CHAPTER 16

The PerlLogHandler and PerlCleanupHandler

Introduction

We now come to the final phases of the Apache request lifecycle—the `PerlLogHandler` and `PerlCleanupHandler`. The `PerlLogHandler` is used for (you guessed it) logging. It allows you to replace the `LogFormat`, `CustomLog`, and other directives provided by the default mod_log_config with logging routines specifically designed for your application. No longer do you have to rely on flat files, pipes, or log rotation scripts to gather useful Web server statistics.

mod_perl provides the `PerlCleanupHandler` as a final stage of the request. Conceptually, the Apache request cycle is complete, so mod_perl gives you the chance to clean up any Perl leftovers. A good example of this can be found in the `Apache::File->tmpfile()` method, which adds a handler to the cleanup pseudo-phase to remove the temporary file it created.

Strictly speaking, the `PerlCleanupHandler` is not a phase of the Apache request cycle. The Apache C API provides a way to run processing at the end of each request, but a C module has to specifically want this processing to occur—it is not called automatically. mod_perl provides access to this API in two ways: the `PerlCleanupHandler` and the `register_cleanup()` method

from the Apache class. `$r->register_cleanup()` is, for the most part, a synonym for the `PerlCleanupHandler`, so it can be used interchangeably with the `$r->push_handlers(PerlCleanupHandler => 'My::Cleanup')` syntax seen elsewhere. Note that this differs from the `register_cleanup()` method from the `Apache::Server` class, which is used to run code when the server is restarted or shut down, as shown in the next chapter.

Because the `PerlCleanupHandler` is not a true phase, you don't see many handlers written exclusively for it. A more frequent and idiomatic approach is illustrated in a few recipes from previous chapters, such as Recipes 2.11, 4.3, and 8.13. In these examples, a cleanup routine is pushed onto the `PerlCleanupHandler` stack to tidy up after some custom processing.

Another idiomatic use for the `PerlCleanupHandler` is actually as a replacement for the `PerlLogHandler`. Despite its name, the `PerlLogHandler` is not necessarily the best phase to insert logging routines, especially ones that are rather process-intensive. As it turns out, the connection to the client is not actually closed until *after* the Apache logging phase is complete. If you have a long-running `PerlLogHandler`, you might notice that the browser sits and waits as though it expects more content (as evident from the moving status bar in some browsers). To get around this minor annoyance you can install any `PerlLogHandler` as a `PerlCleanupHandler` instead. Although there is no performance improvement at all— in both cases the child has not been released to serve other requests—logging from a `PerlCleanupHandler` gives the appearance of a snappy application.

The recipes in this chapter ought to help you on your quest for meaningful and useful logs, which are an extremely important facet of a successfully deployed application.

16.1. Logging to a Database

You are getting tired of continually parsing your `access_log` and want a more flexible solution.

Technique

Use a `PerlLogHandler` to log directly to a database, then use some creative SQL to process and warehouse the data.

```
package Cookbook::SiteLog;

use Apache::Constants qw(OK);

use DBI;
use Time::HiRes qw(time);

use strict;

sub handler {

  my $r = shift;

  my $user  = $r->dir_config('DBUSER');
  my $pass  = $r->dir_config('DBPASS');
  my $dbase = $r->dir_config('DBASE');

  my $dbh = DBI->connect($dbase, $user, $pass,
   {RaiseError => 1, AutoCommit => 1, PrintError => 1}) or die $DBI::errstr;

  # Gather the per-request data and put it into a hash.
  my %columns = ();

  $columns{waittime}   = time - $r->pnotes("REQUEST_START");
  $columns{status}     = $r->status;
  $columns{bytes}      = $r->bytes_sent;
  $columns{browser}    = $r->headers_in->get('User-agent');
  $columns{filename}   = $r->filename;
  $columns{uri}        = $r->uri;
  $columns{referer}    = $r->headers_in->get('Referer');
  $columns{remotehost} = $r->get_remote_host;
  $columns{remoteip}   = $r->connection->remote_ip;
  $columns{remoteuser} = $r->user;
  $columns{hostname}   = $r->get_server_name;
  $columns{encoding}   = $r->headers_in->get('Accept-Encoding');
  $columns{language}   = $r->headers_in->get('Accept-Language');
  $columns{pid}        = $$;

  # Create the SQL
  my $fields = join "$_,", keys %columns;
  my $values = join ', ', ('?') x values %columns;
```

```
  my $sql = qq(
    insert into www.sitelog (hit, servedate, $fields)
    values (hitsequence.nextval, sysdate, $values)
  );

  my $sth = $dbh->prepare($sql);

  $sth->execute(values %columns);

  $dbh->disconnect;

  return OK;
}
1;
```

This class can then be activated with a single line in your `httpd.conf`:

```
PerlLogHandler Cookbook::SiteLog
```

Comments

Although you might be interested in the number of hits your application receives over time for performance reasons, you can rest assured that your marketing department is far more interested in things such as which pages are hit most frequently, impressions per unique user, and which URL brought the user to the site. Unless you want to slice and dice your plain Apache `access_log` in a myriad of ways to meet (changing) marketing requirements, you might want to consider logging requests directly to a database and creating reports using SQL queries.

The solution code is a simple `PerlLogHandler` that extracts some interesting data from the request and inserts it into a table. The table and sequence for this handler was created using the following Oracle-specific SQL, which you can alter to serve the needs of your application and/or platform. You will want to choose the size and other column attributes carefully, and consider using the `substr()` function from the handler to help make sure there are no data overflow errors on inserts.

```
CREATE TABLE WWW.SITELOG (
  HIT        NUMBER(20),
  SERVEDATE  DATE,
  WAITTIME   NUMBER(10,2),
  STATUS     NUMBER(3),
  BYTES      NUMBER(10),
  BROWSER    VARCHAR2(80),
```

```
  FILENAME   VARCHAR2(150),
  URI        VARCHAR2(150),
  REFERER    VARCHAR2(150),
  REMOTEHOST VARCHAR2(80),
  REMOTEIP   VARCHAR2(15),
  REMOTEUSER VARCHAR2(30),
  HOSTNAME   VARCHAR2(50),
  ENCODING   VARCHAR2(50),
  LANGUAGE   VARCHAR2(30),
  PID        NUMBER(10)
)
CREATE SEQUENCE WWW.HITSEQUENCE
  INCREMENT BY 1
  START WITH 1
  MAXVALUE 1.0E28
  MINVALUE 1
  NOCYCLE
  CACHE 20
  ORDER
```

A few modules on CPAN perform similar functions, such as `Apache::DBILogger` and `Apache::DBILogConfig`, but to create a really useful activity log you will want to define fields that make sense to your application and environment. Here, for instance, we took advantage of the `PerlPostReadRequestHandler` described in Recipe 11.3 to obtain a very granular measurement of the time between the start of the request and when the user is able to view the completed page. If you are into exception handling, you might want to create a column that stores a `pnotes()` string set by the exception routine so you can track the exact cause of every error and investigate those that seem to happen frequently. Tracking query strings or skipping logging altogether for images is just as easy after you have the basic framework established—the possibilities are endless and can be custom tailored to your exact needs.

The main benefit in logging to a database is that all the information about the activity of the application is available to you via some relatively simple SQL. For instance, the `Cookbook::LogChart` module from Recipe 15.1 uses the table we created here to render a simple bar chart of requests per hour that can be used to spot peak activity periods. Other trends that might be of interest are the number of users who have their browsers configured to a language other than the default for the application (which might indicate that you are wasting CPU cycles on unnecessary content negotiation), shifts in browser preferences (that could wreck your DHTML), daily `404` reports, and more.

The real disadvantage to this approach is that any database activity is process-intensive, so logging to a database means that the `httpd` child process is not being freed to serve the next request as quickly as it would when writing to a flat file. This is definitely a consideration, as it affects the overall performance of your application, but using a persistent database connection through `Apache::DBI` or other means should lessen the blow. You will also want to consider scheduled jobs to warehouse the data in your ever-growing tables as well as proper indexes to speed up frequent queries. If your database platform supports precompiled stored procedures, using them instead of raw SQL will also improve performance and help keep the insert overhead to a minimum.

16.2. Logging to a Flat File

You do not want to log to a database but you still want to have the control a `PerlLogHandler` offers.

Technique

Install and use the `Apache::LogFile` module, available from CPAN. After it is installed, activate a new logfile by adding a `PerlLogFile` directive to your `httpd.conf`. You will also need to load both the `Apache::LogFile` and `Apache::LogFile::Conf` classes, included with the distribution, using the `PerlModule` directive.

```
PerlModule Apache::LogFile
PerlModule Apache::LogFile::Config

PerlLogFile logs/detailed_log Cookbook::LogHandle
```

This binds the file `logs/detailed_log` to the Perl filehandle `Cookbook::LogHandle`. You could just simply `print()` to that filehandle; however, to make life easier we can create an object-oriented abstraction for our logging. The sample module `Cookbook::DetailedLog` provides just that.

```
package Cookbook::DetailedLog;

use Apache::Constants qw(OK);

use Time::HiRes qw(gettimeofday tv_interval);
```

```
use strict;

sub handler ($$) {

  my $this = shift;
  my $class = ref $this || $this;

  my $r = shift || Apache->request();
  my $self = {};

  $self->{_start}   = [gettimeofday];
  $self->{_request} = $r;

  bless $self, $class;

  $r->pnotes('DETAILED_LOG', $self);

  return OK;
}

sub DESTROY {

  my $self = shift;

  my $r = $self->{_request};

  my $entry = join(' ',
                        $$,
                        $r->uri,
                        time(),
                        tv_interval($self->{_start})
                  );

  print Cookbook::LogHandle $entry;
}
1;
```

To use `Cookbook::DetailedLog`, simply add it to the previous `httpd.conf` configuration as a `PerlPostReadRequestHandler` or a `PerlFixupHandler` depending on whether you want to enclose it within a `<Location>` or `<Directory>` container.

```
<Location /bannerads>
  PerlFixupHandler Cookbook::DetailedLog
</Location>
```

Comments

Apache's built-in file logging is simple and good. However, life isn't simple and sometimes you need to write very specialized log files for specific purposes. You could try doing this yourself, but the complexities of opening and closing log files in a multi-process Apache server are, shall we say, difficult. Instead, using the CPAN module `Apache::LogFile` is quite simple. `Apache::LogFile` binds a log file or log process to a Perl filehandle. After you have the filehandle you can simply print things to it. This is great for little things like debug logs, all the way to specialized log formats.

In our example we bind the file `logs/detailed_log` to the Perl filehandle `Cookbook::LogHandle`. We chose a regular file to log to, but it could just as easily have been a piped log script, just like Apache's logging system. With the binding complete, we can now send data to the file using a simple print statement from within our custom handlers, like

```
print Cookbook::LogHandle 'This goes into logs/detailed_log -- hey!'
```

To add to the functionality of `Apache::LogFile` you can define your own logging object. For this recipe we define `Cookbook::DetailedLog` that saves away the request and the exact time the request started (using the `Time::HiRes` module, available on CPAN). Our Perl method handler can be configured either as a `PerlPostReadRequestHandler` or a `PerlFixupHandler`, depending on whether you want to use it for all requests or just a smaller subset of requests. Keep in mind that when used as a `PerlFixupHandler` the time you are measuring is essentially the time taken to process the content handler and not the entire request. Either way, it creates a new logging object, which we store away using the `pnotes()` method to make sure it does not disappear too soon.

The object is largely forgotten until the request is being destroyed at the end of the Apache request cycle. At this point Perl will try to deallocate our logging object. We are clever and use Perl's built-in concept of the `DESTROY()` method to call code when this happens, letting `DESTROY()` do the actual logging of data. This ensures we don't forget to do it, and allows us to delay logging to the last possible moment. It also gives us an accurate time stamp for calculating the total request processing time, should we choose to use our class from a `PerlPostReadRequestHandler`.

The final step in our `DESTROY()` method is the actual print call. In this example we print the process ID, the request URL, the Unix time stamp, and the detailed transaction time in seconds. A sample looks like this:

```
30323 /xml/captains.html 1001900224 0.438851
30324 /xml/pirates.html 1001900228 0.120550
```

16.3. Altering the Request-Line

You want to change the request URI caught by the default `CustomLog` and `LogFormat` directives. You might need this in cases where you have altered the URI through `$r->uri()` and want your Apache logs to record the altered URI instead of the requested URI.

Technique

Alter the value of the HTTP Request-Line Apache stores internally using the `the_request()` method from the `Apache` class to match the new URI.

```
# We assume that request URI was altered previously,
# such as from a PerlTransHandler.  $r->uri() now contains
# something different than the client request URI.
my $uri = $r->uri;

# Now, make logs to the access_log match the new URI.
(my $request_line = $r->the_request) =~ s/ (.*) / $uri /;
$r->the_request($request_line);
```

Comments

If you have chosen to forego some of the more interesting ways mod_perl allows you to handle your logging needs, you might be relying on the default `access_log` behavior to track traffic on your application. The default logging configuration for Apache looks like

```
LogFormat "%h %l %u %t \"%r\" %>s %b" common
CustomLog /usr/local/apache/logs/access_log common
```

which is perfectly fine—unless you use a `PerlTransHandler` to alter the requested URI. The problem is that `%r` returns the value of the HTTP Request-Line. The Request-Line is stored in the `the_request` slot of the Apache request record and might look like

```
GET /index.html HTTP/1.1
```

The value stored in `the_request` exists independently of the `uri` slot of the request record— if you make changes to the URI behind the scenes using `$r->uri()`, the URI in `%r` still represents the actual request, not the resource that was actually served.

If you are more interested in logging the client request than the served resource, then this disparity is not a big issue. However, if you want to track impressions for individual server resources, then `%r` is not suitable because it does not represent what the client actually saw. This becomes a real issue with the `combined` log format, which uses the value of the Request-Line, and which is relied upon by log analyzers like WebTrends.

One solution is to modify the value for the Request-Line that Apache has stashed away directly using the `the_request()` method, as shown in the solution code. Of course, you could also just create a new `LogFormat` comprised of the individual components of the Request-Line

```
LogFormat "%h %l %u %t \"%m %U %H\" %>s %b" common
```

but that is not nearly as fun. It also still leaves the main issue unresolved—the URI in the `uri` and `the_request` slots of the Apache request record do not match. Because other handlers might use `the_request` in program logic, getting the two to agree can be important. For instance, mod_rewrite offers the ability to use `%{THE_REQUEST}` in a `RewriteCond`. For this reason, it is probably best to use the solution code when you actually change the request URI as to not cause any undue problems.

16.4. Logging Nonstandard Data

You would like to use Apache's built-in logging routines, but you need to log nonstandard data.

Technique

Set the data you want to log with the `subprocess_env()` or `notes()` methods from the `Apache` class, then configure your `LogFormat` line with the appropriate `%{`*NAME*`}e` or `%{`*NAME*`}n` entries.

```
LogFormat "%h %l %u %t \"%r\" %>s %b %{TOTAL_SECS}e %{SESSION}n" common_timed
CustomLog logs/access_log common_timed
```

Comments

If you want to log to files on disk or to `syslog`, you can't get much better than Apache's bundled mod_log_config. It supports all the basic request logging quite well,

and it supports a number of extensions that allow you to add your own custom data to the log files.

One easy way to log your own data is to use the `notes()` method described in Recipe 8.11. Any value stored in the notes table of the Apache request record can be logged; all we need to do is set the value and modify the `LogFormat` directive accordingly. For example, to log an `Apache::Session` generated session ID for the request, we could stash the session away by calling `$r->notes(SESSION => $id)`, then add this value to the log file using `%{SESSION}n` in the `LogFormat` line, as shown earlier.

An alternative way to pass logging data is done by using Apache's environment variables. We can set these by calling the Apache `subprocess_env()` method, as in

```
$r->subprocess_env(TOTAL_SECS => time - $r->pnotes('REQUEST_START'))
```

A corresponding `LogFormat` entry would be `%{TOTAL_SECS}e`, as also shown previously. As already mentioned in Recipe 8.11, the choice of `notes()` over `subprocess_env()` is a matter of personal preference. Note that both of these data elements can be set at any convenient point as long as it's before the final logging phase. Output using the new `common_timed` format might look like

```
127.0.0.1 - - [28/Sep/2001:01:55:05 -0700] "GET /index.htm HTTP/1.0" 200 290
➥0.018991 -
127.0.0.1 - - [28/Sep/2001:01:55:08 -0700] "GET /index.html HTTP/1.0" 200 1469
➥0.030349 0x89AE2234==
```

16.5. Conditional Logging

You want to control the logging behavior of mod_log_config from within mod_perl, enabling and disabling logging based on certain criteria.

Technique

Use the built-in ability of mod_log_config to conditionally log requests by setting an environment trigger using the `subprocess_env()` method from the `Apache` class.

In your `httpd.conf`, add:

```
CustomLog /usr/local/apache/logs/access_log common env=!SKIP
```

Then, use the following snippet in your handler:

```
# $skip_me represents some criterion that means
# we do not want to log the request.
$r->subprocess_env->set(SKIP => 1) if $skip_me;
```

Comments

If you use a `PerlLogHandler` or `PerlCleanupHandler` for logging requests, then turning logging on or off from any point in the request is a simple task. Recipe 8.8 showed how the handler stack can be reset for any phase by setting the phase to `undef`.

```
$r->set_handlers(PerlLogHandler => undef);
```

Although mod_perl provides the power to manipulate the request handlers at runtime, this same ability is not carried over for Apache C modules; generally, after you configure in a C extension module in `httpd.conf` there is nothing more you can do at request time. The rare exception to this is mod_log_config, which recognizes that you might want to conditionally log a request and offers a way to control whether the logging routine runs.

Traditionally, dynamically controlling logging is handled by coupling mod_log_config with modules such as mod_setenvif or mod_rewrite. The `CustomLog` directive has a conditional aspect to it that allows you to toggle logging based on environment variables. For instance, the following configuration allows you to skip the logging request for the pesky `favicon.ico` using mod_rewrite to set the `SKIP` environment variable:

```
CustomLog /usr/local/apache/logs/access_log.skip common env=!SKIP

RewriteEngine On
RewriteRule (/favicon\.ico$) $1 [E=SKIP:1]
```

As discussed in Recipe 8.10, Apache C modules do not really manipulate `%ENV` so much as they populate the `subprocess_env` table in the Apache request record and rely on other processes to pass that on to `%ENV`. This also works in the opposite direction—you cannot set a value in `%ENV` at request time and expect an Apache C module to be able to see it, so setting `$ENV{SKIP}=1` from a handler would not have the same effect as the preceding configuration. This makes it difficult to control conditional logging based from a traditional CGI environment without relying on mod_setenvif or mod_rewrite to populate the `subprocess_env` table. With mod_perl we are not as limited.

mod_log_config follows the same path as mod_setenvif and mod_rewrite in that it is really looking from a value in the `subprocess_env` table rather than a true environment variable. By setting `$r->subprocess_env()` directly, we can use our module in place of mod_setenvif or mod_rewrite. This, coupled with the conditional configuration showed in the solution, allows us to skip over mod_log_config using any request time criteria we choose.

16.6. Intercepting Errors

You want to insert custom processing for the `error_log` similar to the way the `PerlLogHandler` allows for the `access_log`.

Technique

Use low-level Apache routines to redirect errors to a file of your choosing.

The following handler illustrates the use of a new class, `Cookbook::DivertErrorLog`, which is detailed in the following discussion.

```
package Cookbook::ErrorsToIRC;

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

use Cookbook::DivertErrorLog qw(set_error_log restore_error_log);

use Net::IRC ();
use Sys::Hostname ();

our ($irc, $host);

use strict;

sub handler {

  my $r = shift;

  # Create a temporary file for holding the errors for this request.
  my $fh = Apache::File->tmpfile;
```

```
  # Store away the filehandle for later.
  $r->pnotes(ERROR_HANDLE => $fh);

  # Push our log routine if we can divert the error_log to our file.
  $r->register_cleanup(\&send_to_irc) if set_error_log($fh);

  return OK;
}

sub send_to_irc {

  my $r = shift;

  my $irc_host = $r->dir_config('IRCHost') || 'localhost';

  $irc  ||= Net::IRC->new();
  $host ||= Sys::Hostname::hostname();

  # Restore the original error_log.
  # We do this so that the true Apache error_log captures
  # any errors from our processing here.
  my $error_log = restore_error_log;

  # Get the error filehandle we created earlier.
  my $fh = $r->pnotes('ERROR_HANDLE');

  seek($fh, 0, 0); # rewind

  # Open an IRC connection and send the diverted
  # error log across.  This is all pretty standard
  # Net::IRC stuff.
  my $conn = $irc->newconn(Nick    => "log-$$",
                           Server  => $irc_host,
                           Port    =>  6667,
                           Ircname => "Apache Log Bot $$ on $host");
  $conn->add_global_handler('376', \&on_connect);
  $irc->do_one_loop;

  $conn->privmsg('#logs', ('error_log for', $r->uri));
```

```
  while (my $line = <$fh>) {
    $conn->privmsg('#logs', $line);
  }

  $conn->quit();

  return OK;
}

sub on_connect {
  # Callback for the Net::IRC object.

  my $self = shift;

  $self->join('#logs');
}
1;
```

Comments

Unfortunately, although both Apache and mod_perl offer a nice, clean hook into the logging site access, neither provides a reasonable interface into the error-logging process. Of course, you can always define `ErrorLog` as a pipe and process errors that way. However, in doing so you are still left without the transactional concept of a single request, because simultaneous requests will interlace their error output. To process errors on a per-request level, we need a hook into the error-logging process itself.

Intercepting messages sent to the `error_log` is actually more involved than you might think at first. A simple solution is shown in Recipe 4.5, which uses `$s->error_fname()` to retrieve the *name* of the file specified by the `ErrorLog` directive and process that file directly. Although just reading in the error file and writing the results somewhere else is a relatively simple task, knowing exactly how many lines to slurp and write out is rather difficult, because a single request can produce many lines of error or diagnostic messages.

At first, you might want to use a `TIEHANDLE` interface, as in Recipe 6.10. This, however, is insufficient for a number of reasons. First, trapping calls to `warn()` or `die()` in current versions of Perl require you to handle `$SIG{__WARN__}` and `$SIG{__DIE__}` on a global level, which might interfere with other modules that rely on those signals. Additionally, a simple tie to `STDERR` will not capture error messages generated by core

Apache—calls like `$r->log_error()` or `$r->log->warn()` write directly to the file specified by the `ErrorLog` directive, and internal Apache error messages print to the C `stderr` error stream.

Because the usual gambits Perl affords us are not sufficient for intercepting all of these Apache and mod_perl error message variants, we need to take another approach. The following solution is the `Cookbook::DivertErrorLog` class, which combines some XS processing with a little Perl to effectively manipulate the place where errors are sent at request time. This class can then be used in a handler similar to the solution `Cookbook::ErrorsToIRC` module, where errors are collected in a temporary file from each request and sent to a private IRC channel.

To fully understand what is going on in our new class, it is important to know the process Apache uses for its error-logging mechanism. Apache stores two values in the Apache server record that are important to this process: `error_fname`, which specifies the value of the `ErrorLog` directive, and `error_log`, which holds a pointer to the open file or pipe. The file (or pipe) to which Apache will send its errors is opened when Apache starts and before it spawns any child processes. For us to redirect the error stream away from, say `logs/error_log`, we need to replace the `error_log` value in the Apache server record with a pointer we control. For our `Cookbook::DivertErrorLog` class, we are limiting ourselves to just a file implementation, which works out rather nicely as you will see.

As we mentioned, this approach requires both Perl and XS. As with all XS-based modules, it is best to start with `h2xs`.

```
$ h2xs -An Cookbook::DivertErrorLog
Writing Cookbook/DivertErrorLog/DivertErrorLog.pm
Writing Cookbook/DivertErrorLog/DivertErrorLog.xs
Writing Cookbook/DivertErrorLog/Makefile.PL
Writing Cookbook/DivertErrorLog/test.pl
Writing Cookbook/DivertErrorLog/Changes
Writing Cookbook/DivertErrorLog/MANIFEST
```

Here is `DivertErrorLog.pm`, which defines our end-user interface.

Listing 16.1 `DivertErrorLog.pm`

```
package Cookbook::DivertErrorLog;

use DynaLoader ();
use Exporter ();
use 5.006;
use strict;
```

```
our @ISA = qw(DynaLoader Exporter);

our @EXPORT_OK = qw(set_error_log restore_error_log);

our $VERSION = '0.01';

__PACKAGE__->bootstrap($VERSION);

use strict;

sub set_error_log {

  my $arg = shift;

  # The input can be either a filename or an open filehandle.
  # In either case, we need to isolate a file descriptor
  # for the file.
  my $fd  = fileno($arg);

  unless (defined $fd) {
     open(OUT, ">$arg") or return;
     $fd = fileno(*OUT);
  }

  # Call our XS set() routine, passing in an
  # Apache::Server object and the file descriptor.
  set(Apache->server, $fd);
}

sub restore_error_log {
  # Call our XS restore() routine, passing in an
  # Apache::Server object.

  restore(Apache->server);
}
1;
```

`Cookbook::DivertErrorLog` implements two functions: `set_error_log()` and `restore_error_log()`. `set_error_log()` replaces the current value of `error_log` in the Apache server record with a file of your choosing. It can receive either an active filehandle or a filename. `restore_error_log()` unplugs the custom log file set by the

`set_error_log()` function and replaces it with whatever was removed previously by `set_error_log()`.

Each of these Perl functions calls an associated XS function defined in `DivertErrorLog.xs`, which is dynamically pulled in using `DynaLoader`. As we mentioned, we will be directly manipulating the Apache server record. Rather than dig this record out of thin air, both functions rely on receiving an `Apache::Server` object passed in from the Perl code. This enables us to reduce the amount of XS required and hides the implementation magic from the end user of our class, which is always a nice touch.

Although the actual Perl is rather dull, the XS code that does the real work is much more interesting. Here is `DivertErrorLog.xs`

Listing 16.2 `DivertErrorLog.xs`

```
#include "EXTERN.h"
#include "perl.h"
#include "XSUB.h"
#include "mod_perl.h"

static FILE *original_log;

MODULE = Cookbook::DivertErrorLog      PACKAGE = Cookbook::DivertErrorLog

PROTOTYPES: ENABLE

int
set(s, fd)
  Apache::Server s
  int fd

  PREINIT:
    pool *p;

  CODE:
    RETVAL = 1;

    /* Get a memory pool */
    p = perl_get_startup_pool();

    /* Stash away the pointer to the current error_log */
    original_log = s->error_log;
```

Listing 16.2 *(continued)*

```
    /*
     * Open the new error_log descriptor for writing.
     * Make sure the original error_log is restored and
     * return undef on failure
     */
    if (!(s->error_log = ap_pfdopen(p, fd, "w"))) {
      s->error_log = original_log;
      XSRETURN_UNDEF;
    }

    /* Make stderr point to the new error_log as well */
    dup2(fileno(s->error_log), STDERR_FILENO);

  OUTPUT:
    RETVAL

char *
restore(s)
  Apache::Server s

  PREINIT:
    char *fname;

  CODE:
    /* Restore the stashed error_log pointer */
    s->error_log = original_log;

    /* Point stderr back to the original error_log */
    dup2(fileno(s->error_log), STDERR_FILENO);

    /* Return the original error_log file, just to be informative */
    RETVAL = s->error_fname;

  OUTPUT:
    RETVAL
```

The `set()` function does a few things. It stores away the current `error_log` file pointer and resets it to the active filehandle passed in from the Perl routine. To do this, the function relies on the Apache `ap_pfdopen` function, defined in `alloc.c` in the Apache sources. As it turns out, this function expects a memory pool as the first argument. Although presenting a full explanation of how Apache memory pools work is outside

the scope of this book, we chose to get our memory pool from the Perl startup pool instead of the request pool, keeping with the same memory allocation that Apache itself gives `error_log`.

After the `error_log` field of the Apache server record has been set to point to its new location, `set()` needs to do one final thing. As mentioned earlier, to successfully divert all possible sources of error messages we need to capture writes to the `stderr` error stream. `stderr` receives errors on the Perl side (such as `die()` and writes to `STDERR`) as well as error and debug messages from Apache and mod_perl internals. To intercept these types of writes we make `stderr` point to the same place as `s->error_log` does, which should cover all of our bases (as long as `STDERR` is not `tie()`d, that is).

`restore()` is far easier. It simply moves the old value of `error_log` back to the Apache server record, and points `stderr` to the new (old) value of `s->error_log` for the reasons just mentioned. As a convenience, it also returns the value of the `error_fname` slot of the Apache server record to let you know what value of `error_log` was restored.

The only items missing to complete our class are the `Makefile.PL` and `typemap` files. The `Makefile.PL` is essentially the same as the other XS-based modules we have presented so far, but we will show it here for completeness.

Listing 16.3 `Makefile.PL` *for* `Cookbook::DivertErrorLog`

```
#!perl

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

use strict;

my %config;

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

if ($^O =~ /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 } .
```

Listing 16.3 *(continued)*

```
    qq{ -L"$Apache::MyConfig::Setup{MODPERL_LIB}" -lmod_perl};
}

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

The `typemap` file, which converts the incoming `Apache::Server` object to an Apache server record for the XS routine, only contains one element:

Listing 16.4 `typemap` *for* `Cookbook::DivertErrorLog`

```
TYPEMAP
Apache::Server          T_PTROBJ
```

After creating all the relevant pieces, all that is left is to run the standard `perl Makefile.PL` and friends to install the class and it is ready to go.

This brings us back to the code shown in the solution, `Cookbook::ErrorsToIRC`, which uses our snazzy `Cookbook::DivertErrorLog` class to isolate error messages from a specific part of the request cycle. Where this handler is installed is entirely dependent upon what you want to log. Install it as a `PerlPostReadRequestHandler` to capture errors for the entire request, or as a `PerlFixupHandler` for the content-generation phase only.

```
PerlModule Cookbook::DivertErrorLog

PerlModule Cookbook::ErrorsToIRC
PerlPostReadRequestHandler Cookbook::ErrorsToIRC
```

There are really only two steps to consider when using the `Cookbook::DivertErrorLog` API: plugging in the new file to begin collecting errors, and restoring the old value of the `ErrorLog` directive when we are finished. With `ErrorsToIRC.pm`, the `handler()` subroutine calls `set_error_log()`, which replaces the current `ErrorLog` setting with a temporary file produced by `Apache::File->tmpfile()`. If `set_error_log()` returns success, the `send_to_irc()` subroutine is added to the `PerlCleanupHandler` stack using `$r->register_cleanup()`. This restores the `error_log` field of the Apache server

record to the state in which we found it at the end of the request. The restoration of the `error_log` is very important—because the Apache server record has a lifetime greater than a single request, we want to make certain that subsequent requests to the same child use the default `ErrorLog` setting until we specifically choose to override it.

After you have the errors for the request isolated, what you do with them is entirely up to you. In this case, `send_to_irc()` then uses the filehandle stored via the `pnotes()` method to send collected errors to a private IRC channel using `Net::IRC`. As mentioned in Recipe 6.2, `Apache::File->tmpfile()` ensures that the temporary file is indeed removed at the end of the request.

The end results are per-request errors nicely printed to a private IRC channel, enabling support staff to easily monitor the status of the server from just about anywhere. Although what we have described here is rather complex and not really recommended for production sites, it hopefully has managed to introduce some of the more interesting and powerful possibilities that arise when mod_perl is coupled with XS.