CHAPTER 7
Creating Handlers

Introduction

Some of the recipes in the preceding chapters illustrated concepts by using a complete handler as the example, and a few others fully enclosed a handler within a module. Although the term *handler* generally refers to the `handler()` subroutine, the colloquial meaning includes the details of the module that implements the handler. It is this more inclusive meaning that we are examining here—the mod_perl handler as an entity that interacts with a part of the Apache lifecycle, complete with all the bells and whistles.

What makes handlers so powerful is their ability to actually modify how Apache behaves at the server level—the perl interpreter that runs your mod_perl handler is embedded in Apache and uses the Apache API to interact with base server functions. You can choose to override default server behaviors, insert new functionality, or do nothing and let Apache continue doing what it does best. This is completely different from other models, such as SSI, CGI, ASP, and the various Java-based tools, which rely on mechanisms isolated from the actual operation of the server. With mod_perl handlers, functionality that is difficult or impossible for these other platforms becomes rather easy, now that you have the ability to program *within* Apache's framework instead of *around* it.

In its most fundamental form, a mod_perl handler is a Perl module that contains (at least) a single subroutine named `handler()`. This subroutine code is executed during a specific phase of the Apache lifecycle and can be used to create content, alter server behavior, and just about anything else for which the Apache API provides an interface.

The Introduction to Part II provides a basic introduction to handlers and how they interact with the Apache lifecycle at a high level. Part III will expand on that concept further and get into the details of applying handlers to specific operational phases. This chapter presents the fundamentals of creating and configuring handlers so you can fully leverage the mod_perl API. If you have thus far found the notion of a handler to be a bit esoteric, this chapter should solidify things and get you well on your way.

7.1. Creating a mod_perl Handler

You want to create a mod_perl handler.

Technique

Create a Perl package with a subroutine named `handler()`.

```
package Cookbook::Clean;

use Apache::Constants qw( OK DECLINED );
use Apache::File;
use Apache::Log;

use HTML::Clean;

use strict;

sub handler {

  my $r = shift;

  my $log = $r->server->log;

  unless ($r->content_type eq 'text/html') {
    $log->info("Request is not for an html document - skipping...");
    return DECLINED;
  }
  my $fh = Apache::File->new($r->filename);

  unless ($fh) {
    $log->warn("Cannot open request - skipping... $!");
    return DECLINED;
  }

  # Slurp the file (hopefully it's not too big).
  my $dirty = do {local $/; <$fh>};

  # Create the new HTML::Clean object.
  my $h = HTML::Clean->new(\$dirty);

  # Set the level of suds.
  $h->level(3);

  # Clean the HTML.
  $h->strip;

  # Send the crisp, clean data.
  $r->send_http_header('text/html');
  print ${$h->data};

  return OK;
}
1;
__END__

=head1 NAME

Cookbook::Clean - Apache content handler that cleans HTML of cruft

=head1 SYNOPSIS

DocumentRoot /usr/local/apache/htdocs

PerlModule Cookbook::Clean

<Directory /usr/local/apache/htdocs>
  SetHandler perl-script
  PerlHandler Cookbook::Clean
</Directory>

=head1 DESCRIPTION
```

```
Cleans HTML by "scrubbing the deck" of redundant
whitespace and other useless data.  This is basically a
mod_perl interface into HTML::Clean.

=cut
```

Comments

You might have noticed that the preceding chapters have already made rather elaborate use of handlers. Hence, instead of the typical jaded introductory "Hello World!" example, here you will find a simple yet useful mod_perl handler. The handler you see in this recipe is a content handler that is installed as a `PerlHandler`. It sends the requested file to the client after removing all the unsightly HTML added by modern WYSIWYG editors.

A handler is just a `handler()` subroutine contained within a standard Perl module. There really is not much special about it compared to your standard run-of-the-mill Perl module, other than the handler contains code designed to use the features and syntax of the mod_perl API. Actually, the subroutine neither has to be named `handler()` nor be in a module—it really can be any subroutine in a named package, as you will see later. The best way to get a feel for a typical mod_perl handler is to go through the preceding example. Most of this should not be new to anyone already familiar with writing Perl modules, but because modules are such an integral part of mod_perl and provide the foundation for nearly all that the mod_perl programmer does, we can afford to spend some time on the basics here.

First you need to decide on a name for your module. In this example we chose the name `Cookbook::Clean`, referring to the way we "clean" all the excess whitespace and gratuitous use of grandiose tags found in a typical HTML document. For internal applications, sticking to a separate yet appropriate namespace is best, such as the name of your project or company (hence our `Cookbook::` designation). The `Apache::` namespace designation on CPAN means that the module is not intended for use outside of the mod_perl environment. This is merely a convention, but a good one to adopt, especially if you intend to release your module publicly. You will want to come up with your own witty yet descriptive name for your module.

Next create the file corresponding to the module, along with any necessary directories. In the preceding case we would create the file `Clean.pm` located in a `Cookbook/` directory someplace where mod_perl can see it. Recipe 2.10 discusses how to maintain module libraries and the several places where mod_perl can find them by default. For a single module that does not include a `Makefile.PL`, a simple solution is to just place it somewhere beneath the Apache `ServerRoot`, say `/usr/local/apache/lib/perl/Cookbook/Clean.pm`.

Add to this file a standard Perl module skeleton, beginning with a package declaration that matches the name of your file (and directory if necessary). If you plan on writing maintainable code, you should add the line `use strict;` to catch common programming errors. Because all Perl modules need to return a true value, ending the code with `1`; is customary. And everybody loves documentation, so be sure to include some.

Now, somewhere in the middle of all this is where we depart from a standard Perl module. Because this is mod_perl, we start by pulling in a few common Apache modules: `Apache::Constants`, `Apache::File`, and `Apache::Log`, each of which ought to be familiar by now. Next we pull in the CPAN module `HTML::Clean`, which does the actual scrubbing; our `Cookbook::Clean` module merely provides a simple interface to it.

Now we come to the all-important `handler()` subroutine, which is called by mod_perl at request time. The typical handler starts by reading the standard Apache request object parameter into the `$r` variable. Because this is of little use in and of itself, we add some processing that does some typical mod_perl things, such as checking whether the request is for an HTML document before actually operating on the file and sending proper HTTP headers. The nonstandard part of the code is what makes our handler unique; the contents of the requested file are passed to `HTML::Clean`, where they are polished to a Perly white and sent to the client.

Throughout the handler we make certain to pass an appropriate response code back to mod_perl, either `DECLINED` if we are passing control back to the core Apache content handler, or `OK` if everything went as planned.

When all is said and done, you will want to configure your new handler so that mod_perl knows what to do with it at runtime. When the handler is actually executed depends on the way it is implemented in `httpd.conf`. Our documentation says to use it as a `PerlHandler`, which signifies that it will be responsible for the content-generation phase of the request. In our case, we map the URI location `/clean` to `DocumentRoot` and pass the file through our `PerlHandler`, allowing end users to see cluttered or clean HTML, depending on the URL they enter in their browser.

```
PerlModule Cookbook::Clean

Alias /clean /usr/local/apache/htdocs
<Location /clean>
  SetHandler perl-script
  PerlHandler Cookbook::Clean
</Location>
```

Although this example is relatively simple, the handler serves a clearly defined purpose and leverages the power of other Perl modules to do most of the work. After you get a feel for how to write basic mod_perl handlers, you will begin to see an entirely new programming world, rife with possibilities, applications, and elegant solutions to problems practically impossible using conventional CGI.

7.2. Basic Handler Configuration

You want to customize the behavior of your handler without changing the source code on a regular basis.

Technique

Add `PerlSetVar` and/or `PerlAddVar` directives to your `httpd.conf` then use `dir_config()` method from the Apache class to access them.

In `httpd.conf`, add

```
PerlModule Cookbook::Clean

<Location /clean>
  SetHandler perl-script
  PerlHandler Cookbook::Clean

  PerlSetVar CleanLevel 3
  PerlSetVar CleanOption whitespace
  PerlAddVar CleanOption shortertags
</Location>
```

Now, make some slight alterations to your module to use the configured values:

```
  # Set the level of suds.
  $h->level($r->dir_config('CleanLevel') || 1);

  my %options = map { $_ => 1 } $r->dir_config->get('CleanOption');

  # Clean the HTML.
  $h->strip(\%options);
```

Comments

You have many ways to separate configuration information from your module, most of which are unmaintainable. Alternatively, asking the user of your module to modify the source or even create a configuration file is a terrible burden. The best place to configure your mod_perl module is inside Apache's `httpd.conf` file.

As already discussed in Recipe 2.14, back in the days of straight CGI, the only scalable solution for script configuration variables was to make use of `%ENV` using `SetEnv`, `/etc/profile`, or other similar methods. If there is nothing of importance you needed to store, then this is (almost) acceptable, but for controlling things like database passwords, the `%ENV` alternative is almost criminally negligent, since any rogue user capable of running a CGI script can see your passwords. With mod_perl, you have two new options: `PerlSetVar` for setting simple Perl variables, and `PerlAddVar` for pushing data onto an array.

Both `PerlSetVar` and `PerlAddVar` inherit all of Apache's configuration sophistication, including the ability to use `<Directory>` or `<Location>` section for fine-grained control over configuration options, conditional configuration using `<IfDefine>`, virtual host merging, and more.

The first step in configuring your handler is defining what the user can change or override. After you've done that, you need to give each configuration variable a name. You might want to give your variables a special prefix, like we've done: All the options we use will start with `Clean`.

Next we modify our script to use the new variables. Our original `Clean.pm` hardcoded the behavior of the `HTML::Clean` object, which is certainly not ideal. Here, we give the user a choice while providing some defaults. Note the `|| 1` construct; this ensures that the call to `$h->level()` is always set to something meaningful, even if no configuration information was provided.

Although the single configuration value interface provided by `PerlSetVar` is nice (and more common), many situations exist where you might want to support multiple values without creating a large quantity of singular variable names. In this case, `PerlAddVar` offers an elegant solution, although it uses a slightly different interface.

In Recipe 3.14 we introduced the `Apache::Table` class and stressed the importance of understanding this class well. As it turns out, the underlying data object for both `PerlSetVar` and `PerlAddVar` is an `Apache::Table` object, so having a solid understanding of the class and its accessor methods will help you here.

If you looked carefully at the example configuration, you saw that `PerlSetVar` was used both to set a single value for `CleanLevel` and to initialize the array for `CleanOption`.

The actual implementation of the PerlSetVar and PerlAddVar directives is equivalent to the set() and add() methods of the Apache::Table class; initializing the CleanOption array is merely a safeguard against the possibility of inheriting an already populated array from the configuration of the server or parent container. Of course, if configuration inheritance is what you are after, you can simply use PerlAddVar exclusively.

Historically, PerlSetVar was the first to arrive on the scene, and as such the dir_config() method adds some syntactic sugar behind the scenes so that programmers can avoid dealing with the Apache::Table object directly. This, however, is not an option with PerlAddVar. Although $r->dir_config('CleanLevel') and $r->dir_config->get('CleanLevel') are equivalent for variables set with PerlSetVar, PerlAddVar requires the use of Apache::Table's get() method to access the entire array of data—using $r->dir_config('CleanOption') will return only the first value in the array.

As illustrated in Recipe 3.14, that both of these directives are, in fact, manipulatable via the Apache::Table class lends itself to a whole new realm of possibilities, such as the ability to set, modify, or delete PerlSetVar settings across the phases of the request.

7.3. Adding Handlers On-the-Fly

You need to insert a small handler as a stopgap and do not want to write a full module.

Technique

Put the handler right in your httpd.conf using an anonymous subroutine.

```
# Quick! keep external people out of this directory for a while
<Location /public>
  PerlAccessHandler 'sub {                                          \
                     return Apache::Constants::FORBIDDEN            \
                       unless shift->connection->remote_ip =~ m/^\Q10.3.4./; \
                   }'
</Location>
```

Comments

At some point, you might have a need for a specific, short-term solution that does not warrant a full module for one reason or another. For these instances, utilizing standard mod_perl configuration directives with anonymous subroutines is possible, as in the preceding example.

When mod_perl encounters a `Perl*Handler` directive, it actually looks for a subroutine to execute in several different forms. The idiomatic configurations shown thus far have allowed mod_perl to assume the subroutine `handler()`, but in fact you can specify any subroutine name you want.

```
# Idiomatic
PerlHandler My::Dinghy

# The same thing
PerlHandler My::Dinghy::handler

# Specify a different subroutine
PerlHandler My::Dinghy::oars

# Use an object-oriented method handler - see Chapter 10
PerlHandler My::Dinghy->outboard
```

The sample code merely shows an extension to this model in which you are also able to use (possibly anonymous) subroutine references as `Perl*Handlers`. This is actually a rather common occurrence when programming with handlers, because the API for the `push_handlers()` and `set_handlers()` methods, as discussed in the next chapter, requires this syntax.

```
$r->push_handlers(PerlAccessHandler => \&forbidden);
$r->set_handlers(PerlTransHandler => [\&OK]);
```

If we extend this a bit further we can throw in a few more possibilities. Because the `FORBIDDEN` constant is really a constant subroutine exported by the `Apache::Constants` class, we can also do something like

```
<Location /public>
  PerlAccessHandler Apache::Constants::FORBIDDEN
</Location>
```

if we don't need to program any logic around our access control. Yet another, little-known solution is to write the handler right in your `startup.pl`:

```
sub Quick::Forbidden::handler {
  return Apache::Constants::FORBIDDEN
    unless shift->connection->remote_ip =~ m/^\Q10.3.4./;
}
```

and then use it as

```
<Location /forbidden>
  PerlAccessHandler Quick::Forbidden
</Location>
```

At this point the location and meaning of the code are becoming rather obscure, quite to the chagrin of your co-workers. To spare them significant torment, you might as well just write a full, if small, handler processed in the usual fashion and stashed somewhere in `@INC`.

7.4. Preparing a Module for Release

You want to make certain that the module you are about to release to CPAN is as clean as possible.

Technique

Be sure that you `use strict;`, `use warnings;`, and that your code can survive running under `PerlTaintCheck On`.

Comments

Congratulations! You have the foresight to understand that, although CPAN is a wonderful tool, the packages on it do not always represent squeaky-clean code. You aim to be different. As a mod_perl programmer, you want to try to do your best to represent all three communities (Perl, Apache, and mod_perl) in the best possible light, and therefore present the tightest release you can.

With any Perl module, using the `strict` and `warnings` pragmas is good practice and, as has been pointed out in earlier chapters, this is especially true in the mod_perl world. The `strict` pragma will keep you from falling prey to the myriad of scoping and referencing errors that can crop up, and is now generally accepted as a must for writing clean code. You will be a better programmer if you always `use strict;`.

The `warnings` pragma is there to help you uncover errors that might not be immediately obvious while coding. These can include extreme problems such as the notorious *`Variable $foo will not stay shared`* warnings that crop up with nested subroutines, or various other possible sources of errors such as *`Scalar value @foo[1] better written as $foo[1]`* warnings. The nested subroutine problem is far more prevalent when programming `Apache::Registry` scripts than it is with handlers, so if you get this warning from your handler you are *really* doing something wrong.

Unlike with the `strict` pragma, which is essentially under your control, warnings can be enabled in mod_perl from outside of your code using the `PerlWarn` configuration directive. This means that the end user of your CPAN module might have `PerlWarn On` in his configuration, which will immediately illuminate your bad programming practices, such as the popular *`Use of uninitialized value`* warning.

The easiest way to fix these "*`initialized value`*" warnings is to properly initialize your variables at the start of your handler.

```
my $man = "overboard";
my %fleet = ();
```

The use of `PerlTaintCheck On` is, of course, a requirement for any code that uses data supplied by an end-user, especially in a Web environment. If you do not understand Perl's taint mode or why it is important, it is time to read the "Handling Insecure Data" section of *Programming Perl*. You will be glad you did. Recipe 15.5 describes an interesting approach to handling tainted data.

All these features will certainly help you when coding, and naturally you will have subjected your module to a range of tests, trying to anticipate all reasonable (and unreasonable) pitfalls. After a point, though, testing further is hard for an author. When you feel the module is ready, sending a message to the mod_perl mailing list is not uncommon, and perhaps also to the *`comp.lang.perl.modules`* newsgroup, asking for beta testers. This can be a valuable, relatively informal way to get initial feedback, and as well as potentially finding some bugs in this way, you could also obtain some suggestions on the documentation included in your module. See the last recipe in this chapter for details on how to release your module when it is ready for primetime.

7.5. Creating a Release Tarball

You want to create a tarball of your module to release to CPAN or distribute across your internal systems.

Technique

Run the `h2xs` command to create the basic files for your module, and then issue `make dist`.

```
$ h2xs -AXn Cookbook::Clean
Writing Cookbook/Clean/Clean.pm
Writing Cookbook/Clean/Makefile.PL
Writing Cookbook/Clean/README
Writing Cookbook/Clean/test.pl
Writing Cookbook/Clean/Changes
Writing Cookbook/Clean/MANIFEST

[time passes, editing is performed, magical things are created]

$ perl Makefile.PL
Checking if your kit is complete...
Looks good
Writing Makefile for Cookbook::Clean

$ make dist
rm -rf Cookbook-Clean-0.01
/usr/bin/perl -I/usr/local/lib/perl5/5.6.1/i686-linux-thread-multi -
I/usr/local/lib/perl5/5.6.1 -MExtUtils::Manifest=manicopy,maniread \
-e "manicopy(maniread(),'Cookbook-Clean-0.01', 'best');"
mkdir Cookbook-Clean-0.01
tar cvf Cookbook-Clean-0.01.tar Cookbook-Clean-0.01
Cookbook-Clean-0.01/
Cookbook-Clean-0.01/README
Cookbook-Clean-0.01/Makefile.PL
Cookbook-Clean-0.01/Changes
Cookbook-Clean-0.01/MANIFEST
Cookbook-Clean-0.01/test.pl
Cookbook-Clean-0.01/Clean.pm
rm -rf Cookbook-Clean-0.01
gzip --best Cookbook -Clean-0.01.tar
```

Comments

Perl provides a number of tools to package, compile, and test a module. By adapting your module to the standard Perl conventions you get a great many features for little or no work, including the ability to make tarballs and install your module quickly and easily in the standard Perl system library. The Perl utility `h2xs` quickly builds the framework of files and commands needed to build your module. These files include

- `Changes`. Text file with changes between versions
- `Clean.pm`. Our actual code

- `Makefile.PL`. Build and install directives for creating the `Makefile`.
- `MANIFEST`. Complete list of files in this package.
- `README`. Useful documentation for how to install and/or use the module.
- `test.pl`. A simple test harness.

Originally `h2xs` was used to create extension modules associated with C header files (thus the name, header files end with `.h` and Perl extension code ends with `.xs`). These days you can use `h2xs` for any type of module, just as long as you pass the correct parameters. We use the `-AX` arguments, which turn off parsing of C structures and disable `AUTOLOAD` support, resulting in a basic Perl module framework.

After running the `h2xs` command, we need to add our code to the appropriate files. In this case we modify the `Clean.pm` file by copying our old code and modifying the boilerplate text in the supplied file. You might want to take a moment and complete the `POD` documentation for your module at this time (as well as placing something useful in the `README`) because we know you didn't do that before.

Finally, when you finish editing, you can create the `Makefile`. It's as simple as running the command `perl Makefile.PL`. After this you can enter `make` to build the module, or `make dist` to build the tarball, like `Cookbook-Clean-0.01.tar.gz`.

The function `WriteMakefile()`, used by `Makefile.PL` to create the all-important `Makefile` from which everything else flows, has a number of useful attributes that you can use to customize your build. In addition to the ones described in this chapter and elsewhere, some of the more common ones are

- `CCFLAGS`. String indicating the C flags to be used.
- `DEFINE`. String indicating the defines to be used.
- `DIR`. An array reference of directories that include further Makefile.PL files to process.
- `EXE_FILES`. A reference to an array containing a list of executable files to install.
- `INC`. String indicating the directories to search for included files.
- `LIBS`. An anonymous array containing alternative library specifications (typically library directories and names)
- `PREREQ_PM`. A hash reference containing a list of prerequisite modules and versions

As well, if you have more than one module in the distribution, you can place them under a directory called `lib/`, and `Makefile` will install them in the Perl tree, maintaining the same directory structure.

7.6. Creating a Binary PPM Distribution

You want to create a binary PPM distribution of your package.

Technique

Follow the steps in the previous recipe for creating a distribution, and then follow this procedure

```
C:\Cookbook\Clean> perl Makefile.PL BINARY_LOCATION="http://ppm.example.com/
➥ppmpackages/x86/Cookbook-Clean.tar.gz"

Checking if your kit is complete...
Looks good
Writing Makefile for Cookbook::Clean

C:\Cookbook\Clean> nmake
cp Clean.pm blib\lib\Cookbook\Clean.pm
AutoSplitting blib\lib\Cookbook\Clean.pm (blib\lib\auto\Cookbook\Clean)

C:\Cookbook\Clean> nmake ppd

C:\Cookbook\Clean> tar cvf Cookbook-Clean.tar blib
blib/
blib/lib/
blib/lib/Cookbook/
blib/lib/Cookbook/.exists
blib/lib/Cookbook/Clean.pm
blib/lib/auto/
blib/lib/auto/Cookbook/
blib/lib/auto/Cookbook/Clean/
blib/lib/auto/Cookbook/Clean/.exists
blib/lib/auto/Cookbook/Clean/autosplit.ix
blib/arch/
blib/arch/auto/
blib/arch/auto/Cookbook/
```

```
blib/arch/auto/Cookbook/Clean/
blib/arch/auto/Cookbook/Clean/.exists
blib/man3/
blib/man3/.exists

C:\Cookbook\Clean> gzip --best Cookbook-Clean.tar

C:\Cookbook\Clean> copy Cookbook-Clean.ppd \PPMPackages
C:\Cookbook\Clean\Cookbook-Clean.ppd => \PPMPackages\Cookbook-Clean.ppd
     1 file copied

C:\Cookbook\Clean> copy Cookbook-Clean.tar.gz \PPMPackages\x86
C:\Cookbook\Clean\Cookbook-Clean.tar.gz => \PPMPackages\x86\Cookbook-
Clean.tar.gz
     1 file copied
```

Comments

Although normally associated with Win32, ActiveState's PPM (Perl Package Manager) for creating and installing prebuilt packages can be used in principle for any system. Indeed, ActiveState maintains PPM packages for Linux and Solaris as well as Win32; substituting the appropriate `make` and `cp` commands in the preceding dialogue will result in a distribution package for these platforms.

Creating such a distribution follows the preceding procedure for building a `ppd` file. This is an `XML` file containing information on the package:

```
<SOFTPKG NAME="Cookbook-Clean" VERSION="0,1,0,0">
   <TITLE>Cookbook-Clean</TITLE>
   <ABSTRACT>Produce clean HTML</ABSTRACT>
   <AUTHOR>The folks at <authors@modperlcookbook.org></AUTHOR>
   <IMPLEMENTATION>
      <DEPENDENCY NAME="mod_perl" VERSION="1,26,0,0" />
      <OS NAME="MSWin32" />
      <ARCHITECTURE NAME="MSWin32-x86-multi-thread" />
      <CODEBASE HREF="http://ppm.example.com/ppmpackages/x86/
➥Cookbook-Clean.tar.gz" />
   </IMPLEMENTATION>
</SOFTPKG>
```

The value of the `HREF` attribute of the `CODEBASE` field comes from the `BINARY_LOCATION` argument to `perl Makefile.PL`. The `ABSTRACT` and `AUTHOR` fields come from specifying them in `Makefile.PL` as, for example,

```
WriteMakefile(
  NAME          => 'Cookbook::Clean',
  VERSION_FROM  => 'Clean.pm',
  PREREQ_PM     => {mod_perl    => 1.26,
                    HTML::Clean => 0.8, },
  ABSTRACT      => 'Produce Clean HTML',
  AUTHOR        => 'The folks at <authors@modperlcookbook.org>',
);
```

As well, if your module depends on any other modules, and if you specify the needed modules in a PREREQ_PM attribute in Makefile.PL as in the preceding example, these needed modules will appear in a DEPENDENCY field in the ppd file. When installing the module by using the ppm utility, any needed modules not present on the user's system will automatically be installed as well.

When this is all done, one then places the ppd file on a public server, with the .tar.gz file conventionally beneath the ppd/ location in a directory characterizing the target system. For users to install this distribution, they must first get and install the PPM module from CPAN. Installation of a PPM distribution is then a simple matter of using the ppm utility as follows:

```
C:\> ppm install "http://ppm.example.com/ppmpackages/package_name.ppd"
```

The PPM distribution from CPAN also ontains the modules needed to implement a ppm server on your system, so that users can set the repository within the ppm interactive shell utility to your server, from which searches, as well as installations, can be performed.

Despite the convenience of these prebuilt binary packages, there are a few drawbacks as well, especially for modules that require a C compiler (as with those with XS extensions). For these modules, one should, if at all possible, not rely on binary distributions and instead build the extension on your own, as sometimes even minor differences between the system upon which the build is done and the end-user system can result in incompatibilities.

This last point is particularly relevant in the Win32 world, where many users, for lack of experience and/or resources, do not have access to a C compiler, and instead rely almost exclusively on binary distributions. Most of the popular software binaries available, such as ActiveState's Perl and Apache, are compiled with Microsoft's Visual C++. This compiler unfortunately is relatively expensive, and so a user might be tempted to use one of the free compilers available in the Win32 world, such as that of

Borland (`http://www.borland.com/`), Cygwin (`http://www.cygwin.com/`), or mingw32 (`http://agnes.dida.physik.uni-essen.de/~janjaap/mingw32/`), to compile a module extension. Just like in the Unix world, however, mixing code compiled by different compilers generally doesn't work, even on the same machine. Thus, as well as ensuring that the build and end-user platforms are compatible, one should also make certain that any external libraries, and so on, used by an application have been compiled with the same compiler.

7.7. Writing a Live Server Test Suite

You want to write a test for your module that runs against a live Apache server.

Technique

Use the `Apache::Test` module, available from the `httpd-test` distribution, or from the mod_perl 2.0 distribution.

```
$ cvs -d":pserver:anoncvs@cvs.apache.org:/home/cvspublic" checkout httpd-test
cvs server: Updating httpd-test
U httpd-test/CHANGES
U httpd-test/LICENSE
U httpd-test/README
...
U httpd-test/perl-framework/t/ssl/varlookup.t
U httpd-test/perl-framework/t/ssl/verify.t

$ cd httpd-test/perl-framework/Apache-Test
$ perl Makefile.PL
generating script...t/TEST
Checking if your kit is complete...
Looks good
Writing Makefile for Apache::Test

$ make
$ make test
$ su
Password:
# make install
```

Comments

Writing a series of tests that execute against a live Apache server has gotten much simpler since the advent of `Apache::Test`. Originally part of the mod_perl 2.0 development project, the `Apache::Test` module became the basis of the `perl-framework` portion of the `httpd-test` distribution. The Apache HTTP test project, from which the `perl-framework` has stemmed, has amassed a rather astounding amount of development resources. It is full of hundreds of tests for the various Apache extension modules, as well as other useful tools for testing and stressing Apache. The `Apache::Test` part of the distribution is generic enough to be used with virtually any version of Apache, with or without mod_perl enabled. Here, however, we discuss the use of features specific to a mod_perl-enabled server.

Keep in mind that `Apache::Test` is a relatively new project, subject to rapid changes in both features and behavior—the examples here worked at the time of this writing, but changes in the API may mean slight modifications are required on your part for things to run smoothly.

To prepare your own module distribution for the use of `Apache::Test`, you first have to edit the `Makefile.PL` somewhat. Just add the following subroutine anywhere in `Makefile.PL`, which will override the default `make test` routine written by `ExtUtils::MakeMaker` with the `Apache::Test` harness. If the end-user platform does not have `Apache::Test` installed, `make test` simply exits with an informative message.

```
sub MY::test {
  if (eval "require Apache::TestMM") {
    Apache::TestMM::generate_script('t/TEST');
    Apache::TestMM->import(qw(test clean));
    return Apache::TestMM->test;
  }

  # The whitespace in front of @echo MUST be a single tab!
  return <<'EOF';
test::
	@echo This test suite requires Apache::Test
	@echo available from the mod_perl 2.0 sources
	@echo or the httpd-test distribution.
EOF
}
```

Next, you have to create a `t/` subdirectory off of the main directory containing your source code. Very recent versions of `h2xs` create this for you (and place a file named `1.t` in it), but older versions simply create `test.pl` in the main directory of your source tree. In either case you will want to remove the standard file (`1.t` or `test.pl`) before proceeding.

The `t/` directory will eventually contain a number of files and directories, some of which you must create yourself and some of which `Apache::Test` will create for you. The first file that should go into `t/` is called `TEST.PL`, which looks like

Listing 7.1 `t/TEST.PL`

```
#!perl

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

use Apache::TestRunPerl();

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

This is the actual test harness that will be invoked when you issue `make test`. Believe it or not, these few lines do all the intricate work of starting, stopping, configuring, and running your tests. The only real thing you need to worry about at this point is letting `Apache::Test` know the location of your `httpd` binary, which is typically done by setting the `APACHE` environment variable appropriately.

```
$ export APACHE=/usr/local/apache/bin/httpd
```

If `Apache::Test` cannot find a suitable Apache server, it politely lets you know at the start of your tests, so you need not fear the end users of your module are without direction in this regard.

The next step is to define the tests. What type of tests should you write? That depends on how complex your module is, what functions it should perform, what else is installed on the end-user's platform, and so on. For our example, we'll create a few generic tests that illustrate the main features of `Apache::Test`, which you can leverage into something appropriate for your module.

Just about all test suites ought to have a bare-bones test that makes sure their module can be loaded. Additionally, checking for any software version dependencies you might require is important, although the `PREREQ_PM` argument to `WriteMakefile()` can usually enforce this. Here is a minimal test that makes sure our versions of Perl and mod_perl are current, and makes certain that our fictional module, `Cookbook::TestMe` is loadable.

Listing 7.2 `t/01basic.t`

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

use Apache::Test;
```

Listing 7.2 *(continued)*

```
plan tests => 4;

ok require 5.006001;
ok require mod_perl;
ok $mod_perl::VERSION >= 1.26;
ok require Cookbook::TestMe
```

`01basic.t` illustrates a few of the things that will be common to all of our tests. First, we do some bookkeeping and plan the number of tests that will be attempted. After that, we simply call `ok()` followed by our test condition. The syntax of the tests might seem rather odd, but they follow the same pattern as `Test.pm` from the base Perl distribution—`Apache::Test` actually uses `Test.pm` behind the scenes. For the full details of the `ok()` function and its semantics, see the `Test` manpage.

Now it's time to prepare our server for some live tests. `Apache::Test` provides a basic `httpd.conf` configuration, including `DocumentRoot`, `ErrorLog`, `Port`, and other such settings, allowing you to focus on configuring only the settings specific to your needs. To add additional settings to the defaults, we create a `t/conf/extra.conf.in` file. If `Apache::Test` sees `extra.conf.in` it will pull the file into its default configuration using an `Include` directive.

Listing 7.3 `t/conf/extra.conf.in`

```
<Location /hooks>
  SetHandler perl-script
  PerlHandler 'sub { use mod_perl qw(PerlStackedHandlers PerlFileApi); \
                     shift->send_http_header();                       \
                     return Apache::Constants::OK;                    \
               }'
</Location>

Alias /handler @DocumentRoot@
<Location /handler>
  SetHandler perl-script
  PerlHandler Cookbook::TestMe
</Location>

Alias /filter @DocumentRoot@
<Location /filter>
  SetHandler perl-script
  PerlHandler Cookbook::TestMe Cookbook::TestMe
  PerlSetVar Filter On
</Location>
```

As you can see, we are planning on running several live tests with our module. The first configuration is a handler implemented as an anonymous subroutine that merely tests whether mod_perl was compiled with PERL_STACKED_HANDLERS=1 and PERL_FILE_API=1 (or EVERYTHING=1), since our fictitious Cookbook::TestMe module makes copious use of these hooks. The anonymous subroutine shortcut here is rather convenient and keeps us from needing to create a separate file just to test these conditions.

The next configuration is for a direct call to our handler. Notice the @DocumentRoot@ variable in our configuration, which gets expanded to the full path to our (yet to be created) t/htdocs/ directory as part of the Apache::Test magic. The final configuration is for testing whether the platform is capable of handling stacked handlers using Apache::Filter.

Because we are using DocumentRoot in our tests, let's put some content in there. Create the file directory t/htdocs/ and place an index.html file in it. It does not have to contain anything fancy—in fact, the shorter the better, because you will be using the contents of this file as a comparison later on. For our example, we insert the simple phrase "Thanks for using Cookbook::TestMe".

As with httpd.conf, Apache::Test also provides a rudimentary startup.pl file. However, you can augment the basics provided by Apache::Test with your own by creating t/conf/modperl_extra.pl. For our immediate purposes, this file is rather small

Listing 7.4 t/conf/modperl_extra.pl

```
eval "require Apache::Filter";
1;
```

We use modperl_extra.pl as a way to conditionally load Apache::Filter without using the PerlModule directive in our extra_conf.in. As you will see shortly, we can optionally run tests based on various criteria—using an eval() here allows us to satisfy our test conditions for users both with and without Apache::Filter installed.

At this point in the process, your t/ directory should have the following layout:

```
$ ls  t/*
t/01basic.t  t/TEST.PL

t/conf:
extra.conf.in  modperl_extra.pl

t/htdocs:
index.html
```

Now it is time to create some tests that use the layout we just constructed. The format for `01basic.t` was pretty simple, and relatively close to what you would do for a standard test that did not involve a running Apache server. For the remainder of the tests, we will take advantage of the functions provided by `Apache::Test` and its companion modules.

Listing 7.5 `t/02hooks.t`

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

use Apache::Test;
use Apache::TestRequest;

plan tests => 1, \&have_lwp;

ok GET_OK '/hooks';
```

The test file `02hooks.t` is a little different from the file we created for our basic tests. Here, we added `Apache::TestRequest` to our list of required modules. `Apache::TestRequest` provides a number of tools we will need to make requests to our live server. As before, we plan the number of tests that will be attempted. The difference here is that we are choosing to plan the tests only if some additional criteria are met. `plan()` accepts a code reference as a final, optional argument—if the code reference evaluates to true, the tests are planned. Here we use the `have_lwp()` function provided by `Apache::TestRequest`, which checks the availability of modules from the `libwww-perl` distribution. If `have_lwp()` returns true, we know we can take advantage of the shortcuts `Apache::Test` provides instead of implementing our own scheme to initiate requests to the server and parse the response.

After planning our test, we use the shortcut function `GET_OK()` provided by `Apache::TestRequest` to fetch and process our URL. Actually, `Apache::TestRequest` provides a number of different functions for fetching and testing URLs on your test server, some of which are shown in Table 7.1.

Table 7.1 *Some URL-fetching Methods*

Test Function	Details
`GET_BODY()`	Returns the message body of the response.
`GET_OK()`	Returns true on success (`HTTP_OK`) and false otherwise.
`GET_RC()`	Returns the HTTP status code of the response.
`GET_STR()`	Returns the request headers and response message body.

Any of these functions can be used to test the results of your request; it's just a matter of what you want to accomplish and personal preference. For `02hooks.t` we simply want to make sure that the anonymous subroutine handler at the URL `/hooks` returns successfully.

The next test is the one that actually tests our handler. Here, we want to actually fetch a file and compare it to the value we know to be there (hence the terse value for `index.html`). For this we use the `GET_BODY()` method from Table 7.1, along with a conditional form of the `ok()` test function.

Listing 7.6 `t/03handler.t`

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

use Apache::Test;
use Apache::TestRequest;

plan tests => 1, \&have_lwp;

my $content = GET_BODY '/handler/index.html';
chomp $content;
ok ($content eq "Thanks for using Cookbook::TestMe");
```

Finally, in the following `04filter.t` code we use a combination of all the elements presented so far.

Listing 7.7 `t/04filter.t`

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

use Apache::Test;
use Apache::TestRequest;

plan tests => 1, \&have_filter;

my $content = GET_BODY '/filter/index.html';
chomp $content;
ok ($content eq "Thanks for using Cookbook::TestMe");

sub have_filter {
  eval {
    die unless have_lwp();
```

Listing 7.7 *(continued)*

```
    require Apache::Filter;
  };
  return $@ ? 0 : 1;
}
```

We start here by checking the current environment against our own criteria: The platform must support both `LWP` and `Apache::Filter` for the test to be attempted, which we determine using our own `have_filter()` function. Then, similar to `03handler.t`, we use `GET_BODY()` to retrieve the content from the `/filter` location in our `extra.conf.in`, where we chained together two instances of our handler. Assuming that the operations in `Cookbook::TestMe` are basic enough, if `Apache::Filter` is present and working properly, the results should be the same regardless of the number of stacked `PerlHandlers`. The version of `Cookbook::Clean` presented in Recipe 15.4, which is `Apache::Filter` aware, falls into this category. For more elaborate filtering setups, such a generic test may not be possible, but at least you can get a feel for the steps necessary from the example here.

After preparing our tests, a simple call to `make test` should yield a dialogue similar to this:

```
$ make test
/usr/local/bin/perl -Iblib/arch -Iblib/lib \
t/TEST -clean
setting ulimit to allow core files
ulimit -c unlimited
 exec t/TEST -clean
cannot build c-modules without apxs
APACHE_USER= APXS= APACHE_PORT= APACHE_GROUP= APACHE=/usr/local/apache/bin/httpd
\
/usr/local/bin/perl -Iblib/arch -Iblib/lib \
t/TEST
...
waiting for server to warm up...ok
server localhost:8529 started
01basic.............ok
02hooks.............ok
03handler...........ok
04filter............ok
All tests successful.
Files=4, Tests=7,  4 wallclock secs ( 3.65 cusr +  0.23 csys =  3.88 CPU)
server localhost:8529 shutdown
```

If `Apache::Filter` had not been present on the end-user system, `have_filter()` would have returned false and the result of `make test` would have looked like:

```
01basic............ok
02hooks............ok
03handler..........ok
04filter...........skipped: no reason given
All tests successful, 1 test skipped.
```

You can configure many other types of tests with the `Apache::Test` suite; we have only scratched the surface of what is available, and more options are frequently written in. For the latest developments, take a look at the documentation that accompanies the `httpd-test` distribution.

7.8. Adding Custom Configuration Directives

`PerlSetVar` and `PerlAddVar` are a bit too restrictive for your configuration needs, and you are looking for something more flexible.

Technique

Write a directive handler using `Apache::ModuleConfig`, `Apache::ExtUtils`, `h2xs`, and some moxie.

In `httpd.conf` (after many things highly magical)

```
PerlModule Cookbook::Clean

<Location /clean>
  SetHandler perl-script
  PerlHandler Cookbook::Clean

  # Now, our very own Apache directives.
  CleanLevel 3
  CleanOption whitespace shortertags
</Location>
```

Comments

The ability to add custom configuration directives to Apache is an extremely powerful yet seldom used or understood aspect of mod_perl. In reality, the process is not all that difficult mechanically, but the more sophisticated understanding of the inner workings of Apache, Makefile.PL editing, and patience required is somewhat intimidating. On the other hand, the results are quite powerful; implementing directive handlers will give you fine-grained control over your configurations, including the ability to enforce the number, types, and values of the arguments your module receives, as well as overriding core Apache directives with your own, devious substitutes.

The actual number of configuration directives directly handled by the core Apache server is relatively small; of the 210 directives currently provided by the standard Apache distribution, only 76 are from `http_core.c`. Directives such as `SetHandler` and `Alias` are implemented as additions to the core server via C extension modules, such as mod_mime and mod_alias. Apache provides an API for C modules that allows them to add directives Apache will recognize when it parses `httpd.conf`. For C modules everything is quite routine and there is nothing particularly special about the API. For mod_perl programmers, the interface to the Apache directive handler API requires an unusual amount of effort, so some background explanations are due that will add clarity to the process and try to make it a little less intimidating.

If we start by outlining the process Apache uses when it handles configuration directives in general the mod_perl interface into the process will go much more smoothly. As Apache tokenizes `httpd.conf` and encounters various directives, Apache gives each module loaded into the server a chance to process the directive before it tries to handle the directive itself. If neither an extension module nor `http_core` chooses to handle the directive, Apache sends out a warning and the server fails to start:

```
Invalid command 'CleanLevel', perhaps misspelled or defined by a module not
included in the server configuration
/usr/local/apache/bin/apachectl start: httpd could not be started
```

The way a C extension module (such as mod_perl) tells Apache which directives it is responsible for is by populating an Apache module record. Although there is more to the module record than just directive handlers, we are omitting some explanation in favor of focus for the moment.

Apache, in turn, stores the module records from all active modules internally. As Apache tokenizes `httpd.conf`, it traverses this module list, looking for candidates to handle each directive. After a module steps up and accepts responsibility for the directive, it gives Apache a set of rules that govern the directive, such as where it is allowed to appear within `httpd.conf` and what the format of the arguments should be. Apache then applies this set of rules to the directive, and if all looks to be in proper order, Apache passes the configuration data over to the module for processing.

After a module has the configuration data the rest is out of core Apache's hands. The module typically makes decisions about the validity of the arguments then stashes the data away so that it can be used again at request time.

For Perl modules that want to implement their own configuration directives the process is pretty much the same. First, we have to let Apache know about our directive and supply the ruleset that defines its behavior. Then we need to accept and process the configuration data. When these two steps are accomplished, we store the data away and retrieve it again at request time.

As it turns out, the first stage is by far the most difficult and least intuitive part of the process. To let Apache know that our directive exists we have to populate an Apache module record. Like the other Apache records we have encountered thus far, the Apache module record is defined in `http_config.h` in the Apache source distribution and is accessible through a mod_perl API. However, the module record is unique in that it wholly defines the interaction between a C extension module and Apache—it holds all the information about the module that Apache will ever know. Because of this, the Apache module record needs to be fully populated and available to Apache when the server is starting, *before* any requests are served. This requires a bit of chicanery on our part: We have to essentially turn our Perl module into something resembling a C extension module so that our Perl module can be loaded into Apache when the server is started. The solution is to use XS to provide the glue between our Perl world and Apache's C world during the early stages of the Apache lifecycle.

Because we have to use XS for this initial (and most difficult) part of the directive handler API, much of the process occurs outside of the request cycle using a combination of standard Perl tools and a mod_perl-specific interface. In illustration, let's take the `Cookbook::Clean` handler from Recipe 7.2 and alter it to make use of its own custom directives instead of `PerlSetVar` and friends.

Unlike with other mod_perl handlers, to create directive handlers you will need to use `make` and write a `Makefile.PL`. The best way to begin, then, is via `h2xs` using the same `-AX` argument list we used in Recipe 7.5. Although the process will eventually result in the creation of an `.xs` file, the mod_perl API does this on-the-fly so you don't need to concern yourself with it.

After running `h2xs`, the next thing to do is edit the `Makefile.PL`.

Lisiting 7.8 `Makefile.PL` *for* `Cookbook::Clean`

```
package Cookbook::Clean;

use ExtUtils::MakeMaker;
use Apache::ExtUtils qw(command_table);
use Apache::src ();
```

Lisiting 7.8 *(continued)*

```
use Config;

use strict;

my @directives = (
  { name         => 'CleanLevel',
    errmsg       => 'Level of suds',
    args_how     => 'TAKE1',
    req_override => 'OR_ALL', },

  { name         => 'CleanOption',
    errmsg       => 'Specific detergent to use when cleaning',
    args_how     => 'ITERATE',
    req_override => 'OR_ALL', },
);

command_table(\@directives);

my %config;

$config{INC} = Apache::src->new->inc;

if ($^O =~ m/Win32/) {
  require Apache::MyConfig;

  $config{DEFINE}  = ' -D_WINSOCK2API_ -D_MSWSOCK_ ';
  $config{DEFINE} .= ' -D_INC_SIGNAL -D_INC_MALLOC '
    if $Config{usemultiplicity};

  $config{LIBS} =
    qq{ -L"$Apache::MyConfig::Setup{APACHE_LIB}" -lApacheCore } .
    qq{ -L"$Apache::MyConfig::Setup{MODPERL_LIB}" -lmod_perl};
}

WriteMakefile(
  NAME            => 'Cookbook::Clean',
  VERSION_FROM    => 'Clean.pm',
  PREREQ_PM       => { mod_perl    => 1.26,
                       HTML::Clean => 0.8, },
  ABSTRACT        => 'An XS-based Apache module',
  AUTHOR          => 'authors@modperlcookbook.org',
  clean           => { FILES => '*.xs*' },
  %config,
);
```

For the most part, this looks the same as the Makefile.PL structure introduced in Recipe 3.19. There are, however, a few differences that are important and essential to using the custom directive API.

The first change to note is that the standard *shebang* line has been replaced with the package keyword. This needs to match the name of the package that defines your directive handler, Cookbook::Clean in our case. Additionally, we have imported the command_table() function from the Apache::ExtUtils class. More on the need for both of these changes shortly.

The WriteMakefile() function from ExtUtils::MakeMaker has been augmented to include the PREREQ_PM and clean elements. clean and PREREQ_PM are not required keys but are included for good measure to make sure we have a recent version of mod_perl and that we remove the Clean.xs file generated later. We also need an INC key so make can find all of the Apache and mod_perl header files that it will need. We actually sneak the INC key into the %config hash, along with some Win32 specific data that helps make the Makefile.PL platform independent, using the inc() utility function from the Apache::src class.

The rest of the code at the top of the Makefile.PL is the magic that ties your custom directive into the Apache module record, and requires a rather lengthy explanation.

Let's take a moment here to examine the high-level Apache process again. When Apache parses httpd.conf it decides whether a directive can appear within a <Directory> container or an .htaccess file, as well as whether the directive takes a single argument or a list of values. As we already mentioned, all our module needs to do is supply the set of rules that define these aspects of the custom directive and Apache takes care of the rest, which is very convenient and removes a large burden from module developers.

The rules that govern our directive are held in an additional record: the Apache command record, also defined in httpd_config.h, which occupies a slot of the Apache module record. The command record specifies the behavior of the directive, such as the number of arguments it expects, where it can appear within httpd.conf, and which callback routine is passed the argument list when the directive is encountered.

The command_table() function is the real workhorse of the entire process. It creates the Apache module and command records by generating lots of XS glue required to tie our module to Apache. The @directives array we pass (by reference) to command_table() is an array of hash references, each hash representing a separate custom directive, and each hash key representing a field in the Apache command record. Table 7.2 lists the keys and a brief description of their meaning.

Table 7.2 `command_table()` *Parameters*

Apache Command Record Field	Short Description
`args_how`	The directive prototype
`cmd_data`	Additional configuration data
`errmsg`	Description of the directive
`func`	Name of routine that handles the directive
`name`	Name of the directive as it will appear in the configuration
`req_override`	Options that specify where in the configuration the directive might be located

The `name` field is the name of the directive to be implemented, such as our `CleanLevel`. To avoid future namespace clashes, prepending your module name to the front of the directive is a good convention to stick to. As will be discussed in Recipe 7.11, the ability to choose the name of an existing Apache directive to override its behavior is possible.

`errmsg` is a description of the directive. This can be anything you like, but it will be the message displayed when Apache encounters an error processing the directive, such as a prototype mismatch. It is also displayed by mod_info on the `/server-info` page.

By default, the Perl subroutine that is passed the configuration data is the same as the name you give your directive. However, you may also specify a different name for the directive handler using the `func` key. This is useful for providing backward compatibility and support for older directive names when your module is hauled out for its annual rewrite; there is no reason why two directives cannot point to the same subroutine.

The `cmd_data` field is actually seldom used, but can contain additional data that you want available when the directive is configured. For example, mod_access actually uses the same handler to process the `Allow` and `Deny` configuration directives, letting the `cmd_data` field serve as the distinguishing marker. Creative uses for this field are shown in Recipe 12.6.

Now for the tricky stuff. The `args_how` key defines the prototype for the directive. This affects not only the number of arguments received by your Perl subroutine, but also the number of times your routine is called when the directive is encountered. The possible values for this field are defined in `http_config.h` in the `cmd_how` enumeration. In our example we specify both `TAKE1` and `ITERATE`. `TAKE1` specifies that the directive takes one argument, which means your directive handler will be called once for each time the directive is encountered. The ITERATE prototype signifies that, for each time the directive is encountered, the directive handler subroutine is to be called once for each argument until the argument list is exhausted. Thus, the number of callbacks to your subroutine can vary greatly depending on the ruleset you specify with command_table(). There are also other possible prototypes, such as those that enforce two parameters or others that accept only "On" or "Off". For a more complete explanation of the various prototypes, as well as sample usages, see Appendix B.

Although the args_how values correspond to constants exported by the Apache::Constants :args_how import tag, note that at this point they are merely strings, not constant subroutines, so actually use()ing Apache::Constants is not required. Apache::ExtUtils, as part of its wand waving, takes care of translating these strings into numerical constants that mod_perl understands.

The final key, req_override, signifies where in the Apache configuration the directive can reside. A directive may appear in four different logical areas of the configuration: the base server, a virtual host, a container directive, or an .htaccess file. Additionally, a directive's presence within an .htaccess file is restricted based on the values set by the AllowOverride core directive. All these permutations are captured in the req_override bitmask. Permissible values for this field are (you guessed it) defined in http_config.h and may be logically ORed together to determine the appropriate level of containment.

In our Makefile.PL, both directives are capable of being placed anywhere within any configuration file the administrator pleases due to the OR_ALL override setting. If we had wanted our directives to be allowed only on a per-server basis (outside of any container directive, such as <Location>), we could have used the RSRC_CONF designation instead.

As with args_how, these options all correspond to constants available from Apache::Constants, can be imported using the :override tag, and are fully listed in Appendix B.

We mentioned that the req_override values can be logically combined to form an access bitmask. In reality, however, the only combination you will ever likely encounter is RSRC_CONF|ACCESS_CONF, which means the directive is allowed any place other than in an .htaccess file. In fact, there is no other combination of overrides in use by any module in the standard Apache distribution, and most other combinations tend to be misleading or redundant. Keep in mind that, however hard you might try, there is no way to distinguish the different container directives using override flags; if a directive is allowed in a <Location>, it is allowed within <Files>, <DirectoryMatch>, and all the others as well.

After you have determined where your directive can live and what types of arguments you require, you can simply call `Apache::ExtUtils`'s `command_table()` function, issue `perl Makefile.PL`, and *voila!* mod_perl has magically created everything Apache needs to know about your new directives. If you are interested, take a look at the generated `Clean.xs` and compare it to a simple module in the Apache distribution, such as `mod_dir.c`; it is almost like you wrote the extension in C! Almost.

Although most of the hard work is complete at this point, much still remains. We still have to process the incoming data, store it away, and retrieve it again at request time. The good news is that each of these functions is handled back in familiar Perl territory, within our module.

Listing 7.9 `Clean.pm`

```
package Cookbook::Clean;

use Apache::Constants qw( OK DECLINED );
use Apache::File;
use Apache::Log;
use Apache::ModuleConfig;

use DynaLoader ();
use HTML::Clean;

use 5.006;

our $VERSION = '0.01';
our @ISA = qw(DynaLoader);

__PACKAGE__->bootstrap($VERSION);

use strict;

sub handler {

  my $r = shift;

  my $log = $r->server->log;

  my $cfg = Apache::ModuleConfig->get($r, __PACKAGE__);

  unless ($r->content_type eq 'text/html') {
    $log->info("Request is not for an html document - skipping...");
    return DECLINED;
  }
```

Listing 7.9 *(continued)*

```
  my $fh = Apache::File->new($r->filename);

  unless ($fh) {
    $log->warn("Cannot open request - skipping... $!");
    return DECLINED;
  }

  # Slurp the file (hopefully it's not too big).
  my $dirty = do {local $/; <$fh>};

  # Create the new HTML::Clean object.
  my $h = HTML::Clean->new(\$dirty);

  # Set the level of suds.
  $h->level($cfg->{_level} || 1);

  # Make sure that we have a hash reference to dereference.
  my %options = $cfg->{_options} ? %{$cfg->{_options}} : ();

  # Clean the HTML.
  $h->strip(\%options);

  # Send the crisp, clean data.
  $r->send_http_header('text/html');
  print ${$h->data};

  return OK;
}

sub CleanLevel ($$$) {

  my ($cfg, $parms, $arg) = @_;

  die "Invalid CleanLevel $arg!" unless $arg =~ m/^[1-9]$/;

  $cfg->{_level}  = $arg;
}

sub CleanOption ($$@) {

  my ($cfg, $parms, $arg) = @_;
```

Listing 7.9 *(continued)*

```
  my %possible = map {$_ => 1} qw(whitespace shortertags blink contenttype
                                  comments entities dequote defcolor
                                  javascript htmldefaults lowercasetags);

  if ($possible{lc $arg}) {
    $cfg->{_options}{lc $arg} = 1;
  }
  else {
    die "Invalid CleanOption $arg!";
  }
}
1;
```

With the exception of some additional code sandwiching the code from Recipe 7.1, things look pretty much the same. We have added the global variables `$VERSION` and `@ISA` and used them with the call to `DynaLoader`'s `bootstrap()` method. This is what ties your Perl XS extension module to Apache. When you issue `make`, a shared object file is created using the `.xs` file generated by `command_table()`. At runtime, the `bootstrap()` method takes care of loading this shared object into the current environment. If you don't fully understand the mechanism here, that's okay—it's a realm best left to the conjurers and prestidigitators of the Perl internals world.

At the end of our module rests our actual directive handlers, with names matching those we entered into the Apache command record using `command_table()`. These contain the code that will give meaning to the directive and make it possible for our `handler()` subroutine to access the configuration data at request time.

Again, let's take a high-level approach before continuing. Configuration directives are generally followed by a series of arguments. For instance, `ExtendedStatus` takes either `On` or `Off`, whereas the argument list for `AddType` is a single value (the MIME type) followed by a list of values (the extensions to associate with that type). Your directive handler will need to know how Apache will present the configuration data to make intelligent decisions about how to store the data. The way your directive handler interacts with Apache and the argument list is actually configured by the value specified in the `args_how` key in the `Makefile.PL` and the prototype given to your directive handler in your module.

As with request-time handlers, which receive the Apache request object as their first argument, directive handlers also receive a Perl object by default. Actually they receive two: an object for storing away data, and an object containing information about the directive itself that can only be known by Apache, such as the server under which the directive is configured. By convention these two objects are placed into the $cfg and $parms variables. The data that follows these two parameters is whatever information accompanied the directive in the httpd.conf.

For the moment, we can safely ignore $parms: A more detailed discussion is forthcoming in Recipe 7.10. The object held in $cfg, however, is of the utmost importance, because it is what you will use to store your configuration data so that you can access it again at request time.

$cfg actually contains a reference to a hash bless()ed into the class of our directive handler. Similar to other handlers, there are two ways to retrieve this object. The first way is to pull it from the argument list:

```
my ($cfg, $parms, $arg) = @_;
```

The other way is by retrieving the object directly using the Apache::ModuleConfig class, passing the current request and your package name to its get() method:

```
my $cfg = Apache::ModuleConfig->get(Apache->request, __PACKAGE__);
```

These two forms are analogous to the shift() and Apache->request() idioms used to access the Apache request object. Just as Apache->request() always retrieves the *same* request object rather than creating a *new* object, so does Apache::ModuleConfig's get() method always dig out the configuration data for your module. At this point, however, there is no data in $cfg. You can specify any behavior in your directive handler you want, but this example is in need of nothing terribly complex. If the incoming arguments pass muster, both the CleanLevel() and CleanOption() directive handlers populate keys within the hash reference held in $cfg.

Remember that if the number of arguments fails to meet the prototype, or the directive appears someplace other than an area allowed by req_override, Apache will handle the exception by halting its startup routine and displaying an informative message. In cases where you might want to halt Apache yourself due to an invalid argument or other such data error, the appropriate action is to simply die() with an error message.

The point of this entire exercise has been to define directives that supply meaningful data to your handler, so we need a method for extracting the data from within our handler() subroutine at request time. For this we use the explicit call to Apache::ModuleConfig->get() just mentioned, which returns the same object populated by our directive handler.

The very last step after running the canonical perl Makefile.PL, make and friends (we promise) is to alter your httpd.conf to reflect the new directives, as shown in the solution to this recipe.

There are a few things to note about our new configuration. The first is that you *must* use the PerlModule directive to load your module, even though for modules without custom directives a use() call within a startup.pl is all that is normally required. Because we are using some trickery to make Apache think our Perl module is a really a C module, the PerlModule directive also has to appear before any directives that are implemented by your directive handlers, as shown in the solution configuration.

The other, more important, item is that although you have implemented a new directive, nothing has been said about which phase your module will handle. If you examined the generated Clean.xs file closely, you would have seen that all the handler slots were set to NULL. This means that despite all of your hard work, you still need mod_perl's Perl*Handler directives to add your module to a particular request phase.

Even though the meal we have prepared here is rather dense and hard to digest, don't reach for the antacid too soon. In addition to the recipes presented in the remainder of this chapter, many examples are available of custom directives in action, both in *Writing Apache Modules with Perl and C* and several modules on CPAN, including Apache::Dispatch, Apache::Language, and Apache::RefererBlock to name only a few. Looking at the code provided by all of these sources ought to put the wind in your sails and get you on your way.

7.9. Expanding Custom Directive Prototypes

None of the args_how options seem to fit what you want to do—you need an unavailable prototype.

Technique

Use the RAW_ARGS prototype—it's not just for containers.

```
sub UserDir ($$$;*) {
  # Provide a subset of mod_userdir support, eg
  # UserDir public_html ./ (ITERATE-esque)
  # UserDir disable root ftp (ITERATE2-esque)
  my ($cfg, $parms, $args, $fh) = @_;

  # UserDir is implemented as a PerlTransHandler
  # so we can use a per-server configuration.
  $cfg = Apache::ModuleConfig->get($parms->server, __PACKAGE__);

  my @directives = split " ", $args;

  if ($directives[0] =~ m/^disabled?$/i) {
    # Continue along...
  }
}
```

Comments

You probably paid it no attention, but now that you are starting to write your own Apache directives you might wonder exactly how `mod_userdir` implements its `UserDir` directive. The directive is documented to take either a list of subdirectories or the keywords `disabled` or `enabled` followed by an optional list of usernames. Each of these three options is covered by an existing prototype, but the combination of all of them does not fit into an existing model. As it turns out, `mod_userdir` handles the multiple-prototype situation deftly using `RAW_ARGS` and parsing the argument string itself.

The typical example given for `RAW_ARGS` is for the creation of container directives. In these cases the fourth argument passed to the directive handler, a filehandle corresponding to the configuration file, is read and processed until an enclosing block is found.

```
sub Balast ($$$;*) {

  my ($cfg, $parms, $args, $fh) = @_;

  (my $boat = $args) =~ s/>$//;      # strip the trailing >

  while (my $line = <$fh>) {
    last if $line =~ m!</Balast>!;   # exit if the end tag is found
    next if $line =~ m!^\s*#!;       # skip over comments

    # Do something useful with $line...
  }
}
```

However, the third argument passed back to a RAW_ARGS prototype is the remainder of the line containing the directive itself after the directive token has been removed. In the case of container directives, this represents the focus of the container, such as /usr/local/apache/htdocs> for a <Directory> corresponding to DocumentRoot. Pay attention to that final >— this truly is *raw* data that you have before you, capable of being manipulated however you want.

Using RAW_ARGS for directives other than containers is a convenient way of dealing with prototypes that do not fit neatly into any of the other models. Directives that occupy only a single line can safely ignore the input filehandle $fh and operate only on $args, as in the example UserDir() subroutine, which splits the argument list on whitespace and decides what to do from there.

7.10. Merging Custom Configuration Directives

You want your custom directives to properly inherit from parent directories and/or servers.

Technique

Create DIR_CREATE() and DIR_MERGE(), or SERVER_CREATE() and SERVER_MERGE() subroutines.

```
package Cookbook::Clean;

use Apache::Constants qw( OK DECLINED );
use Apache::File;
use Apache::Log;
use Apache::ModuleConfig;

use DynaLoader ();
use HTML::Clean;

use 5.006;

our $VERSION = '0.02';
our @ISA = qw(DynaLoader);

__PACKAGE__->bootstrap($VERSION);
```

```
use strict;

sub handler {

  my $r = shift;

  my $log = $r->server->log;

  my $cfg = Apache::ModuleConfig->get($r, __PACKAGE__);

  unless ($r->content_type eq 'text/html') {
    $log->info("Request is not for an html document - skipping...");
    return DECLINED;
  }

  my $fh = Apache::File->new($r->filename);

  unless ($fh) {
    $log->warn("Cannot open request - skipping... $!");
    return DECLINED;
  }

  # Slurp the file (hopefully it's not too big).
  my $dirty = do {local $/; <$fh>};

  # Create the new HTML::Clean object.
  my $h = HTML::Clean->new(\$dirty);

  # Set the level of suds.
  $h->level($cfg->{_level});

  # No need to check before dereferencing since we can now
  # initialize our data in DIR_CREATE().
  $h->strip($cfg->{_options});

  # Send the crisp, clean data.
  $r->send_http_header('text/html');
  print ${$h->data};

  return OK;
}
```

```
sub CleanLevel ($$$) {

  my ($cfg, $parms, $arg) = @_;

  die "Invalid CleanLevel $arg!" unless $arg =~ m/^[1-9]$/;

  $cfg->{_level}  = $arg;
}

sub CleanOption ($$@) {

  my ($cfg, $parms, $arg) = @_;

  my %possible = map {$_ => 1} qw(whitespace shortertags blink contenttype
                                  comments entities dequote defcolor
                                  javascript htmldefaults lowercasetags);

  if ($possible{lc $arg}) {
    $cfg->{_options}{lc $arg} = 1;
  }
  else {
    die "Invalid CleanOption $arg!";
  }
}

sub DIR_CREATE {
  # Initialize an object instead of using the mod_perl default.

  my $class = shift;
  my %self  = ();

  $self{_level}   = 1;   # default to 1
  $self{_options} = {};  # now we don't have to check when dereferencing

  return bless \%self, $class;
}

sub DIR_MERGE {
  # Allow the subdirectory to inherit the configuration
  # of the parent, while overriding with anything more specific.

  my ($parent, $current) = @_;
  my %new = (%$parent, %$current);

  return bless \%new, ref($parent);
}
1;
```

Comments

For most applications, the simple directive implementation given in Recipe 7.8 is usually enough However, because the explanation there was already sufficiently intense, we purposefully left out some rather complex details that are important if you want to be able to have your directives merge through your configuration in the same manner that, say, `PerlSetVar` does.

Consider the situation where you have a `<Directory>` with one set of custom directives and an `.htaccess` file with a partial list of directives for the same module. Apache's default behavior is to apply *only* those directives in the `.htaccess` file and ignore any defined in the parent `<Directory>` container. Although this might not seem all too reasonable, Apache is about flexibility: You have the ability to supply your own merge routines if the default behavior does not suit your needs. As with the aspects of the directive handler API we have discussed so far, directive merging is not as simple a concept as some of the other programming techniques in this book, but its application is rather straightforward. If you already have working custom directives, then the hard part is (far) behind you.

The mechanism by which Apache allows modules to define their own merging behavior is separated into four separate routines: server configuration creation, directory configuration creation, and merging for each. As we hinted in Recipe 7.8, there is more to the Apache module record than the Apache command record. The module record is also used to define the routines that will handle each of these phases. As before, because Apache needs a fully populated module record prior to request time, the real work is done over in XS-land with the call to `command_table()` within the `Makefile.PL`.

The good news is that there is no additional fiddling that needs to be done to the `Makefile.PL`; all is handled from within your Perl module. You will need to define any combination of these four subroutines, each corresponding to a phase of the Apache configuration process outlined previously: `DIR_CREATE()`, `DIR_MERGE()`, `SERVER_CREATE()`, and/or `SERVER_MERGE()`. If you were wondering why the `Makefile.PL` had to contain a package declaration, this is the reason: `command_table()`

checks whether your module, for instance, `can('SERVER_MERGE')` and populates the Apache module record accordingly in the `.xs` file it generates. Tricky.

These two sets of routines perform essentially the same function. The `DIR_CREATE()` and `SERVER_CREATE()` subroutines are used to create the storage object for your module's configuration data. For Perl this is a relatively simple and idiomatic task—merely `bless()` a hash reference into the current class and return it. The object you create in these routines will supercede the default `$cfg` object created by mod_perl we used in Recipe 7.8. However, mod_perl will still manage it for your class so that calls to `Apache::ModuleConfig->get()` behave just as they did before. If you want to define any default values for your directive you can do so here, as we did by initializing both `$cfg->{_level}` and `$cfg->{_options}`, which frees us of the need to check before dereferencing them in our `handler()`.

The `DIR_MERGE()` and `SERVER_MERGE()` subroutines define how directives will merge when configurations overlap. They both receive two objects in their argument list: the object from the parent configuration, as well as that from the current configuration (if one exists). They can then decide on an appropriate course of action. Typically, this is to summarily override the parent configuration with the current configuration, while allowing the parent to fill in any empty values.

Even though Apache has placed directive merging completely under your control at this point, you certainly do not have to follow this sweeping model. The following example allows users to decide whether they want to inherit from the parent configuration.

```
sub SERVER_MERGE {
  # Require the SubMerge flag to be set before merging directives.

  my ($parent, $current) = @_;

  if ($current->{_merge}) {
    my %new = (%$parent, %$current);
    return bless \%new, ref($parent);
  }

  return $current;
}
```

Although `DIR_CREATE()` and `SERVER_CREATE()` are functionally equivalent, there are differences in when they come into play and how you must interact with them in your handler. `SERVER_CREATE()` is called when Apache is started, once for the main server, and once for each virtual host. `SERVER_MERGE()` is also called at server startup, where it then merges any configuration data found in the virtual hosts with that from the main server.

For per-directory configurations things are slightly different. Like `SERVER_CREATE()`, `DIR_CREATE()` is called once for each configured server when Apache is started. However, it is also called once at startup for each `<Location>` or `<Directory>` where a custom directive appears. `DIR_CREATE()` further differs in that it is also called at request time whenever Apache encounters an `.htaccess` file. `DIR_MERGE()` is called at request time, running whenever a request enters a `<Location>`, `<Directory>`, or other container that can potentially be merged.

As you recall from Recipe 7.8, the configuration object on the argument list to our directive handler was the same one that could be retrieved directly using the Apache request object and the `Apache::ModuleConfig` class,

```
my $cfg = Apache::ModuleConfig->get($r, __PACKAGE__);
```

which is the syntax we use at request time. As it turns out, this is the per-directory configuration object, created either with `DIR_CREATE()` or by mod_perl's default routine. Thus, interacting with per-directory configurations is exactly the same as in Recipe 7.8. In fact, in Recipe 7.8 we were working on a per-directory basis all along, you just didn't know it!

Dealing with per-server configurations is a bit more complex, but the basic steps are the same. First, we have to populate a per-server configuration object within our directive handler. Then, at request time, we need to retrieve the same per-server object to access our data. Because per-directory configurations are the default, mod_perl offers a few shortcuts to them, such as passing the per-directory object (`$cfg`) to our directive handler through the argument list. However, for per-server configurations we have to do things explicitly.

It is easier to begin this part of the discussion with what happens at request time. As we discussed in Chapter 4, data related to the server configuration is available through the Apache server record. This happens to include any per-server configuration data. The object containing the per-server configuration for your module can be retrieved by passing `Apache::ModuleConfig->get()` an `Apache::Server` object as the first argument:

```
my $scfg = Apache::ModuleConfig->get($r->server, __PACKAGE__);
```

This leaves only one piece remaining—how to differentiate between per-server and per-directory configurations within the actual directive handler. Well, this leads us to explain some of the details we left out of Recipe 7.8.

For per-server configurations, the solution rests in the second argument passed to your directive handler: the `Apache::CmdParms` object `$parms`, which represents yet another Apache record (`cmd_parms` to be specific). The full list of information available through this object is outlined in Table 7.3.

Table 7.3 `Apache::CmdParms` *Methods*

Method	Description
`cmd()`	An `Apache::Command` object, which provides access to the Apache command record for this directive.
`getline()`	Provides direct access to the `httpd.conf`.
`info()`	Data corresponding the `cmd_data` field in the Apache command record for this directive.
`limited()`	A bitmask representing any `<Limit>` directives that apply to this directive.
`override()`	A bitmask representing the values set in `req_override` in the Apache command record for this directive.
`path()`	The `<Location>` or `<Directory>` to which the directive applies.
`server()`	Returns an `Apache::Server` object corresponding to the server to which the directive applies.

As we hinted in Recipe 7.8, in practice much of the information available through the `Apache::CmdParms` class is rarely used. The notable exception to this is the `server()` method, which contains an `Apache::Server` object for the server to which the directive is being applied. This is used to access the per-server configuration directive object created by `SERVER_CREATE()`. For instance, if we had wanted to implement `CleanLevel` on a per-server basis instead, we could have used the following:

```
sub CleanLevel ($$$) {

  my ($cfg, $parms, $arg) = @_;

  # Get the per-server configuration from the current Apache server record.
  # We ignore the passed in, per-directory object $cfg.
  my $scfg = Apache::ModuleConfig->get($parms->server, __PACKAGE__);

  # CleanLevel and CleanWithBleach are equivalent directives, but
  # we like to know which they used anyway.  We can tell by getting
  # the data from the cmd_data slot.
  $scfg->{_bleach} = $parms->info;

  # Continue along...
}
```

The per-server configuration object can then be accessed in your handler via `Apache::ModuleConfig->get()` class using an `Apache::Server` object as the first argument, as previously illustrated.

Now that you have the ability to create both per-server and per-directory configurations, you might find yourself wondering whether to use one, the other, or both. Because limiting your runtime overhead wherever possible makes sense, if your directive is going to be applied on only a per-server basis, using only the `SERVER_CREATE()` and `SERVER_MERGE()` routines and limiting where the directive can occur via the `req_override` setting in the Apache command record is the correct approach. This might happen if you are configuring a `PerlTransHandler` or `PerlPostReadRequestHandler`, both of which are incapable of residing inside of a `<Location>` or other container directive.

Obviously, if you want to perform per-directory merges you will want to stick with `DIR_CREATE()` and `DIR_MERGE()`. One thing that may not be immediately obvious, however, is that you do not have to manage both per-directory and per-server configurations unless you want to enforce separate and distinct behaviors—per-directory directives that exist on a per-server level are merged into `<Location>` and friends due to a single `DIR_MERGE` call for each virtual host at startup.

7.11. Overriding Core Directives

You want to transparently override a core server directive using your own custom directive.

Technique

Go ahead.

```
package Cookbook::WinBitHack;

BEGIN {
  eval{
    require Win32::File;
    Win32::File->import(qw(READONLY ARCHIVE));
  };
}
```

```
use Apache::Constants qw(OK DECLINED OPT_INCLUDES DECLINE_CMD);
use Apache::File;
use Apache::ModuleConfig;

use DynaLoader;

use 5.006;

use strict;

our $VERSION = '0.01';
our @ISA = qw(DynaLoader);

__PACKAGE__->bootstrap($VERSION);

sub handler {
  # Implement XBitHack on Win32.
  # Usage: PerlModule Cookbook::WinBitHack
  #        PerlFixupHandler Cookbook::WinBitHack
  #        XBitHack On|Off|Full

  my $r = shift;

  my $cfg = Apache::ModuleConfig->get($r, __PACKAGE__);

  return DECLINED unless (
     $^O =~ m/Win32/                  &&    # we're on Win32
     -f $r->finfo                     &&    # the file exists
     $r->content_type eq 'text/html'  &&    # and is HTML
     $r->allow_options & OPT_INCLUDES &&    # and we have Options +Includes
     $cfg->{_state} ne 'OFF');              # and XBitHack On or Full

  # Gather the file attributes.
  my $attr;
  Win32::File::GetAttributes($r->filename, $attr);

  # Return DECLINED if the file has the ARCHIVE attribute set,
  # which is the usual case.
  return DECLINED if $attr & ARCHIVE();

  # Set the Last-Modified header unless the READONLY attribute is set.
  if ($cfg->{_state} eq 'FULL') {
    $r->set_last_modified((stat _)[9]) unless $attr & READONLY();
  }

  # Make sure mod_include picks it up.
  $r->handler('server-parsed');

  return OK;
}

sub DIR_CREATE {

  my $class = shift;
  my %self  = ();

  # XBitHack is disabled by default.
  $self{_state} = "OFF";

  return bless \%self, $class;
}

sub DIR_MERGE {

  my ($parent, $current) = @_;

  my %new = (%$parent, %$current);

  return bless \%new, ref($parent);
}

sub XBitHack ($$$) {

  my ($cfg, $parms, $arg) = @_;

  # Let mod_include do the Unix stuff - we only do Win32.
  return DECLINE_CMD unless $^O =~ m/Win32/;

  if ($arg =~ m/^(On|Off|Full)$/i) {
    $cfg->{_state} = uc($arg);
  }
  else {
    die "Invalid XBitHack $arg!";
  }
}
1;
```

Comments

Recipe 6.5 discussed how the XBitHack directive is essentially useless on the Win32 platform. Whereas in the last chapter we implemented a PerlFixupHandler to remedy the problem, the issue remained that the XBitHack directive still pointed to the mod_include implementation; there is the (albeit slight) possibility that unexpected behaviors could arise where the two implementations collide. A better solution would be to override the default XBitHack directive with our own implementation so that mod_include is sure not to be in the way.

Our new Cookbook::WinBitHack combines attributes from all the other custom directive examples we have seen so far. It bootstraps itself, merges directives on a per-directory basis, and provides a directive handler subroutine. The only thing new here is the inclusion of the DECLINE_CMD constant, which is similar to the standard DECLINE constant except that DECLINE_CMD is designated for use within directive handlers. Basically, we are telling Apache that if a certain criterion is met (the platform is not Win32), we would like to decline handling this directive and instead pass it back to Apache, which will then seek another handler to process it.

The only pitfall to be wary of when choosing to decline processing a directive comes when using RAW_ARGS to implement a container directive. As you recall from Recipe 7.9, the RAW_ARGS prototype passes the directive handler an open filehandle as the final argument in the argument list. This filehandle is actually a tied filehandle that reads from httpd.conf using a native Apache utility routine. Because of this, there is no way to inspect the raw data within a container directive and seek() back to return what you read. Thus, you should *not* use data read from $fh to determine whether you will return DECLINE_CMD.

The use of DECLINE_CMD here, along with some nonstandard syntax to bring in Win32::File constants, allows us to reuse the same httpd.conf for both Win32 and Unix servers. If we are running on Win32 everything proceeds as planned: the XBitHack directive is intercepted and our PerlFixupHandler is run. If we are on some other platform, the code will still compile, but Cookbook::WinBitHack will not interfere, and instead pass all XBitHack processing over to mod_include. Both Apache::AutoIndex and Apache::Language use a similar approach to pass the Perl implementations of mod_autoindex and mod_mime over to their faster C counterparts.

```
return DECLINE_CMD if Apache->module('mod_autoindex.c');
```

For the sake of clarity, here is the corresponding Makefile.PL for our new Cookbook::WinBitHack.

Listing 7.10 `Makefile.PL` *for* `Cookbook::WinBitHack`

```
package Cookbook::WinBitHack;

use ExtUtils::MakeMaker;
use Apache::ExtUtils qw(command_table);
use Apache::src ();
use Config;

use strict;

my @directives = (
  { name         => 'XBitHack',
    errmsg       => 'Off, On, or Full - On and Full are equivalent',
    args_how     => 'TAKE1',
    req_override => 'OR_OPTIONS', },
);

command_table(\@directives);

my %config;

$config{INC} = Apache::src->new->inc;

if ($^O =~ m/Win32/) {
  require Apache::MyConfig;

  $config{DEFINE}  = ' -D_WINSOCK2API_ -D_MSWSOCK_ ';
  $config{DEFINE} .= ' -D_INC_SIGNAL -D_INC_MALLOC '
    if $Config{usemultiplicity};

  $config{LIBS} =
    qq{ -L"$Apache::MyConfig::Setup{APACHE_LIB}" -lApacheCore } .
    qq{ -L"$Apache::MyConfig::Setup{MODPERL_LIB}" -lmod_perl};
}

WriteMakefile(
  NAME           => 'Cookbook::WinBitHack',
  VERSION_FROM   => 'WinBitHack.pm',
  PREREQ_PM      => { mod_perl => 1.26_01 },
  ABSTRACT       => 'An XS-based Apache module',
  AUTHOR         => 'authors@modperlcookbook.org',
  clean          => { FILES => '*.xs*' },
  %config,
);
```

If you are salivating at the thought that, because Perl can pass off processing to C on-the-fly, perhaps there is a way to remove extraneous C modules from the server altogether when the functionality is implemented in Perl. Well, it's quite devious, but it is possible using the `Apache::Module` class, which is not part of the mod_perl distribution but is available from CPAN. At the time of writing, `Apache::Module` has not yet been ported to Win32.

The `Apache::Module` class provides an interface into the Apache module record we have been tiptoeing around. The Apache module record defines exactly what phases of the Apache lifecycle the module will be entering, which routines it will use to handle these phases, and some additional information important to either Apache or the module itself. Again, the exact structure of the module record can be found in `http_config.h`, along with some helpful documentation.

Internally, Apache maintains a linked list of module records for all the active modules. This list is not necessarily the same as the modules compiled into the server, but represents the modules Apache will consider when it comes across a directive token in `httpd.conf` or when serving a request. The `Apache::Module` class provides hooks into the Apache module record and allows you to inspect it and (rarely and unwisely) manipulate it.

Because mod_include implements the actual Server Side Include engine used to implement our version of `XBitHack`, you probably would not want to remove mod_include from your configuration altogether. However, if you really want to steal the wind from another module, using `Apache::Module->remove()` outside of a `handler()` subroutine will deactivate the module in a manner similar to the `ClearModuleList` directive, but for a single module.

```
use Apache::Module ();

my $modp = Apache::Module->find('userdir');
$modp->remove if $modp;

sub handler {

  my $r = shift;

  # Continue along...
}
```

Although novel, operating on the Apache module record in this manner is generally very unwise; a better solution would be to use `ClearModuleList` and `AddModule`, which will almost certainly result in less segfaults.

While we are on the topic, one of the more constructive uses for `Apache::Module` is within a directive handler. The `Apache::Module` distribution also provides the `Apache::Command` class, which provides the runtime interface for the Apache command record. You will remember this record as where all the settings that you specified in `Makefile.PL` and passed to `command_table()` finally reside. Each of the methods from the `Apache::Command` class corresponds to the name of a field in the Apache command record as given in Recipe 7.8, so there is no reason to list them here. And as previously mentioned, you can obtain an `Apache::Command` object by calling the `cmd()` method on the `Apache::CmdParms` object passed to your subroutine (`$parms` in our examples).

Although the route to the `Apache::Command` object is rather circuitous, it can be somewhat useful in exception handling during server startup. The following code will allow you to not have to repeat the usage for your directive a second time. It pulls the information right from the data you provided in your `Makefile.PL`.

```
sub XBitHack ($$$) {

  my ($cfg, $parms, $arg) = @_;

  if ($arg =~ m/^(On|Off|Full)$/i) {
    $cfg->{_state} = uc($arg);
  }
  else {
    die "Invalid $arg! ", (join " ", $parms->cmd->name,
                                     $parms->cmd->errmsg);
  }
}
```

7.12. Adding Unique Server Tokens

You want to modify the outgoing Server response header to represent your module.

Technique

Use the `Apache::add_version_component()` function.

```
package Apache::WinBitHack;

use strict;
```

```
our $VERSION = '0.01';
our @ISA = qw(DynaLoader);

my ($module) = __PACKAGE__ =~ /.*::(.*)/;
Apache::add_version_component("$module/$VERSION");

# Continue along...
```

Comments

Although we certainly do not advocate that you maim the `Server` header for every module that you write, if you put a great deal of effort into an application, you might want to put your mark on it. Using the `add_version_component()` function will add your token to the end of the `Server` header, resulting in something similar to this:

```
Server: Apache/1.3.22 (Unix) mod_perl/1.26 WinBitHack/0.01
```

If you tried simply modifying the `Server` header for the response using `$r->headers_out->set()`, you quickly found it had no effect (though we are proud of you for trying). This is because Apache overwrites the `Server` header with whatever was populated using the official Apache API when you call `$r->send_http_header()`.

If you are interested in finding out what the `Server` header will be at runtime, you can use the `SERVER_VERSION` constant from `Apache::Constants`, which is really a call to `ap_get_server_version()` from the Apache C API.

One final option for varying the `Server` header is to set the `$Apache::Server::AddPerlVersion` global to a true value in your `startup.pl`, which will signal mod_perl to add the version of perl that is embedded in Apache as well.

7.13. Releasing a Module to CPAN

You want to release your module to CPAN under the Apache namespace.

Technique

Make up a distribution tarball as in Recipe 7.5, and then follow these instructions.

Comments

Generally, it is a good idea before releasing a module to CPAN to discuss it in an appropriate forum and get some initial feedback. For most Perl modules the newsgroup `comp.lang.perl.modules` is the place to provide an RFC describing the nature of your module, the needs it fills that cannot be provided for by other modules, decide on the namespace the module ought to occupy, and so on.

The approach for mod_perl modules is slightly different than that of the rest of CPAN. The `Apache::` namespace has been reserved for modules that cannot exist outside of the mod_perl environment (with a few historical exceptions, like `Apache::Session`). As such, the `Apache` tree maintains its own module list, `apache-modlist.html`, which comes as part of the mod_perl distribution. Because the mod_perl community is essentially a self-governing subset of CPAN, it is normal practice before releasing your module to present it as an RFC to the mod_perl mailing list, `modperl@perl.apache.org`. Unlike many other Perl mailing lists, the mod_perl list tends to be friendly and flame-free. The people there spend an inordinate amount of time assisting both newbies and seasoned programmers to the benefit of all, so you shouldn't feel intimidated.

Releasing an RFC will accomplish a few objectives. First, it will help you determine whether the concept you are proposing is too broad, too narrow, not extensible enough, or duplicates an existing module. It will also give you a chance to improve your module almost immediately due to the aggregate knowledge available that comes from an open-source approach.

After receiving the feedback and coming to an agreement with the community, you can safely release your module to CPAN following the normal procedures listed in `http://www.cpan.org/modules/04pause.html`. After you have received a confirmation e-mail acknowledging the success of your upload, send an e-mail to the mod_perl mailing list using a subject line similar to

```
[ANNOUNCE] Apache::Pollywog-0.01
```

The final step is to edit `apache-modlist.html` to include the details of your module and e-mail a `diff` generated patch to the mod_perl development mailing list at `dev@perl.apache.org`.

```
$ cp apache-modlist.html apache-modlist.html~
$ vi apache-modlist.html~
$ diff -u apache-modlist.html apache-modlist.html~ > apache-modlist.diff
```

Now you are free to fix the bugs the users of your module will uncover.