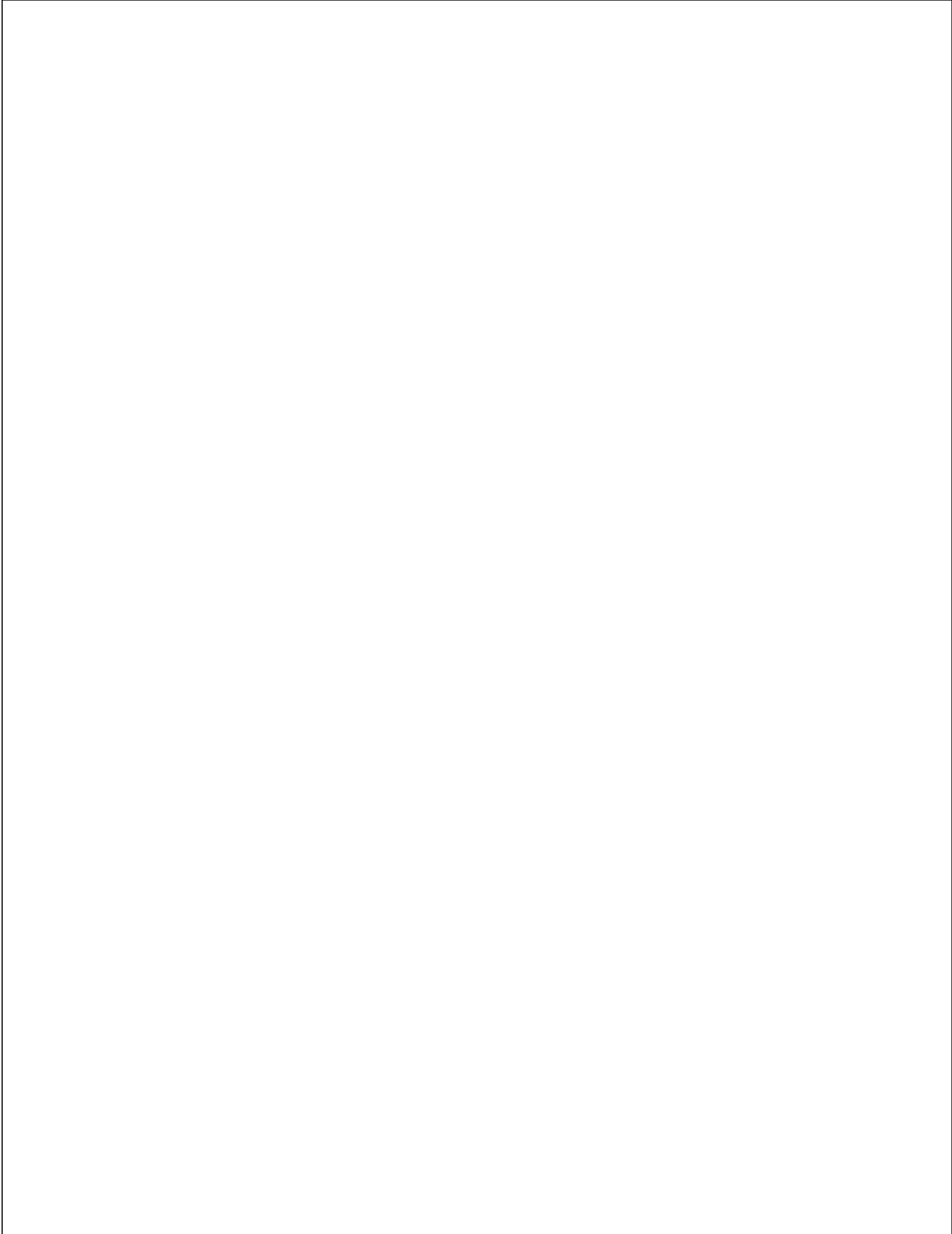


The ApacheCon 2000
March 8, 2000
Orlando, Florida

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

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



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

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

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

1 Getting Started Fast

1.1 mod_perl in Four Slides

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

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

The four slides (sections) are:

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

1.2 What is mod_perl?

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

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

Among various attempts to improve on mod_cgi's shortcomings, mod_perl has proved to be one of the better ones and has been widely adopted by CGI developers. According to the <http://perl.apache.org/netcraft/> about 412000 hosts use mod_perl. Doug MacEachern fathered the core code of this Apache module and licensed it under the “Artistic License” as Perl itself.

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

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

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

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

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

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

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

1.3 Installation

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

```
% cd /usr/src
% lwp-download http://www.apache.org/dist/apache_x.x.x.tar.gz
% lwp-download http://perl.apache.org/dist/mod_perl-x.xx.tar.gz
% tar xzvf apache_x.x.x.tar.gz
% tar xzvf mod_perl-x.xx.tar.gz
% cd mod_perl-x.xx
% perl Makefile.PL APACHE_SRC=../apache_x.x.x/src \
  DO_HTTPD=1 USE_APACI=1 EVERYTHING=1
% make && make test && make install
% cd ../apache_x.x.x
% make install
```

That's all!

- Of course you must replace `x.x.x` with the actual version numbers of the `mod_perl` and Apache releases that you use.

- The GNU `tar` utility knows how to uncompress a gzipped tar archive (use the `z` option).

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

1.4 Configuration

Add the following to the configuration file *httpd.conf*:

```
# for Apache::Registry mode
Alias /perl/ /home/httpd/perl/

PerlModule Apache::Registry
<Location /perl>
  SetHandler perl-script
  PerlHandler Apache::Registry
  Options ExecCGI
  allow from all
  PerlSendHeader On
</Location>
```

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

1.5 The "mod_perl rules" Apache::Registry Scripts

You can write plain perl/CGI scripts just as under `mod_cgi`:

```
mod_perl_rules1.pl
-----
print "Content-type: text/plain\r\n\r\n";
print "mod_perl rules!\n";
```

Of course you can write them in the Apache Perl API:

```
mod_perl_rules2.pl
-----
my $r = shift;
$r->send_http_header('text/plain');
$r->print("mod_perl rules!\n");
```

Save both files under the */home/httpd/perl* directory, make them executable and readable by server, and issue these requests using your favorite browser:

```
http://localhost/perl/mod\_perl\_rules1.pl
http://localhost/perl/mod\_perl\_rules2.pl
```

In both cases you will see on the following response:

```
mod_perl rules!
```

1.6 The "mod_perl rules" Apache Perl Module

To create an Apache Perl module, all you have to do is to wrap the code into a handler subroutine and return the status to the server.

```
ModPerl/Rules.pm
-----
use Apache::Constants;

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

Create a directory called *ModPerl* under one of the directories in @INC, and put *Rules.pm* into it. Then add the following snippet to *httpd.conf*:

```
PerlModule ModPerl::Rules
<Location /mod_perl_rules>
    SetHandler perl-script
    PerlHandler ModPerl::Rules
</Location>
```

Now you can issue a request to:

```
http://localhost/perl/mod\_perl\_rules
```

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

```
mod_perl rules!
```

as the response.

1.7 Is That All I Need To Know About mod_perl?

Definitely not!

These slides are intended to show you that you can install and start using a mod_perl server within 30 minutes of downloading the sources.

There is much more to mod_perl than this, you will need to plan your study around the projects you want to implement. Fortunately, there are many resources and lots of help freely available to you.

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

;o)

2 mod_perl Installation

2.1 What we will learn in this chapter

- mod_perl Installation scenario
- The Gory Details
- mod_perl Installation with CPAN.pm's Interactive Shell
- Installation Without Superuser Privileges
- Miscellaneous issues

2.2 mod_perl Installation scenario

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

```
% cd /usr/src
% lwp-download http://www.apache.org/dist/apache_x.x.x.tar.gz
% lwp-download http://perl.apache.org/dist/mod_perl-x.xx.tar.gz
% tar zvxf apache_x.x.x.tar.gz
% tar zvxf mod_perl-x.xx.tar.gz
% cd mod_perl-x.xx
% perl Makefile.PL APACHE_SRC=../apache_x.x.x/src \
  DO_HTTPD=1 USE_APACI=1 EVERYTHING=1
% make && make test && make install
% cd ../apache_x.x.x
% make install
```

That's all!

- Of course replace *x.x.x* with the real version numbers of mod_perl and Apache.
- GNU tar utility knows to uncompress as well (with z flag).

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

If you have stumbled upon a problem at any of the above steps, don't despair -- the next section will explain in details each and every step.

2.3 The Gory Details

We saw that the basic mod_perl installation is quite simple and takes about 10 command that can be copied and pasted from these pages. However, sometimes you need to make different optimizations by passing only specific parameters (compared to `EVERYTHING=1`), bundling other components with mod_perl and etc. You may want to build mod_perl as loadable object, that can be upgraded without rebuilding the Apache itself.

To accomplish this you will want to understand various techniques for mod_perl configuration and building. You need to know what configuration parameters are available and when each of them should be used.

We can clearly separate the installation process into the following stages: Sources Configuration, Building, Testing and Installation itself.

2.3.1 Sources Configuration (*perl Makefile.PL ...*)

Before building and installing mod_perl you have to configure it. You configure mod_perl as any other Perl module:

```
% perl Makefile.PL [parameters]
```

In this section we will go through most of the parameters mod_perl can accept and explain each one of them.

2.3.1.1 Configuration parameters

perl Makefile.PL accepts various parameters. In this section we will learn what are they and when should they be used.

- **APACHE_SRC**

You will be asked the following question during the configuration stage:

```
"Configure mod_perl with ../apache_xxx/src ?"
```

APACHE_SRC should be used to define the Apache's source tree directory. For example:

```
APACHE_SRC=../apache-x.x.x/src
```

Unless APACHE_SRC specified, *Makefile.PL* makes an intelligent guess by looking at the directories at the same level as mod_perl sources and suggests a directory with the highest version of Apache found there.

Answering 'y' confirms either *Makefile.PL*'s guess about the location of the tree, or the directory you have specified with APACHE_SRC.

If you use DO_HTTPD=1 or NO_HTTPD -- the first apache source tree found or the one you have defined will be used for the rest of the build process.

- **DO_HTTPD, NO_HTTPD, PREP_HTTPD**

Unless any of DO_HTTPD, NO_HTTPD or PREP_HTTPD used you will be prompted by the following question:

```
"Shall I build httpd in ../apache-x.x.x/src for you?"
```

Answering 'y' will make sure an httpd binary will be built in `../apache-x.x.x/src` when running `make`.

To avoid this prompt when the answer is *Yes* use:

```
DO_HTTPD=1
```

Note that if you set `DO_HTTPD=1`, but not used `APACHE_SRC=../apache-x.x.x/src` -- the first apache source tree found will be used to configure and build against.

`PREP_HTTPD=1` just means default 'n' to the second prompt -- meaning, *do not build httpd (make) in the apache source tree*. But it still will ask you about Apache's source location even if you have used the `APACHE_SRC` parameter. Providing the `APACHE_SRC` parameter will just save `perl Makefile.PL` a need to make a guess.

To avoid the two prompts and avoid building httpd, use:

```
NO_HTTPD=1
```

If you choose not to build the binary, you will have to do that manually. We will talk about it later. In any case, you need to run `make install` in the `mod_perl` source tree, so the perl side of `mod_perl` will be installed. Certainly, `make test` wouldn't work until before you get the server built.

● Callback Hooks

By default, all callback hooks except for `PerlHandler` are turned off. You may edit `src/modules/perl/Makefile`, or enable when running `perl Makefile.PL`.

Possible parameters are:

```
PERL_POST_READ_REQUEST
PERL_TRANS
PERL_INIT

PERL_HEADER_PARSER
PERL_AUTHEN
PERL_AUTHZ
PERL_ACCESS
PERL_TYPE
PERL_FIXUP
PERL_LOG
PERL_CLEANUP
PERL_CHILD_INIT
PERL_CHILD_EXIT
PERL_DISPATCH

PERL_STACKED_HANDLERS
PERL_METHOD_HANDLERS
PERL_SECTIONS
PERL_SSI
```

As with any parameters that are either defined or not, use `foo=1` to enable them (e.g. `PERL_AUTHEN=1`).

To enable all callback hooks use:

```
ALL_HOOKS=1
```

- **EVERYTHING**

To enable all possible hooks, set:

```
EVERYTHING=1
```

- **APACHE_PREFIX**

If you want to use a non-default Apache installation prefix, use `APACHE_PREFIX` parameter, e.g.:

```
% perl Makefile.PL APACHE_PREFIX=/usr/local/ [...]
```

- **APACI_ARGS**

When you use `<USE_APACI=1>` parameter, you can tell the `perl Makefile.PL` to pass any arguments you want to the Apache's `./configure` utility, e.g:

```
% perl Makefile.PL USE_APACI=1 \  
APACI_ARGS=--sbindir=/usr/local/sbin/httpd_perl, \  
            --sysconfdir=/usr/local/etc/httpd_perl, \  
            --localstatedir=/usr/local/var/httpd_perl, \  
            --runtimedir=/usr/local/var/httpd_perl/run, \  
            --logfiledir=/usr/local/var/httpd_perl/logs, \  
            --proxycachedir=/usr/local/var/httpd_perl/proxy
```

Notice that **all** `APACI_ARGS` (above) must be passed as one long line if you work with `t?csh!!!` However it works correctly the way it shown above with `(ba)?sh` (by breaking the long lines with `'\``). If you work with `t?csh` it does not work, since `t?csh` passes `APACI_ARGS` arguments to `./configure` by keeping the new lines untouched, but stripping the original `'\``, which makes the all the arguments but the first one, ignored by the configuration process.

2.3.1.2 Reusing Configuration Parameters

It's quite hard to remember what parameters were used in `mod_perl` build, when you have to upgrade the server. So it's better to save them into a file. For example if you create a file at `~/mod_perl_build_options`, with contents:

```
APACHE_SRC=../apache_x.x.x/src DO_HTTPD=1 USE_APACI=1 \  
EVERYTHING=1
```

You can build the server with the following command:

```
% perl Makefile.PL `cat ~/.mod_perl_build_options`
% make && make test && make install
```

But wait, `mod_perl` has a standard method to perform the above trick. If a file name `makepl_args.mod_perl` is found in the same directory as the `mod_perl` build location with any of these options, it will be read in by `Makefile.PL`.

```
% ls -l /usr/src
apache_x.x.x/
makepl_args.mod_perl
mod_perl-x.xx/

% cat makepl_args.mod_perl
APACHE_SRC=../apache_x.x.x/src DO_HTTPD=1 USE_APACI=1 \
EVERYTHING=1

% cd mod_perl-x.xx
% perl Makefile.PL
% make && make test && make install
```

Now the parameters from `makepl_args.mod_perl` file will be used, as if they were directly typed in.

There is a sample `makepl_args.mod_perl` in the `eg/` directory of `mod_perl` distribution package, in which you might find a few options to enable experimental features to play with too!

But if you have found yourself with a compiled `mod_perl` and no traces of the specified parameters left, usually you can still find them out, if the sources were not `make clean`'d. You will find the Apache specific parameters in `apache_x.x.x/config.status` and `mod_perl`'s at in `mod_perl_x.xx/apaci/mod_perl.config`.

2.3.2 *mod_perl Building (make)*

After configuration completion you build the server, by calling:

```
% make
```

which compiles the source files and creates an `httpd` binary or/and a separate library for each module, which can be loaded at run time or inserted into the `httpd` binary sometime later when the `make` will be called from Apache source directory.

Note: it's important that you don't put the `mod_perl` source tree, inside the Apache's sources subdirectory -- since `Apache::src` seems to not work then!

2.3.3 *Built Server Testing (make test)*

After building the server, it's a good idea to thoroughly test it, by calling:

```
% make test
```

Fortunately `mod_perl` comes with a bunch of tests, which attempt to try to use all the features you asked for at the configuration stage. If any of the test fails, the `make test` stage would fail.

Running `make test` will start a freshly built `httpd` on port 8529 running under the `uid` and `gid` of the `perl Makefile.PL` process, the `httpd` will be terminated when the tests are finished.

Each file in the testing suite generally includes more than one test, but when you do the testing, the program will solely report how many were passed and the total number of tests defined in the test file. However if not all the tests in the file fail you want to know which ones did. To gain this information, you should run the tests in a verbose mode. You can enable this mode by using `TEST_VERBOSE` parameter:

```
% make test TEST_VERBOSE=1
```

To change the default port the testing happens on (8529 as of this writing), do:

```
% perl Makefile.PL PORT=xxxx
```

To simply start the newly built `httpd` run:

```
% make start_httpd
```

To shutdown this `httpd` run:

```
% make kill_httpd
```

NOTE to Ben-SSL users: `httpsd` does not seem to handle `/dev/null` as the location of certain files, you'll have to change these by hand. Tests are run with `SSLDisable` directive.

2.3.3.1 Manual Testing

Tests are invoked by running the `./TEST` script located at `./t` directory. Use `-v` option for verbose tests. You might run an individual test like this:

```
% t/TEST -v modules/file.t
```

or all tests in a test sub-directory:

```
% t/TEST modules
```

`TEST` script worries to start the server before the test is getting executed. If for some reason it fails, use `make start_httpd` to start it explicitly.

2.3.4 Installation (make install)

After testing the server, the last step left is to install it. First install all the perl side files:

```
% make install
```

The go to the Apache source tree and complete the Apache files installation (config files, httpd and other utilities):

```
% cd ../apache_x.x.x
% make install
```

Now the installation should be considered completed. You may configure your server now and start using it.

2.4 mod_perl Installation with CPAN.pm's Interactive Shell

To install mod_perl and all the required packages is much easier with help of CPAN.pm module, which provides you among other features a shell interface to a CPAN repository (CPAN = Comprehensive Perl Archive Network, which is a repository of thousands Perl modules, scripts and documentation. See <http://cpan.org> for more info)

First thing first is to download an Apache source code, unpack it into a directory the name of which you will need very soon.

Now execute:

```
% perl -MCPAN -eshell
```

If it's a first time that you use it, it will ask you about 10 questions to configure the module. It's quite easy to accomplish this task, when following the very helpful hints coming along with the questions. When you done, you will see a cpan prompt:

```
cpan>
```

CPAN will download mod_perl for you, unpack it, will check prerequisites, detect the missing third party modules if any, download and install them. All you need to install mod_perl is to type at the prompt:

```
cpan> install mod_perl
```

You will see (I'll use x.xx instead of real version numbers, since these change very frequently):

```
Running make for DOUGM/mod_perl-x.xx.tar.gz
Fetching with LWP:
http://www.perl.com/CPAN-local/authors/id/DOUGM/mod\_perl-x.xx.tar.gz

CPAN.pm: Going to build DOUGM/mod_perl-x.xx.tar.gz

Enter 'q' to stop search
Please tell me where I can find your apache src
[../apache-x.x.x/src]
```

It will search for a latest apache sources and suggest a directory. Here you need to type in the directory you have unpacked the apache in unless it CPAN detected and suggested the right directory... The next question is about the src directory which resides at the root level of the unpacked Apache distribution. In most cases CPAN would "guess" the correct directory.

```
Please tell me where I can find your apache src
[../apache-x.x.x/src]
```

Answer yes to all the following questions, unless you have a reason not to do that.

```
Configure mod_perl with /usr/src/apache_x.x.x/src ? [y]
Shall I build httpd in /usr/src/apache_x.x.x/src for you? [y]
```

Now it will build the apache with enabled mod_perl. The only thing left to do is to go to apache sources root directory (when you quit CPAN shell or use using another terminal) and run:

```
% make install
```

which will complete the installation by installing Apache headers and the binary at the appropriate directories.

The only caveat of described process is that you don't have a control over a configuration process. Actually, it's an easy to solve problem -- you can tell <CPAN.pm> to pass whatever parameters you want to perl Makefile.PL. You do this with `o conf makepl_arg` command:

```
cpan> o conf makepl_arg 'DO_HTTPD=1 USE_APACI=1 EVERYTHING=1'
```

You just enlist all the parameters like you were to pass to a familiar perl Makefile.PL. If you add `APACHE_SRC=/usr/src/apache_x.x.x/src` and `DO_HTTPD=1` parameters, you will be not asked a single question. Of course use a correct path to the apache source distribution.

Now proceed with `install mod_perl`, like before. When the installation is completed, remember to unset the `makepl_arg` variable, by executing:

```
cpan> o conf makepl_arg ''
```

In case you have the `makepl_arg` previously (before you altered it for a mod_perl installation) set to some value, you will probably want to save it somewhere, and restore when you done with mod_perl installation. To read the original value, use:

```
cpan> o conf makepl_arg
```

You can install all the modules you might want to use with mod_perl. You install them all by typing a single command:

```
cpan> install Bundle::Apache
```

It'll install mod_perl if isn't yet installed and many other packages like: `ExtUtils::Embed`, `MIME::Base64`, `URI::URL`, `Digest::MD5`, `Net::FTP`, `LWP`, `HTML::TreeBuilder`, `CGI`, `Devel::Symdump`, `Apache::DB`, `Tie::IxHash`, `Data::Dumper` and etc.

A helpful hint: If you have a system with all the perl modules you use and you want to replicate them all at some other place, and if you cannot just copy the whole `/usr/lib/perl5` directory because of a possible binary incompatibility of the other system, making your own bundle comes as a handy solution. To accomplish that the command `autobundle` can be used on the CPAN shell command line. This command writes a bundle definition file for all modules that are installed for the currently running perl

interpreter.

With a clever bundle file you can then simply say

```
cpan> install Bundle::my_bundle
```

then answer a few questions and then go out for a coffee.

2.5 Installation Without Superuser Privileges

As you have already learned, mod_perl enabled Apache consists of two main components: perl modules and Apache itself. Let's tackle each task at a time.

I'll show a complete installation example using a *stas* as a username, and assume that */home/stas* is a home directory of that user.

2.5.1 Installing Perl Modules into a Directory of Choice

Since without a superuser permissions you aren't allowed to install modules into a system directories like */usr/lib/perl5*, you need to find out how to install the modules under your home directory. The task is a very one.

First you have to decide where the modules to be installed. The simplest approach is to simulate a relevant to perl portion of the / file system, under your home directory. Actually we need only two directories:

```
/home/stas/bin  
/home/stas/lib
```

But we don't have to create them, since it'll be done automatically when the first module will be installed. 99% of the files will go into the *lib* directory, occasionally when some module comes with perl scripts, these will go into a *bin* directory, and the directory itself will be created if it wasn't there before.

Let's install a *CGI.pm* package, which among *CGI.pm* includes a few other *CGI::** modules. As usually, download the package from CPAN repository, unpack it and *chdir* to the created directory.

Now we do a standard `perl Makefile.PL` to prepare a *Makefile*, but this time we tell the *MakeMaker* to use non-default perl installation directories.

```
% perl Makefile.PL PREFIX=/home/stas
```

`PREFIX=/home/stas` is the only different part of the standard perl modules installation process. Note that if you don't like how *MakeMaker* choose to select the rest of the directories or if you are using an older version of it, which requires an explicit declaration of all target directories you should do:

```
% perl Makefile.PL PREFIX=/home/stas \
INSTALLPRIVLIB=/home/stas/lib/perl5 \
INSTALLSCRIPT=/home/stas/bin \
INSTALLSITELIB=/home/stas/lib/perl5/site_perl \
INSTALLBIN=/home/stas/bin \
INSTALLMAN1DIR=/home/stas/lib/perl5/man \
INSTALLMAN3DIR=/home/stas/lib/perl5/man3
```

The rest is as usual:

```
% make
% make test
% make install
```

We see that `make install` installs all the files in my private repository. Note that all the missing directories are created automatically, so there is no need to create them in first place. Here is what it does (this is a slightly truncated output):

```
Installing /home/stas/lib/perl5/CGI/Cookie.pm
Installing /home/stas/lib/perl5/CGI.pm
Installing /home/stas/lib/perl5/man3/CGI.3
Installing /home/stas/lib/perl5/man3/CGI::Cookie.3
Writing /home/stas/lib/perl5/auto/CGI/.packlist
Appending installation info to /home/stas/lib/perl5/perllocal.pod
```

If you have to use the explicit target parameters, instead of a single `PREFIX` parameter, you will find it useful to create a file called for example `~/.perl_dirs` (where `~` is `/home/stas` in our example) and to populate it with:

```
PREFIX=/home/stas \
INSTALLPRIVLIB=/home/stas/lib/perl5 \
INSTALLSCRIPT=/home/stas/bin \
INSTALLSITELIB=/home/stas/lib/perl5/site_perl \
INSTALLBIN=/home/stas/bin \
INSTALLMAN1DIR=/home/stas/lib/perl5/man \
INSTALLMAN3DIR=/home/stas/lib/perl5/man3
```

From now on any time you want to install perl modules locally you simply execute:

```
% perl Makefile.PL `cat ~/.perl_dirs`
% make
% make test
% make install
```

Using the last tip, you can easily maintain several Perl module repositories, for example one for production perl and another for development. When the only difference is either you call:

```
% perl Makefile.PL `cat ~/.perl_dirs.production`
```

or

```
% perl Makefile.PL `cat ~/.perl_dirs.develop`
```

2.5.2 Making Your Scripts Find the Locally Installed Modules

Perl modules are generally being dispatched into a five main directories. You find out these directories, execute:

```
% perl -V
```

and in the generated output, among other important information about your perl installation you will see at the end:

```
Characteristics of this binary (from libperl):
Built under linux
Compiled at Apr  6 1999 23:34:07
@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
 .
```

It shows us the content of the @INC perl special variable, which is being used by perl to look for its modules, as an equivalent to a PATH environment variable in Unix shells which is being used to find the binaries to be executed.

Of course this is the information of the 5.00503 version of perl installed on my x86 architecture PC running Linux. That's why you see *i386-linux* and 5.00503. If your system runs a different operating system, processor or chipset architecture and version of perl, directories would have a different names.

I also have a perl-5.00561 installed under /usr/local/lib/ so when I do:

```
% /usr/local/bin/perl5.00561 -V
```

I see:

```
@INC:
 /usr/local/lib/perl5/5.00561/i586-linux
 /usr/local/lib/perl5/5.00561
 /usr/local/lib/site_perl/5.00561/i586-linux
 /usr/local/lib/site_perl
```

Notice, that it's still *linux* but a newer perl version uses a version of my Pentium processor (thus the *i586* and not *i386* as it was before), which makes a use of compiler optimization for a Pentium processors, when the binary perl extensions are being created.

i386-linux like directories are the ones, where all the platform specific files are supposed to go, such as compiled C files glued to Perl with XS or SWIG.

The above discussion is important to us, because since we have installed the perl modules into a non-standard directories, somehow we have to make Perl know where to look for the four directories. There are two ways to accomplish this task. You should either set the `PERL5LIB` environment variable or modify the `@INC` variable in yours scripts.

Assuming that we use perl-5.00503, in our example the directories are:

```
/home/sbekman/lib/perl5/5.00503/i386-linux
/home/sbekman/lib/perl5/5.00503
/home/sbekman/lib/perl5/site_perl/5.005/i386-linux
/home/sbekman/lib/perl5/site_perl/5.005
```

As I've mentioned it before, you find out the exact directories by executing `perl -V` and replacing the global's perl installation's base directory with your home directory.

Modifying `@INC` is quite easy. The best approach is to use `lib` module, by adding the following snippet at the top of all your scripts that require the locally installed modules.

```
use lib qw(/home/stas/lib/perl5/5.00503/
           /home/stas/lib/perl5/site_perl/5.005);
```

Another way is to explicitly write the code to alter `@INC`:

```
BEGIN {
  unshift @INC,
    qw(/home/stas/lib/perl5/5.00503
       /home/stas/lib/perl5/5.00503/i386-linux
       /home/stas/lib/perl5/site_perl/5.005
       /home/stas/lib/perl5/site_perl/5.005/i386-linux);
}
```

Notice, that with `lib` module, we don't have to enlist the corresponding architecture specific directories, since it adds them automatically if they are exist (well, to be exact, when `$dir/$archname/auto` directory exists).

Also, notice that both approaches prepend the directories to be searched to `@INC`, which allows you to install a more recent module into your local repository and perl will use it instead of the older one installed in the main system repository.

Both approaches, modify the value of `@INC` at the compilation time, `lib` module uses the `BEGIN` block as well, but internally.

Now, let's assume the following scenario. I have installed LWP package in my local repository. Now I want to install another module (e.g. `mod_perl`) and it has LWP listed in its prerequisites list. I know that I've LWP installed, but when I run `perl Makefile.PL` for the module I'm about to install, I'm being told that I don't have LWP installed.

If we think for a moment, there is no way for Perl to know that we have some locally installed modules. All it does, is searching the directories listed in `@INC` and since the latter contains only the default five directories, no wonder it cannot find locally installed LWP package. There is no script we could add the `@INC` modification code, but there is a `PERL5LIB` variable that I've mentioned before, that solves this

problem. If you are using a `tcsh` for interactive work, do:

```
setenv PERL5LIB /home/stas/lib/perl5/5.00503:
/home/stas/lib/perl5/site_perl/5.005
```

It should be a single line with directories separated by colons (`:`) and no spaces. If you are a `bash` user, do:

```
export PERL5LIB=/home/stas/lib/perl5/5.00503:
/home/stas/lib/perl5/site_perl/5.005
```

Again make it a single line. Actually `bash` allows to have a multi-line settings with help of backslash (`\`). So you can set it this way:

```
export PERL5LIB=/home/stas/lib/perl5/5.00503:\
/home/stas/lib/perl5/site_perl/5.005
```

As with `use lib, perl` automatically prepends the architecture specific directories to `@INC` if those exist.

When you have done with this setting, verify the value of the newly configured `@INC`, by executing `perl -V` as before. Now you should see the modified value of `@INC`:

```
% perl -V

Characteristics of this binary (from libperl):
Built under linux
Compiled at Apr  6 1999 23:34:07
%ENV:
  PERL5LIB="/home/stas/lib/perl5/5.00503:/home/stas/lib/perl5/site_perl/5.005"
@INC:
  /home/stas/lib/perl5/5.00503/i386-linux
  /home/stas/lib/perl5/5.00503
  /home/stas/lib/perl5/site_perl/5.005/i386-linux
  /home/stas/lib/perl5/site_perl/5.005
  /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
.
```

The moment everything works as you want it to, add this setting into a `.tcshrc` or `.bashrc` file, according to the interactive shell you use, so the next time you open a new shell, this setting would be already in place.

Note that if you have a `PERL5LIB` setting, you don't need to alter the `@INC` value in your scripts, only if you are executing them from the interactive shell or in any other way that sets the `PERL5LIB` variable. For example, if someone else tries to execute your scripts but doesn't have this setting in the shell she attempts to execute the script from, Perl will fail to find your locally installed modules.

So the best approach is to have both: the `PERL5LIB` environment variable and the explicit `@INC` extension code at the beginning of the scripts as described before.

2.5.3 Making a Local Apache Installation

Just like with perl modules, when you don't have permissions to install files into a system area, you have to install them locally under your home directory. It's almost the same as a plain installation, but you will have to run the server listening to port number > 1024, since these are the ports only root processes can listen to.

Another important issue you would have to solve is how to add an automatic startup and shutdown scripts to the directories use by the rest of the system services. You will have to ask your system administrator to assist you with this issue.

Now to install Apache locally, all you have to do is to tell a `.configure` script in the Apache source directory what target directories to be used. If following a convention that I use, which makes your home directory looking like the `/` (base) directory, the invocation parameters would be:

```
./configure --prefix=/home/stas
```

Apache will use the prefix for the rest of its target directories instead of the default `/usr/local/apache`. If you want to see what are they, before you proceed, add the `--show-layout` option:

```
./configure --prefix=/home/stas --show-layout
```

You might want to put all the Apache files under `/home/stas/apache` following the Apache's defaults convention. To accomplish that do:

```
./configure --prefix=/home/stas/apache
```

If you want to modify some or all of the automatically created names of directories, when you omit their explicit parameters, just set them to the desired values, e.g:

```
./configure --prefix=/home/stas/apache \  
--sbindir=/home/stas/apache/sbin \  
--sysconfdir=/home/stas/apache/etc \  
--localstatedir=/home/stas/apache/var \  
--runtimedir=/home/stas/apache/var/run \  
--logfiledir=/home/stas/apache/var/logs \  
--proxycachedir=/home/stas/apache/var/proxy
```

That's all!

Also remember that you can start the script only under a user and group you belong to. Set the appropriate `User` and `Group` directives in the `httpd.conf` to correct values.

2.5.4 Actual Local `mod_perl` Enabled Apache Installation

Now when we have learned how to install perl modules and Apache locally, let's see how we use the acquired knowledge to install `mod_perl` enabled Apache in our home directory. It's almost as simple as doing each one at separate, but a single nuance you should know about and I'll mention it at the end of this section.

So if you have unpacked Apache and mod_perl sources under the /home/stas/src directory and they look like:

```
% ls /home/stas/src
/home/stas/src/apache_x.x.x
/home/stas/src/mod_perl-x.xx
```

where x.xx are the version numbers as usual and you want the perl modules from the mod_perl package to be installed under /home/stas/lib/perl5 and Apache files under /home/stas/apache, the following commands will do that for you.

```
% perl Makefile.PL \
PREFIX=/home/stas \
APACHE_PREFIX=/home/stas/apache \
APACHE_SRC=./apache_x.x.x/src \
DO_HTTPD=1 \
USE_APACI=1 \
EVERYTHING=1
% make && make test && make install
% cd ../apache_x.x.x
% make install
```

If you need something to be passed to .configure script as we have seen in the previous section use the APACI_ARGS parameter, e.g:

```
APACI_ARGS=--sbindir=/home/stas/apache/sbin, \
--sysconfdir=/home/stas/apache/etc, \
--localstatedir=/home/stas/apache/var, \
--runtimedir=/home/stas/apache/var/run, \
--logfiledir=/home/stas/apache/var/logs, \
--proxycachedir=/home/stas/apache/var/proxy
```

Note that the above multiline splitting will work only with bash shell, tcsh users have to list all the parameters in a single line.

Basically the installation is complete. The only nuance is a @INC variable, that wouldn't be correctly set if you rely on the PERL5LIB environment variable, unless you set it explicitly in the startup file, which is get required before any other module that resides in your local repository is being loaded. But a much nicer approach is to use the lib pragma as we saw before, but in a little different way - we use it in the startup file and it affects all the code that will be executed under mod_perl handlers. e.g:

```
PerlRequire /home/stas/apache/perl/startup.pl
```

where startup.pl starts with:

```
use lib qw(/home/stas/lib/perl5/5.00503/
/home/stas/lib/perl5/site_perl/5.005);
```

Note that you can still use the hard-coded @INC modifications in the scripts themselves, but you should know that @INC would be reset to its original value after the scripts would be compiled for the first time and all the hard-coded settings of @INC would be forgot.

That's because scripts modify @INC in BEGIN blocks and mod_perl executes the BEGIN blocks only when it does script compilation, that's why when you execute the script for a second time, @INC would be reset to its original value.

The only place you can alter this "original" value is during the server configuration stage either in the startup file or by setting:

```
PerlSetEnv Perl5LIB /home/stas/lib/perl5/5.00503/:/home/stas/lib/perl5/site_perl/5.005
```

in the httpd.conf.

Now the rest of the mod_perl configuration and using is absolutely the same as if you were installing mod_perl as a super user.

One more important thing to keep in mind is a system resources consuming. mod_perl is memory hungry -- if you run a lot of mod_perl processes on a public, multiuser (not dedicated) machine -- most likely the system administrator of this machine will ask you to use less resources and even to shut down your mod_perl server and to find another home for it. You have a few solutions:

- Reduce resources usage using the modules `Apache::SizeLimit`, `Apache::GTopLimit`.
- Ask your ISP whether they can setup a dedicated machine for you in their computer room, so you will be able to install as much memory as you need and have the ISP to administer the system. But if you get a dedicated machine chances are that you will want to have a root access if you are able to manage the administering your self, keeping on the list of ISP's responsibilities only the following items: keeping a constant electricity supply, making sure that the network link is up, and protecting the machine from possible physical break-ins (when someone breaks into a computer room either to steal the information from your machine, or to damage it physically). Another good idea is to let the ISP to install security patches if you have a trust in them or just incapable of doing that.
- Look for another ISP with lots of resources or one that supports mod_perl. You can find a list of these ISP at <http://perl.apache.org>.

2.6 Miscellaneous issues

2.6.1 *Should I rebuild mod_perl if I have upgraded my perl?*

Yes, you should. You have to rebuild mod_perl enabled server since it has a hard coded @INC which points to the old perl and it is probably linked to the an old `libperl` library. You can try to modify the @INC in the startup script (if you keep the old perl version around), but it is better to build a fresh one to save you a mess.

2.6.2 Should I Build mod_perl with gcc or cc?

Since mod_perl includes C code, to make it binary compatible with Perl, on most systems the same compiler should be used as the one Perl was built with. So if your Perl was built with `gcc`, it will pick the same compiler when you do `perl Makefile.PL` To find out which compiler it was built with, run `perl -V` at the command prompt.

Sometimes Perl's configuration will choose one compiler, e.g. `cc`, but Apache's configuration chooses a different one, e.g. `gcc`. If you run into this problem, consult Perl's and Apache's *INSTALL* documents on how to ensure both are built with the same compiler.

;o)

3 mod_perl Configuration

3.1 What we will learn in this chapter

- Apache Configuration
- mod_perl Configuration
- Start-up File
- <Perl>...</Perl> Sections
- Miscellaneous issues

3.2 Server Configuration

The next step after building and installing your new mod_perl enabled Apache server, is to configure the server. The configuration process consists of two parts: Apache and mod_perl specific directives configuration.

Prior to version 1.3.4, the default Apache install used three configuration files -- *httpd.conf*, *srm.conf*, and *access.conf*. The 1.3.4 version began distributing the configuration directives in a single file -- *httpd.conf*. The tutorial uses the *httpd.conf* in its examples.

So as you have already understood, the only file that you should edit is *httpd.conf* that by default is put into a *conf* directory under the document root. The document root is the directory that you choose for Apache installation or the default one, which is */usr/local/apache/* on many UNIX platforms.

3.3 Apache Configuration

To minimize the number of things that can go wrong, it can be a good idea to configure the Apache itself first (like there is no mod_perl at all) and make sure that it works.

Apache distribution comes with an extensive configuration manual and in addition each section of the configuration file includes helpful comments explaining how every directive should be configured and what are the defaults values.

3.3.1 Configuration Directives

If you didn't move Apache directories around, the installation program already has configured everything for you. Just start the server and test it working. To start the server use the `apachectl` utility which comes bundled with Apache distribution and resides in the same directory with `httpd` (the Apache server itself). Go to this directory and execute:

```
/usr/local/apache/bin/apachectl start
```

Now you can test the server, by trying to access it from <http://localhost> .

For a basic setup there are just a few things to configure. If you have moved directories you have to update them in *httpd.conf*. There are many of them, listing just a few of them:

```
ServerRoot    "/usr/local/apache"  
DocumentRoot "/home/httpd/docs"
```

You should set a name of your machine as it's known to the external world if it's not a testing machine and referring to it as `localhost` isn't good enough for you.

```
ServerName www.example.com
```

If you want to run it on a different from port 80, edit the `Port` directive.

```
Port 8080
```

You might want to change the user and group names the server will run under. Note that if started as *root* user (which is generally the case), the parent process will continue to run as *root*, but children will run as the user and group you have specified. For example:

```
User nobody  
Group nobody
```

There are other directives that you might need to configure as well, as mentioned earlier you will find them all in *httpd.conf*.

After single valued directives come the `Directory` and `Location` sections of configuration. That's the place where for each directory and location you supply its unique behaviour, that applies to every request that happens to fall into its domain.

3.4 mod_perl Configuration

When you have tested that the Apache server works on your machine, it's a time to step forward and configure the `mod_perl` side. Part of the configuration directives are already familiar to you, however `mod_perl` introduces a few new ones.

It can be a good idea to keep all the `mod_perl` stuff at the end of the file, after the native Apache configurations.

3.4.1 Alias Configurations

First, you need to specify the locations on a file-system for the scripts to be found.

Add the following configuration directives:

```
# for plain cgi-bin:
ScriptAlias /cgi-bin/ /usr/local/myproject/cgi/

# for Apache::Registry mode
Alias /perl/ /usr/local/myproject/cgi/

# Apache::PerlRun mode
Alias /cgi-perl/ /usr/local/myproject/cgi/
```

Alias provides a mapping of URL to file system object under mod_perl. ScriptAlias is being used for mod_cgi.

Alias defines the start of the URL path to the script you are referencing. For example, using the above configuration, fetching *http://www.nowhere.com/perl/test.pl*, will cause the server to look for the file *test.pl* at */usr/local/myproject/cgi*, and execute it as an `Apache::Registry` script if we define `Apache::Registry` to be the handler of */perl* location (see below).

The URL *http://www.nowhere.com/perl/test.pl* will be mapped to */usr/local/myproject/cgi/test.pl*. This means that you can have all your CGIs located at the same place in the file-system, and call the script in any of three modes simply by changing the directory name component of the URL (*cgi-bin/perl/cgi-perl*) - is not this neat? (That is the configuration you see above - all three Aliases point to the same directory within your file system, but of course they can be different).

If your script does not seem to be working while running under mod_perl, you can easily call the script in straight mod_cgi mode without making any script changes (in most cases), but rather by changing the URL you invoke it by.

ScriptAlias is actually:

```
Alias /foo/ /path/to/foo/
SetHandler cgi-handler
```

where `SetHandler cgi-handler` invokes mod_cgi. The latter will be overwritten if you enable `Apache::Registry`. In other words, ScriptAlias does not work for mod_perl, it only appears to work when the additional configuration is in there. If the `Apache::Registry` configuration came before the ScriptAlias, scripts would be run under mod_cgi. While handy, ScriptAlias is a known kludge--it's always better to use Alias and SetHandler.

Of course you can choose any other alias (will be used later in configuration). All three modes or part of them can be used. But you should remember that it is undesirable to run scripts in plain mod_cgi from a mod_perl-enabled server--the price is too high, it is better to run these on plain Apache server.

3.4.2 <Location> Configuration

As we know <Location> section assigns a number of rules the server should follow when the request's URI matches the Location domain. It's widely accepted to use */perl* as a base URI of the perl scripts running under mod_perl, like */cgi-bin* for mod_cgi. Let's explain the following very widely used <Location> section:

```
<Location /perl>
  SetHandler perl-script
  PerlHandler Apache::Registry
  Options ExecCGI
  allow from all
  PerlSendHeader On
</Location>
```

This configuration causes all requests' URI starting with */perl* to be handled by `mod_perl` Apache module with handler from the `Apache::Registry` Perl module. Let's go through the directives inside the `<Location>` section in the example:

```
<Location /perl>
```

Remember the **Alias** from the above section? We use the same `Alias` here, if you use `Location` that does not have the same `Alias`, the server will fail to locate the script in the file system. You needed the `Alias` setting only if the code that should be executed is located in the file. So `Alias` just provides the URI to filepath translation rule.

Sometimes there is no script to be executed. Instead there is some module that its method is being executed, similar to */perl-status*, where the code is stored in Apache module. If this is the case, we don't need `Alias` settings for such a `<Location>`.

```
  SetHandler perl-script
```

assigns `mod_perl` Apache module to handle the content generation phase.

```
  PerlHandler Apache::Registry
```

tells Apache to use `Apache::Registry` Perl module for the actual content generation.

```
  Options ExecCGI
```

`Options` directive accepts a few different parameters (options), the `ExecCGI` option tells the server that the file is a program and should be executed, instead of just displayed like plain html file. If you omit this option depending on clients configuration, the script will either be rendered as a plain text or trigger a *Save-As* dialog.

```
  allow from all
```

This directive is in charge of access control based on domain. The above settings allows to run the script for client from any domain.

```
  PerlSendHeader On
```

`PerlSendHeader On` tells the server to send an HTTP header to the browser on every script invocation. You will want to turn this off for `nph` (non-parsed-headers) scripts.

`PerlSendHeader On` setting invokes `ap_send_http_header()` after parsing your script headers. It is only meant for CGI emulation, it's always better to use `CGI->header` from `CGI.pm` module or `$r->send_http_header` directly to send the HTTP header.

```
</Location>
```

closes the `<Location>` section definition.

Note that sometimes you will have to preload the module before using it in the `<Location>` section, in case of `Apache::Registry` the configuration will look like this:

```
PerlModule Apache::Registry
<Location /perl>
  SetHandler perl-script
  PerlHandler Apache::Registry
  Options ExecCGI
  allow from all
  PerlSendHeader On
</Location>
```

`PerlModule` is equal to Perl's native `use()` function call.

You have nothing to do about `/cgi-bin` location (`mod_cgi`), since it has nothing to do with `mod_perl`.

Let's see another very similar example with `Apache::PerlRun`.

```
<Location /cgi-perl>
  SetHandler perl-script
  PerlHandler Apache::PerlRun
  Options ExecCGI
  allow from all
  PerlSendHeader On
</Location>
```

The only difference from the `Apache::Registry` configuration is the argument of the `PerlHandler` directive, where `Apache::Registry` was replaced by `Apache::PerlRun`.

3.4.3 *PerlModule and PerlRequire Directives*

As we saw earlier the module should be loaded before allowed to be used, the `PerlModule` and `PerlRequire` are the two `mod_perl` directives equivalent to the Perl's `use()` and `require()` respectively. Since they are equivalent, the same rules apply to their arguments. You pass `Apache::DBI` as an argument for `PerlModule`, and `Apache/DBI.pm` for `PerlRequire`.

You may load modules from the config file at server startup via:

```
PerlModule Apache::DBI CGI DBD::Mysql
```

Generally the modules are preloaded from the startup script, usually named `startup.pl`. This is a file with plain perl code which is executed through the `PerlRequire` directive. For example:

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

As with any file with Perl code that gets `require()`'d--it must return a *true* value. To ensure that this happens don't forget to add `1;` at the end of file.

3.4.4 Perl*Handlers

As you know Apache specifies about 11 phases of the request loop, namely in that order: Post-Read-Request, URI Translation, Header Parsing, Access Control, Authentication, Authorization, MIME type checking, FixUp, Response (Content phase). Logging and finally Cleanup. These are the stages of a request where the Apache API allows a module to step in and do something. There is a dedicated PerlHandler for each of these stages. Namely:

```
PerlChildInitHandler
PerlPostReadRequestHandler
PerlInitHandler
PerlTransHandler
PerlHeaderParserHandler
PerlAccessHandler
PerlAuthenHandler
PerlAuthzHandler
PerlTypeHandler
PerlFixupHandler
PerlHandler
PerlLogHandler
PerlCleanupHandler
PerlChildExitHandler
```

The first 4 handlers cannot be used in the `<Location>`, `<Directory>`, `<Files>` and `.htaccess` file, the main reason is all the above require a known path to the file in order to bind a requested path with one or more of the identifiers above. Starting from `PerlHeaderParserHandler` (5th) URI is already being mapped to a physical pathname, thus can be used to match the `<Location>`, `<Directory>` or `<Files>` configuration section, or to look at `.htaccess` file if exists at the specified directory in the translated path.

The Apache documentation (or even better -- the “Writing Apache Modules with Perl and C” book by Doug MacEachern and Lincoln Stein) will tell you all about those stages and what your modules can do. By default, these hooks are disabled at compile time, see the `INSTALL` document for information on enabling these hooks.

Note that by default Perl API expects a subroutine called `handler` to handle the request in the registered PerlHandler module. Thus if your module implements this subroutine, you can register the handler as simple as writing:

```
Perl*Handler Apache::SomeModule
```

replace *Perl*Handler* with a wanted name of the handler. `mod_perl` will preload the specified module for you. But if you decide to give the handler code a different name, like `my_handler`, you must preload the module and to write explicitly the chosen name.

```
PerlModule Apache::SomeModule
Perl*Handler Apache::SomeModule::my_handler
```

Please note that the former approach will not preload the module at the startup, so either explicitly preload it with `PerlModule` directive, add it to the startup file or use a nice shortcut the `Perl*Handler` syntax suggests:

```
Perl*Handler +Apache::SomeModule
```

Notice the leading `+` character. It's equal to:

```
PerlModule Apache::SomeModule
Perl*Handler Apache::SomeModule
```

If a module wishes to know what handler is currently being run, it can find out with the `current_callback` method. This method is most useful to *PerlDispatchHandlers* who wish to only take action for certain phases.

```
if($r->current_callback eq "PerlLogHandler") {
    $r->warn("Logging request");
}
```

3.4.5 Stacked Handlers

With the `mod_perl` stacked handlers mechanism, it is possible for more than one `Perl*Handler` to be defined and run during each stage of a request.

`Perl*Handler` directives can define any number of subroutines, e.g. (in config files)

```
PerlTransHandler OneTrans TwoTrans RedTrans BlueTrans
```

With the method, `Apache->push_handlers()`, callbacks can be added to the stack by scripts at runtime by `mod_perl` scripts.

`Apache->push_handlers()` takes the callback hook name as its first argument and a subroutine name or reference as its second. e.g.:

```
Apache->push_handlers("PerlLogHandler", \&first_one);

$r->push_handlers("PerlLogHandler", sub {
    print STDERR "__ANON__ called\n";
    return 0;
});
```

After each request, this stack is cleared out.

All handlers will be called unless a handler returns a status other than `OK` or `DECLINED`.

example uses:

CGI.pm maintains a global object for its plain function interface. Since the object is global, it does not go out of scope, DESTROY is never called. CGI->new can call:

```
Apache->push_handlers("PerlCleanupHandler", \&CGI::_reset_globals);
```

This function will be called during the final stage of a request, refreshing CGI.pm's globals before the next request comes in.

Apache::DCELogin establishes a DCE login context which must exist for the lifetime of a request, so the DCE::Login object is stored in a global variable. Without stacked handlers, users must set

```
PerlCleanupHandler Apache::DCELogin::purge
```

in the configuration files to destroy the context. This is not "user-friendly". Now, Apache::DCELogin::handler can call:

```
Apache->push_handlers("PerlCleanupHandler", \&purge);
```

Persistent database connection modules such as Apache::DBI could push a PerlCleanupHandler handler that iterates over %Connected, refreshing connections or just checking that ones have not gone stale. Remember, by the time we get to PerlCleanupHandler, the client has what it wants and has gone away, we can spend as much time as we want here without slowing down response time to the client (but the process is unavailable for serving new request before the operation is completed).

PerlTransHandlers may decide, based on URI or other condition, whether or not to handle a request, e.g. Apache::MsqlProxy. Without stacked handlers, users must configure:

```
PerlTransHandler Apache::MsqlProxy::translate
PerlHandler      Apache::MsqlProxy
```

PerlHandler is never actually invoked unless translate() sees the request is a proxy request (\$r->proxyreq), if it is a proxy request, translate() sets \$r->handler("perl-script"), only then will PerlHandler handle the request. Now, users do not have to specify PerlHandler Apache::MsqlProxy, the translate() function can set it with push_handlers().

Includes, footers, headers, etc., piecing together a document, imagine (no need for SSI parsing!):

```
PerlHandler My::Header Some::Body A::Footer
```

A little test:

```
#My.pm
package My;
```

```

sub header {
    my $r = shift;
    $r->content_type("text/plain");
    $r->send_http_header;
    $r->print("header text\n");
}
sub body { shift->print("body text\n") }
sub footer { shift->print("footer text\n") }
1;
__END__

```

```

#in config
<Location /foo>
SetHandler "perl-script"
PerlHandler My::header My::body My::footer
</Location>

```

Parsing the output of another PerlHandler? this is a little more tricky, but consider:

```

<Location /foo>
    SetHandler "perl-script"
    PerlHandler OutputParser SomeApp
</Location>

<Location /bar>
    SetHandler "perl-script"
    PerlHandler OutputParser AnotherApp
</Location>

```

Now, OutputParser goes first, but it `untie()`'s `*STDOUT` and `re-tie()`'s to its own package like so:

```

package OutputParser;

sub handler {
    my $r = shift;
    untie *STDOUT;
    tie *STDOUT => 'OutputParser', $r;
}

sub TIEHANDLE {
    my($class, $r) = @_;
    bless { r => $r }, $class;
}

sub PRINT {
    my $self = shift;
    for (@_) {
        #do whatever you want to $_
        $self->{r}->print($_ . "[insert stuff]");
    }
}

1;
__END__

```

To build in this feature, configure with:

```
% perl Makefile.PL PERL_STACKED_HANDLERS=1 [PERL_FOO_HOOK=1,etc]
```

Another method `Apache->can_stack_handlers` will return `TRUE` if `mod_perl` was configured with `PERL_STACKED_HANDLERS=1`, `FALSE` otherwise.

3.4.6 *PerlFreshRestart*

To reload `PerlRequire`, `PerlModule`, other `use()`'d modules and flush the `Apache::Registry` cache on server restart, add:

```
PerlFreshRestart On
```

Not all Perl modules can stand the reload, that's why it's better to avoid enabling this directive.

3.4.7 *PerlSetVar, PerlSetEnv and PerlPassEnv*

```
PerlSetEnv key val
PerlPassEnv key
```

`PerlPassEnv` passes, `PerlSetEnv` sets and passes the *ENV* variables to your scripts. you can access them in your scripts through `%ENV` (e.g. `$ENV{"key"}`).

Regarding the setting of `PerlPassEnv PERL5LIB` in `httpd.conf` If you turn on taint checks (`PerlTaintMode On`), `$ENV{PERL5LIB}` will be ignored (unset).

`PerlSetVar` is very similar to `PerlSetEnv`, but you extract it with another method. In `<Perl>` sections:

```
push @{ $Location{"/"}->{PerlSetVar} }, [ 'FOO' => BAR ];
```

and in the code you read it with:

```
my $r = Apache->request;
print $r->dir_config('FOO');
```

3.4.8 *PerlWarn and PerlTaintCheck*

To enable the warnings and taint mode globally to all server children use:

```
PerlWarn On
PerlTaintCheck On
```

3.5 Start-up File

There is more to do at the server startup, than just preloading files. You might want to initialize RDMS connections, tie read-only dbm files and etc. Startup file is an ideal place to put the code that should be executed when the server starts. Once you prepared the code, load it before the rest of the mod_perl configuration directives with:

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

I must stress that all the code that is run at the server initialization time is run with root privileges if you are executing it as a root user (you have to, unless you choose to run the server on an unprivileged port, above 1024). This means that anyone who has write access to a script or module that is loaded by `PerlModule` or `PerlRequire`, effectively has root access to the system. You might want to take a look at the new and experimental `PerlOpmask` directive and `PERL_OPMASK_DEFAULT` compile time option to try to disable some dangerous operators.

Since the startup file is a file written in plain perl, one can validate its syntax with:

```
% perl -c /home/httpd/perl/lib/startup.pl
```

3.5.1 *The Sample Start-up File*

Let's look at a real world startup file:

```
startup.pl
-----
use strict;

# extend @INC if needed
use lib qw(/dir/foo /dir/bar);

# make sure we are in a sane environment.
$ENV{GATEWAY_INTERFACE} =~ /^CGI-Perl/
  or die "GATEWAY_INTERFACE not Perl!";

# for things in the "/perl" URL
use Apache::Registry;

#load perl modules of your choice here
#this code is interpreted *once* when the server starts
use LWP::UserAgent ();
use Apache::DBI ();
use DBI ();

# tell me more about warnings
use Carp ();
$SIG{__WARN__} = \&Carp::cluck;

# Load CGI.pm and call its compile() method to precompile
# (but not to import) its autoloaded methods.
use CGI ();
CGI->compile(':all');
```

```

# init the connections for each child
Apache::DBI->connect_on_init
( "DBI:mysql:$Match::Config::c{db}{DB_NAME}::$Match::Config::c{db}{SERVER}",
  $Match::Config::c{db}{USER},
  $Match::Config::c{db}{USER_PASSWD},
  {
    PrintError => 1, # warn() on errors
    RaiseError => 0, # don't die on error
    AutoCommit => 1, # commit executes immediately
  }
);

```

Let's go and explain the reasons of my decision to include this specific code in the startup file.

```
use strict;
```

As with every script longer than five lines I have a habit to use this pragma, which saves me from a lot for troubles in a long run. The startup file is not different from any other perl code that I write.

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

The only chance to permanently modify the @INC before the server was started is with this command. Later the running code can modify @INC just for a moment it `require()`'s some file, and than its value gets reset to the previous one.

```

$ENV{GATEWAY_INTERFACE} =~ /^CGI-Perl/
  or die "GATEWAY_INTERFACE not Perl!";

```

A sanity check, if Apache wasn't properly built, the above code will abort the server startup.

```

use Apache::Registry;
use LWP::UserAgent ();
use Apache::DBI ();
use DBI ();

```

Preload the modules that get used by our Perl code serving the requests. Unless you need the symbols (variables and subroutines) exported by the modules you preload to do accomplish something within the startup file, don't import them, since it's just a waste of startup time. Instead use empty list `()` to tell the `import()` function not to import a thing.

```

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

```

This is a useful snippet to enable extended warnings logged in the `error_log` file. In addition to same basic warning, a trace of calls would be added which makes the tracking of the potential problem a much easier task, since you know who called whom. For example, with normal warnings you might see:

```

Use of uninitialized value at
  /usr/lib/perl5/site_perl/5.005/Apache/DBI.pm line 110.

```

but you have no idea where it was called from. When we use the Carp as shown above we might see:

```
Use of uninitialized value at
  /usr/lib/perl5/site_perl/5.005/Apache/DBI.pm line 110.
Apache::DBI::connect(undef, 'mydb::localhost', 'user',
  'passwd', 'HASH(0x87a5108)') called at
  /usr/lib/perl5/site_perl/5.005/i386-linux/DBI.pm line 382
DBI::connect('DBI', 'DBI:mysql:mydb::localhost', 'user',
  'passwd', 'HASH(0x8375e4c)') called at
  /usr/lib/perl5/site_perl/5.005/Apache/DBI.pm line 36
Apache::DBI::__ANON__('Apache=SCALAR(0x87a50c0)') called at
  PerlChildInitHandler subroutine
  'Apache::DBI::__ANON__' line 0
eval {...} called at PerlChildInitHandler subroutine
  'Apache::DBI::__ANON__' line 0
```

we clearly see that the warning was triggered by `eval()`uating the `Apache::DBI::__ANON__` which called `DBI::connect` with the arguments that we see as well, which in turn called `Apache::DBI::connect` method. Now we know where to look for a problem.

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

Some modules get their subroutines created at the run time to improve the loading time. This helps when the module includes many subroutines, but only a few are actually get used. `CGI.pm` falls into this category. Since with `mod_perl` the module is loaded only once, it might be a good idea to precompile all or a part of its methods.

`CGI.pm`'s `compile()` method performs this task. Notice that this is a proprietary function of this module, other modules can implement this feature or not and use this other name for the subroutine. As with all module we preloaded in the startup file, we don't import symbols from them as they are all get lost when they go out of the file's scope.

Note that starting with `$CGI::VERSION 2.46`, the recommended method to precompile the code in `CGI.pm` is:

```
use CGI qw(-compile :all);
```

But the old method is still available for backward compatibility.

3.5.2 What Modules Should You Add to the Start-up File and Why

Every module loaded at the server startup will be shared among server children, saving a lot of RAM for you. Usually I put most of the code I develop into modules and preload them.

You can even preload your CGI script with `Apache::RegistryLoader` and preopen the DB connections with `Apache::DBI`.

3.5.3 *The Confusion with use() at the Server Start-up File*

Some people wonder, why there is a need for a duplication of `use()` clause in startup file and in the script itself. The confusion rises from misunderstanding of the `use()` function. `use()` consists of two other functions, namely `require()` and `import()`, called within a `BEGIN` block.

So, if the module in question imports some symbols into your code's namespace, you have to write the `use()` statement once again in your code. The module will not be loaded once again, only the `import()` call will be called. For example in the startup file you write:

```
use CGI ();
```

since you probably don't need any symbols to be exported there. But in your code you probably would write:

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

Since the symbols that you might import into a startup's script namespace will be visible by none of the children, scripts that need a `FOO`'s module exported tags have to pull it in like if you did not preload `FOO` at the startup file. For example, just because you have `use()`'d `Apache::Constants` in the startup file, does not mean you can have the following handler:

```
package MyModule;
sub {
    my $r = shift;
    ## Cool stuff goes here
    return OK;
}
1;
```

You would either need to add:

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

Or use a fully qualified notation:

```
return Apache::Constants::OK;
```

If you want to use the function interface without exporting the symbols, use a fully qualified functions, e.g. `CGI::param`. The same rule applies to variables, you can import variables and you can access them by their full name. e.g. `$My::Module::bar`. When you use the object oriented (methods) interface you don't need to export the method symbols as well.

In both later cases technically you aren't required to return on `use()` statement in your code, if it was already loaded at the startup. But you do have to write your code like id there is no code preloaded. Because you or someone else will read your code at some point and will not understand how can you use this or that method, without first loading the module containing it.

Read the `Exporter` and `perlmod` manpages for more information about `import()`.

3.5.4 *The Confusion with Global Variables in Start-up File*

`PerlRequire` allows you to execute code that preloads modules and does more things. Imported or defined variables are visible in the scope of the startup file. It is a wrong assumption that global variables that were defined in the startup file, will be accessible by child processes.

You do have to define/import variables in your scripts and they will be visible inside a child process who run this script. They will be not shared between siblings. Remember that every script is running in a specially (uniquely) named package - so it cannot access variables from other packages unless it inherits from them or `use()`'s them.

3.6 <Perl>...</Perl> Sections

With <Perl>...</Perl> sections, it is possible to configure your server entirely in Perl.

3.6.1 Usage

<Perl> sections can contain *any* and as much Perl code as you wish. These sections are compiled into a special package whose symbol table `mod_perl` can then walk and grind the names and values of Perl variables/structures through the Apache core configuration gears. Most of the configurations directives can be represented as scalars (`$scalar`) or lists (`@list`). A `@List` inside these sections is simply converted into a space delimited string for you inside. Here is an example:

```
httpd.conf
-----
<Perl>
@PerlModule = qw(Mail::Send Devel::Peek);

#run the server as whoever starts it
$User = getpwuid($>) || $>;
$Group = getgrgid($>) || $>;

$ServerAdmin = $User;

</Perl>
```

Block sections such as <Location>..</Location> are represented in a `%Location` hash, e.g.:

```
$Location{"/~doug/"} = {
  AuthUserFile => '/tmp/htpasswd',
  AuthType => 'Basic',
  AuthName => 'test',
  DirectoryIndex => [qw(index.html index.htm)],
  Limit => {
    METHODS => 'GET POST',
    require => 'user dougm',
  },
};
```

If an Apache directive can take two or three arguments you may push strings and the lowest number of arguments will be shifted off the @List or use array reference to handle any number greater than the minimum for that directive:

```
push @Redirect, "/foo", "http://www.foo.com/";;
push @Redirect, "/imdb", "http://www.imdb.com/";;
push @Redirect, [qw(temp "/here" "http://www.there.com");];
```

Other section counterparts include %VirtualHost, %Directory and %Files.

To pass all environment variables to the children with a single configuration directive, rather than listing each one via PassEnv or PerlPassEnv, a <Perl> section could read in a file and:

```
push @PerlPassEnv, [$key => $val];
```

or

```
Apache->httpd_conf("PerlPassEnv $key $val");
```

These are somewhat simple examples, but they should give you the basic idea. You can mix in any Perl code your heart desires. See *eg/httpd.conf.pl* and *eg/perl_sections.txt* in mod_perl distribution for more examples.

A tip for syntax checking outside of httpd:

```
<Perl>
# !perl

#... code here ...

__END__
</Perl>
```

Now you may run:

```
perl -cx httpd.conf
```

3.6.2 Enabling

To enable <Perl> sections you should build mod_perl with perl Makefile.PL PERL_SECTIONS=1.

3.6.3 Verifying

You can watch how have you configured the <Perl> sections through the /perl-status location, by choosing the **Perl Sections** from the menu.

You can dump the configuration by `<Perl>` sections configuration this way:

```
<Perl>
use Apache::PerlSections();
...
# Configuration Perl code here
...
print STDERR Apache::PerlSections->dump();
</Perl>
```

Alternatively you can store it in a file:

```
Apache::PerlSections->store("httpd_config.pl");
```

You can then `require()` that file in some other `<Perl>` section.

3.7 Miscellaneous issues

3.7.1 *Validating the Configuration Syntax*

`apachectl configtest` tests the configuration file without starting the server. You can safely modify the configuration file on your production server, if you run this test before you restart the server. Of course it is not 100% error prone, but it will reveal any syntax errors you might make while editing the file.

'`apachectl configtest`' is the same as '`httpd -t`' and it actually executes the code in `startup.pl`, not just parses it. `<Perl>` configuration has always started Perl during the configuration read, `Perl{Require,Module}` do so as well.

If you want your startup code to get a control over the `-t` (`configtest`) server launch, start the server configuration test with:

```
httpd -t -Dsyntax_check
```

and in your startup file, add (at the top):

```
return if Apache->define('syntax_check');
```

if you want to prevent the code in the file from being executed.

3.7.2 *Testing the mod_perl Server*

Assuming that we have configured the `/perl` URI base to invoke scripts under `Apache::Registry` handler, let's create a simple CGI script and put a test script into `/home/httpd/perl/` directory:

```

test.pl
-----
#!/usr/bin/perl -w
use strict;
print "Content-type: text/html\r\n\r\n";
print "It worked!!!\n";

```

Make it executable and readable by server, if your server is running as user nobody (hint: look for User directive in `httpd.conf` file), do the following:

```

% chown nobody /home/httpd/perl/test.pl
% chmod u+rx /home/httpd/perl/test.pl

```

Test that the script is running from the command line, by executing it:

```

% /home/httpd/perl/test.pl

```

You should see:

```

Content-type: text/html

It worked!!!

```

Now it is a time to test our `mod_perl` server, assuming that your config file includes `Port 80`, go to your favorite Netscape browser and fetch the following URL (after you have started the server):

```

http://localhost/perl/test.pl

```

Make sure that you have a loop-back device configured, if not -- use the real server name for this test, for example:

```

http://www.example.com/perl/test.pl

```

You should see:

```

It worked!!!

```

If something went wrong, go through the installation process again, and make sure you didn't make a mistake. If that doesn't help, read the `INSTALL` pod document (`perlpod INSTALL`) in the `mod_perl` distribution directory.

3.7.3 Publishing Port Numbers Different from 80

It is advised not to publish the 8080 (or alike) port number in URLs, but rather using a proxying rewrite rule in the thin (`httpd_docs`) server:

```

RewriteRule */perl/(.*) http://my.url:8080/perl/$1 [P]

```

One problem with publishing 8080 port numbers is that I was told that IE 4.x has a bug when re-posting data to a non-port-80 url. It drops the port designator, and uses port 80 anyway.

The other reason is that firewalls the users work from behind might have all ports closed, but 80.

3.7.4 Apache Restarts Twice On Start

When the server is restarted, the configuration and module initialization phases are called again (twice in total) before children get forked. The restart is done in order to ensure that the future restart will work out correctly, by making sure that all modules can survive a restart (SIGHUP). This is very important if you restart a production server.

You can control what code to execute only on the start or only on restart by checking the value of `$Apache::Server::Starting` and `$Apache::Server::ReStarting` respectively. The former variable is *true* when the server is starting and the latter when it's restarting.

;o)

4 Choosing the Right Strategy

4.1 What we will learn in this chapter

- Deployment of mod_perl in Overview, with the pros and cons.
- Standalone mod_perl Enabled Apache Server
- One Plain Apache and One mod_perl-enabled Apache Servers
- One light non-Apache and One mod_perl enabled Apache Servers
- Proxy servers (Squid, and Apache's mod_proxy).

I will present a few ways of using standalone mod_perl, and some combinations of mod_perl and other technologies. I'll describe how these things work together, and offer my opinions on the pros and cons of each, the relative degree of difficulty in installing and maintaining them, and some hints on approaches that should be used and things to avoid.

4.2 mod_perl Deployment Overview

There are several different ways to build, configure and deploy your mod_perl enabled server. Some of them are:

1. Having one binary and one configuration file (one big binary for mod_perl).
2. Having two binaries and two configuration files (one big binary for mod_perl and one small binary for static objects like images.)
3. Having one DSO-style binary and two configuration files, with mod_perl available as a loadable object.
4. Any of the above plus a reverse proxy server in http accelerator mode.

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

1. The first option will kill your production site if you serve a lot of static data from large (4 to 15MB) webserver processes. On the other hand, while testing you will have no other server interaction to mask or add to your errors.
2. This option allows you to tune the two servers individually, for maximum performance.

However, you need to choose between running the two servers on multiple ports, multiple IPs, etc., and you have the burden of administering more than one server. You have to deal with proxying or fancy site design to keep the two servers in synchronization.

3. With DSO, modules can be added and removed without recompiling the server, and their code is even shared among multiple servers.

You can compile just once and yet have more than one binary, by using different configuration files to load different sets of modules. The different Apache servers loaded in this way can run simultaneously to give a setup such as described in the second option above.

On the down side, you are playing at the bleeding edge.

You are dealing with a new solution that has weak documentation and is still subject to change. It is still somewhat platform specific. Your mileage may vary.

The DSO module (`mod_so`) adds size and complexity to your binaries.

4. The fourth option (proxy in http accelerator mode), once correctly configured and tuned, improves the performance of any of the above three options by caching and buffering page results.

4.3 Alternative architectures for running one and two servers

The next part of this chapter discusses the pros and the cons of each of these presented configurations.

We will look at the following installations:

- **Standalone `mod_perl` Enabled Apache Server**
- **One Plain Apache and One `mod_perl`-enabled Apache Servers**
- **One light non-Apache and One `mod_perl` enabled Apache Servers**
- **Adding a Proxy Server in http Accelerator Mode**

4.3.1 *Standalone `mod_perl` Enabled Apache Server*

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

The advantages:

- Simplicity. You just follow the installation instructions, configure it, restart the server and you are done.
- No network changes. You do not have to worry about using additional ports as we will see later.

- Speed. You get a very fast server, you see an enormous speedup from the first moment you start to use it.

The disadvantages:

- The process size of a mod_perl-enabled Apache server is huge (maybe 4Mb at startup and growing to 10Mb and more, depending on how you use it) compared to the typical plain Apache. Of course if memory sharing is in place, RAM requirements will be smaller.

You probably have a few tens of child processes. The additional memory requirements add up in direct relation to the number of child processes. Your memory demands are growing by an order of magnitude, but this is the price you pay for the additional performance boost of mod_perl. With memory prices so cheap nowadays, the additional cost is low -- especially when you consider the dramatic performance boost mod_perl gives to your services with every 100Mb of RAM you add.

While you will be happy to have these monster processes serving your scripts with monster speed, you should be very worried about having them serve static objects such as images and html files. Each static request served by a mod_perl-enabled server means another large process running, competing for system resources such as memory and CPU cycles. The real overhead depends on static objects request rate. Remember that if your mod_perl code produces HTML code which includes images, each one will turn into another static object request. Having another plain webserver to serve the static objects solves this unpleasant obstacle. Having a proxy server as a front end, caching the static objects and freeing the mod_perl processes from this burden is another solution. We will discuss both below.

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

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

If you are new to mod_perl, this is probably the best way to get yourself started.

And of course, if your site is serving only mod_perl scripts (close to zero static objects, like images), this might be the perfect choice for you!

4.3.2 One Plain Apache and One mod_perl-enabled Apache Servers

As I have mentioned before, when running scripts under mod_perl, you will notice that the httpd processes consume a huge amount of virtual memory, from 5Mb to 15Mb and even more. That is the price you pay for the enormous speed improvements under mod_perl. (Again -- shared memory keeps the real memory that is being used much smaller :)

Using these large processes to serve static objects like images and html documents is overkill. A better approach is to run two servers: a very light, plain apache server to serve static objects and a heavier mod_perl-enabled apache server to serve requests for dynamic (generated) objects (aka CGI).

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

The advantages:

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

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

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

`http://www.nowhere.com:8080/perl/index.pl`) all relative URLs in the dynamically generated HTML will be served by the heavy mod_perl processes. You must use fully qualified URLs and not relative ones! `http://www.nowhere.com/icons/arrow.gif` is a full URL, while `/icons/arrow.gif` is a relative one. Using `<BASE HREF="http://www.nowhere.com/">` in the generated HTML is another way to handle this problem. Also the httpd_perl server could rewrite the requests back to httpd_docs (much slower) and you still need the attention of the heavy servers. This is not an issue if you hide the internal port implementations, so the client sees only one server running on port 80.

The disadvantages:

- An administration overhead.
 - The need for two different sets of configuration, log and other files. We need a special directory layout to manage these. While some directories can be shared between the two servers (like the `include` directory, containing the apache include files -- assuming that both are built from the same source distribution), most of them should be separated and the configuration files updated to reflect the changes.
 - The need for two sets of controlling scripts (startup/shutdown) and watchdogs.
 - If you are processing log files, now you probably will have to merge the two separate log files into one before processing them.

- Just as in the one server approach, we still have the problem of a mod_perl process spending its precious time serving slow clients, when the processing portion of the request was completed a long time ago. Deploying a proxy solves this, and will be covered in the next section.

As with the single server approach, this is not a major disadvantage if you are on a fast network (i.e. Intranet). It is likely that you do not want a buffering server in this case.

Before you go on with this solution you really want to look at the Adding a *Proxy Server in http Accelerator Mode* section.

4.3.3 One light non-Apache and One mod_perl enabled Apache Servers

If the only requirement from the light server is for it to serve static objects, then you can get away with non-apache servers having an even smaller memory footprint. `thttpd` has been reported to be about 5 times faster than apache (especially under a heavy load), since it is very simple and uses almost no memory (260k) and does not spawn child processes.

Meta: Hey, No personal experience here, only rumours. Please let me know if I have missed some pros/cons here. Thanks!

The Advantages:

- All the advantages of the 2 servers scenario.
- More memory saving. Apache is about 4 times bigger than **thttpd**, if you spawn 30 children you use about 30M of memory, while **thttpd** uses only 260k - 100 times less! You could use the 30M you've saved to run a few more mod_perl servers.

The memory savings are significantly smaller if your OS supports memory sharing with Dynamically Shared Objects (DSO) and you have configured apache to use it. If you do allow memory sharing, 30 light apache servers ought to use only about 3 to 4Mb, because most of it will be shared. There is no memory sharing if apache modules are statically compiled into httpd.

- Reported to be about 5 times faster than plain apache serving static objects.

The Disadvantages:

- Lacks some of apache's features, like access control, error redirection, customizable log file formats, and so on.

4.4 Adding a Proxy Server in http Accelerator Mode

At the beginning there were 2 servers: one plain apache server, which was *very light*, and configured to serve static objects, the other mod_perl enabled (*very heavy*) and configured to serve mod_perl scripts. We named them `httpd_docs` and `httpd_perl` respectively.

The two servers coexist at the same IP address by listening to different ports: `httpd_docs` listens to port 80 (e.g. <http://www.nowhere.com/images/test.gif>) and `httpd_perl` listens to port 8080 (e.g. <http://www.nowhere.com:8080/perl/test.pl>). Note that I did not write <http://www.nowhere.com:80> for the first example, since port 80 is the default port for the http service. Later on, I will be changing the configuration of the `httpd_docs` server to make it listen to port 81.

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

- Allow serving of static objects from the proxy's cache (objects that previously were entirely served by the `httpd_docs` server).
- You get less I/O activity reading static objects from the disk (proxy serves the most "popular" objects from RAM - of course you benefit more if you allow the proxy server to consume more RAM). Since you do not wait for the I/O to be completed you are able to serve static objects much faster.
- The proxy server acts as a sort of output buffer for the dynamic content. The `mod_perl` server sends the entire response to the proxy and is then free to deal with other requests. The proxy server is responsible for sending the response to the browser. So if the transfer is over a slow link, the `mod_perl` server is not waiting around for the data to move.

Using numbers is always more convincing :) Let's take a user connected to your site with 28.8 kbps (bps == bits/sec) modem. It means that the speed of the user's link is $28.8/8 = 3.6$ kbytes/sec. I assume an average generated HTML page to be of 10kb (kb == kilobytes) and an average script that generates this output in 0.5 secs. How long will the server wait before the user gets the whole output response? A simple calculation reveals pretty scary numbers - it will have to wait for another 6 secs ($20\text{kb}/3.6\text{kb}$), when it could serve another 12 ($6/0.5$) dynamic requests in this time.

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

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

Many users like to open many browser windows and do many things at once (download files and browse graphically *heavy* sites). So the speed of 3.6kb/sec we were assuming before, may often be 5-10 times slower.

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

- For security reasons, using any httpd accelerator (or a proxy in httpd accelerator mode) is essential because you do not let your internal server get directly attacked by arbitrary packets from whomever. The httpd accelerator and internal server communicate in expected HTTP requests. This allows for only your public “bastion” accelerating www server to get hosed in a successful attack, while leaving your internal data safe.

The disadvantages are:

- Of course there are drawbacks. Luckily, these are not functionality drawbacks, but they are more administration hassle. You have another daemon to worry about, and while proxies are generally stable, you have to make sure to prepare proper startup and shutdown scripts, which are run at boot and reboot as appropriate. Also, you might want to set up the crontab to run a watchdog script.
- Proxy servers can be configured to be light or heavy, the admin must decide what gives the highest performance for his application. A proxy server like squid is light in the concept of having only one process serving all requests. But it can appear pretty heavy when it loads objects into memory for faster service.

Have I succeeded in convincing you that you want a proxy server?

If you are on a local area network (LAN), then the big benefit of the proxy buffering the output and feeding a slow client is gone. You are probably better off sticking with a straight mod_perl server in this case.

4.5 Implementations of Proxy Servers

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

4.5.1 *The Squid Server*

The Advantages:

- Caching of static objects. These are served much faster, assuming that your cache size is big enough to keep the most frequently requested objects in the cache.
- Buffering of dynamic content, by taking the burden of returning the content generated by mod_perl servers to slow clients, thus freeing mod_perl servers from waiting for the slow clients to download the data. Freed servers immediately switch to serve other requests, thus your number of required servers goes down dramatically.
- Non-linear URL space / server setup. You can use Squid to play some tricks with the URL space and/or domain based virtual server support.

The Disadvantages:

- Proxying dynamic content is not going to help much if all the clients are on a fast local net. Also, a message on the squid mailing list implied that squid only buffers in 16k chunks so it would not allow a mod_perl to complete immediately if the output is larger.
- Speed. Squid is not very fast today when compared with the plain file based web servers available. Only if you are using a lot of dynamic features such as mod_perl or similar is there a reason to use Squid, and then only if the application and the server are designed with caching in mind.
- Memory usage. Squid uses quite a bit of memory.
- HTTP protocol level. Squid is pretty much a HTTP/1.0 server, which seriously limits the deployment of HTTP/1.1 features.
- HTTP headers, dates and freshness. The squid server might give out stale pages, confusing downstream/client caches.(You update some documents on the site, but squid will still serve the old ones.)
- Stability. Compared to plain web servers, Squid is not the most stable.

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

4.5.2 Apache's mod_proxy

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

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

The difficult case is where:

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

The Advantages:

- No additional server is needed. We keep the one plain plus one mod_perl enabled apache servers. All you need is to enable mod_proxy in the httpd_docs server and add a few lines to httpd.conf file.
- The ProxyPass and ProxyPassReverse directives allow you to hide the internal redirects, so if http://nowhere.com/modperl/ is actually http://localhost:81/modperl/, it will be absolutely transparent to the user. ProxyPass redirects the request to the mod_perl server, and when it gets the response, ProxyPassReverse rewrites the URL back to the original one, e.g:

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

- It does mod_perl output buffering like squid does.
- It even does caching. You have to produce correct Content-Length, Last-Modified and Expires http headers for it to work. If some of your dynamic content does not change frequently, you can dramatically increase performance by caching it with ProxyPass.
- ProxyPass happens before the authentication phase, so you do not have to worry about authenticating twice.
- Apache is able to accelerate secure HTTP requests completely, while also doing accelerated HTTP. With squid you have to use an external redirection program for that.
- The latest (apache 1.3.6 and later) Apache proxy accelerated mode is reported to be very stable.

The Disadvantages:

- Users have reported that it might be a bit slow, but the latest version is fast enough.

;o)

5 Real World Scenarios Implementation

5.1 What we will learn in this chapter

- Standalone mod_perl Enabled Apache Server
- One Plain and One mod_perl enabled Apache Servers
- Running 2 webservers and squid in httpd accelerator mode
- Running 1 webserver and squid in httpd accelerator mode
- One Light and One Heavy Server where ALL htmls are Perl-Generated
- Building and Using mod_proxy

5.2 Standalone mod_perl Enabled Apache Server

We saw the implementation in the Installation section at the beginning of Tutorial. A quick reminder of what it takes to build a standalone server:

```
% cd /usr/src
% lwp-download http://www.apache.org/dist/apache_x.x.x.tar.gz
% lwp-download http://perl.apache.org/dist/mod_perl-x.xx.tar.gz
% tar zvxf apache_x.xx.tar.gz
% tar zvxf mod_perl-x.xx.tar.gz
% cd mod_perl-x.xx
% perl Makefile.PL APACHE_SRC=../apache_x.x.x/src \
  DO_HTTPD=1 USE_APACI=1 PERL_MARK_WHERE=1 EVERYTHING=1
% make && make test && make install
% cd ../apache_x.x.x
% make install
```

5.3 One Plain and One mod_perl enabled Apache Servers

Since we are going to run two apache servers we will need two different sets of configuration, log and other files. We need a special directory layout. While some of the directories can be shared between the two servers (assuming that both are built from the same source distribution), others should be separated. From now on I will refer to these two servers as **httpd_docs** (vanilla Apache) and **httpd_perl** (Apache/mod_perl).

For this illustration, we will use `/usr/local` as our *root* directory. The Apache installation directories will be stored under this root (`/usr/local/bin`, `/usr/local/etc` and etc...)

First let's prepare the sources. We will assume that all the sources go into `/usr/src` dir. It is better when you use two separate copies of apache sources. Since you probably will want to tune each apache version at separate and to do some modifications and recompilations as the time goes. Having two independent source trees will prove helpful, unless you use DSO, which is covered later in this section.

Make two subdirectories:

```
% mkdir /usr/src/httpd_docs
% mkdir /usr/src/httpd_perl
```

Put the Apache sources into a /usr/src/httpd_docs directory:

```
% cd /usr/src/httpd_docs
% gzip -dc /tmp/apache_x.x.x.tar.gz | tar xvf -
```

If you have a gnu tar:

```
% tar xvzf /tmp/apache_x.x.x.tar.gz
```

Replace /tmp directory with a path to a downloaded file and x.x.x with the version of the server you have.

```
% cd /usr/src/httpd_docs

% ls -l
drwxr-xr-x  8 stas  stas 2048 Apr 29 17:38 apache_x.x.x/
```

Now we will prepare the httpd_perl server sources:

```
% cd /usr/src/httpd_perl
% gzip -dc /tmp/apache_x.x.x.tar.gz | tar xvf -
% gzip -dc /tmp/modperl-x.xx.tar.gz | tar xvf -

% ls -l
drwxr-xr-x  8 stas  stas 2048 Apr 29 17:38 apache_x.x.x/
drwxr-xr-x  8 stas  stas 2048 Apr 29 17:38 modperl-x.xx/
```

Time to decide on the desired directory structure layout (where the apache files go):

```
ROOT = /usr/local
```

The two servers can share the following directories (so we will not duplicate data):

```
/usr/local/bin/
/usr/local/lib
/usr/local/include/
/usr/local/man/
/usr/local/share/
```

Important: we assume that both servers are built from the same Apache source version.

Servers store their specific files either in httpd_docs or httpd_perl sub-directories:

```
/usr/local/etc/httpd_docs/
                    httpd_perl/

/usr/local/sbin/httpd_docs/
                    httpd_perl/
```

```

/usr/local/var/httpd_docs/logs/
    proxy/
    run/
httpd_perl/logs/
    proxy/
    run/

```

After completion of the compilation and the installation of the both servers, you will need to configure them. To make things clear before we proceed into details, you should configure the `/usr/local/etc/httpd_docs/httpd.conf` as a plain apache and `Port` directive to be 80 for example. And `/usr/local/etc/httpd_perl/httpd.conf` to configure for `mod_perl` server and of course whose `Port` should be different from the one `httpd_docs` server listens to (e.g. 8080). The port numbers issue will be discussed later.

The next step is to configure and compile the sources: Below are the procedures to compile both servers taking into account the directory layout I have just suggested to use.

5.3.1 Configuration and Compilation of the Sources.

Let's proceed with installation. I will use `x.x.x` instead of real version numbers so this document will never become obsolete :).

5.3.1.1 Building the `httpd_docs` Server

- Sources Configuration:

```

% cd /usr/src/httpd_docs/apache_x.x.x
% make clean
% env CC=gcc \
./configure --prefix=/usr/local \
--sbindir=/usr/local/sbin/httpd_docs \
--sysconfdir=/usr/local/etc/httpd_docs \
--localstatedir=/usr/local/var/httpd_docs \
--runtimedir=/usr/local/var/httpd_docs/run \
--logfiledir=/usr/local/var/httpd_docs/logs \
--proxycachedir=/usr/local/var/httpd_docs/proxy

```

If you need some other modules, like `mod_rewrite` and `mod_include` (SSI), add them here as well:

```
--enable-module=include --enable-module=rewrite
```

Note: `gcc` -- compiles `httpd` by 100K+ smaller than `cc` on AIX OS. Remove the line `env CC=gcc` if you want to use the default compiler. If you want to use it and you are a `(ba)?sh` user you will not need the `env` function, `t?csh` users will have to keep it in.

Note: add `--layout` to see the resulting directories' layout without actually running the configuration process.

- **Sources Compilation:**

```
% make
% make install
```

Rename httpd to http_docs

```
% mv /usr/local/sbin/httpd_docs/httpd \
/usr/local/sbin/httpd_docs/httpd_docs
```

Now update an **apachectl** utility to point to the renamed httpd via your favorite text editor or by using perl:

```
% perl -p -i -e 's|httpd_docs/httpd|httpd_docs/httpd_docs|' \
/usr/local/sbin/httpd_docs/apachectl
```

5.3.1.2 Building the httpd_perl (mod_perl enabled) Server

Before you start to configure the mod_perl sources, you should be aware that there are a few Perl modules that have to be installed before building mod_perl. You will be alerted if any required modules are missing when you run the perl Makefile.PL command line below. If you discover that some are missing, pick them from your nearest CPAN repository (if you do not know what is it, make a visit to <http://www.perl.com/CPAN>) or run the CPAN interactive shell via the command line perl -MCPAN -e shell.

Make sure the sources are clean:

```
% cd /usr/src/httpd_perl/apache_x.x.x
% make clean
% cd /usr/src/httpd_perl/mod_perl-x.xx
% make clean
```

It is important to **make clean** since some of the versions are not binary compatible (e.g apache 1.3.3 vs 1.3.4) so any “third-party” C modules need to be re-compiled against the latest header files.

Here I did not find a way to compile with gcc (my perl was compiled with cc so we have to compile with the same compiler!!!

```
% cd /usr/src/httpd_perl/mod_perl-x.xx

% /usr/local/bin/perl Makefile.PL \
APACHE_PREFIX=/usr/local/ \
APACHE_SRC=../apache_x.x.x/src \
DO_HTTPD=1 \
USE_APACI=1 \
PERL_MARK_WHERE=1 \
PERL_STACKED_HANDLERS=1 \
ALL_HOOKS=1 \
APACI_ARGS=--sbindir=/usr/local/sbin/httpd_perl, \
--sysconfdir=/usr/local/etc/httpd_perl, \
```

```
--localstatedir=/usr/local/var/httpd_perl, \
--runtimedir=/usr/local/var/httpd_perl/run, \
--logfiledir=/usr/local/var/httpd_perl/logs, \
--proxycachedir=/usr/local/var/httpd_perl/proxy
```

Notice that **all** APACI_ARGS (above) must be passed as one long line if you work with `t?csh!!!` However it works correctly the way it shown above with `(ba)?sh` (by breaking the long lines with `'\``). If you work with `t?csh` it does not work, since `t?csh` passes APACI_ARGS arguments to `./configure` by keeping the new lines untouched, but stripping the original `'\``, thus breaking the configuration process.

As with `httpd_docs` you might need other modules like `mod_rewrite`, so add them here:

```
--enable-module=rewrite
```

Note: `PERL_STACKED_HANDLERS=1` is needed for `Apache::DBI`

Now, build, test and install the `httpd_perl`.

```
% make && make test && make install
```

Note: apache puts a stripped version of `httpd` at `/usr/local/sbin/httpd_perl/httpd`. The original version which includes debugging symbols (if you need to run a debugger on this executable) is located at `/usr/src/httpd_perl/apache_x.x.x/src/httpd`.

Note: You may have noticed that we did not run `make install` in the apache's source directory. When `USE_APACI` is enabled, `APACHE_PREFIX` will specify the `--prefix` option for apache's `configure` utility, specifying the installation path for apache. When this option is used, `mod_perl`'s `make install` will also make `install` on the apache side, installing the `httpd` binary, support tools, along with the configuration, log and document trees.

If `make test` fails, look into `t/logs` and see what is in there.

While doing `perl Makefile.PL ... mod_perl` might complain by warning you about missing `libgdbm`. Users reported that it is actually crucial, and you must have it in order to successfully complete the `mod_perl` building process.

Now rename the `httpd` to `httpd_perl`:

```
% mv /usr/local/sbin/httpd_perl/httpd \
/usr/local/sbin/httpd_perl/httpd_perl
```

Update the `apachectl` utility to point to renamed `httpd` name:

```
% perl -p -i -e 's|httpd_perl/httpd|httpd_perl/httpd_perl|' \
/usr/local/sbin/httpd_perl/apachectl
```

5.3.2 Configuration of the servers

Now when we have completed the building process, the last stage before running the servers, is to configure them.

5.3.2.1 Basic httpd_docs Server's Configuration

Configuring of `httpd_docs` server is a very easy task. Open `/usr/local/etc/httpd_docs/httpd.conf` into your favorite editor (starting from version 1.3.4 of Apache - there is only one file to edit). And configure it as you always do. Make sure you configure the log files and other paths according to the directory layout we decided to use.

Start the server with:

```
/usr/local/sbin/httpd_docs/apachectl start
```

5.3.2.2 Basic httpd_perl Server's Configuration

Here we will make a basic configuration of the `httpd_perl` server. We edit the `/usr/local/etc/httpd_perl/httpd.conf` file. As with `httpd_docs` server configuration, make sure that `ErrorLog` and other file's location directives are set to point to the right places, according to the chosen directory layout.

The first thing to do is to set a `Port` directive - it should be different from 80 since we cannot bind 2 servers to use the same port number on the same machine. Here we will use 8080. Some developers use port 81, but you can bind to it, only if you have root permissions. If you are running on multiuser machine, there is a chance someone already uses that port, or will start using it in the future - which as you understand might cause a collision. If you are the only user on your machine, basically you can pick any not used port number. Port number choosing is a controversial topic, since many organizations use firewalls, which may block some of the ports, or enable only a known ones. From my experience the most used port numbers are: 80, 81, 8000 and 8080. Personally, I prefer the port 8080. Of course with 2 server scenario you can hide the nonstandard port number from firewalls and users, by either using the `mod_proxy`'s `ProxyPass` or proxy server like squid.

Now we proceed to `mod_perl` specific directives. A good idea will be to add them all at the end of the `httpd.conf`, since you are going to fiddle a lot with them at the beginning.

First, you need to specify the location where all `mod_perl` scripts will be located.

Add the following configuration directive:

```
# mod_perl scripts will be called from
Alias /perl/ /usr/local/myproject/perl/
```

From now on, all requests starting with `/perl` will be executed under `mod_perl` and will be mapped to the files in `/usr/local/myproject/perl/`.

Now we should configure the `/perl` location.

```
PerlModule Apache::Registry

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

This configuration causes all scripts that are called with a `/perl` path prefix to be executed under the `Apache::Registry` module and as a CGI (so the `ExecCGI`, if you omit this option the script will be printed to the user's browser as a plain text or will possibly trigger a 'Save-As' window). `Apache::Registry` module lets you run almost unaltered CGI/perl scripts under `mod_perl`. `PerlModule` directive is an equivalent of perl's `require()`. We load the `Apache::Registry` module before we use it in the `PerlHandler` in the `Location` configuration.

`PerlSendHeader On` tells the server to send an HTTP header to the browser on every script invocation. You will want to turn this off for `nph` (non-parsed-headers) scripts.

Now start the server with:

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

5.4 Running 2 webservers and squid in httpd accelerator mode

While I have detailed the `mod_perl` server installation, you are on your own with installing the squid server. I run linux, so I downloaded the rpm package, installed it, configured the `/etc/squid/squid.conf`, fired off the server and was all set. Basically once you have the squid installed, you just need to modify the default `squid.conf` the way I will explain below, then you are ready to run it.

First, let's understand what do we have in hands and what do we want from squid. We have an `httpd_docs` and `httpd_perl` servers listening on ports 81 and 8080 accordingly (we have to move the `httpd_docs` server to port 81, since port 80 will be taken over by squid). Both reside on the same machine as squid. We want squid to listen on port 80, forward a single static object request to the port `httpd_docs` server listens to, and dynamic request to `httpd_perl`'s port. Both servers return the data to the proxy server (unless it is already cached in the squid), so user never sees the other ports and never knows that there might be more than one server running. Proxy server makes all the magic behind it transparent to user. Do not confuse it with **mod_rewrite**, where a server redirects the request somewhere according to the rules and forgets about it. The described functionality is being known as `httpd accelerator mode` in proxy dialect.

You should understand that squid can be used as a straight forward proxy server, generally used at companies and ISPs to cut down the incoming traffic by caching the most popular requests. However we want to run it in the `httpd accelerator` mode. Two directives: `httpd_accel_host` and `httpd_accel_port` enable this mode. We will see more details in a few seconds. If you are currently using the squid in the regular proxy mode, you can extend its functionality by running both modes concurrently. To accomplish this, you extend the existent squid configuration with `httpd accelerator mode`'s related directives or you just create one from scratch.

As stated before, squid listens now to the port 80, we have to move the `httpd_docs` server to listen for example to the port 81 (your mileage may vary :). So you have to modify the `httpd.conf` in the `httpd_docs` configuration directory and restart the `httpd_docs` server (But not before we get the squid running if you are working on the production server). And as you remember `httpd_perl` listens to port 8080.

Let's go through the changes we should make to the default configuration file. Since this file (`/etc/squid/squid.conf`) is huge (about 60k+) and we would not use 95% of it, my suggestion is to write a new one including only the modified directives.

We want to enable the redirect feature, to be able to serve requests, by more then one server (in our case we have `httpd_docs` and `httpd_perl`) servers. So we specify `httpd_accel_host` as `virtual`. This assumes that your server has multiple interfaces - Squid will bind to all of them.

```
httpd_accel_host virtual
```

Then we define the default port - by default, if not redirected, `httpd_docs` will serve the pages. We assume that most requests will be of the static nature. We have our `httpd_docs` listening on port 81.

```
httpd_accel_port 81
```

And as described before, squid listens to port 80.

```
http_port 80
```

We do not use `icp` (`icp` used for cache sharing between neighbor machines), which is more relevant in the proxy mode.

```
icp_port 0
```

`hierarchy_stoplist` defines a list of words which, if found in a URL, causes the object to be handled directly by this cache. In other words, use this to not query neighbor caches for certain objects. Note that I have configured the `/cgi-bin` and `/perl` aliases for my dynamic documents, if you named them in a different way, make sure to use the correct aliases here.

```
hierarchy_stoplist /cgi-bin /perl
```

Now we tell squid not to cache dynamic pages.

```
acl QUERY urlpath_regex /cgi-bin /perl
no_cache deny QUERY
```

Please note that the last two directives are controversial ones. If you want your scripts to be more complying with the HTTP standards, the headers of your scripts should carry the `Caching Directives` according to the HTTP specs. You will find a complete tutorial about this topic in `Tutorial on HTTP Headers for mod_perl users` by Andreas J. Koenig (at <http://perl.apache.org>). If you set the headers correctly there is no need to tell squid accelerator to **NOT** try to cache something. The headers I am talking about are `Last-Modified` and `Expires`. What are they good for? Squid would not bother your mod_perl server a second time if a request is (a) cachable and (b) still in the cache. Many mod_perl applications will produce identical results on identical requests at least if not much time goes by between the requests. So your squid might have a hit ratio of 50%, which means that mod_perl servers will have as twice as less work to do than before. This is only possible by setting the headers correctly.

Even if you insert user-ID and date in your page, caching can save resources when you set the expiration time to 1 second. A user might double click where a single click would do, thus sending two requests in parallel, squid could serve the second request.

But if you are lazy, or just have too many things to deal with, you can leave the above directives the way I described. But keep in mind that one day you will want to reread this snippet and the Andreas' tutorial and squeeze even more power from your servers without investing money for additional memory and better hardware.

While testing you might want to enable the debugging options and watch the log files in `/var/log/squid/`. But turn it off in your production server. I list it commented out. (`28 == access control routes`).

```
# debug_options ALL, 1, 28, 9
```

We need to provide a way for squid to dispatch the requests to the correct servers, static object requests should be redirected to `httpd_docs` (unless they are already cached), while dynamic should go to the `httpd_perl` server. The configuration below tells squid to fire off 10 redirect daemons at the specified path of the redirect daemon and disables rewriting of any `Host:` headers in redirected requests (as suggested by squid's documentation). The redirection daemon script is enlisted below.

```
redirect_program /usr/lib/squid/redirect.pl
redirect_children 10
redirect_rewrites_host_header off
```

Maximum allowed request size in kilobytes. This one is pretty obvious. If you are using POST to upload files, then set this to the largest file's size plus a few extra kbytes.

```
request_size 1000 KB
```

Then we have access permissions, which I will not explain. But you might want to read the documentation so to avoid any security flaws.

```
acl all src 0.0.0.0/0.0.0.0
acl manager proto cache_object
acl localhost src 127.0.0.1/255.255.255.255
acl myserver src 127.0.0.1/255.255.255.255
acl SSL_ports port 443 563
acl Safe_ports port 80 81 8080 81 443 563
```

```
acl CONNECT method CONNECT

http_access allow manager localhost
http_access allow manager myserver
http_access deny manager
http_access deny !Safe_ports
http_access deny CONNECT !SSL_ports
# http_access allow all
```

Since squid should be run as non-root user, you need these if you are invoking the squid as root.

```
cache_effective_user squid
cache_effective_group squid
```

Now configure a memory size to be used for caching. A squid documentation warns that the actual size of squid can grow three times larger than the value you are going to set.

```
cache_mem 20 MB
```

Keep pools of allocated (but unused) memory available for future use. Read more about it in the squid documents.

```
memory_pools on
```

Now tight the runtime permissions of the cache manager CGI script (`cachemgr.cgi`, that comes bundled with squid) on your production server.

```
cachemgr_passwd disable shutdown
#cachemgr_passwd none all
```

Now the redirection daemon script (you should put it at the location you have specified by `redirect_program` parameter in the config file above, and make it executable by webserver of course):

```
#!/usr/local/bin/perl

$|=1;

while (<>) {
    # redirect to mod_perl server (httpd_perl)
    print($_, next if s|(:81)?/perl/|:8080/perl/|o;

    # send it unchanged to plain apache server (http_docs)
    print;
}
```

In my scenario the proxy and the apache servers are running on the same machine, that's why I just substitute the port. In the presented squid configuration, requests that passed through squid are converted to point to the **localhost** (which is 127.0.0.1). The above redirector can be more complex of course, but you know the perl, right?

A few notes regarding redirector script:

You must disable buffering. `$|=1;` does the job. If you do not disable buffering, the `STDOUT` will be flushed only when the buffer becomes full and its default size is about 4096 characters. So if you have an average URL of 70 chars, only after 59 (4096/70) requests the buffer will be flushed, and the requests will finally achieve the server in target. Your users will just wait till it will be filled up.

If you think that it is a very ineffective way to redirect, I'll try to prove you the opposite. The redirector runs as a daemon, it fires up N redirect daemons, so there is no problem with perl interpreter loading, exactly like `mod_perl -- perl` is loaded all the time and the code was already compiled, so redirect is very fast (not slower if redirector was written in C or alike). Squid keeps an open pipe to each redirect daemon, thus there is even no overhead of the expensive system calls.

Now it is time to restart the server, at linux I do it with:

```
/etc/rc.d/init.d/squid restart
```

Now the setup is complete ...

Almost... When you try the new setup, you will be surprised and upset to discover a port 81 showing up in the URLs of the static objects (like `htmls`). Hey, we did not want the user to see the port 81 and use it instead of 80, since then it will bypass the squid server and the hard work we went through was just a waste of time?

The solution is to run both squid and `httpd_docs` at the same port. This can be accomplished by binding each one to a specific interface. Modify the `httpd.conf` in the `httpd_docs` configuration directory:

```
Port 80
BindAddress 127.0.0.1
Listen 127.0.0.1:80
```

Modify the `squid.conf`:

```
http_port 80
tcp_incoming_address 123.123.123.3
tcp_outgoing_address 127.0.0.1
httpd_accel_host 127.0.0.1
httpd_accel_port 80
```

Where `123.123.123.3` should be replaced with IP of your main server. Now restart squid and `httpd_docs` in either order you want, and voila the port number has gone.

You must also have in the `/etc/hosts` an entry (most chances that it's already there):

```
127.0.0.1 localhost.localdomain localhost
```

Now if your scripts were generating HTML including fully qualified self references, using the 8080 or other port -- you should fix them to generate links to point to port 80 (which means not using the port at all). If you do not, users will bypass squid, like if it was not there at all, by making direct requests to the `mod_perl` server's port.

The only question left is what to do with users who bookmarked your services and they still have the port 8080 inside the URL. Do not worry about it. The most important thing is for your scripts to return a full URLs, so if the user comes from the link with 8080 port inside, let it be. Just make sure that all the consecutive calls to your server will be rewritten correctly. During a period of time users will change their bookmarks. What can be done is to send them an email if you have one, or to leave a note on your pages asking users to update their bookmarks. You could avoid this problem if you did not publish this non-80 port in first place.

To save you some keystrokes, here is the whole modified `squid.conf`:

```
http_port 80
tcp_incoming_address 123.123.123.3
tcp_outgoing_address 127.0.0.1
httpd_accel_host 127.0.0.1
httpd_accel_port 80

icp_port 0

hierarchy_stoplist /cgi-bin /perl
acl QUERY urlpath_regex /cgi-bin /perl
no_cache deny QUERY

# debug_options ALL,1 28,9

redirect_program /usr/lib/squid/redirect.pl
redirect_children 10
redirect_rewrites_host_header off

request_size 1000 KB

acl all src 0.0.0.0/0.0.0.0
acl manager proto cache_object
acl localhost src 127.0.0.1/255.255.255.255
acl myserver src 127.0.0.1/255.255.255.255
acl SSL_ports port 443 563
acl Safe_ports port 80 81 8080 81 443 563
acl CONNECT method CONNECT

http_access allow manager localhost
http_access allow manager myserver
http_access deny manager
http_access deny !Safe_ports
http_access deny CONNECT !SSL_ports
# http_access allow all

cache_effective_user squid
cache_effective_group squid

cache_mem 20 MB

memory_pools on

cachemgr_passwd disable shutdown
```

Note that all directives should start at the beginning of the line.

5.5 Running 1 webserver and squid in httpd accelerator mode

When I was first told about squid, I thought: “Hey, Now I can drop the `httpd_docs` server and to have only squid and `httpd_perl` servers“. Since all my static objects will be cached by squid, I do not need the light `httpd_docs` server. But it was a wrong assumption. Why? Because you still have the overhead of loading the objects into squid at first time, and if your site has many of them -- not all of them will be cached (unless you have devoted a huge chunk of memory to squid) and my heavy `mod_perl` servers will still have an overhead of serving the static objects. How one would measure the overhead? The difference between the two servers is memory consumption, everything else (e.g. I/O) should be equal. So you have to estimate the time needed for first time fetching of each static object at a peak period and thus the number of additional servers you need for serving the static objects. This will allow you to calculate additional memory requirements. I can imagine, this amount could be significant in some installations.

So I have decided to have even more administration overhead and to stick with squid, `httpd_docs` and `httpd_perl` scenario, where I can optimize and fine tune everything. Of course this can be not your case. If you are feeling that the scenario from the previous section is too complicated for you, make it simpler. Have only one server with `mod_perl` built in and let the squid to do most of the job that plain light apache used to do. As I have explained in the previous paragraph, you should pick this lighter setup only if you can make squid cache most of your static objects. If it cannot, your `mod_perl` server will do the work we do not want it to.

If you are still with me, install apache with `mod_perl` and squid. Then use a similar configuration from the previous section, but now `httpd_docs` is not there anymore. Also we do not need the redirector anymore and we specify `httpd_accel_host` as a name of the server and not `virtual`. There is no need to bind two servers on the same port, because we do not redirect and there is neither `Bind` nor `Listen` directives in the `httpd.conf` anymore.

The modified configuration (see the explanations in the previous section):

```
httpd_accel_host put.your.hostname.here
httpd_accel_port 8080
http_port 80
icp_port 0

hierarchy_stoplist /cgi-bin /perl
acl QUERY urlpath_regex /cgi-bin /perl
no_cache deny QUERY

# debug_options ALL, 1, 28, 9

# redirect_program /usr/lib/squid/redirect.pl
# redirect_children 10
# redirect_rewrites_host_header off

request_size 1000 KB
```

```

acl all src 0.0.0.0/0.0.0.0
acl manager proto cache_object
acl localhost src 127.0.0.1/255.255.255.255
acl myserver src 127.0.0.1/255.255.255.255
acl SSL_ports port 443 563
acl Safe_ports port 80 81 8080 81 443 563
acl CONNECT method CONNECT

http_access allow manager localhost
http_access allow manager myserver
http_access deny manager
http_access deny !Safe_ports
http_access deny CONNECT !SSL_ports
# http_access allow all

cache_effective_user squid
cache_effective_group squid

cache_mem 20 MB

memory_pools on

cachemgr_passwd disable shutdown

```

5.6 Building and Using mod_proxy

To build it into apache just add **--enable-module=proxy** during the apache **configure** stage.

Now we will talk about apache's mod_proxy and understand how it works.

The server on port 80 answers http requests directly and proxies the mod_perl enabled server in the following way:

```

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

```

PPR is the saving grace here, that makes apache a win over Squid. It rewrites the redirect on its way back to the original URI.

You can control the buffering feature with ProxyReceiveBufferSize directive:

```

ProxyReceiveBufferSize 16384

```

The above setting will set a buffer size to be of 16Kb. If it is not set explicitly or set to 0, then the default buffer size is used. It may not be smaller than 512 and it should be a number that it's a multiplicative of 512.

Both the default and the maximum possible value are depend on OS. For example on linux OS with kernel 2.2.5 the maximum and default values are either 32k or 64k (hint: grep the kernel sources for SK_RMEM_MAX variable). If you set the value bigger than limit, the default one will be used.

Under FreeBSD it's possible to configure kernel to have bigger socket buffers:

```
% sysctl -w kern.ipc.maxsockbuf=2621440
```

When you tell the kernel to use bigger sockets you can set bigger values for *ProxyReceiveBufferSize*. i.e. 1048576 (1Mb) and bigger.

So basically to get an immediate release of the mod_perl server from stale awaiting, *ProxyReceiveBufferSize* should be set to a value greater than the biggest generated response produced by any mod_perl script but not bigger than the limit. But even if not all the requests' output will be small enough or the buffer big enough to absorb it all, you've got an improve since the processes that generated smaller responses will be immediately released.

As the name states, its buffering feature applies only to **downstream data** (coming from the origin server to the proxy) and not upstream (i.e. buffering the data being uploaded from the client browser to the proxy, thus freeing the httpd_perl origin server from being tied up during a large POST such as a file upload).

Apache does caching as well. It's relevant to mod_perl only if you produce proper headers, so your scripts' output can be cached. See apache documentation for more details on configuration of this capability.

Ask Bjoern Hansen has written a `mod_proxy_add_forward` module for apache, that sets the `X-Forwarded-For` field when doing a `ProxyPass`, similar to what squid can do. (Its location is specified in the help section). Basically, that module adds an extra HTTP header to proxying requests. You can access that header in the mod_perl-enabled server, and set the IP of the remote server. You won't need to compile anything into the back-end server, if you are using `Apache::Registry,PerlRun` just put something like the following into `start-up.pl`:

```
sub My::ProxyRemoteAddr ($) {
    my $r = shift;

    # we'll only look at the X-Forwarded-For header if the requests
    # comes from our proxy at localhost
    return OK unless ($r->connection->remote_ip eq "127.0.0.1");

    # Select last value in the chain -- original client's ip
    if (my ($ip) = $r->headers_in->{'X-Forwarded-For'} =~ /([\^,\s]+)$/) {
        $r->connection->remote_ip($ip);
    }

    return OK;
}
```

And in `httpd.conf`:

```
PerlPostReadRequestHandler My::ProxyRemoteAddr
```

Different sites have different needs. If you're using the header to set the IP address, apache believes it is dealing with (in the logging and stuff), you really don't want anyone but your own system to set the header. That's why the above "recommended code" checks where the request is really coming from,

before changing the `remote_ip`.

Generally you shouldn't trust the `X-Forwarded-For` header. You only want to rely on `X-Forwarded-For` headers from proxies you control yourself. If you know how to spoof a cookie you've probably got the general idea on making HTTP headers and can spoof the `X-Forwarded-For` header as well. The only address **you** can count on as being a reliable value is the one from `r->connection->remote_ip`.

From that point on, the remote IP address is correct. You should be able to access `REMOTE_ADDR` as usual.

It was reported that Ben Laurie's Apache-SSL does not seem to put the IPs in the `X-Forwarded-For` header (it does not set up such a header at all). However, the `REMOTE_ADDR` it sets up and contains the IP of the original client machine.

You could do the same thing with other environment variables (though I think several of them are preserved, you will want to run some tests to see which ones).

;o)

6 mod_perl for ISPs

6.1 What we will learn in this chapter

- ISPs providing mod_perl services - a fantasy or reality

6.2 ISPs providing mod_perl services - a fantasy or reality.

You have fallen in love with mod_perl from the first sight, since the moment you have installed it at your home box. But when you wanted to convert your CGI scripts, currently running on your favorite ISPs machine, to run under mod_perl - you have discovered, your ISPs either have never heard of such a beast, or refuse 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 provides, to attract more clients to become your users and hopefully to have a bigger slice than a neighbor ISP.

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 3 choices to choose from:

1. ISP cannot afford having a user, running scripts under mod_perl, on the main server, since it will die very soon for one of the many reasons: either sloppy programming, or user testing just updated script which probably has some syntax errors and etc, no need to explain why if you are familiar with mod_perl peculiarities. The only scripts that **CAN BE ALLOWED** to use, are the ones that were written by ISP and are not being modified by user (guest books, counters and etc - the same standard scripts ISPs providing since they were born). So you have to say **NO** for this choice.

More things to think about are file permissions (any user who is allowed to write and run CGI script, can at least read if not write any other files that has a permissions of the web server. This has nothing to do with mod_perl, and there are solutions for that suEXEC and cgiwrap for example) and Apache::DBI connections (You can pick a connection from the pool of cached connections, opened by someone else by hacking the Apache::DBI code).

2. But, hey why I cannot let my user to run his own server, so I clean my hands off and do not care how dirty and sloppy user's code is (assuming that user is running the server by his own username).

This option is fine as long as you are concerned about your new system requirements. If you have even some very limited experience with mod_perl, you know that mod_perl enabled apache servers while freeing up your CPU and lets you run scripts much much faster, has a huge memory demands (5-20 times the plain apache uses). The size depends on the code length, sloppiness of the programmer, 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 demanding 10Mb of memory which spawns 10 children, already rises your memory requirements by 100Mb (the real requirements are actually smaller if your OS allows code sharing between processes and a programmer exploits these features in her code). Now multiply the received number by the number of users you intend to have and you will get the memory requirements. Since ISPs never say no, you better use an opposite approach -

think of a largest memory size you can afford then divide it by one user's requirements as I have shown in example, and you will know how much mod_perl users you can afford :)

But who am I to prognosticate how much memory your user may use. His requirement from a single server can be very modest, but do you know how many of servers he will run (after all she has all the control over httpd.conf - and it has to be that way, since this is very essential for the user running mod_perl)?

All this rumbling about memory leads to a single question: Can you restrict user from using more than X memory? Or another variation of the question: Assuming you have as much memory as you want, can you charge user for the average memory usage?

If the answer for either of the above question is positive, you are all set and your clients will prize your name for letting them run mod_perl! There are tools to restrict resources' usage (See for example man pages for `ulimit(3)`, `getrlimit(2)`, `setrlimit(2)` and `sysconf(3)`).

If you have picked this choice, you have to provide your client:

- Shutdown/startup scripts installed together with the rest of your daemon startup scripts (e.g. `/etc/rc.d` directory) scripts, so when you reboot your machine user's server will be correctly shutdown and will be back online the moment your system comes back online. Also make sure to start each server under username the server belongs to, if you are not looking for a big trouble.
- Proxy (in a forward or httpd accelerator mode) services for user's virtual host. Since user will have to run her server on unprivileged port (>1024), you will have to forward all requests from `user.given.virtual.hostname:80` (which is `user.given.virtual.hostname` without port - 80 is a default) to `your.machine.ip:port_assigned_to_user` and user to code his scripts to write self referencing URLs to be of `user.given.virtual.hostname` base of course.

Letting user to run a mod_perl server, immediately adds a requirement for user to be able to restart and configure their own server. But only root can bind port 80. That is why user has to use ports numbers >1024.

- Another problem you will have to solve is how to assign ports between users. Since user can pick any port above 1024 to run his server on, you will have to make some regulation here. A simple example will stress the importance of this problem: I am a malicious user or I just a rival of some fellow who runs his own server on your ISP. All I should do is to find out what port his server is listening to (e.g. with help of `netstat(8)`) and configure my own server to listen on the same port. While I am unable to bind to this same port, imagine what will happen when you reboot your system and my startup script happen to be run before my rivals! I get the port first, now all requests will be redirected to my server and let your imagination go wild about what nasty things might happen then. Of course the ugly things will be revealed pretty soon, but the damage has been done.

3. A much better, but costly solution is **co-location**. Let user to hook her (or ISP's) stand alone machine into your network, and forget about this user. Of course either user or you will have to make all the system administration chores and it will cost your client more money.

All in all, who are the people who seek the mod_perl support? The ones who run serious projects/businesses, who can afford a stand alone box, thus gaining their goal of self autonomy and keeping their ISP happy. So money is not an obstacle.

;o)

7 Getting Help and Further Learning

7.1 What we will learn in this chapter

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

7.2 Getting help

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

7.3 Get help with mod_perl

- **mod_perl home**

<http://perl.apache.org>

- **mod_perl Garden project**

<http://modperl.sourcegarden.org>

- **mod_perl Books**

- **'Apache Modules' Book**

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

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

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

○ **'Enabling web services with mod_perl' Book**

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

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

● **mod_perl Guide**

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

● **mod_perl FAQ**

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

● **mod_perl performance tuning guide**

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

● **mod_perl plugin reference guide**

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

● **Quick guide for moving from CGI to mod_perl**

at http://perl.apache.org/dist/cgi_to_mod_perl.html .

● **mod_perl_traps, common traps and solutions for mod_perl users**

at http://perl.apache.org/dist/mod_perl_traps.html .

● **mod_perl Quick Reference Card**

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

● **mod_perl Resources Page**

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

● **mod_perl mailing list**

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

Subject and with Body:

```
subscribe modperl
```

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

More archives available:

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

7.4 Get help with Perl

- **The Perl FAQ**

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

- **The Perl home**

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

- **The Perl Journal**

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

- **Perl Module Mechanics**

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

7.5 Get help with Perl/CGI

- **Perl/CGI FAQ**

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

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

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

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

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

- **WWW Security FAQ**

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

- **CGI/Perl Taint Mode FAQ**

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

7.6 Get help with Apache

- **Apache Project's Home**

<http://www.apache.org>

- **Apache Quick Reference Card**

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

- **The Apache FAQ**

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

- **Apache Server Documentation**

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

- **Apache Handlers**

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

- **mod_rewrite Guide**

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

7.7 Get help with DBI

- **Perl DBI examples**

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

- **DBI Homepage**

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

- **DBI mailing list information**

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

- **DBI mailing list archives**

<http://outside.organic.com/mail-archives/dbi-users/> <http://www.xray.mpe.mpg.de/mailling-lists/dbi/>

- **Persistent connections with mod_perl**

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

7.8 Get help with Squid - Internet Object Cache

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

;o)

Table of Contents:

Tutorial: Getting Started with mod_perl (Part I of II)	1
mod_perl tutorial: Getting Started Fast	4
1 Getting Started Fast	4
1.1 mod_perl in Four Slides	5
1.2 What is mod_perl?	5
1.3 Installation	6
1.4 Configuration	7
1.5 The "mod_perl rules" Apache::Registry Scripts	7
1.6 The "mod_perl rules" Apache Perl Module	8
1.7 Is That All I Need To Know About mod_perl?	8
mod_perl tutorial: mod_perl Installation	10
2 mod_perl Installation	10
2.1 What we will learn in this chapter	11
2.2 mod_perl Installation scenario	11
2.3 The Gory Details	11
2.3.1 Sources Configuration (perl Makefile.PL ...)	12
2.3.1.1 Configuration parameters	12
2.3.1.2 Reusing Configuration Parameters	14
2.3.2 mod_perl Building (make)	15
2.3.3 Built Server Testing (make test)	15
2.3.3.1 Manual Testing	16
2.3.4 Installation (make install)	16
2.4 mod_perl Installation with CPAN.pm's Interactive Shell	17
2.5 Installation Without Superuser Privileges	19
2.5.1 Installing Perl Modules into a Directory of Choice	19
2.5.2 Making Your Scripts Find the Locally Installed Modules	21
2.5.3 Making a Local Apache Installation	24
2.5.4 Actual Local mod_perl Enabled Apache Installation	24
2.6 Miscellaneous issues	26
2.6.1 Should I rebuild mod_perl if I have upgraded my perl?	26
2.6.2 Should I Build mod_perl with gcc or cc?	27
mod_perl tutorial: mod_perl Configuration	28
3 mod_perl Configuration	28
3.1 What we will learn in this chapter	29
3.2 Server Configuration	29
3.3 Apache Configuration	29
3.3.1 Configuration Directives	29
3.4 mod_perl Configuration	30
3.4.1 Alias Configurations	30
3.4.2 <Location> Configuration	31
3.4.3 PerlModule and PerlRequire Directives	33
3.4.4 Perl*Handlers	34
3.4.5 Stacked Handlers	35
3.4.6 PerlFreshRestart	38

3.4.7	PerlSetVar, PerlSetEnv and PerlPassEnv	38
3.4.8	PerlWarn and PerlTaintCheck	38
3.5	Start-up File	39
3.5.1	The Sample Start-up File	39
3.5.2	What Modules Should You Add to the Start-up File and Why	41
3.5.3	The Confusion with use() at the Server Start-up File	42
3.5.4	The Confusion with Global Variables in Start-up File	43
3.6	<Perl>...</Perl> Sections	43
3.6.1	Usage	43
3.6.2	Enabling	44
3.6.3	Verifying	44
3.7	Miscellaneous issues	45
3.7.1	Validating the Configuration Syntax	45
3.7.2	Testing the mod_perl Server	45
3.7.3	Publishing Port Numbers Different from 80	46
3.7.4	Apache Restarts Twice On Start	47
	mod_perl tutorial: Choosing the Right Strategy	48
4	Choosing the Right Strategy	48
4.1	What we will learn in this chapter	49
4.2	mod_perl Deployment Overview	49
4.3	Alternative architectures for running one and two servers	50
4.3.1	Standalone mod_perl Enabled Apache Server	50
4.3.2	One Plain Apache and One mod_perl-enabled Apache Servers	51
4.3.3	One light non-Apache and One mod_perl enabled Apache Servers	53
4.4	Adding a Proxy Server in http Accelerator Mode	53
4.5	Implementations of Proxy Servers	55
4.5.1	The Squid Server	55
4.5.2	Apache's mod_proxy	56
	mod_perl tutorial: Real World Scenarios Implementation	58
5	Real World Scenarios Implementation	58
5.1	What we will learn in this chapter	59
5.2	Standalone mod_perl Enabled Apache Server	59
5.3	One Plain and One mod_perl enabled Apache Servers	59
5.3.1	Configuration and Compilation of the Sources.	61
5.3.1.1	Building the httpd_docs Server	61
5.3.1.2	Building the httpd_perl (mod_perl enabled) Server	62
5.3.2	Configuration of the servers	64
5.3.2.1	Basic httpd_docs Server's Configuration	64
5.3.2.2	Basic httpd_perl Server's Configuration	64
5.4	Running 2 webservers and squid in httpd accelerator mode	65
5.5	Running 1 webserver and squid in httpd accelerator mode	71
5.6	Building and Using mod_proxy	72
	mod_perl tutorial: mod_perl for ISPs	75
6	mod_perl for ISPs	75
6.1	What we will learn in this chapter	76
6.2	ISPs providing mod_perl services - a fantasy or reality.	76

mod_perl tutorial: Getting Help and Further Learning	79
7 Getting Help and Further Learning	79
7.1 What we will learn in this chapter	80
7.2 Getting help	80
7.3 Get help with mod_perl	80
7.4 Get help with Perl	82
7.5 Get help with Perl/CGI	82
7.6 Get help with Apache	83
7.7 Get help with DBI	83
7.8 Get help with Squid - Internet Object Cache	84