	161
7.14.1 Handling Log Files . . . . .	161
7.14.1.1 Log Rotation . . . . .	162
7.14.1.2 Non-Scheduled Emergency Log Rotation . . . . .	164
7.15 Swapping Prevention . . . . .	165
7.16 Preventing mod_perl Processes From Going Wild . . . . .	168
7.16.1 All RAM Consumed . . . . .	168
7.17 Maintainers . . . . .	168
7.18 Authors . . . . .	168
<b>mod_perl Advocacy . . . . .</b>	<b>169</b>
8 mod_perl Advocacy . . . . .	169
8.1 Description . . . . .	170
8.2 Thoughts about scalability and flexibility . . . . .	170
8.3 The boss, the developer and advocacy . . . . .	170
8.4 A summary of perl/CGI discussion at slashdot.org . . . . .	171
8.5 Maintainers . . . . .	172
8.6 Authors . . . . .	172
<b>Popular Perl Complaints and Myths . . . . .</b>	<b>173</b>
9 Popular Perl Complaints and Myths . . . . .	173
9.1 Description . . . . .	174
9.2 Abbreviations . . . . .	174
9.2.1 Interpreted vs. Compiled . . . . .	174
9.2.1.1 Interpreted vs. Compiled (More Gory Details) . . . . .	174
9.2.2 Perl is overly memory intensive making it unscalable . . . . .	175
9.2.2.1 More Tuning Advice: . . . . .	175
9.2.3 Not enough support, or tools to develop with Perl. (Myth) . . . . .	175
9.2.4 If Perl scales so well, how come no large sites use it? (myth) . . . . .	175
9.2.5 Perl even with mod_perl, is always slower then C. . . . .	176

Table of Contents:

9.2.6 Java does away with the need for Perl. . . . .	176
9.2.7 Perl can't create advanced client side applications . . . . .	176
9.2.8 ASP makes Perl obsolete as a web programming language. . . . .	176
9.3 Credits . . . . .	177
9.4 Maintainers . . . . .	177
9.5 Authors . . . . .	178