1 Getting Your Feet Wet

1.1 Description

This chapter gives you step-by-step instructions to get a basic statically-compiled mod_perl-enabled Apache server up and running. Having a running server allows you to experiment with mod_perl as you learn more about it.

(Of course, you'll be experimenting on a private machine, not on a production server, right?) The remainder of the guide, along with the documentation supplied with mod_perl, gives the detailed information required for fine-tuning your mod_perl-enabled server to deliver the best possible performance.

Although there are binary distributions of mod_perl-enabled Apache servers available for various platforms, we recommend that you always build mod_perl from source. It's simple to do (provided you have all the proper tools on your machine), and building from source circumvents possible problems with the binary distributions (such as the reported bugs with the RPM packages built for RedHat Linux).

The mod_perl installation that follows has been tested on many mainstream Unix and Linux platforms. Unless you're using a very non-standard system, you should have no problems when building the basic mod_perl server.

For Windows users, the simplest solution is to use the binary package. Windows users may skip to the Installing mod_perl for Windows.

1.2 Installing mod_perl in Three Steps

You can install mod_perl in three easy steps: Obtaining the source files required to build mod_perl, building mod_perl, and installing it.

Building mod_perl from source requires a machine with basic development tools. In particular, you will need an ANSI-compliant C compiler (such as *gcc*) and the *make* utility. All standard Unix-like distributions include these tools. If a required tool is not already installed, then use the package manager that is provided with the system (*rpm*, *apt*, *yast*, etc.) to install them.

A recent version of Perl is also required, at least version 5.004. Perl is available as an installable package, although most Unix-like distributions will have installed Perl by default. To check that the tools are available and learn about their version numbers, try:

```
% make -v
% gcc -v
% perl -v
```

If any of these responds with `Command not found`, it will need to be installed.

Once all the tools are in place the installation process can begin. Experienced Unix users will need no explanation of the commands that follow and can simply copy and paste them into a terminal window to get the server installed.

Acquire the source code distrubutions of Apache and mod_perl from the Internet, using your favorite web browser or one of the command line clients like *wget*, *lwp-download*, etc. These two distributions are available from *http://www.apache.org/dist/httpd/* and *http://apache.org/dist/perl/*.

Remember that mod_perl 1.0 works only with Apache 1.3, and mod_perl 2.0 requires Apache 2.0. In this chapter we talk about mod_perl 1.0/Apache 1.3, hence the packages that you want are named *apache_1.3.xx.tar.gz* and *mod_perl-1.xx.tar.gz*, where *xx* should be replaced with the real version numbers of mod_perl and Apache.

Move the downloaded packages into a directory of your choice, *e.g.*, */home/stas/src/*, proceed with the described steps and you will have mod_perl installed:

```
% cd /home/stas/src
% tar -zvxf apache_1.3.xx.tar.gz
% tar -zvxf mod_perl-1.xx.tar.gz
% cd mod_perl-1.xx
% perl Makefile.PL APACHE_SRC=../apache_1.3.xx/src \
    APACHE_PREFIX=/home/httpd DO_HTTPD=1 USE_APACI=1 EVERYTHING=1
% make && make test
% su
# make install
```

That's all!

All that remains is to add a few configuration lines to the Apache configuration file, (*/usr/local/apache/conf/httpd.conf*), start the server, and enjoy mod_perl.

The following detailed explanation of each step should help you solve any problems that may have arisen when executing the commands above.

1.3 Installing mod_perl for Unix Platforms

Here is a more detailed explanation of the installation process, with each step explained in detail and with some troubleshooting advice. If the build worked and you are in a hurry to boot your new *httpd*, you may skip to the next section, talking about configuration of the server.

Before installing Apache and mod_perl, you usually have to become *root* so that the files can be installed in a protected area. A user who does not have *root* access, however, can still install all files under their home directory. The proper approach is to build Apache in an unprivileged location and then only use *root* access to install it. We will talk about the nuances of this approach in the dedicated installation process chapter.

1.3.1 Obtaining and Unpacking the Source Code

The first step is to obtain the source code distributions of Apache, available from http://www.apache.org/dist/httpd/ and mod_perl, available from http://apache.org/dist/perl/. Of course you can use your favorite mirror sites to get these distributions.

Even if you have the Apache server running on your machine, usually you need to download its source distribution, because you need to re-build it from scratch with mod_perl.

The source distributions of Apache and mod_perl should be downloaded into directory of your choice. For the sake of consistency, we assume throughout the guide that all builds are being done in the */home/stas/src* directory. However, using a different location is perfectly possible and merely requires changing this part of the path in the examples to the actual path being used.

The next step is to move to the directory with the source files:

```
% cd /home/stas/src
```

Uncompress and untar both sources. GNU *tar* allows this using a single command per file:

```
% tar -zvxf apache_1.3.xx.tar.gz
% tar -zvxf mod_perl-1.xx.tar.gz
```

For non-GNU *tar*'s, you may need to do this with two steps (which you can combine via a pipe):

```
% gzip -dc apache_1.3.xx.tar.gz | tar -xvf -
% gzip -dc mod_perl-1.xx.tar.gz | tar -xvf -
```

Linux distributions supply `tar` and `gzip` and install them by default; for other systems these utilities should be obtained from *http://www.gnu.org/* or other sources, and installed--the GNU versions are available for every platform that Apache supports.

1.3.2 Building mod_perl

Move into the */home/stas/src/mod_perl-1.xx/* source distribution directory:

```
% cd mod_perl-1.xx
```

The next step is to create the *Makefile*. This step is no different in principle from the creation of the *Makefile* for any other Perl module.

```
% perl Makefile.PL APACHE_SRC=../apache_1.3.xx/src \
  DO_HTTPD=1 USE_APACI=1 EVERYTHING=1
```

(Replace *x.x.x* with the Apache distribution version number.)

mod_perl accepts a variety of parameters. The options specified above will enable almost everything that mod_perl offers. There are many other options for fine-tuning mod_perl to suit particular circumstances; these are explained in detail in the dedicated installation process chapter.

Running *Makefile.PL* will cause Perl to check for prerequisites and identify any required software packages which are missing. If it reports missing Perl packages, these will have to be installed before proceeding. They are all available from CPAN (*http://cpan.org/*) and can be easily downloaded and installed.

An advantage to installing mod_perl with the help of the `CPAN.pm` module is that all the missing modules will be installed with the `Bundle::Apache` bundle:

```
% perl -MCPAN -e 'install("Bundle::Apache")'
```

We will talk about using `CPAN.pm` in-depth in the installation process chapter.

Running *Makefile.PL* also transparently executes the *./configure* script from Apache's source distribution directory, which prepares the Apache build configuration files. If parameters must be passed to Apache's `./configure` script, they can be passed as options to *Makefile.PL.*

The *httpd* executable can now be built by using the `make` utility. (Note that the current working directory is still */home/stas/src/mod_perl-1.xx/*):

```
% make
```

This command prepares the mod_perl extension files, installs them in the Apache source tree and builds the *httpd* executable (the web server itself) by compiling all the required files. Upon completion of the *make* process, the working directory is restored to */home/stas/src/mod_perl-1.xx/.*

Running `make test` will execute various mod_perl tests on the freshly built *httpd* executable.

```
% make test
```

This command starts the server on a non-standard port (8529) and tests whether all parts of the built server function correctly. The process will report anything that does not work properly.

1.3.3 Installing mod_perl

Running `make install` completes the installation process of mod_perl by installing all the Perl files required for mod_perl to run--and, of course, the server documentation (man pages). Typically, you need to be *root* to have permission to do this, but another user account can be used if the appropriate options were set on the `perl Makefile.PL` command line. To become *root*, use the *su* command.

```
% su
# make install
```

If you have the proper permission, you might also chain all three *make* commands into a single command line:

```
# make && make test && make install
```

`&&` in shell program is similar to Perl's `&&`. Each section of the statement will be executed left to right, untill all sections will be executed and return true (success) or one of them will return false (failure).

The single-line version simplifies the installation, since there is no need to wait for each command to complete before starting the next one. Of course, if you need to become *root* in order to run *make install*, you'll either need to run it as a separate command or become *root* before running the single-line version.

If the all-in-one approach is chosen and any of the make commands fail, execution will stop at that point. For example if make alone fails then make test and make install will not be attempted; similarly if make test fails then make install will not be attempted.

Finally, change to the Apache source distribution directory and run make install to create the Apache directory tree and install Apache's header files (*.h), default configuration files (*.conf), the *httpd* executable, and a few other programs.

```
# cd ../apache_1.3.xx
# make install
```

Note that, as with a plain Apache installation, any configuration files left from a previous installation will not be overwritten by this process. So there is no need to backup the previously working configuration files before the installation, although backing up is never unwise.

At the end of the *make install* process, it will list the path to the apachectl utility that you can use to start and stop the server, and the path to the installed configuration files. It is important to write down these paths since they will frequently be needed when maintaining and configuring Apache. On our machines these two important paths are:

```
/usr/local/apache/bin/apachectl
/usr/local/apache/conf/httpd.conf
```

The mod_perl Apache server is now built and installed. All that needs to be done before it can be run is to edit the configuration file *httpd.conf* and write a little test script.

1.3.4 Configuring and Starting the mod_perl Server

The first thing to do is ensure that Apache was built correctly and that it can serve plain HTML files. This helps to minimize the number of possible problem areas: once you have confirmed that Apache can serve plain HTML files, you know that any problems with mod_perl are with mod_perl itself.

Apache should be configured just the same as when it did not have mod_perl. Set the Port, User, Group, ErrorLog and other directives in the *httpd.conf* file. Use the defaults as suggested, customizing only when necessary. Values that will probably need to be customized are ServerName, Port, User, Group, ServerAdmin, DocumentRoot and a few others. There are helpful hints preceding each directive in the configuration files themselves, with further information in Apache's documentation. Follow the advice in the files and documentation if in doubt.

When the configuration file has been edited, it is time to start the server. One of the ways to start and stop the server is to use the apachectl utility. This can be used to start the server with:

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

And stop it with:

```
# /usr/local/apache/bin/apachectl stop
```

Note that if the server is going to listen on port `80` or another privileged port (Any port with a number less than 1024 can be accessed only by the programs running as *root*.), the user executing `apachectl` must be *root*.

After the server has started, check in the *error_log* file (*/usr/local/apache/logs/error_log* by default) to see if the server has indeed started. Do not rely on the status `apachectl` reports. The *error_log* should contain something like the following:

```
[Thu Jun 22 17:14:07 2000] [notice] Apache/1.3.12 (Unix)
mod_perl/1.24 configured -- resuming normal operations
```

Now point the browser to *http://localhost/* or *http://example.name/* as configured with the `ServerName` directive. If the `Port` directive has been set with a value different from `80`, add this port number at the end of the server name. For example, if the port is 8080, test the server with *http://localhost:8080/* or *http://example.com:8080/*. The infamous *"It worked"* page should appear in the browser, which is an `index.html` file that `make  install` in the Apache source tree installs automatically. If this page does not appear, something went wrong and the contents of the *error_log* file should be checked. The path of the error log file is specified in the `ErrorLog` directive section in *httpd.conf*.

If everything works as expected, shut the server down, open *httpd.conf* with a plain text editor, and scroll to the end of the file. The mod_perl configuration directives are added to the end of *httpd.conf* by convention. It is possible to place mod_perl's configuration directives anywhere in *httpd.conf*, but adding them at the end seems to work best in practice.

Assuming that all the scripts that should be executed by the mod_perl enabled server are located in the */home/stas/modperl* directory, add the following configuration directives:

```
Alias /perl/ /home/stas/modperl/

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

Save the modified file.

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

1.4 Installing mod_perl for Windows

Apache runs on many flavors of Unix and Unix-like operating systems. Version 1.3 introduced a port to the Windows family of operating systems, often named `Win32` after the name of the common API. Because of many deep differences between Unix and Windows, the Win32 port of Apache is still branded as beta quality, because it hasn't yet reached the stability and performance of the native Unix counterpart.

Another hindrance to using mod_perl 1.0 on Windows is that current versions of Perl are not thread-safe on Win32. As a consequence, mod_perl calls to the embedded Perl interpreter must be serialized, i.e. executed one at a time. See the discussion on multithreading on Win32 mod_perl 1.xx for details. This situation changes with Apache/mod_perl 2.0, which is based on a multi-process/multi-thread approach using a native Win32 threads implementation - see the mod_perl 2 overview for more details, and the discussion of modperl 2.0 in Win32 on getting modperl-2 for Win32 in particular.

Building mod_perl from source on Windows is a bit of a challenge. Development tools such as a C compiler are not bundled with the operating system, and most users expect a point-and-click installation as with most Windows software. Additionally, all software packages need to be built with the same compiler and compile options. This means building Perl, Apache, and mod_perl from source, quite a daunting task. For details on building mod_perl on Windows, see the documentation for modperl 1.0 in Win32 or modperl 2.0 in Win32.

For those who prefer binary distributions, there are a number of alternatives. For mod_perl 1.0, one can obtain either mod_perl 1.0 PPM packages, suitable for use with ActivePerl's `ppm` utility, or else mod_perl 1.0 all-in-one packages containing binaries of Perl and Apache, including mod_perl. For mod_perl 2.0, similar mod_perl 2.0 PPM packages and mod_perl 2.0 all-in-one packages are available.

1.5 Preparing the Scripts Directory

Now you have to select a directory where all the mod_perl scripts and modules will be placed to. We usually use create a directory *modperl* under our home directory, e.g., */home/stas/modperl*, for this purpose. Your mileage may vary.

First create this directory if it doesn't yet exist.

```
% mkdir /home/stas/modperl
```

What about file permissions? Remember that when scripts are executed from a shell, they are being executed with permissions of the user's account. Usually you want to have a read, write and execute access for yourself, but only read and execute permissions for the server. When the scripts are run by Apache, however, the server needs to be able to read and execute them. Apache runs under an account specified by the `User` directive, typically *nobody*. If you modify the `User` directive to run the server under your username, e.g.,

```
User stas
```

the permissions on all files and directories should usually be `rwx------`, so we can set the directory permissions to:

```
% chmod 0700 /home/stas/modperl
```

Now, no-one, but you and the server can access the files in this directory. You should set the same permissions for all the files you place under this directory. (You don't need to set the *x* bit for files that aren't going to be executed, for those mode `0600` would be sufficient.)

If the server is running under account *nobody*, you have to set the permissions to `rwxr-xr-x` or `0755` for your files and directories, which is insecure since other users on the same machine can read your files.

```
# chmod 0755  /home/stas/modperl
```

If you aren't running the server with your username, you have to set these permissions for all the files created under this directory, so Apache can read and execute these.

If you need to have an Apache write files you have to set the file permissions to `rwxrwxrwx` or `0777` which is very undesirable, since any user on the same machine can read and write your files. If this is the case, you should run the server under your username, and then only you and the server have a write access to your files. (Assuming of course that other users have no access to your server, since if they do, they can access your files through this server.)

In the following examples we assume that you run the server under your username, and hence we set the scripts permissions to `0700`.

1.6 A Sample Apache::Registry Script

One of mod_perl's benefits is that it can run existing CGI scripts written in Perl which were previously used under mod_cgi (the standard Apache CGI handler). Indeed mod_perl can be used for running CGI scripts without taking advantage of any of mod_perl's special features, while getting the benefit of the potentially huge performance boost. Here is an example of a very simple CGI-style mod_perl script:

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

Save this script in the */home/stas/modperl/mod_perl_rules1.pl* file. Notice that the `#!` line (colloquially known as the *shebang* line) is not needed with mod_perl, although having one causes no problems, as can be seen here:

```
mod_perl_rules1.pl
------------------
#!/usr/bin/perl
print "Content-type: text/plain\n\n";
print "mod_perl rules!\n";
```

Now make the script executable and readable by the server, as explained in the previous section.

```
# chmod 0700   /home/stas/modperl/mod_perl_rules1.pl
```

The *mod_perl_rules1.pl* script can be tested from the command line, since it is essentially a regular Perl script.

```
% perl /home/stas/modperl/mod_perl_rules1.pl
```

This should produce the following output:

```
Content-type: text/plain

mod_perl rules!
```

Make sure the server is running and issue these requests using a browser:

```
http://localhost/perl/mod_perl_rules1.pl
```

If you see it--**congratulations**! You have a working mod_perl server.

If something went wrong, go through the installation process again, making sure that none of the steps are missed and that each is completed successfully. If this does not solve the problem, the installation chapter will attempt to salvage the situation.

If the port being used is not 80, for example 8080, then the port number should be included in the URL:

```
http://localhost:8080/perl/mod_perl_rules1.pl
```

The `localhost` approach will work only if the browser is running on the same machine as the server. If not, use the real server name for this test. For example:

```
http://example.com/perl/mod_perl_rules1.pl
```

If there is any problem please refer to the *error_log* file for the error messages.

Jumping a little bit ahead, we would like to show the same CGI script written using mod_perl API:

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

mod_perl API needs a request object $r to communicate with Apache. The script gets this object first thing, after that it uses the object to send the HTTP header and print the irrefutable fact about mod_perl's coolness.

This script generates the same output as the previous one.

As you can see it's not much harder to write your code in mod_perl API, all you need to know is the API, but the concepts are the same. As we will show in the later chapters, usually you will want to use mod_perl API for a better performance or when you need a functionality that plain Perl API doesn't provide.

1.6.1 Porting Existing CGI Scripts to run under mod_perl

Now it is time to move any existing CGI scripts from the */somewhere/cgi-bin* directory to */home/stas/modperl*. Once moved they should run much faster when requested from the newly configured base URL (*/perl/*). For example, a CGI script called *test.pl* that was previously accessed as */cgi-bin/test.pl* can now be accessed under mod_perl as */perl/test.pl*.

Some of the scripts might not work immediately and may require some minor tweaking or even a partial rewrite to work properly with mod_perl. We will talk in-depth about these things in the Coding chapter. Most scripts that have been written with care and especially developed with enabled warnings and the `strict` pragma will probably work without any modifications at all.

A quick solution that avoids most rewriting or editing of existing scripts that do not run properly under `Apache::Registry` is to run them under `Apache::PerlRun`. This can be achieved by simply replacing `Apache::Registry` with `Apache::PerlRun` in *httpd.conf*. Put the following configuration directives instead in *httpd.conf* and restart the server:

```
Alias /perl/ /home/stas/modperl/
PerlModule Apache::PerlRun
<Location /perl/>
    SetHandler perl-script
    PerlHandler Apache::PerlRun
    Options ExecCGI
    PerlSendHeader On
    allow from all
</Location>
```

Almost every script should now run without problems; the few exceptions will almost certainly be due to the few minor limitations that mod_perl or its handlers have, but these are all solvable and covered in Coding chapter.

`Apache::PerlRun` is usually useful in the transition period, while the scripts are being cleaned up to run properly under `Apache::Registry`. Though it gives you a significant speedup over mod_cgi, it's still not as fast as `Apache::Registry` and mod_perl handlers.

1.7 A Simple Apache Perl Content Handler

As we mentioned in the beginning of this chapter, mod_perl lets you run both scripts and handlers. The previous example showed a script, since that is probably the most familiar approach to web programming, but the more advanced use of mod_perl involves writing handlers. Have no fear: writing handlers is almost as easy as writing scripts and offers a level of access to Apache's internals that is simply not possible with conventional CGI scripts.

To create a mod_perl handler module, all that is necessary is to wrap the code that would have been the body of a script into a `handler` subroutine, add a statement to return the status to the server when the subroutine has successfully completed, and add a package declaration at the top of the code.

Just as with scripts, the familiar CGI API may be used:

```
ModPerl/Rules1.pm
-----------------
package ModPerl::Rules1;
use Apache::Constants qw(:common);

sub handler {
    print "Content-type: text/plain\n\n";
    print "mod_perl rules!\n";
    return OK; # We must return a status to mod_perl
}
1; # This is a perl module so we must return true to perl
```

Alternatively, the mod_perl API can be used. This API provides almost complete access to the Apache core. In the simple example shown, using either approach is fine, but when lower level access to Apache is required the mod_perl API must be used.

```
ModPerl/Rules2.pm
----------------
package ModPerl::Rules2;
use Apache::Constants qw(:common);

sub handler {
    my $r = shift;
    $r->send_http_header('text/plain');
    $r->print("mod_perl rules!\n");
    return OK; # We must return a status to mod_perl
}
1; # This is a perl module so we must return true to perl
```

Create a directory called *ModPerl* under one of the directories in @INC (e.g. under */usr/lib/perl5/site_perl/5.005/*), and put *Rules1.pm* and *Rules2.pm* into it (Note that you will need a *root* access in order to do that.). The files should include the code from the above examples. To find out what the @INC directories are, execute:

```
% perl -le 'print join "\n", @INC'
```

On our machine it reports:

```
/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
.
```

So on our machine, we might place the files in the directory */usr/lib/perl5/site_perl/5.005/ModPerl*. By default when you work as *root* the files are created with permissions allowing everybody to read them, so here we don't have to adjust the file permissions, since the server only needs to be able to read those.

Now add the following snippet to */usr/local/apache/conf/httpd.conf* to configure mod_perl to execute the ModPerl::Rules::handler subroutine whenever a request to *mod_perl_rules1* is made:

```
PerlModule ModPerl::Rules1
<Location /mod_perl_rules1>
    SetHandler perl-script
    PerlHandler ModPerl::Rules1
    PerlSendHeader On
</Location>
```

Now issue a request to:

```
http://localhost/mod_perl_rules1
```

and just as with the *mod_perl_rules.pl* scripts,

```
mod_perl rules!
```

should be rendered as the response. (Don't forget to include the port number if not using port 80; from now on we will assume this is done, e.g. http://localhost:8080/mod_perl_rules1.)

To test the second module `ModPerl::Rules2` add a similar configuration, while replacing all 1's with 2's:

```
PerlModule ModPerl::Rules2
<Location /mod_perl_rules2>
    SetHandler perl-script
    PerlHandler ModPerl::Rules2
    PerlSendHeader On
</Location>
```

As we will see later in Configuration chapter you can remove the `PerlSendHeader` directive for this particular module.

And to test use the URI:

```
http://localhost/mod_perl_rules2
```

You should see the same response from the server as we have seen when issuing a request for the former mod_perl handler.

1.8 Is This All we Need to Know About mod_perl?

Obviously the next question is: *"Is this all I need to know about mod_perl?"*.

The answer is: "Yes and No".

The *Yes* part:

- Just like with Perl, really cool stuff can be written even with very little mod_perl knowledge. With the basic unoptimized setup presented in this chapter, visitor counters and guest books and any other CGI scripts will run much faster and amaze friends and colleagues, usually without changing a single line of code.

The *No* part:

- A 50 times improvement in guest book response times is great--but when deploying a very heavy service with thousands of concurrent users, a delay of even a few milliseconds might lose a customer, and probably many of them.

Of course when testing a single script with the developer as the only user, squeezing yet another millisecond from the response time seems unimportant. But it becomes a real issue when these milliseconds add up at the production site, with hundreds or thousands of users concurrently generating requests to various scripts on the site. Users are not merciful nowadays. If there is another site that provides the same kind but a significantly faster service, chances are that users will switch to the competing site.

Testing scripts on an unloaded machine can be very misleading, Everything might seem so perfect. But when they are moved into a production machine, things do not behave as well as they did on the development box. For example, the production machine may run out of memory on very busy services. In the performance tuning chapter it will be explained how to optimize code to use less memory and how to make as much memory as possible shared.

Debugging is something that some developers prefer not to think about, since the process can be very tedious at times. Learning how to make the debugging process simpler and efficient is essential for web programmers. This task can be difficult enough when debugging CGI scripts, but it can be even more complicated with mod_perl. The Debugging Chapter explains how to approach debugging in the mod_perl environment.

mod_perl has many features unavailable under mod_cgi when working with databases. Amongst others the most important are persistent database connections. Persistent database connections require a slightly different approach and this is explained in the Databases chapter.

Most web services, especially those which are aimed at an international audience, must run non-stop 24x7. But at the same time new scripts may need to be added, old ones removed, and the server software will need upgrades and security fixes. And if the server goes down, fast recovery is essential. These issues are considered in the Controlling your server chapter.

Finally, the most important aspect of mod_perl is the Apache Perl API, which allows intervention at any or every stage of request processing. This provides incredible flexibility, allowing the creation of scripts and processes which would simply be impossible with mod_cgi.

There are many more things to learn about mod_perl and web programming in general. The rest of this guide will attempt to provide as much information as possible about these and other related matters.

1.9 References

- CPAN is the Comprehensive Perl Archive Network. Comprehensive: its aim is to contain all the Perl material you will need. Archive: close to a gigabyte in size at the time of this writing. Network: CPAN is mirrored at more than one hundred sites around the world.

The CPAN home page: http://cpan.org/

- The libwww-perl distribution is a collection of Perl modules and programs which provide a simple and consistent programming interface (API) to the World Wide Web. The main focus of the library is to provide classes and functions that facilitate writing WWW clients, thus libwww-perl is said to be a WWW client library. The library also contains modules that are of more general use, as well as some useful programs.

  The libwww-perl home page: http://www.linpro.no/lwp/

Table of Contents: