

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.

# 1 Getting Started Fast

# 1.1 mod\_perl in Four Slides

- Installation
- Configuration
- The “mod\_perl rules” Apache::Registry Scripts
- The “mod\_perl rules” Apache Perl Module

# 1.2 What is mod\_perl?

Solves numerous mod\_cgi shortcomings:

- Embedded Perl Interpreter -- no loading overhead
- Code compiled only once per process life -- no compilation overhead
- No forking per request -- process reuse
- Response processing is now reduced to running your code.
- Response times improve by a factor of 10 to 100

- A bigger size, but just a few processes can handle a much bigger load
- `mod_cgi` compatibility preserved (Apache::Registry and Apache::PerlRun modules)
- Persistent database connections

## Extended mod\_cgi's functionality:

- A complete Perl API added to the Apache core
- Handling of all phases of request processing in Perl.
- Writing complete Apache modules in Perl
- Complete server configuration in Perl.
- Numerous 3rd party modules are available

## Logistics:

- Developed by Doug MacEachern
- Licensed under the “Artistic License” as Perl itself.
- Home page <http://perl.apache.org>
- Mailing list: send email to [modperl-subscribe@apache.org](mailto:modperl-subscribe@apache.org) with the string “*subscribe modperl*” in the body.
- December 1999 -- 412000 mod\_perl hosts (according to <http://perl.apache.org/netcraft/>)

# 1.3 Installation

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

That's all!

# 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>
```

# 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_rules1.pl)

[http://localhost/perl/mod\\_perl\\_rules2.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:

```
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](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.
- Fortunately, there are many resources and lots of help freely available to you. See the last chapter of this tutorial for the help references.

;o)

# 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

- 10 minutes/10 commands installation

```
% 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
```

## 2.3 The Gory Details

- We can clearly separate the installation process into the following stages:
- Sources Configuration
- Building
- Testing
- Installation

## 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]
```

## 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

- Should be used to define the Apache's source tree directory.
- Non-Interactive

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

- Interactive. *Makefile.PL* makes an intelligent guess and suggests a directory with the highest version of Apache found there.
- "Configure mod\_perl with ../apache\_xxx/src ?"
- Answer 'y' to confirm the guess or offer your alternative.

## **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
```

- `PREP_HTTPD=1 == do not build httpd (make) in the apache source tree`

- 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.

## Callback Hooks

- By default, all callback hooks except for `PerlHandler` are turned off.
- 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 and options, 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

- Use `<USE_APACI=1>` parameter to 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 `'\'`).

## 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
```

- 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 the httpd binary or/and a separate library for mod\_perl and its modules.
- 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
```

- `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.
- if some tests fail, run them in the verbose mode
  - % **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
```

## 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

- Comprehensive Perl Archive Network [CPAN], is a repository of thousands Perl modules, scripts and documentation.
- See <http://cpan.org> for more info.
- 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.
- 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
```

- 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:

```
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
```

- 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 and run:  
  

```
% make install
```
- which will complete the installation by installing Apache headers and the binary at the appropriate directories.

## Getting control over CPAN shell

- 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`.

## Installing Bundle::Apache

```
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.

## 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:
- Username: `stas`
- Home directory: `/home/stas`

## 2.5.1 *Installing Perl Modules into a Directory of Choice*

- You aren't allowed to install into `/usr/lib/perl5`
- Solution: Install the modules under your home directory
- Choosing Dir Layout:
- The simplest approach is to simulate the 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

- 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.

- If you want to change the default dir layout or have an old MakeMaker version:

```
% 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 `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
```

- Use shortcuts if you need to specify the params explicitly

- Populate `~/.perl_dirs` with all the params:

```
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

- You have to tell your scripts where to find locally installed modules.
- How do you know the directory version numbers? Run:

```
% perl -V
```

- On my machine I see:

```
/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
```

- Therefore I add to the top of my scripts:

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

- I also add to my *.tcshrc* (in one line):

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

- Which is a must, so when you install new modules locally the already installed local modules will be found by Perl.
- bash users will use:

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

- There is more interesting discussion going in your handouts, we skip it here due to the lack of time.

## 2.5.3 Making a Local Apache Installation

- Almost no difference between the normal installation and this one.
- Only the target directory is the one that you are allowed to install to
- Using home directory as /  
• `/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:
  - `./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.

- **Must choose port > 1024**
- **Adding 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.**

## 2.5.4 Actual Local mod\_perl Enabled Apache Installation

- Combining the knowledge of the installing Perl modules and Apache locally we now can install mod\_perl locally
- Unpack the Apache and mod\_perl sources under your home dir.

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

- Configure and Install:

```
% 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.
- Now tell mod\_perl where the local Perl modules can be found:

```
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 forgotten.
- 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.

## 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.

## Solutions

- Reduce resources usage using the modules `Apache::SizeLimit`, `Apache::GTopLimit`.
- Ask your ISP whether they can setup a dedicated box 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.
- 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
- 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`?

- When you run `perl Makefile.PL`, a *Makefile* get created.
- This *Makefile* includes the same compilation options that were used to build Perl itself.
- (These are stored in *Config.pm* module and can be displayed with `Perl -V` command).
- So all these options are re-applied when compiling Perl modules.

- If you use a different compiler to build Perl extensions, chances are that the options that a different compiler uses won't be the same, or even worse, they might be interpreted in a completely different way.
- So the code either won't compile, would dump core file when used or behave in the most unexpected ways.
- Since `mod_perl` uses Perl, Apache and third party modules, and they all work together, it's a must thing to use the same compiler while building each of the components.
- But you shouldn't worry about when compiling Perl modules since Perl will choose what's right automatically, unless you override things.

- If you do this, you are on your own...
- If you compile a non-Perl component separately, you should worry to use the same compiler and the same options, used to build Perl.
- Hint: Take a look at the *Config.pm* module or the output if

```
perl -V.  
  
;0)
```

# 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 configuration process consists of two parts: Apache and `mod_perl` specific directives configuration.
- `httpd.conf`, `srm.conf`, and `access.conf` before Apache 1.3.4
- 1.3.4+ a single `httpd.conf` file
- By default located at `/usr/local/apache/conf`

## 3.3 Apache Configuration

- First make sure that the plain Apache configuration works.
- Do this in order to minimize the number of things that can go wrong with more complicated setup.

## 3.3.1 Configuration Directives

- If you didn't move Apache directories around, the installation program already has configured everything for you.

- To start the server use the `apachectl` utility

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

- Test the server by accessing:

<http://localhost/>

- If you did move things around, update the `httpd.conf`.

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

- Name of the machine as it's known to the external world

**ServerName** **www.example.com**

- **Port,**  
**Port** **8080**
- **User and Group**
- **Note that if started as *root* user the parent process will continue to run as *root***

- Children will run as the user and group you have specified.

**User nobody**

**Group nobody**

- More directives 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

- 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. YMMV.

## 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:
- if we define `Apache::Registry` to be the handler of `/perl` location.  
<http://www.example.com/perl/test.pl>  
`==> /usr/local/myproject/cgi/test.pl`
- and execute it as an Apache: `Registry` script

- 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*).
- 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:**

**ScriptAlias /cgi-bin/ /usr/local/myproject/cgi/**

- is equal to:

**Alias /cgi-bin/ /usr/local/myproject/cgi/  
SetHandler cgi-handler**

- where `SetHandler cgi-handler` invokes `mod_cgi`.

## 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:

```
PerlModule Apache::Registry  
<Location /perl>  
SetHandler perl-script  
PerlHandler Apache::Registry
```

```
Options ExecCGI
allow from all
PerlSendHeader On
</Location>
```

- 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

- Each module should be loaded before it gets used,
- `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()` 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*

- These are 11 stages of a request where the Apache API allows a module to step in and do something.
- Namely in that order:

**Post-Read-Request**  
**URI Translation**  
**Header Parsing**  
**Access Control**  
**Authentication**  
**Authorization**  
**MIME type checking**

**FixUp  
Response (Content phase)  
Logging  
Cleanup.**

- 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.

- 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**

## 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.

- After each request, this stack is cleared out.
- All handlers will be called unless a handler returns a status other than OK or DECLINED.
- 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 the existing *ENV* environment variables to your scripts.
- `PerlSetEnv` sets and passes the *ENV* environment variables to your scripts.
- You can access them in your scripts through `%ENV` (`($ENV{ "key" })`).

- `PerlSetVar` is very similar to `PerlSetEnv`, but you extract it with another method.

`PerlSetVar FOO BAR`

- or 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, 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 autoloading 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
  }
);

```

## 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 might write:

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

- For example, just because you have:

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

- 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;
```

## 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
$user = getpwnid($> || $>;
$Group = getgrgid($> || $>;

$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'` 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`, 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!!!**

## 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 workout 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,
- 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

Now will look at the following installations and discuss the pros and the cons of each one.

- **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:

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

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

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

- **Serving slow clients**

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

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

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

## 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.
- Using these large processes to serve static objects like images and html documents is overkill.

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

## The advantages:

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

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

## Relative URLs:

<http://www.example.com:8080/perl/example.pl>

- The above URL returns a page with relative links to images
- Where the images will be brought from?
- Of course from the mod\_perl heavy server  
<http://www.example.com:8080> --
- Solution: using fully qualified URIs

[s#/img/foo\ .gif#http://www.example.com/img/foo.gif#;](#)

## The disadvantages:

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

- **Two sets of scripts**

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

- **Logs merging**

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

- **Serving Slow Clients**

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

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

## 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.

### **The Advantages:**

- **All the advantages of the two 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.
- **Speed**
  - Reported to be about 5 times faster than plain apache serving static objects.

## **The Disadvantages:**

- **Inferior features set**
  - 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
- and `httpd_perl` listens to port 8080
- Now I am going to convince you that you **want** to use a proxy server (in the `http accelerator mode`).

## The advantages:

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

- **Output Buffering**

- The proxy server acts as a sort of output buffer for the dynamic content.
- The mod\_perl server sends the entire response to the proxy and is then free to deal with other requests.
- The proxy server is responsible for sending the response to the browser.
- So if the transfer is over a slow link, the mod\_perl server is not waiting around for the data to move.
- Using numbers is always more convincing :)

- 28.8 kbps connection => **3.6 kbytes/sec**.
- An average generated HTML page to be of **10kb**
- 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:  
$$(0.5 \text{ sec} * 10 \text{ kb}) / 3.6 \text{ kb/sec} \sim 6 \text{ sec}$$
- It will have to wait for another 6 secs (20kb/3.6kb)
- When it could serve 11 more dynamic requests in this time.

6 sec / 0.5 sec - 1 = 11

- But the generated pages are generally much bigger than 10Kb
- and users tend to open more than one browser at the same time
- Result: The waiting time can grow 10 times and more

- **Hiding Implementation Details**

- We are going to hide the details of the server's implementation.
- Users will never see ports in the URLs (more on that topic later).

- You can have a few boxes serving the requests, and only one serving as a front end, which spreads the jobs between the servers in a way that you can control.
- You can actually shut down a server, without the user even noticing, because the front end server will dispatch the jobs to other servers.
- (This is called a Load Ballancing and it's a pretty big issue, which will not be discussed here)

- **Security protection**

- For security reasons, using any httpd accelerator (or a proxy in httpd accelerator mode) is essential because you do not let your internal server get directly attacked by arbitrary packets from whomever.

- The httpd accelerator and internal server communicate in expected HTTP requests.
- This allows for only your public “bastion” accelerating www server to get hosed in a successful attack, while leaving your internal data safe.

## The disadvantages

- **Administration overhead**
  - You have another daemon to worry about, and while proxies are generally stable,
  - You have to make sure to prepare proper startup and shutdown scripts, which are run at boot and reboot as appropriate.
  - Also, you might want to set up the crontab to run a watchdog script.

- **Memory Usage**

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

- Have I succeeded in convincing you that you want a proxy server?
- If you are on a local area network (LAN), then the big benefit of the proxy buffering the output and feeding a slow client is gone.
- You are probably better off sticking with a straight mod\_perl server in this case.

# 4.5 Implementations of Proxy Servers

- Nowadays 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:

- **Fast Local Net**
  - 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*

- The difference in speed between apache's **mod\_proxy** and **squid** is not 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:

- **Setup Simplicity**
  - 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.
- **Hiding Internal Redirects**
  - 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/
```

- **Output Buffering**
  - It does `mod_perl` output buffering like squid does.

- **Object Caching**

- 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`.

- **Authentication**

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

- **Handling https requests**

- 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.

- **Stability**

- The latest (apache 1.3.11) 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 Getting Help and Further Learning

# 5.1 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**

## **5.2 Getting help**

1. Searchable Mailing list archive
2. The Books and Online Documentation
3. Mailing List as a last resort.

## 5.3 Get help with mod\_perl

- **mod\_perl home**

<http://perl.apache.org>

- **mod\_perl Garden project**

<http://modperl.sourceforge.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](mailto: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 Kherra at <http://perl.apache.org/tuning/> .

- **mod\_perl plugin reference guide**

by Doug MacEachern at [http://perl.apache.org/src/mod\\_perl.html](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](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](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/](http://www.perlreference.com/mod_perl/)

- **mod\_perl mailing list**

The Apache/Perl mailing list ([modperl@apache.org](mailto: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](mailto: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/>

## 5.4 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](http://world.std.com/~swmcd/steven/perl/module_mechanics.html)

- This page describes the mechanics of creating, compiling, releasing and maintaining Perl modules.

## 5.5 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://www.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)

## 5.6 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/>

## 5.7 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](http://perl.apache.org/src/mod_perl.html#PERSISTENT_DATABASE_CONNECTIONS)

## 5.8 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)