CHAPTER 3

Installing mod_perl

In Chapter 2, we presented a basic mod_perl installation. In this chapter, we will talk about various ways in which mod_perl can be installed (using a variety of installation parameters), as well as prepackaged binary installations, and more.

Chapter 2 showed you the following commands to build and install a basic mod_perl-enabled Apache server on almost any standard flavor of Unix.

First, download *http://www.apache.org/dist/httpd/apache_1.3.xx.tar.gz* and *http://perl.apache.org/dist/mod_perl-1.xx.tar.gz*. Then, issue the following commands:

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

As usual, replace *1.xx* and *1.3.xx* with the real version numbers of mod_perl and Apache, respectively.

You can then add a few configuration lines to *httpd.conf* (the Apache configuration file), start the server, and enjoy mod_perl. This should work just fine. Why, then, are you now reading a 50-page chapter on installing mod_perl?

You're reading this chapter for the same reason you bought this book. Sure, the instructions above will get you a working version of mod_perl. But the average reader of this book won't want to stop there. If you're using mod_perl, it's because you want to improve the performance of your web server. And when you're concerned with performance, you're always looking for ways to eke a little bit more out of your server. In essence, that's what this book is about: getting the most out of your mod_perl-enabled Apache server. And it all starts at the beginning, with the installation of the software.

In the basic mod_perl installation, the parameter `EVERYTHING=1` enables a lot of options for you, whether you actually need them or not. You may want to enable only the required options, to squeeze even more juice out of mod_perl. You may want to build mod_perl as a loadable object instead of compiling it into Apache, so that it can be upgraded without rebuilding Apache itself. You may also want to install other Apache components, such as PHP or mod_ssl, alongside mod_perl.

To accomplish any of these tasks, you will need to understand various techniques for mod_perl configuration and building. You need to know what configuration parameters are available to you and when and how to use them.

As with Perl, in mod_perl simple things are simple. But when you need to accomplish more complicated tasks, you may have to invest some time to gain a deeper understanding of the process. In this chapter, we will take the following route. We'll start with a detailed explanation of the four stages of the mod_perl installation process, then continue on with the different paths each installation might take according to your goal, followed by a few copy-and-paste real-world installation scenarios. Toward the end of the chapter we will show you various approaches that might make the installation easier, by automating most of the steps. Finally, we'll cover some of the general issues that new users might stumble on while installing mod_perl.

Configuring the Source

Before building and installing mod_perl you will have to configure it, as you would configure any other Perl module:

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

Perl Installation Requirements

Make sure you have Perl installed! Use the latest stable version, if possible. To determine your version of Perl, run the following command on the command line:

```
panic% perl -v
```

You will need at least Perl Version 5.004. If you don't have it, install it. Follow the instructions in the distribution's *INSTALL* file. The only thing to watch for is that during the configuration stage (while running *./Configure*) you make sure you can dynamically load Perl module extensions. That is, answer `YES` to the following question:

```
Do you wish to use dynamic loading? [y]
```

In this section, we will explain each of the parameters accepted by the *Makefile.PL* file for mod_perl First, however, lets talk about how the mod_perl configuration dovetails with Apache's configuration. The source configuration mechanism in Apache 1.3 provides four major features (which of course are available to mod_perl):

- Apache modules can use per-module configuration scripts to link themselves into the Apache configuration process. This feature lets you automatically adjust the configuration and build parameters from the Apache module sources. It is triggered by *ConfigStart/ConfigEnd* sections inside *modulename.module* files (e.g., see the file *libperl.module* in the mod_perl distribution).
- The *APache AutoConf-style Interface* (APACI) is the top-level *configure* script from Apache 1.3; it provides a GNU Autoconf-style interface to the Apache configuration process. APACI is useful for configuring the source tree without manually editing any *src/Configuration* files. Any parameterization can be done via command-line options to the *configure* script. Internally, this is just a nifty wrapper over the old *src/Configure* script.

  Since Apache 1.3, APACI is the best way to install mod_perl as cleanly as possible. However, the complete Apache 1.3 source configuration mechanism is available only under Unix at this writing—it doesn't work on Win32.
- *Dynamic shared object* (DSO) support is one of the most interesting features in Apache 1.3. It allows Apache modules to be built as so-called DSOs (usually named *modulename.so*), which can be loaded via the `LoadModule` directive in Apache's *httpd.conf* file. The benefit is that the modules become part of the *httpd* executable only on demand; they aren't loaded into the address space of the *httpd* executable until the user asks for them to be. The benefits of DSO support are most evident in relation to memory consumption and added flexibility (in that you won't have to recompile your *httpd* each time you want to add, remove, or upgrade a module).

  The DSO mechanism is provided by Apache's `mod_so` module, which needs to be compiled into the *httpd* binary with:

  ```
  panic% ./configure --enable-module=so
  ```

  The usage of any *--enable-shared* option automatically implies an *--enable-module=so* option, because the bootstrapping module `mod_so` is always needed for DSO support. So if, for example, you want the module `mod_dir` to be built as a DSO, you can write:

  ```
  panic% ./configure --enable-shared=dir
  ```

  and the DSO support will be added automatically.
- The *APache eXtension Support* tool (`APXS`) is a tool from Apache 1.3 that can be used to build an Apache module as a DSO even outside the Apache source tree. `APXS` is to Apache what `MakeMaker` and `XS` are to Perl.[*] It knows the platform-dependent build parameters for making DSO files and provides an easy way to run the build commands with them.

[*] `MakeMaker` allows easy, automatic configuration, building, testing, and installation of Perl modules, while `XS` allows you to call functions implemented in C/C++ from Perl code.

Pros and Cons of Building mod_perl as a DSO

As of Apache 1.3, the configuration system supports two optional features for taking advantage of the modular DSO approach: compilation of the Apache core program into a DSO library for shared usage, and compilation of the Apache modules into DSO files for explicit loading at runtime.

Should you build mod_perl as a DSO? Let's study the pros and cons of this installation method, so you can decide for yourself.

Pros:

- The server package is more flexible because the actual server executable can be assembled at runtime via `LoadModule` configuration commands in *httpd.conf* instead of via `AddModule` commands in the *Configuration* file at build time. This allows you to run different server instances (e.g., standard and SSL servers, or servers with and without mod_perl) with only one Apache installation; the only thing you need is different configuration files (or, by judicious use of `IfDefine`, different startup scripts).
- The server package can easily be extended with third-party modules even after installation. This is especially helpful for vendor package maintainers who can create an Apache core package and additional packages containing extensions such as PHP, mod_perl, mod_fastcgi, etc.
- DSO support allows easier Apache module prototyping, because with the DSO/`APXS` pair you can work outside the Apache source tree and need only an *apxs -i* command followed by an *apachectl restart* to bring a new version of your currently developed module into the running Apache server.

Cons:

- The DSO mechanism cannot be used on every platform, because not all operating systems support shared libraries.
- The server starts up approximately 20% slower because of the overhead of the symbol-resolving the Unix loader now has to do.
- The server runs approximately 5% slower on some platforms, because position-independent code (PIC) sometimes needs complicated assembler tricks for relative addressing, which are not necessarily as fast as those for absolute addressing.

—continued—

- Because DSO modules cannot be linked against other DSO-based libraries (*ld -lfoo*) on all platforms (for instance, *a.out*-based platforms usually don't provide this functionality, while ELF-based platforms do), you cannot use the DSO mechanism for all types of modules. In other words, modules compiled as DSO files are restricted to use symbols only from the Apache core, from the C library (*libc*) and from any other dynamic or static libraries used by the Apache core, or from static library archives (*libfoo.a*) containing position-independent code. The only way you can use other code is to either make sure the Apache core itself already contains a reference to it, load the code yourself via `dlopen( )`, or enable the `SHARED_CHAIN` rule while building Apache (if your platform supports linking DSO files against DSO libraries). This, however, won't be of much significance to you if you're writing modules only in Perl.
- Under some platforms (e.g., many SVR4 systems), there is no way to force the linker to export all global symbols for use in DSOs when linking the Apache *httpd* executable program. But without the visibility of the Apache core symbols, no standard Apache module could be used as a DSO. The only workaround here is to use the `SHARED_CORE` feature, because in this way the global symbols are forced to be exported. As a consequence, the Apache *src/Configure* script automatically enforces `SHARED_CORE` on these platforms when DSO features are used in the *Configuration* file or on the *configure* command line.

Together, these four features provide a way to integrate mod_perl into Apache in a very clean and smooth way. No patching of the Apache source tree is usually required, and for `APXS` support, not even the Apache source tree is needed.

To benefit from the above features, a hybrid build environment was created for the Apache side of mod_perl. See the section entitled "Installation Scenarios for Standalone mod_perl," later in this chapter, for details.

Once the overview of the four building steps is complete, we will return to each of the above configuration mechanisms when describing different installation passes.

Controlling the Build Process

The configuration stage of the build is performed by the command *perl Makefile.PL*, which accepts various parameters. This section covers all of the configuration parameters, grouped by their functionality.

Of course, you should keep in mind that these options are cumulative. We display only one or two options being used at once, but you should use the ones you want to enable all at once, in one call to *perl Makefile.PL*.

`APACHE_SRC`, `DO_HTTPD`, `NO_HTTPD`, `PREP_HTTPD`

These four parameters are tightly interconnected, as they control the way in which the Apache source is handled.

Typically, when you want mod_perl to be compiled statically with Apache without adding any extra components, you specify the location of the Apache source tree using the `APACHE_SRC` parameter and use the `DO_HTTPD=1` parameter to tell the installation script to build the *httpd* executable:

```
panic% perl Makefile.PL APACHE_SRC=../apache_1.3.xx/src DO_HTTPD=1
```

If no `APACHE_SRC` is specified, *Makefile.PL* makes an intelligent guess by looking at the directories at the same level as the mod_perl sources and suggesting a directory with the highest version of Apache found there.

By default, the configuration process will ask you to confirm whether the location of the source tree is correct before continuing. If you use `DO_HTTPD=1` or `NO_HTTPD=1`, the first Apache source tree found or the one you specified will be used for the rest of the build process.

If you don't use `DO_HTTPD=1`, you will be prompted by the following question:

```
Shall I build httpd in ../apache_1.3.xx/src for you?
```

Note that if you set `DO_HTTPD=1` but do not use `APACHE_SRC=../apache_1.3.xx/src`, the first Apache source tree found will be used to configure and build against. Therefore, you should always use an explicit `APACHE_SRC` parameter, to avoid confusion.

If you don't want to build the *httpd* in the Apache source tree because you might need to add extra third-party modules, you should use `NO_HTTPD=1` instead of `DO_HTTPD=1`. This option will install all the files that are needed to build mod_perl in the Apache source tree, but it will not build *httpd* itself.

`PREP_HTTPD=1` is similar to `NO_HTTPD=1`, but if you set this parameter you will be asked to confirm the location of the Apache source directory even if you have specified the `APACHE_SRC` parameter.

If you choose not to build the binary, you will have to do that manually. Building an *httpd* binary is covered in an upcoming section. In any case, you will need to run *make install* in the mod_perl source tree so the Perl side of mod_perl will be installed. Note that mod_perl's *make test* won't work until you have built the server.

`APACHE_HEADER_INSTALL`

When Apache and mod_perl are installed, you may need to build other Perl modules that use Apache C functions, such as `HTML::Embperl` or `Apache::Peek`. These modules usually will fail to build if Apache header files aren't installed in the Perl tree. By default, the Apache source header files are installed into the *$Config{sitearchexp}/auto/Apache/include* directory.[*] If you don't want or need

[*] *%Config* is defined in the *Config.pm* file in your Perl installation.

these headers to be installed, you can change this behavior by using the `APACHE_HEADER_INSTALL=0` parameter.

`USE_APACI`

The `USE_APACI` parameter tells mod_perl to configure Apache using the flexible APACI. The alternative is the older system, which required a file named *src/Configuration* to be edited manually. To enable APACI, use:

```
panic% perl Makefile.PL USE_APACI=1
```

`APACI_ARGS`

When you use the `USE_APACI=1` parameter, you can tell *Makefile.PL* to pass any arguments you want to the Apache *./configure* utility. For example:

```
panic% perl Makefile.PL USE_APACI=1 \
    APACI_ARGS='--sbindir=/home/httpd/httpd_perl/sbin, \
        --sysconfdir=/home/httpd/httpd_perl/etc'
```

Note that the `APACI_ARGS` argument must be passed as a single long line if you work with a C-style shell (such as *csh* or *tcsh*), as those shells seem to corrupt multi-lined values enclosed inside single quotes.

Of course, if you want the default Apache directory layout but a different root directory (*/home/httpd/httpd_perl/*, in our case), the following is the simplest way to do so:

```
panic% perl Makefile.PL USE_APACI=1 \
    APACI_ARGS='--prefix=/home/httpd/httpd_perl'
```

`ADD_MODULE`

This parameter enables building of built-in Apache modules. For example, to enable the mod_rewrite and mod_proxy modules, you can do the following:

```
panic% perl Makefile.PL ADD_MODULE=proxy,rewrite
```

If you are already using `APACI_ARGS`, you can add the usual Apache *./configure* directives as follows:

```
panic% perl Makefile.PL USE_APACI=1 \
    APACI_ARGS='--enable-module=proxy --enable-module=rewrite'
```

`APACHE_PREFIX`

As an alternative to:

```
APACI_ARGS='--prefix=/home/httpd/httpd_perl'
```

you can use the `APACHE_PREFIX` parameter. When `USE_APACI` is enabled, this attribute specifies the same *--prefix* option.

Additionally, the `APACHE_PREFIX` option automatically executes *make install* in the Apache source directory, which makes the following commands:

```
panic% perl Makefile.PL APACHE_SRC=../apache_1.3.xx/src \
    DO_HTTPD=1 USE_APACI=1 EVERYTHING=1 \
    APACI_ARGS='--prefix=/home/httpd/httpd_perl'
panic% make && make test
panic# make install
panic# cd ../apache_1.3.xx
panic# make install
```

equivalent to these commands:

```
panic% perl Makefile.PL APACHE_SRC=../apache_1.3.xx/src \
    DO_HTTPD=1 USE_APACI=1 EVERYTHING=1 \
    APACHE_PREFIX=/home/httpd/httpd_perl
panic% make && make test
panic# make install
```

`PERL_STATIC_EXTS`

Normally, if a C code extension is statically linked with Perl, it is listed in *Config.pm*'s *$Config{static_exts}*, in which case mod_perl will also statically link this extension with *httpd*. However, if an extension is statically linked with Perl after it is installed, it will not be listed in *Config.pm*. You can either edit *Config.pm* and add these extensions, or configure mod_perl like this:

```
panic% perl Makefile.PL "PERL_STATIC_EXTS=DBI DBD::Oracle"
```

`DYNAMIC`

This option tells mod_perl to build the `Apache::*` API extensions as shared libraries. The default is to link these modules statically with the *httpd* executable. This can save some memory if you use these API features only occasionally. To enable this option, use:

```
panic% perl Makefile.PL DYNAMIC=1
```

`USE_APXS`

If this option is enabled, mod_perl will be built using the `APXS` tool. This tool is used to build C API modules in a way that is independent of the Apache source tree. mod_perl will look for the *apxs* executable in the location specified by `WITH_APXS`; otherwise, it will check the *bin* and *sbin* directories relative to `APACHE_PREFIX`. To enable this option, use:

```
panic% perl Makefile.PL USE_APXS=1
```

`WITH_APXS`

This attribute tells mod_perl the location of the *apxs* executable. This is necessary if the binary cannot be found in the command path or in the location specified by `APACHE_PREFIX`. For example:

```
panic% perl Makefile.PL USE_APXS=1 WITH_APXS=/home/httpd/bin/apxs
```

`USE_DSO`

This option tells mod_perl to build itself as a DSO. Although this reduces the apparent size of the *httpd* executable on disk, it doesn't actually reduce the memory consumed by each *httpd* process. This is recommended only if you are going to be using the mod_perl API only occasionally, or if you wish to experiment with its features before you start using it in a production environment. To enable this option, use:

```
panic% perl Makefile.PL USE_DSO=1
```

`SSL_BASE`

When building against a mod_ssl-enabled server, this option will tell Apache where to look for the SSL *include* and *lib* subdirectories. For example:

```
panic% perl Makefile.PL SSL_BASE=/usr/share/ssl
```

`PERL_DESTRUCT_LEVEL={1,2}`
: When the Perl interpreter shuts down, this level enables additional checks during server shutdown to make sure the interpreter has done proper bookkeeping. The default is `0`. A value of `1` enables full destruction, and `2` enables full destruction with checks. This value can also be changed at runtime by setting the environment variable `PERL_DESTRUCT_LEVEL`. We will revisit this parameter in Chapter 5.

`PERL_TRACE`
: To enable mod_perl debug tracing, configure mod_perl with the `PERL_TRACE` option:

```
panic% perl Makefile.PL PERL_TRACE=1
```

To see the diagnostics, you will also need to set the `MOD_PERL_TRACE` environment variable at runtime.

We will use mod_perl configured with this parameter enabled to show a few debugging techniques in Chapter 21.

`PERL_DEBUG`
: This option builds mod_perl and the Apache server with C source code debugging enabled (the *-g* switch). It also enables `PERL_TRACE`, sets `PERL_DESTRUCT_LEVEL` to `2`, and links against the debuggable *libperld* Perl interpreter if one has been installed. You will be able to debug the Apache executable and each of its modules with a source-level debugger, such as the GNU debugger *gdb*. To enable this option, use:

```
panic% perl Makefile.PL PERL_DEBUG=1
```

We will discuss this option in Chapter 21, as it is extremely useful to track down bugs or report problems.

Activating Callback Hooks

A callback hook (also known simply as a *callback*) is a reference to a subroutine. In Perl, we create subroutine references with the following syntax:

```
$callback = \&subroutine;
```

In this example, `$callback` contains a reference to the subroutine called `subroutine`. Another way to create a callback is to use an anonymous subroutine:

```
$callback = sub { 'some code' };
```

Here, `$callback` contains a reference to the anonymous subroutine. Callbacks are used when we want some action (subroutine call) to occur when some event takes place. Since we don't know exactly when the event will take place, we give the event handler a reference to the subroutine we want to be executed. The handler will call our subroutine at the right time, effectively *calling back* that subroutine.

By default, most of the callback hooks except for PerlHandler, PerlChildInitHandler, PerlChildExitHandler, PerlConnectionApi, and PerlServerApi are turned off. You may enable them via options to *Makefile.PL*.

Here is the list of available hooks and the parameters that enable them. The Apache request prcessing phases were explained in Chapter 1.

```
Directive/Hook              Configuration Option
--------------------------------------------------------
PerlPostReadRequestHandler  PERL_POST_READ_REQUEST
PerlTransHandler            PERL_TRANS
PerlInitHandler             PERL_INIT
PerlHeaderParserHandler     PERL_HEADER_PARSER
PerlAuthenHandler           PERL_AUTHEN
PerlAuthzHandler            PERL_AUTHZ
PerlAccessHandler           PERL_ACCESS
PerlTypeHandler             PERL_TYPE
PerlFixupHandler            PERL_FIXUP
PerlHandler                 PERL_HANDLER
PerlLogHandler              PERL_LOG
PerlCleanupHandler          PERL_CLEANUP
PerlChildInitHandler        PERL_CHILD_INIT
PerlChildExitHandler        PERL_CHILD_EXIT
PerlDispatchHandler         PERL_DISPATCH
```

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

To enable all callback hooks, use:

```
ALL_HOOKS=1
```

There are a few more hooks that won't be enabled by default, because they are experimental.

If you are using:

```
panic% perl Makefile.PL EVERYTHING=1 ...
```

it already includes the ALL_HOOKS=1 option.

Activating Standard API Features

The following options enable various standard features of the mod_perl API. While not absolutely needed, they're very handy and there's little penalty in including them. Unless specified otherwise, these options are all disabled by default. The EVERYTHING=1 or DYNAMIC=1 options will enable them en masse. If in doubt, include these.

PERL_FILE_API=1
: Enables the Apache::File class, which helps with the handling of files under mod_perl.

`PERL_TABLE_API=1`
: Enables the `Apache::Table` class, which provides tied access to the Apache Table structure (used for HTTP headers, among others).

`PERL_LOG_API=1`
: Enables the `Apache::Log` class. This class allows you to access Apache's more advanced logging features.

`PERL_URI_API=1`
: Enables the `Apache::URI` class, which deals with the parsing of URIs in a similar way to the Perl `URI::URL` module, but much faster.

`PERL_UTIL_API=1`
: Enables the `Apache::Util` class, allowing you to use various functions such as HTML escaping or date parsing, but implemented in C.

`PERL_CONNECTION_API=1`
: Enables the `Apache::Connection` class. This class is enabled by default. Set the option to `0` to disable it.

`PERL_SERVER_API=1`
: Enables the `Apache::Server` class. This class is enabled by default. Set the option to `0` to disable it.

Please refer to Lincoln Stein and Doug MacEachern's *Writing Apache Modules with Perl and C* (O'Reilly) for more information about the Apache API.

Enabling Extra Features

mod_perl comes with a number of other features. Most of them are disabled by default. This is the list of features and options to enable them:

- `<Perl>` sections give you a way to configure Apache using Perl code in the *httpd.conf* file itself. See Chapter 4 for more information.

  ```
  panic% perl Makefile.PL PERL_SECTIONS=1 ...
  ```

- With the `PERL_SSI` option, the mod_include module can be extended to include a `#perl` directive.

  ```
  panic% perl Makefile.PL PERL_SSI=1
  ```

  By enabling `PERL_SSI`, a new `#perl` element is added to the standard mod_include functionality. This element allows server-side includes to call Perl subroutines directly. This feature works only when mod_perl is not built as a DSO (i.e., when it's built statically).

- If you develop an Apache module in Perl and you want to create custom configuration directives[*] to be recognized in *httpd.conf*, you need to use `Apache::`

[*] See Chapters 8 and 9 of *Writing Apache Modules with Perl and C* (O'Reilly).

`ModuleConfig` and `Apache::CmdParms`. For these modules to work, you will need to enable this option:

```
panic% perl Makefile.PL PERL_DIRECTIVE_HANDLERS=1
```

- The stacked handlers feature explained in Chapter 4 requires this parameter to be enabled:

```
panic% perl Makefile.PL PERL_STACKED_HANDLERS=1
```

- The method handlers feature discussed in Chapter 4 requires this parameter to be enabled:

```
panic% perl Makefile.PL PERL_METHOD_HANDLERS=1
```

- To enable all phase callback handlers, all API modules, and all miscellaneous features, use the "catch-all" option we used when we first compiled mod_perl:

```
panic% perl Makefile.PL EVERYTHING=1
```

Reusing Configuration Parameters

When you have to upgrade the server, it's sometimes hard to remember what parameters you used in the previous mod_perl build. So it's a good idea to save them in a file.

One way to save parameters is to create a file (e.g., *~/.mod_perl_build_options*) with the following contents:

```
APACHE_SRC=../apache_1.3.xx/src DO_HTTPD=1 USE_APACI=1 \
EVERYTHING=1
```

Then build the server with the following command:

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

But mod_perl has a standard method to perform this trick. If a file named *makepl_args.mod_perl* is found in the same directory as the mod_perl build location, it will be read in by *Makefile.PL*. Parameters supplied at the command line will override the parameters given in this file.

The *makepl_args.mod_perl* file can also be located in your home directory or in the ../ directory relative to the mod_perl distribution directory. The filename can also start with a dot (*.makepl_args.mod_perl*), so you can keep it nicely hidden along with the rest of the dot files in your home directory. So, *Makefile.PL* will look for the following files (in this order), using the first one it comes across:

```
./makepl_args.mod_perl
../makepl_args.mod_perl
./.makepl_args.mod_perl
../.makepl_args.mod_perl
$ENV{HOME}/.makepl_args.mod_perl
```

For example:

```
panic% ls -1 /home/stas/src
apache_1.3.xx/
makepl_args.mod_perl
mod_perl-1.xx/

panic% cat makepl_args.mod_perl
APACHE_SRC=../apache_1.3.xx/src
DO_HTTPD=1
USE_APACI=1
EVERYTHING=1

panic% cd mod_perl-1.xx
panic% perl Makefile.PL
panic% make && make test
panic# make install
```

Now the parameters from the *makepl_args.mod_perl* file will be used automatically, as if they were entered directly.

In the sample *makepl_args.mod_perl* file in the *eg/* directory of the mod_perl distribution package, you might find a few options enabling some experimental features for you to play with, too!

If you are faced with a compiled Apache and no trace of the parameters used to build it, you can usually still find them if *make clean* was not run on the sources. You will find the Apache-specific parameters in *apache_1.3.xx/config.status* and the mod_perl parameters in *mod_perl-1.xx/apaci/mod_perl.config*.

Discovering Whether a Feature Was Enabled

mod_perl Version 1.25 introduced `Apache::MyConfig`, which provides access to the various hooks and features set when mod_perl was built. This circumvents the need to set up a live server just to find out if a certain callback hook is available.

To see whether some feature was built in or not, check the `%Apache::MyConfig::Setup` hash. For example, suppose we install mod_perl with the following options:

```
panic% perl Makefile.PL EVERYTHING=1
```

but the next day we can't remember which callback hooks were enabled. We want to know whether the `PERL_LOG` callback hook is available. One of the ways to find an answer is to run the following code:

```
panic% perl -MApache::MyConfig -e 'print $Apache::MyConfig::Setup{PERL_LOG}'
```

If it prints `1`, that means the `PERL_LOG` callback hook is enabled (which it should be, as `EVERYTHING=1` enables them all).

Another approach is to configure `Apache::Status` (see Chapter 9) and run *http://localhost/perl-status?hooks* to check for enabled hooks.

If you want to check for the existence of various hooks within your handlers, you can use the script shown in Example 3-1.

Example 3-1. test_hooks.pl

```
use mod_perl_hooks;

for my $hook (mod_perl::hooks( )) {
    if (mod_perl::hook($hook)) {
        print "$hook is enabled\n";
    }
    else {
        print "$hook is not enabled\n";
    }
}
```

You can also try to look at the symbols inside the *httpd* executable with the help of *nm(1)* or a similar utility. For example, if you want to see whether you enabled `PERL_LOG=1` while building mod_perl, you can search for a symbol with the same name but in lowercase:

```
panic% nm httpd | grep perl_log
08071724 T perl_logger
```

This shows that `PERL_LOG=1` was enabled. But this approach will work only if you have an unstripped *httpd* binary. By default, *make install* strips the binary before installing it, thus removing the symbol names to save space. Use the *--without-execstrip ./configure* option to prevent stripping during the *make install* phase. [*]

Yet another approach that will work in most cases is to try to use the feature in question. If it wasn't configured, Apache will give an error message.

Using an Alternative Configuration File

By default, mod_perl provides its own copy of the *Configuration* file to Apache's *configure* utility. If you want to pass it your own version, do this:

```
panic% perl Makefile.PL CONFIG=Configuration.custom
```

where *Configuration.custom* is the pathname of the file *relative* to the Apache source tree you build against.

perl Makefile.PL Troubleshooting

During the configuration (*perl Makefile.PL*) stage, you may encounter some of these problems. To help you avoid them, let's study them, find out why they happened, and discuss how to fix them.

[*] You might need the unstripped version for debugging reasons too.

A test compilation with your Makefile configuration failed...

When you see the following error during the *perl Makefile.PL* stage:

```
** A test compilation with your Makefile configuration
** failed. This is most likely because your C compiler
** is not ANSI. Apache requires an ANSI C Compiler, such
** as gcc. The above error message from your compiler
** will also provide a clue.
 Aborting!
```

it's possible that you have a problem with a compiler. It may be improperly installed or not installed at all. Sometimes the reason is that your Perl executable was built on a different machine, and the software installed on your machine is not the same. Generally this happens when you install prebuilt packages, such as *rpm* or *deb*. You may find that the dependencies weren't properly defined in the Perl binary package and you were allowed to install it even though some essential packages were not installed.

The most frequent pitfall is a missing `gdbm` library (see the next section).

But why guess, when we can actually see the real error message and understand what the real problem is? To get a real error message, edit the Apache *src/Configure* script. Around line 2140, you should see a line like this:

```
if ./helpers/TestCompile sanity; then
```

Add the *-v* option, as follows:

```
if ./helpers/TestCompile -v sanity; then
```

and try again. Now you should get a useful error message.

Missing or misconfigured libgdbm.so

On some Red Hat Linux systems, you might encounter a problem during the *perl Makefile.PL* stage, when Perl was installed from an *rpm* package built with the `gdbm` library, but `libgdbm` isn't actually installed. If this happens to you, make sure you install it before proceeding with the build process.

You can check how Perl was built by running the *perl -V* command:

```
panic% perl -V | grep libs
```

You should see output similar to this:

```
libs=-lnsl -lndbm -lgdbm -ldb -ldl -lm -lc -lposix -lcrypt
```

Sometimes the problem is even more obscure: you do have `libgdbm` installed, but it's not installed properly. Do this:

```
panic% ls /usr/lib/libgdbm.so*
```

If you get at least three lines, like we do:

```
lrwxrwxrwx   /usr/lib/libgdbm.so -> libgdbm.so.2.0.0
lrwxrwxrwx   /usr/lib/libgdbm.so.2 -> libgdbm.so.2.0.0
-rw-r--r--   /usr/lib/libgdbm.so.2.0.0
```

you are all set. On some installations, the *libgdbm.so* symbolic link is missing, so you get only:

```
lrwxrwxrwx   /usr/lib/libgdbm.so.2 -> libgdbm.so.2.0.0
-rw-r--r--   /usr/lib/libgdbm.so.2.0.0
```

To fix this problem, add the missing symbolic link:

```
panic% cd /usr/lib
panic% ln -s libgdbm.so.2.0.0 libgdbm.so
```

Now you should be able to build mod_perl without any problems.

Note that you might need to prepare this symbolic link as well:

```
lrwxrwxrwx   /usr/lib/libgdbm.so.2 -> libgdbm.so.2.0.0
```

with the command:

```
panic% ln -s libgdbm.so.2.0.0 libgdbm.so.2
```

Of course, if a new version of the `libgdbm` library was released between the moment we wrote this sentence and the moment you're reading it, you will have to adjust the version numbers. We didn't use the usual *xx.xx* version replacement here, to make it easier to understand how the symbolic links should be set.

About the gdbm, db, and ndbm Libraries

If you need to have the `dbm` library linked in, you should know that both the `gdbm` and `db` libraries offer `ndbm` emulation, which is the interface that Apache actually uses. So when you build mod_perl, you end up using whichever library was linked first by the Perl compilation. If you build Apache without mod_perl, you end up with whatever appears to be be your `ndbm` library, which will vary between systems, and especially Linux distributions. So you may have to work a bit to get both Apache and Perl to use the same library, and you are likely to have trouble copying the *dbm* file from one system to another or even using it after an upgrade.

Undefined reference to 'PL_perl_destruct_level'

When manually building mod_perl using the shared library:

```
panic% cd mod_perl-1.xx
panic% perl Makefile.PL PREP_HTTPD=1
panic% make && make test
panic# make install

panic% cd ../apache_1.3.xx
panic% ./configure --with-layout=RedHat --target=perlhttpd
    --activate-module=src/modules/perl/libperl.a
```

you might see the following output:

```
gcc -c  -I./os/unix -I./include   -DLINUX=2 -DTARGET=\"perlhttpd\"
-DUSE_HSREGEX -DUSE_EXPAT -I./lib/expat-lite `./apaci` buildmark.c
gcc  -DLINUX=2 -DTARGET=\"perlhttpd\" -DUSE_HSREGEX -DUSE_EXPAT
-I./lib/expat-lite `./apaci`    \
      -o perlhttpd buildmark.o modules.o modules/perl/libperl.a
modules/standard/libstandard.a main/libmain.a ./os/unix/libos.a ap/libap.a
regex/libregex.a lib/expat-lite/libexpat.a  -lm -lcrypt
modules/perl/libperl.a(mod_perl.o): In function `perl_shutdown':
mod_perl.o(.text+0xf8): undefined reference to `PL_perl_destruct_level'
mod_perl.o(.text+0x102): undefined reference to `PL_perl_destruct_level'
mod_perl.o(.text+0x10c): undefined reference to `PL_perl_destruct_level'
mod_perl.o(.text+0x13b): undefined reference to `Perl_av_undef'
[more errors snipped]
```

This happens when Perl was built statically linked, with no shared `libperl.a`. Build a dynamically linked Perl (with `libperl.a`) and the problem will disappear.

Building mod_perl (make)

After completing the configuration, it's time to build the server by simply calling:

```
panic% make
```

The *make* program first compiles the source files and creates a mod_perl library file. Then, depending on your configuration, this library is either linked with *httpd* (statically) or not linked at all, allowing you to dynamically load it at runtime.

You should avoid putting the mod_perl source directory inside the Apache source directory, as this confuses the build process. The best choice is to put both source directories under the same parent directory.

What Compiler Should Be Used to Build mod_perl?

All Perl modules that use C extensions must be compiled using the compiler with which your copy of Perl was built.

When you run *perl Makefile.PL*, a *Makefile* is created. This *Makefile* includes the same compilation options that were used to build Perl itself. They are stored in the `Config.pm` module and can be displayed with the *Perl -V* command. All these options are reapplied when compiling Perl modules.

If you use a different compiler to build Perl extensions, chances are that the options this compiler uses won't be the same, or they might be interpreted in a completely different way. So the code may not compile, may dump core, or may behave in unexpected ways.

Since Perl, Apache, and third-party modules all work together under mod_perl, it's essential to use the same compiler while building each of the components.

,ch03.21292 Page 59 Thursday, November 18, 2004 12:35 PM

If you compile a non-Perl component separately, you should make sure to use both the same compiler and the same options used to build Perl. You can find much of this information by running *perl -V*.

make Troubleshooting

The following errors are the ones that frequently occur during the *make* process when building mod_perl.

Undefined reference to 'Perl_newAV'

This and similar error messages may show up during the *make* process. Generally it happens when you have a broken Perl installation. If it's installed from a broken *rpm* or another precompiled binary package, build Perl from source or use another properly built binary package. Run *perl -V* to learn what version of Perl you are using and other important details.

Unrecognized format specifier for...

This error is usually reported due to the problems with some versions of the `SFIO` library. Try to use the latest version to get around this problem or, if you don't really need `SFIO`, rebuild Perl without this library.

Testing the Server (make test)

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

```
panic% make test
```

Fortunately, mod_perl comes with a big collection of tests, which attempt to exercise all the features you asked for at the configuration stage. If any of the tests fails, the *make test* step will fail.

Running *make test* will start the freshly built *httpd* on port 8529 (an unprivileged port), running under the UID (user ID) and GID (group ID) of the *perl Makefile.PL* process. The *httpd* will be terminated when the tests are finished.

To change the default port (8529) used for the tests, do this:

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

Each file in the testing suite generally includes more than one test, but when you do the testing, the program will report only how many tests were passed and the total number of tests defined in the test file. To learn which ones failed, run the tests in verbose mode by using the `TEST_VERBOSE` parameter:

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

As of mod_perl v1.23, you can use the environment variables `APACHE_USER` and `APACHE_GROUP` to override the default `User` and `Group` settings in the *httpd.conf* file used for *make test*. These two variables should be set before the *Makefile* is created to take effect during the testing stage. For example, if you want to set them to *httpd*, you can do the following in the Bourne-style shell:

```
panic% export APACHE_USER=httpd
panic% export APACHE_GROUP=httpd
panic% perl Makefile.PL ...
```

Manual Testing

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

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

or all tests in a test subdirectory:

```
panic% perl t/TEST modules
```

The *TEST* script starts the server before the test is executed. If for some reason it fails to start, use *make start_httpd* to start it manually:

```
panic% make start_httpd
```

To shut down Apache when the testing is complete, use *make kill_httpd*:

```
panic% make kill_httpd
```

make test Troubleshooting

The following sections cover problems that you may encounter during the testing stage.

make test fails

`make test` requires Apache to be running already, so if you specified `NO_HTTPD=1` during the *perl Makefile.PL* stage, you'll have to build *httpd* independently before running *make test*. Go to the Apache source tree and run *make*, then return to the mod_perl source tree and continue with the server testing.

If you get an error like this:

```
still waiting for server to warm up...............not ok
```

you may want to examine the *t/logs/error_log* file, where all the *make test*–stage errors are logged. If you still cannot find the problem or this file is completely empty, you may want to run the test with *strace* (or *truss*) in the following way (assumming that you are located in the root directory of the mod_perl source tree):

```
panic% make start_httpd
panic% strace -f -s1024 -o strace.out -p `cat t/logs/httpd.pid` &
```

```
panic% make run_tests
panic% make kill_httpd
```

where the *strace -f* option tells *strace* to trace child processes as they are created, *-s1024* allows trace strings of a maximum of 1024 characters to be printed (it's 32 by default), *-o* gives the name of the file to which the output should be written, *-p* supplies the PID of the parent process, and *&* puts the job in the background.

When the tests are complete, you can examine the generated *strace.out* file and hopefully find the problem. We talk about creating and analyzing trace outputs in Chapter 21.

mod_perl.c is incompatible with this version of Apache

If you had a stale Apache header layout in one of the *include* paths during the build process, you will see the message "mod_perl.c is incompatible with this version of Apache" when you try to execute *httpd*. Find the file *ap_mmn.h* using *find*, *locate*, or another utility. Delete this file and rebuild Apache. The Red Hat Linux distribution usually installs it in */usr/local/include*.

Before installing mod_perl-enabled Apache from scratch, it's a good idea to remove all the pre-installed Apache modules, and thus save the trouble of looking for files that mess up the build process. For example, to remove the precompiled Apache installed as a Red Hat Package Manager (RPM) package, as *root* you should do:

```
panic# rpm -e apache
```

There may be other RPM packages that depend on the Apache RPM package. You will be notified about any other dependent packages, and you can decide whether to delete them, too. You can always supply the *--nodeps* option to tell the RPM manager to ignore the dependencies.

apt users would do this instead:

```
panic# apt-get remove apache
```

make test......skipping test on this platform

make test may report some tests as *skipped*. They are skipped because you are missing the modules that are needed for these tests to pass. You might want to peek at the contents of each test; you will find them all in the *./t* directory. It's possible that you don't need any of the missing modules to get your work done, in which case you shouldn't worry that the tests are skipped.

If you want to make sure that all tests pass, you will need to figure out what modules are missing from your installation. For example, if you see:

```
modules/cookie......skipping test on this platform
```

you may want to install the `Apache::Cookie` module. If you see:

```
modules/request.....skipping test on this platform
```

Apache::Request is missing.[*] If you see:

```
modules/psections...skipping test on this platform
```

Devel::Symdump and Data::Dumper are needed.

Chances are that all of these will be installed if you use CPAN.pm to install Bundle::Apache. We talk about CPAN installations later in this chapter.

make test fails due to misconfigured localhost entry

The *make test* suite uses *localhost* to run the tests that require a network. Make sure you have this entry in */etc/hosts*:

```
127.0.0.1       localhost.localdomain   localhost
```

Also make sure you have the loopback device *lo* configured. If you aren't sure, run:

```
panic% /sbin/ifconfig lo
```

This will tell you whether the loopback device is configured.

Installation (make install)

After testing the server, the last step is to install it. First install all the Perl files (usually as *root*):

```
panic# make install
```

Then go to the Apache source tree and complete the Apache installation (installing the configuration files, *httpd*, and utilities):

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

Of course, if you have used the APACHE_PREFIX option as explained earlier in this chapter, you can skip this step.

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

Manually Building a mod_perl-Enabled Apache

If you want to build *httpd* separately from mod_perl, you should use the NO_HTTPD=1 option during the *perl Makefile.PL* (mod_perl build) stage. Then you will have to configure various things by hand and proceed to build Apache. You shouldn't run *perl Makefile.PL* before following the steps described in this section.

If you choose to manually build mod_perl, there are three things you may need to set up before the build stage:

[*] Apache::Cookie and Apache::Request are both part of the *libapreq* distribution.

mod_perl's Makefile

When *perl Makefile.PL* is executed, *$APACHE_SRC/modules/perl/Makefile* may need to be modified to enable various options (e.g., `ALL_HOOKS=1`).

Optionally, instead of tweaking the options during the *perl Makefile.PL* stage, you can edit *mod_perl-1.xx/src/modules/perl/Makefile* before running *perl Makefile.PL*.

Configuration

Add the following to *apache_1.3.xx/src/Configuration*:

```
AddModule modules/perl/libperl.a
```

We suggest you add this entry at the end of the *Configuration* file if you want your callback hooks to have precedence over core handlers.

Add the following to `EXTRA_LIBS`:

```
EXTRA_LIBS=`perl -MExtUtils::Embed -e ldopts`
```

Add the following to `EXTRA_CFLAGS`:

```
EXTRA_CFLAGS=`perl -MExtUtils::Embed -e ccopts`
```

mod_perl source files

Return to the mod_perl directory and copy the mod_perl source files into the Apache build directory:

```
panic% cp -r src/modules/perl apache_1.3.xx/src/modules/
```

When you are done with the configuration parts, run:

```
panic% perl Makefile.PL NO_HTTPD=1 DYNAMIC=1  EVERYTHING=1 \
    APACHE_SRC=../apache_1.3.xx/src
```

`DYNAMIC=1` enables a build of the shared mod_perl library. Add other options if required.

```
panic# make install
```

Now you may proceed with the plain Apache build process. Note that in order for your changes to the *apache_1.3.xx/src/Configuration* file to take effect, you must run *apache_1.3.xx/src/Configure* instead of the default *apache_1.3.xx/configure* script:

```
panic% cd ../apache_1.3.xx/src
panic% ./Configure
panic% make
panic# make install
```

Installation Scenarios for Standalone mod_perl

When building mod_perl, the mod_perl C source files that have to be compiled into the *httpd* executable usually are copied to the subdirectory *src/modules/perl/* in the Apache source tree. In the past, to integrate this subtree into the Apache build process, a lot of adjustments were done by mod_perl's *Makefile.PL*. *Makefile.PL* was also responsible for the Apache build process.

This approach is problematic in several ways. It is very restrictive and not very clean, because it assumes that mod_perl is the only third-party module that has to be integrated into Apache.

A new hybrid build environment was therefore created for the Apache side of mod_perl, to avoid these problems. It prepares only the *src/modules/perl/* subtree inside the Apache source tree, without adjusting or editing anything else. This way, no conflicts can occur. Instead, mod_perl is activated later (via APACI calls when the Apache source tree is configured), and then it configures itself.

There are various ways to build Apache with the new hybrid build environment (using `USE_APACI=1`):

- Build Apache and mod_perl together, using the default configuration.
- Build Apache and mod_perl separately, allowing you to plug in other third-party Apache modules as needed.
- Build mod_perl as a DSO inside the Apache source tree using APACI.
- Build mod_perl as a DSO outside the Apache source tree with APXS.

The All-in-One Way

If your goal is just to build and install Apache with mod_perl out of their source trees, and you have no interest in further adjusting or enhancing Apache, proceed as we described in Chapter 2:

```
panic% tar xzvf apache_1.3.xx.tar.gz
panic% tar xzvf mod_perl-1.xx.tar.gz
panic% cd mod_perl-1.xx
panic% perl Makefile.PL APACHE_SRC=../apache_1.3.xx/src \
    DO_HTTPD=1 USE_APACI=1 EVERYTHING=1
panic% make && make test
panic# make install
panic# cd ../apache_1.3.xx
panic# make install
```

This builds Apache statically with mod_perl, installs Apache under the default */usr/local/apache* tree, and installs mod_perl into the *site_perl* hierarchy of your existing Perl installation.

Building mod_perl and Apache Separately

However, sometimes you might need more flexibility while building mod_perl. If you build mod_perl into the Apache binary (*httpd*) in separate steps, you'll also have the freedom to include other third-party Apache modules. Here are the steps:

1. Prepare the Apache source tree.

   As before, first extract the distributions:

   ```
   panic% tar xvzf apache_1.3.xx.tar.gz
   panic% tar xzvf mod_perl-1.xx.tar.gz
   ```

2. Install mod_perl's Perl side and prepare the Apache side.

   Next, install the Perl side of mod_perl into the Perl hierarchy and prepare the *src/modules/perl/* subdirectory inside the Apache source tree:

   ```
   panic% cd mod_perl-1.xx
   panic% perl Makefile.PL \
       APACHE_SRC=../apache_1.3.xx/src \
       NO_HTTPD=1   \
       USE_APACI=1  \
       PREP_HTTPD=1 \
       EVERYTHING=1 \
       [...]
   panic% make
   panic# make install
   ```

   The APACHE_SRC option sets the path to your Apache source tree, the NO_HTTPD option forces this path and only this path to be used, the USE_APACI option triggers the new hybrid build environment, and the PREP_HTTPD option forces preparation of the *$APACHE_SRC/modules/perl/* tree but no automatic build.

   This tells the configuration process to prepare the Apache side of mod_perl in the Apache source tree, but doesn't touch anything else in it. It then just builds the Perl side of mod_perl and installs it into the Perl installation hierarchy.

   Note that if you use PREP_HTTPD as described above, to complete the build you must go into the Apache source directory and run *make* and *make install*.

3. Prepare other third-party modules.

   Now you have a chance to prepare any other third-party modules you might want to include in Apache. For instance, you can build PHP separately, as you did with mod_perl.

4. Build the Apache package.

   Now it's time to build Apache, including the Apache side of mod_perl and any other third-party modules you've prepared:

   ```
   panic% cd apache_1.3.xx
   panic% ./configure \
       --prefix=/path/to/install/of/apache \
       --activate-module=src/modules/perl/libperl.a \
       [...]
   panic% make
   panic# make install
   ```

   You must use the *--prefix* option if you want to change the default target directory of the Apache installation. The *--activate-module* option activates mod_perl for the configuration process and thus also for the build process. If you choose *--prefix=/usr/share/apache*, the Apache directory tree will be installed in */usr/share/apache*.

If you add other third-party components, such as PHP, include a separate *--activate-module* option for each of them. (See the module's documentation for the actual path to which *--activate-module* should point.) For example, for mod_php4:

```
--activate-module=src/modules/php4/libphp4.a
```

Note that the files activated by *--activate-module* do not exist at this time. They will be generated during compilation.

You may also want to go back to the mod_perl source tree and run *make test* (to make sure that mod_perl is working) before running *make install* inside the Apache source tree.

For more detailed examples on building mod_perl with other components, see the section later in this chapter entitled "Building mod_perl with Other Components."

When DSOs Can Be Used

If you want to build mod_perl as a DSO, you must make sure that Perl was built with the system's native `malloc()`. If Perl was built with its own `malloc()` and *-Dbincompat5005*, it pollutes the main *httpd* program with *free* and *malloc* symbols. When *httpd* starts or restarts, any references in the main program to *free* and *malloc* become invalid, causing memory leaks and segfaults.

Notice that mod_perl's build system warns about this problem.

With Perl 5.6.0+ this pollution can be prevented by using *-Ubincompat5005* or *-Uusemymalloc* for any version of Perl. However, there's a chance that *-Uusemymalloc* might hurt performance on your platform, so *-Ubincompat5005* is likely a better choice.

If you get the following reports with Perl version 5.6.0+:

```
% perl -V:usemymalloc
usemymalloc='y';
% perl -V:bincompat5005
bincompat5005='define';
```

rebuild Perl with *-Ubincompat5005*.

For pre-5.6.x Perl versions, if you get:

```
% perl -V:usemymalloc
usemymalloc='y';
```

rebuild Perl with *-Uusemymalloc*.

Now rebuild mod_perl.

Building mod_perl as a DSO via APACI

We have already mentioned that the new mod_perl build environment (with `USE_APACI`) is a hybrid. What does that mean? It means, for instance, that you can use the same *src/modules/perl/* configuration to build mod_perl as a DSO or not, without having to edit any files. To build `libperl.so`, just add a single option, depending on which method you used to build mod_perl.

libperl.so and libperl.a

The static mod_perl library is called *libperl.a*, and the shared mod_perl library is called *libperl.so*. Of course, *libmodperl* would have been a better prefix, but *libperl* was used because of prehistoric Apache issues. Be careful that you don't confuse mod_perl's *libperl.a* and *libperl.so* files with the ones that are built with the standard Perl installation.

If you choose the "standard" all-in-one way of building mod_perl, add:

```
USE_DSO=1
```

to the *perl Makefile.PL* options.

If you choose to build mod_perl and Apache separately, add:

```
--enable-shared=perl
```

to Apache's *configure* options when you build Apache.

As you can see, whichever way you build mod_perl and Apache, only one additional option is needed to build mod_perl as a DSO. Everything else is done automatically: mod_so is automatically enabled, the *Makefile*s are adjusted, and the *install* target from APACI installs *libperl.so* into the Apache installation tree. Additionally, the `LoadModule` and `AddModule` directives (which dynamically load and insert mod_perl into *httpd*) are automatically added to *httpd.conf*.

Building mod_perl as a DSO via APXS

We've seen how to build mod_perl as a DSO *inside* the Apache source tree, but there is a nifty alternative: building mod_perl as a DSO *outside* the Apache source tree via the new Apache 1.3 support tool called `APXS`. The advantage is obvious: you can extend an already installed Apache with mod_perl even if you don't have the sources (for instance, you may have installed an Apache binary package from your vendor or favorite distribution).

Here are the build steps:

```
panic% tar xzvf mod_perl-1.xx.tar.gz
panic% cd mod_perl-1.xx
```

```
panic% perl Makefile.PL \
    USE_APXS=1 \
    WITH_APXS=/path/to/bin/apxs \
    EVERYTHING=1 \
    [...]
panic% make && make test
panic# make install
```

This will build the DSO *libperl.so* outside the Apache source tree and install it into the existing Apache hierarchy.

Building mod_perl with Other Components

mod_perl is often used with other components that plug into Apache, such as PHP and SSL. In this section, we'll show you a build combining mod_perl with PHP. We'll also show how to build a secure version of Apache with mod_perl support using each of the SSL options available for Apache today (mod_ssl, Apache-SSL, Stronghold, and Covalent).

Since you now understand how the build process works, we'll present these scenarios without much explanation (unless they involve something we haven't discussed yet).

All these scenarios were tested on a Linux platform. You might need to refer to the specific component's documentation if something doesn't work for you as described here. The intention of this section is not to show you how to install other non-mod_perl components alone, but how to do this in a bundle with mod_perl.

Also, notice that the links we've used are very likely to have changed by the time you read this document. That's why we have used the *x.xx* convention instead of using hardcoded version numbers. Remember to replace the *x.xx* placeholders with the version numbers of the distributions you are going to use. To find out the latest stable version number, visit the components' sites—e.g., if we say *http://perl.apache.org/dist/mod_perl-1.xx.tar.gz*, go to *http://perl.apache.org/download/* to learn the version number of the latest stable release of mod_perl 1, and download the appropriate file.

Unless otherwise noted, all the components install themselves into a default location. When you run *make install*, the installation program tells you where it's going to install the files.

Installing mod_perl with PHP

The following is a simple installation scenario of a combination mod_perl and PHP build for the Apache server. We aren't going to use a custom installation directory, so Apache will use the default */usr/local/apache* directory.

1. Download the latest stable source releases:

```
Apache:   http://www.apache.org/dist/httpd/
mod_perl: http://perl.apache.org/download/
PHP:      http://www.php.net/downloads.php
```

2. Unpack them:

   ```
   panic% tar xvzf mod_perl-1.xx
   panic% tar xvzf apache_1.3.xx.tar.gz
   panic% tar xvzf php-x.x.xx.tar.gz
   ```

3. Configure Apache:

   ```
   panic% cd apache_1.3.xx
   panic% ./configure
   ```

4. Build mod_perl:

   ```
   panic% cd ../mod_perl-1.xx
   panic% perl Makefile.PL APACHE_SRC=../apache_1.3.xx/src NO_HTTPD=1 \
       USE_APACI=1 PREP_HTTPD=1 EVERYTHING=1
   panic% make
   ```

5. Build mod_php:

   ```
   panic% cd ../php-x.x.xx
   panic% ./configure --with-apache=../apache_1.3.xx \
       --with-mysql --enable-track-vars
   panic% make
   panic# make install
   ```

   (mod_php doesn't come with a *make test* suite, so we don't need to run one.)

6. Reconfigure Apache to use mod_perl and PHP, and then build it:

   ```
   panic% cd ../apache_1.3.xx
   panic% ./configure \
       --activate-module=src/modules/perl/libperl.a \
       --activate-module=src/modules/php4/libphp4.a
   panic% make
   ```

   Note that if you are building PHP3, you should use *php3/libphp3.a*. Also remember that *libperl.a* and *libphp4.a* do not exist at this time. They will be generated during compilation.

7. Test and install mod_perl:

   ```
   panic% cd ../mod_perl-1.xx
   panic% make test
   panic# make install
   ```

8. Complete the Apache installation:

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

Now when you start the server:

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

you should see something like this in */usr/local/apache/logs/error_log*:

```
[Sat May 18 11:10:31 2002] [notice]
Apache/1.3.24 (Unix) PHP/4.2.0 mod_perl/1.26
configured -- resuming normal operations
```

If you need to build mod_ssl as well, make sure that you add the mod_ssl component first (see the next section).

Installing mod_perl with mod_ssl (+openssl)

mod_ssl provides strong cryptography for the Apache 1.3 web server via the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols. mod_ssl uses the open source SSL/TLS toolkit OpenSSL, which is based on SSLeay, by Eric A. Young and Tim J. Hudson. As in the previous installation scenario, the default installation directory is used in this example.

1. Download the latest stable source releases. For mod_ssl, make sure that the version matches your version of Apache (e.g., get *mod_ssl-2.8.8-1.3.24.tar.gz* if you have Apache 1.3.24).

```
Apache:   http://www.apache.org/dist/httpd/
mod_perl: http://perl.apache.org/download/
mod_ssl:  http://www.modssl.org/source/
openssl:  http://www.openssl.org/source/
```

2. Unpack the sources:

```
panic% tar xvzf mod_perl-1.xx.tar.gz
panic% tar xvzf apache_1.3.xx.tar.gz
panic% tar xvzf mod_ssl-x.x.x-1.3.xx.tar.gz
panic% tar xvzf openssl-x.x.x.tar.gz
```

3. Configure, build, test, and install *openssl* if it isn't already installed:

```
panic% cd openssl-x.x.x
panic% ./config
panic% make && make test
panic# make install
```

   (If you already have the *openssl* development environment installed, you can skip this stage.)

4. Configure `mod_ssl`:

```
panic% cd mod_ssl-x.x.x-1.3.xx
panic% ./configure --with-apache=../apache_1.3.xx
```

5. Configure, build, test, and install mod_perl:

```
panic% cd ../mod_perl-1.xx
panic% perl Makefile.PL USE_APACI=1 EVERYTHING=1 \
    DO_HTTPD=1 SSL_BASE=/usr/local/ssl \
    APACHE_SRC=../apache_1.3.xx/src \
    APACI_ARGS='--enable-module=ssl'
panic% make && make test
panic# make install
```

6. Create an SSL certificate and install Apache and certificate files:

```
panic% cd ../apache_1.3.xx
panic% make certificate
panic# make install
```

7. Now proceed with the mod_ssl and mod_perl parts of the server configuration in *httpd.conf*. The next chapter provides in-depth information about mod_perl configuration. For mod_ssl configuration, please refer to the mod_ssl documentation available from *http://www.modssl.org/*.

Now when you start the server:

```
panic# /usr/local/apache/bin/apachectl startssl
```

you should see something like this in */usr/local/apache/logs/error_log*:

```
[Fri May 18 11:10:31 2001] [notice]
Apache/1.3.24 (Unix) mod_perl/1.26 mod_ssl/2.8.8
OpenSSL/0.9.6c configured -- resuming normal operations
```

If you used the default configuration, the SSL part won't be loaded if you use *apachectl start* and not *apachectl startssl*.

This scenario also demonstrates the fact that some third-party Apache modules can be added to Apache by just enabling them (as with mod_ssl), while others need to be separately configured and built (as with mod_perl and PHP).

Installing mod_perl with Apache-SSL (+openssl)

Apache-SSL is a secure web server based on Apache and SSLeay/OpenSSL. It is licensed under a BSD-style license, which means that you are free to use it for commercial or non-commercial purposes as long as you retain the copyright notices.

Apache-SSL provides similar functionality to mod_ssl. mod_ssl is what is known as a *split*—i.e., it was originally derived from Apache-SSL but has been extensively redeveloped so the code now bears little relation to the original. We cannot advise you to use one over another—both work fine with mod_perl, so choose whichever you want. People argue about which one to use all the time, so if you are interested in the finer points, you may want to check the mailing list archives of the two projects (*http://www.apache-ssl.org/#Mailing_List* and *http://www.modssl.org/support/*).

To install mod_perl with Apache-SSL:

1. Download the sources. You'll need to have matching Apache-SSL and Apache versions.

   ```
   Apache:     http://www.apache.org/dist/httpd/
   mod_perl:   http://perl.apache.org/download/
   openssl:    http://www.openssl.org/source/
   Apache-SSL: http://www.apache-ssl.org/#Download
   ```

2. Unpack the sources:

   ```
   panic% tar xvzf mod_perl-1.xx
   panic% tar xvzf apache_1.3.xx.tar.gz
   panic% tar xvzf openssl-x.x.x.tar.gz
   ```

3. Configure and install *openssl*, if necessary:

   ```
   panic% cd openssl-x.x.x
   panic% ./config
   panic% make && make test
   panic# make install
   ```

   If you already have the *openssl* development environment installed, you can skip this stage.

4. Apache-SSL comes as a patch to Apache sources. First unpack the Apache-SSL sources inside the Apache source tree and make sure that the Apache source is clean (in case you've used this source to build Apache before). Then run *./FixPatch* and answer y to proceed with the patching of Apache sources:

   ```
   panic% cd apache_1.3.xx
   panic% make clean
   panic% tar xzvf ../apache_1.3.xx+ssl_x.xx.tar.gz
   panic% ./FixPatch
   Do you want me to apply the fixed-up Apache-SSL patch for you? [n] y
   ```

5. Proceed with mod_perl configuration. The notable addition to the usual configuration parameters is that we use the `SSL_BASE` parameter to point to the directory in which *openssl* is installed:

   ```
   panic% cd ../mod_perl-1.xx
   panic% perl Makefile.PL USE_APACI=1 EVERYTHING=1 \
       DO_HTTPD=1 SSL_BASE=/usr/local/ssl \
       APACHE_SRC=../apache_1.3.xx/src
   ```

6. Build, test, and install mod_perl:

   ```
   panic% make && make test
   panic# make install
   ```

7. Create an SSL certificate and install Apache and the certificate files:

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

8. Now proceed with the configuration of the Apache-SSL and mod_perl parts of the server configuration files before starting the server. Refer to the Apache-SSL documentation to learn how to configure the SSL section of *httpd.conf*.

Now start the server:

```
panic# /usr/local/apache/bin/httpsdctl start
```

Note that by default, Apache-SSL uses *httpsdctl* instead of *apachectl*.

You should see something like this in */usr/local/apache/logs/httpsd_error_log*:

```
[Sat May 18 14:14:12 2002] [notice]
Apache/1.3.24 (Unix) mod_perl/1.26 Ben-SSL/1.48 (Unix)
configured -- resuming normal operations
```

Installing mod_perl with Stronghold

Stronghold is a secure SSL web server for Unix that allows you to give your web site full-strength, 128-bit encryption. It's a commercial product provided by Red Hat. See *http://www.redhat.com/software/apache/stronghold/* for more information.

To install Stronghold:

1. First, build and install Stronghold without mod_perl, following Stronghold's installation procedure.

2. Having done that, download the mod_perl sources:

   ```
   panic% lwp-download http://perl.apache.org/dist/mod_perl-1.xx.tar.gz
   ```

3. Unpack mod_perl:

   ```
   panic% tar xvzf mod_perl-1.xx.tar.gz
   ```

4. Configure mod_perl with Stronghold (assuming that you have the Stronghold sources extracted to */usr/local/stronghold*):

   ```
   panic% cd mod_perl-1.xx
   panic% perl Makefile.PL APACHE_SRC=/usr/local/stronghold/src \
       DO_HTTPD=1 USE_APACI=1 EVERYTHING=1
   ```

5. Build mod_perl:

   ```
   panic% make
   ```

6. Before running *make test*, add your *StrongholdKey* to *t/conf/httpd.conf*. If you are configuring by hand, be sure to edit *src/modules/perl/Makefile* and uncomment the `#APACHE_SSL` directive.

7. Test and install mod_perl:

   ```
   panic% make test
   panic# make install
   ```

8. Install Stronghold:

   ```
   panic# cd /usr/local/stronghold
   panic# make install
   ```

Note for Solaris 2.5 Users

There has been a report that after building Apache with mod_perl, the the `REGEX` library that comes with Stronghold produces core dumps. To work around this problem, change the following line in *$STRONGHOLD/src/Configuration*:

```
Rule WANTHSREGEX=default
```

to:

```
Rule WANTHSREGEX=no
```

Now start the server:

```
panic# /usr/local/stronghold/bin/start-server
```

It's possible that the start script will have a different name on your platform.

You should see something like this in */usr/local/stronghold/logs/error_log*:

```
[Sun May 19 11:54:39 2002] [notice]
StrongHold/3.0 Apache/1.3.24 (Unix) mod_perl/1.26
configured -- resuming normal operations
```

Installing mod_perl with the CPAN.pm Interactive Shell

Installation of mod_perl and all the required packages is much easier with the help of the CPAN.pm module, which provides, among other features, a shell interface to the CPAN repository (see the Preface).

First, download the Apache source code and unpack it into a directory (the name of which you will need very soon).

Now execute:

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

You will see the cpan prompt:

```
cpan>
```

All you need to do to install mod_perl is to type:

```
cpan> install mod_perl
```

You will see something like the following:

```
Running make for DOUGM/mod_perl-1.xx.tar.gz
Fetching with LWP:
http://www.cpan.org/authors/id/DOUGM/mod_perl-1.xx.tar.gz

CPAN.pm: Going to build DOUGM/mod_perl-1.xx.tar.gz
```

(As with earlier examples in this book, we use *x.xx* as a placeholder for real version numbers, because these change very frequently.)

CPAN.pm will search for the latest Apache sources and suggest a directory. If the CPAN shell did not find your version of Apache and suggests the wrong directory name, type the name of the directory into which you unpacked Apache:

```
Enter 'q' to stop search
Please tell me where I can find your apache src
[../apache_1.3.xx/src]
```

Answer yes to the following questions, unless you have a good reason not to:

```
Configure mod_perl with /home/stas/src/apache_1.3.xx/src ? [y]
Shall I build httpd in /home/stas/src/apache_1.3.xx/src for you? [y]
```

After you have built mod_perl and Apache, tested mod_perl, and installed its Perl modules, you can quit the CPAN shell and finish the installation. Go to the Apache source root directory and run:

```
cpan> quit
panic% cd /home/stas/src/apache_1.3.xx
panic% make install
```

This will complete the installation by installing Apache's headers and the *httpd* binary into the appropriate directories.

The only caveat of the process we've just described is that you don't have control over the configuration process. But that problem is easy to solve—you can tell CPAN.pm to pass whatever parameters you want to *perl Makefile.PL*. You do this with the *o conf makepl_arg* command:

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

If you had previously set *makepl_arg* to some value, you will probably want to save it somewhere so that you can restore it when you have finished with the mod_perl installation. In that case, type the following command first:

```
cpan> o conf makepl_arg
```

and copy its value somewhere before unsetting the variable.

List all the parameters as if you were passing them to the familiar *perl Makefile.PL*. If you add the APACHE_SRC=/home/stas/src/apache_1.3.xx/src and DO_HTTPD=1 parameters, you will not be asked a single question.

Now proceed with *install mod_perl* as before. When the installation is complete, remember to reset the makepl_arg variable by executing:

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

Note that if there was a previous value, use that instead of ''. You can now install all the modules you want to use with mod_perl. You can install them all at once with a single command:

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

This will install mod_perl if hasn't already been installed. It will also install many other packages, such as ExtUtils::Embed, MIME::Base64, URI::URL, Digest::MD5, Net::FTP, LWP, HTML::TreeBuilder, CGI, Devel::Symdump, Apache::DB, Tie::IxHash, Data::Dumper, and so on.

Bundling Modules

If you have a system that's already configured with all the Perl modules you use, making your own bundle is a way to replicate them on another system without worrying about binary incompatibilities. To accomplish this, the command *autobundle* can be used on the CPAN shell command line. This command writes a bundle definition file for all modules that are installed for the currently running Perl interpreter.

With the clever bundle file you can then simply say:

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

and, after answering a few questions, go out for a coffee.

Installing mod_perl on Multiple Machines

You may want to build *httpd* once and then copy it to other machines. But the Perl side of mod_perl needs the Apache header files to compile. To avoid dragging and build Apache on all your other machines, there are a few *Makefile* targets in mod_perl to help you out:

```
panic% make tar_Apache
```

This will make a *tar* file (*Apache.tar*) of all the files mod_perl installs in your Perl's *site_perl* directory. You can then unpack this under the *site_perl* directory on another machine:

```
panic% make offsite-tar
```

This will copy all the header files from the Apache source directory against which you configured mod_perl. It will then run *make dist*, which creates a *mod_perl-1.xx.tar.gz* file, ready to unpack on another machine to compile and install the Perl side of mod_perl.

If you really want to make your life easy, you should use one of the more advanced packaging systems. For example, almost all Linux distributions use packaging tools on top of plain *tar.gz*, allowing you to track prerequisites for each package and providing for easy installation, upgrade, and cleanup. One of the most widely used packagers is the Red Hat Package Manager (RPM). See *http://www.rpm.org/* for more information.

Under RPM, all you have to do is prepare a source distribution package (SRPM) and then build a binary release. The binary can be installed on any number of machines in a matter of seconds.

RPM will even work on live production machines. Suppose you have two identical machines (with identical software and hardware, although, depending on your setup, identical hardware may be less critical). Let's say that one is a live server and the other is for development. You build an RPM with a mod_perl binary distribution, install it on the development machine, and make sure that it is working and stable. You can then install the RPM package on the live server without any fear. Make sure that *httpd.conf* is correct, since it generally specifies parameters that are unique to the live machine (for example, the hostname).

When you have installed the package, just restart the server. It's a good idea to keep a package of the previous system, in case something goes wrong. You can then easily remove the installed package and put the old one back in case of problems.

Installation into a Nonstandard Directory

There are situations when you need to install mod_perl-enabled Apache and other components (such as Perl libraries) into nonstandard locations. For example, you might work on a system to which you don't have *root* access, or you might need to install more than one set of mod_perl-enabled Apache and Perl modules on the same machine (usually when a few developers are using the same server and want to have their setups isolated from each other, or when you want to test a few different setups on the same machine).

We have already seen that you can install mod_perl-enabled Apache into different directories on the system (using the `APACHE_PREFIX` parameter of *Makefile.PL*). Until now, all our scenarios have installed the Perl files that are part of the mod_perl package into the same directory as the system Perl files (usually */usr/lib/perl5*).

Now we are going to show how can you install both the Apache and the Perl files into a nonstandard directory. We'll show a complete installation example using *stas* as a username, assuming that */home/stas* is the home directory of that user.

Installing Perl Modules into a Nonstandard Directory

Before we proceed, let's look at how to install any Perl module into a nonstandard directory. For an example, let's use the package that includes `CGI.pm` and a few other `CGI::*` modules.

First, you have to decide where to install the modules. The simplest approach is to simulate the portion of the / filesystem relevant to Perl under your home directory. Actually, we need only two directories:

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

We don't have to create them, as they are created automatically when the first module is installed. Ninety-nine percent of the files will go into the *lib* directory. Only occasionally does a module distribution come with Perl scripts that are installed into the *bin* directory, at which time *bin* will be created if it doesn't exist.

As usual, download the package from the CPAN repository (*CGI.pm-x.xx.tar.gz*), unpack it, and *chdir* to the newly created directory.

Now do a standard *perl Makefile.PL* to create the *Makefile*, but this time make use of your nonstandard Perl installation directory instead of the default one:

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

Specifying `PREFIX=/home/stas` is the only part of the installation process that is different from usual. Note that if you don't like how *Makefile.PL* chooses the rest of the directories, or if you are using an older version of it that requires an explicit declaration of all the target directories, you should do this:

```
panic% 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:

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

make install installs all the files in the private repository. Note that all the missing directories are created automatically, so you don't need to create them beforehand. Here is what it does (slightly edited):

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

If you have to use explicit target parameters instead of a single PREFIX parameter, you will find it useful to create a file called something like *~/.perl_dirs* (where ~ is */home/stas* in our example), containing:

```
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, simply execute:

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

Using this technique, you can easily maintain several Perl module repositories. For example, you could have one for production and another for development:

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

or:

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

Finding Modules Installed in Nonstandard Directories

Installing Perl modules into nonstandard directories is only half the battle. We also have to let Perl know what these directories are.

Perl modules are generally placed in four main directories. To find these directories, execute:

```
panic% perl -V
```

The output contains important information about your Perl installation. At the end you will see:

```
Characteristics of this binary (from libperl):
Built under linux
Compiled at Oct 14 2001 17:59:15
@INC:
  /usr/lib/perl5/5.6.1/i386-linux
  /usr/lib/perl5/5.6.1
  /usr/lib/perl5/site_perl/5.6.1/i386-linux
  /usr/lib/perl5/site_perl/5.6.1
  /usr/lib/perl5/site_perl
  .
```

This shows us the content of the Perl special variable `@INC`, which is used by Perl to look for its modules. It is equivalent to the `PATH` environment variable, used to find executable programs in Unix shells.

Notice that Perl looks for modules in the . directory too, which stands for the current directory. It's the last entry in the above output.

This example is from Perl Version 5.6.1, installed on our x86 architecture PC running Linux. That's why you see *i386-linux* and *5.6.1*. If your system runs a different version of Perl, or a different operating system, processor, or chipset architecture, then some of the directories will have different names.

All the platform-specific files (such as compiled C files glued to Perl with `XS`, or some *.h* header files) are supposed to go into the *i386-linux*-like directories. Pure Perl modules are stored in the non-platform-specific directories.

As mentioned earlier, you find the exact directories used by your version of Perl by executing *perl -V* and replacing the global Perl installation's base directory with your home directory. Assuming that we use Perl 5.6.1, in our example the directories are:

```
/home/stas/lib/perl5/5.6.1/i386-linux
/home/stas/lib/perl5/5.6.1
/home/stas/lib/perl5/site_perl/5.6.1/i386-linux
/home/stas/lib/perl5/site_perl/5.6.1
/home/stas/lib/perl5/site_perl
```

There are two ways to tell Perl about the new directories: you can either modify the `@INC` variable in your scripts or set the `PERL5LIB` environment variable.

Modifying @INC

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

```
use lib qw(/home/stas/lib/perl5/5.6.1/
           /home/stas/lib/perl5/site_perl/5.6.1
           /home/stas/lib/perl5/site_perl
);
```

Another way is to write code to modify @INC explicitly:

```
BEGIN {
    unshift @INC,
        qw(/home/stas/lib/perl5/5.6.1/i386-linux
           /home/stas/lib/perl5/5.6.1
           /home/stas/lib/perl5/site_perl/5.6.1/i386-linux
           /home/stas/lib/perl5/site_perl/5.6.1
           /home/stas/lib/perl5/site_perl
        );
}
```

Note that with the lib module, we don't have to list the corresponding architecture-specific directories—it adds them automatically if they exist (to be exact, when *$dir/$archname/auto* exists). It also takes care of removing any duplicated entries.

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

Both approaches modify the value of @INC at compilation time. The lib module uses the BEGIN block internally.

Using the PERL5LIB environment variable

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

There is no way for Perl to know that we have some locally installed modules. All it does is search the directories listed in @INC, and because @INC contains only the default four directories (plus the . directory), it cannot find the locally installed LWP package. We cannot solve this problem by adding code to modify @INC, but changing the PERL5LIB environment variable will do the trick.

How to define an environment variable varies according to which shell you use. Bourne-style shell users can split a long line using the backslash (\):

```
panic% export PERL5LIB=/home/stas/lib/perl5/5.6.1:\
/home/stas/lib/perl5/site_perl/5.6.1:\
/home/stas/lib/perl5/site_perl
```

In the C-style shells, however, you'll have to make sure that the value of the PERL5LIB environment variable is specified as one continuous line with no newlines or spaces:

```
panic% setenv PERL5LIB /home/stas/lib/perl5/5.6.1:
/home/stas/lib/perl5/site_perl/5.6.1:
/home/stas/lib/perl5/site_perl
```

(In this example, the lines were split to make them fit on the page.)

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

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

```
panic% perl -V

Characteristics of this binary (from libperl):
Built under linux
Compiled at Apr  6 1999 23:34:07
%ENV:
  PERL5LIB="/home/stas/lib/perl5/5.6.1:
  /home/stas/lib/perl5/site_perl/5.6.1:
  /home/stas/lib/perl5/site_perl"
@INC:
  /home/stas/lib/perl5/5.6.1/i386-linux
  /home/stas/lib/perl5/5.6.1
  /home/stas/lib/perl5/site_perl/5.6.1/i386-linux
  /home/stas/lib/perl5/site_perl/5.6.1
  /home/stas/lib/perl5/site_perl
  /usr/lib/perl5/5.6.1/i386-linux
  /usr/lib/perl5/5.6.1
  /usr/lib/perl5/site_perl/5.6.1/i386-linux
  /usr/lib/perl5/site_perl/5.6.1
  /usr/lib/perl5/site_perl
  .
```

When everything works as you want it to, add these commands to your *.tcshrc*, *.bashrc*, *C:\autoexec.bat* or another equivalent file.* The next time you start a shell, the environment will be ready for you to work with the new Perl directories.

Note that if you have a `PERL5LIB` setting, you don't need to alter the `@INC` value in your scripts. But if someone else (who doesn't have this setting in the shell) tries to execute your scripts, Perl will fail to find your locally installed modules. This includes *cron* scripts, which *might* use a different shell environment (in which case the `PERL5LIB` setting won't be available).

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

Using the CPAN.pm Shell with Nonstandard Installation Directories

As we saw previously in this chapter, using the `CPAN.pm` shell to install mod_perl saves a great deal of time. It does the job for us, even detecting the missing modules

* These files are run by the shell at startup and allow you to set environment variables that might be useful every time you use your shell.

listed in prerequisites, fetching them, and installing them. So you might wonder whether you can use `CPAN.pm` to maintain your local repository as well.

When you start the CPAN interactive shell, it searches first for the user's private configuration file and then for the system-wide one. For example, for a user *stas* and Perl Version 5.6.1, it will search for the following configuration files:

```
/home/stas/.cpan/CPAN/MyConfig.pm
/usr/lib/perl5/5.6.1/CPAN/Config.pm
```

If there is no CPAN shell configured on your system, when you start the shell for the first time it will ask you a dozen configuration questions and then create the *Config.pm* file for you.

If the CPAN shell is already configured system-wide, you should already have a */usr/lib/perl5/5.6.1/CPAN/Config.pm* file. (As always, if you have a different Perl version, the path will include a different version number.) Create the directory for the local configuration file as well:

```
panic% mkdir -p /home/stas/.cpan/CPAN
```

(On many systems, *mkdir -p* creates the whole path at once.)

Now copy the system-wide configuration file to your local one:

```
panic% cp /usr/lib/perl5/5.6.1/CPAN/Config.pm /home/stas/.cpan/CPAN/MyConfig.pm
```

The only thing left is to change the base directory of *.cpan* in your local file to the one under your home directory. On our machine, we replace */root/.cpan* (which is where our system's *.cpan* directory resides) with */home/stas*. Of course, we use Perl to edit the file:

```
panic% perl -pi -e 's|/root|/home/stas|' \
    /home/stas/.cpan/CPAN/MyConfig.pm
```

Now that you have the local configuration file ready, you have to tell it what special parameters you need to pass when executing *perl Makefile.PL*. Open the file in your favorite editor and replace the following line:

```
'makepl_arg' => q[ ],
```

with:

```
'makepl_arg' => q[PREFIX=/home/stas],
```

Now you've finished the configuration. Assuming that you are logged in with the same username used for the local installation (*stas* in our example), start it like this:

```
panic% perl -MCPAN -e shell
```

From now on, any module you try to install will be installed locally. If you need to install some system modules, just become the superuser and install them in the same way. When you are logged in as the superuser, the system-wide configuration file will be used instead of your local one.

If you have used more than just the PREFIX variable, modify *MyConfig.pm* to use the other variables. For example, if you have used these variables during the creation of the *Makefile*:

```
panic% 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
```

replace PREFIX=/home/stas in the line:

```
'makepl_arg' => q[PREFIX=/home/stas],
```

with all the variables from above, so that the line becomes:

```
'makepl_arg' => q[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
],
```

If you arrange all the above parameters in one line, you can remove the backslashes (\).

Making a Local Apache Installation

Just as with Perl modules, if you don't have the permissions required to install Apache into the system area, you have to install them locally under your home directory. It's almost the same as a plain installation, but you have to run the server listening to a port number greater than 1024 (only *root* processes can listen to lower-numbered ports).

Another important issue you have to resolve is how to add startup and shutdown scripts to the directories used by the rest of the system services. Without *root* access, you won't be able to do this yourself; you'll have to ask your system administrator to assist you.

To install Apache locally, all you have to do is to tell *./configure* in the Apache source directory what target directories to use. If you are following the convention that we use, which makes your home directory look like the / (base) directory, the invocation parameters will be:

```
panic% ./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 they are, add the *--show-layout* option before you proceed:

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

You might want to put all the Apache files under */home/stas/apache*, following Apache's convention:

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

If you want to modify some or all of the names of the automatically created directories, use the *--sbindir*, *--sysconfdir*, and *--logfiledir* options:

```
panic% ./configure --prefix=/home/stas/apache \
    --sbindir=/home/stas/apache/sbin          \
    --sysconfdir=/home/stas/apache/conf       \
    --logfiledir=/home/stas/apache/logs
```

Refer to the output of *./configure --help* for all available options.

Also remember that you can start the script only under a user and group to which you belong, so you must set the `User` and `Group` directives in *httpd.conf* to appropriate values.

Furthermore, as we said before, the `Port` directive in *httpd.conf* must be adjusted to use an unused port above 1024, such as 8080. This means that when users need to access the locally installed server, their URLs need to specify the port number (e.g., *http://www.example.com:8080/*). Otherwise, browsers will access the server running on port 80, which isn't the one you installed locally.

Nonstandard mod_perl-Enabled Apache Installation

Now that we know how to install local Apache and Perl modules separately, let's see how to install mod_perl-enabled Apache in our home directory. It's almost as simple as doing each one separately, but there is one wrinkle. We'll talk about it at the end of this section.

Let's say you have unpacked the Apache and mod_perl sources under */home/stas/src* and they look like this:

```
panic% ls /home/stas/src
/home/stas/src/apache_1.3.xx
/home/stas/src/mod_perl-1.xx
```

where *x.xx* are replaced by the real version numbers, as usual. You want the Perl modules from the mod_perl package to be installed under */home/stas/lib/perl5* and the Apache files to go under */home/stas/apache*. The following commands will do that for you:

```
panic% perl Makefile.PL \
    PREFIX=/home/stas \
    APACHE_PREFIX=/home/stas/apache \
    APACHE_SRC=../apache_1.3.xx/src \
    DO_HTTPD=1 \
    USE_APACI=1 \
    EVERYTHING=1
```

```
panic% make && make test && make install
panic% cd ../apache_1.3.xx
panic% make install
```

If you need some parameters to be passed to the *./configure* script, as we saw in the previous section, use APACI_ARGS. For example:

```
APACI_ARGS='--sbindir=/home/stas/apache/sbin  \
   --sysconfdir=/home/stas/apache/conf       \
   --logfiledir=/home/stas/apache/logs'
```

Note that the above multiline splitting will work only with Bourne-style shells. C-style shell users will have to list all the parameters on a single line.

Basically, the installation is complete. The only remaining problem is the @INC variable. This won't be correctly set if you rely on the PERL5LIB environment variable unless you set it explicitly in a startup file that is required before loading any other module that resides in your local repository. A much nicer approach is to use the lib pragma, as we saw before, but in a slightly different way—we use it in the startup file and it affects all the code that will be executed under mod_perl handlers. For example:

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

where *startup.pl* starts with:

```
use lib qw(/home/stas/lib/perl5/5.6.1/
           /home/stas/lib/perl5/site_perl/5.6.1
           /home/stas/lib/perl5/site_perl
);
```

Note that you can still use the hardcoded @INC modifications in the scripts themselves, but be aware that scripts modify @INC in BEGIN blocks and mod_perl executes the BEGIN blocks only when it performs script compilation. As a result, @INC will be reset to its original value after the scripts are compiled, and the hardcoded settings will be forgotten.

The only time you can alter the "original" value is during the server configuration stage, either in the startup file or by putting the following line in *httpd.conf*:

```
PerlSetEnv Perl5LIB \
/home/stas/lib/perl5/5.6.1/:/home/stas/lib/perl5/site_perl/5.6.1
```

But the latter setting will be ignored if you use the PerlTaintcheck setting, and we hope you do use it. See the *perlrun* manpage for more information.

The rest of the mod_perl configuration can be done just as if you were installing mod_perl as *root*.

Resource Usage

Another important thing to keep in mind is the consumption of system resources. mod_perl is memory-hungry. If you run a lot of mod_perl processes on a public, multiuser machine, most likely the system administrator of this machine will ask you to use fewer resources and may even shut down your mod_perl server and ask you to find another home for it. You have a few options:

- Reduce resource usage as explained in Chapter 21.
- Ask your ISP's system administrator whether she can set up a dedicated machine for you, so that you will be able to install as much memory as you need. If you get a dedicated machine, chances are that you will want to have *root* access, so you may be able to manage the administration yourself. You should also make sure the system administrator is responsible for a reliable electricity supply and a reliable network link. The system administrator should also make sure that the important security patches get applied and the machine is configured to be secure (not to mention having the machine physically protected, so no one will turn off the power or break it).
- The best solution might be to look for another ISP with lots of resources or one that supports mod_perl. You can find a list of these ISPs at *http://perl.apache.org/*.

Nonstandard mod_perl-Enabled Apache Installation with CPAN.pm

Again, CPAN makes installation and upgrades simpler. You have seen how to install a mod_perl-enabled server using `CPAN.pm`'s interactive shell. You have seen how to install Perl modules and Apache locally. Now all you have to do is to merge these techniques.

Assuming that you have configured `CPAN.pm` to install Perl modules locally, the installation is very simple. Start the CPAN shell, set the arguments to be passed to *perl Makefile.PL* (modify the example setting to suit your needs), and tell `CPAN.pm` to do the rest for you:

```
panic% perl -MCPAN -eshell
cpan> o conf makepl_arg 'DO_HTTPD=1 USE_APACI=1 EVERYTHING=1 \
      PREFIX=/home/stas APACHE_PREFIX=/home/stas/apache'
cpan> install mod_perl
```

When you use `CPAN.pm` for local installation, you need to make sure that the value of `makepl_arg` is restored to its original value after the mod_perl installation is complete, because if you install other Perl modules you probably don't want to pass mod_perl flags to them. The simplest way to do this is to quit the interactive shell and then re-enter it. There is another way to do it without quitting, but it's very cumbersome—if you want to learn about the other option, refer to the `CPAN.pm` manpage.

How Can I Tell if mod_perl Is Running?

There are several ways to find out if mod_perl is enabled in your version of Apache. In older versions of Apache (versions earlier than 1.3.6), you could check by running *httpd -v*, but that no longer works. Now you should use *httpd -l*.

It is not enough to know that mod_perl is installed—the server needs to be configured for mod_perl as well. Refer to Chapter 4 to learn about mod_perl configuration.

Checking the error_log File

One way to check for mod_perl is to check the *error_log* file for the following message at server startup:

```
[Sat May 18 18:08:01 2002] [notice]
Apache/1.3.24 (Unix) mod_perl/1.26 configured
  -- resuming normal operations
```

Testing by Viewing /perl-status

Assuming that you have configured the `<Location /perl-status>` section in the server configuration file as explained in Chapter 9, fetch *http://www.example.com/perl-status/* using your favorite browser.

You should see something like this:

```
Embedded Perl version 5.6.1 for Apache/1.3.24 (Unix)
mod_perl/1.26 process 50880,
running since Sat May 18 18:08:01 2002
```

Testing via Telnet

Knowing the port you have configured Apache to listen on, you can use `Telnet` to talk directly to it.

Assuming that your mod_perl-enabled server listens to port 8080,* telnet to your server at port 8080, type `HEAD / HTTP/1.0`, and then press the Enter key twice:

```
panic% telnet localhost 8080
HEAD / HTTP/1.0

```

You should see a response like this:

```
HTTP/1.1 200 OK
Date: Mon, 06 May 2002 09:49:41 GMT
Server: Apache/1.3.24 (Unix) mod_perl/1.26
```

* If in doubt, try port 80, which is the standard HTTP port.

```
Connection: close
Content-Type: text/html; charset=iso-8859-1

Connection closed.
```

The line:

```
Server: Apache/1.3.24 (Unix) mod_perl/1.26
```

confirms that you have mod_perl installed and that its version is 1.26.

Testing via a CGI Script

Another method to test for mod_perl is to invoke a CGI script that dumps the server's environment.

We assume that you have configured the server so that scripts running under the location */perl/* are handled by the `Apache::Registry` handler and that you have the `PerlSendHeader` directive set to `On`.

Copy and paste the script below. Let's say you name it *test.pl* and save it at the root of the CGI scripts, which is mapped directly to the */perl* location of your server.

```
print "Content-type: text/plain\n\n";
print "Server's environment\n";
foreach ( keys %ENV ) {
    print "$_\t$ENV{$_}\n";
}
```

Make it readable and executable by the server (you may need to tune these permissions on a public host):

```
panic% chmod a+rx test.pl
```

Now fetch the URL *http://www.example.com:8080/perl/test.pl* (replacing 8080 with the port your mod_perl-enabled server is listening to). You should see something like this (the output has been edited):

```
SERVER_SOFTWARE Apache/1.3.24 (Unix) mod_perl/1.26
GATEWAY_INTERFACE       CGI-Perl/1.1
DOCUMENT_ROOT   /home/httpd/docs
REMOTE_ADDR     127.0.0.1
[more environment variables snipped]
MOD_PERL        mod_perl/1.21_01-dev
[more environment variables snipped]
```

If you see the that the value of `GATEWAY_INTERFACE` is `CGI-Perl/1.1`, everything is OK.

If there is an error, you might have to add a shebang line (`#!/usr/bin/perl`) as the first line of the CGI script and then try it again. If you see:

```
GATEWAY_INTERFACE       CGI/1.1
```

it means you have configured this location to run under mod_cgi and not mod_perl.

Also note that there is a $ENV{MOD_PERL} environment variable if you run under a mod_perl handler. This variable is set to the mod_perl/1.xx string, where 1.xx is the version number of mod_perl.

Based on this difference, you can write code like this:

```
BEGIN {
    # perl5.004 or better is a must under mod_perl
    require 5.004 if $ENV{MOD_PERL};
}
```

If you develop a generic Perl module aimed at mod_perl, mod_cgi, and other runtime environments, this information comes in handy, because it allows you to do mod_perl-specific things when running under mod_perl. For example, CGI.pm is mod_perl-aware: when CGI.pm knows that it is running under mod_perl, it registers a cleanup handler for its global $Q object, retrieves the query string via Apache->request->args, and does a few other things differently than when it runs under mod_cgi.

Testing via lwp-request

Assuming you have the libwww-perl (LWP) package installed, you can run the following tests. (Most likely you do have it installed, since you need it to pass mod_perl's *make test*.)

```
panic% lwp-request -e -d http://www.example.com
```

This shows you just the headers; the *-d* option disables printing the response content. If you just want to see the server version, try:

```
panic% lwp-request -e -d http://www.example.com | egrep '^Server:'
```

Of course, you should use *http://www.example.com:port_number* if your server is listening to a port other than port 80.

General Notes

This section looks at some other installation issues you may encounter.

How Do I Make the Installation More Secure?

Unix systems usually provide *chroot* or *jail* mechanisms, which allow you to run subsystems isolated from the main system. So if a subsystem gets compromised, the whole system is still safe.

The section titled "Apache" in Chapter 23 includes a few references to articles discussing these mechanisms.

Can I Run mod_perl-Enabled Apache as suExec?

The answer is definitively "no." You can't *suid* a part of a process. mod_perl lives inside the Apache process, so its UID and GID are the same as those of the Apache process.

You have to use mod_cgi if you need this functionality. See Appendix C for other possible solutions.

Should I Rebuild mod_perl if I Have Upgraded Perl?

Yes, you should. You have to rebuild the mod_perl-enabled server, because it has a hardcoded `@INC` variable. This points to the old Perl and is probably linked to an old `libperl` library. If for some reason you need to keep the old Perl version around, you can modify `@INC` in the startup script, but it is better to build afresh to save you from getting into a mess.

mod_auth_dbm Nuances

If you are a `mod_auth_dbm` or `mod_auth_db` user, you may need to edit Perl's `Config` module. When Perl is configured, it attempts to find libraries for `ndbm`, `gdbm`, `db`, etc. for the `DB*_File` modules. By default, these libraries are linked with Perl and remembered by the `Config.pm` module. When mod_perl is configured with Apache, the `ExtUtils::Embed` module requires these libraries to be linked with *httpd* so Perl extensions will work under mod_perl. However, the order in which these libraries are stored in *Config.pm* may confuse `mod_auth_db*`. If `mod_auth_db*` does not work with mod_perl, take a look at the order with the following command:

```
panic% perl -V:libs
```

Here's an example:

```
libs='-lnet -lnsl_s -lgdbm -lndbm -ldb -ldld -lm -lc -lndir -lcrypt';
```

If `-lgdbm` or `-ldb` is before `-lndbm` (as it is in the example), edit *Config.pm* and move `-lgdbm` and `-ldb` to the end of the list. Here's how to find *Config.pm*:

```
panic% perl -MConfig -e 'print "$Config{archlibexp}/Config.pm\n"'
```

Under Solaris, another solution for building mod_perl- and mod_auth_dbm-enabled Apache is to remove the DBM and NDBM "emulation" from *libgdbm.a*. It seems that Solaris already provides its own DBM and NDBM, and in our installation we found there's no reason to build GDBM with them.

In our *Makefile* for GDBM, we changed:

```
OBJS = $(DBM_OF) $(NDBM_OF) $(GDBM_OF)
```

to:

```
    OBJS = $(GDBM_OF)
```

Then rebuild `libgdbm` before building mod_perl-enabled Apache.

References

- Apache Toolbox (*http://apachetoolbox.com/*) provides a means to easily compile Apache with about 60 different Apache modules. It is fully customizable and menu-driven. Everything is compiled from source. It checks for RPMs that might cause problems and uses *wget* to download the source automatically if it's missing.
- Several Apache web server books that discuss the installation details are listed at *http://httpd.apache.org/info/apache_books.html*.