APPENDIX A

# mod_perl Recipes

This appendix acts as a mini-cookbook for mod_perl. As we've mentioned many times in this book, the mod_perl mailing list is a terrific resource for anyone working with mod_perl. Many very useful code snippets and neat techniques have been posted to the mod_perl mailing list. In this appendix, we present the techniques that you will find most useful in your day-to-day mod_perl programming.

## Emulating the Authentication Mechanism

You can authenticate users with your own mechanism (instead of the standard one) but still make Apache think that the user was authenticated by the standard mechanism. Set the username with:

```
$r->connection->user('username');
```

Now you can use this information, for example, during logging, so that you can have your "username" passed as if it was transmitted to Apache through HTTP authentication.

## Reusing Data from POST Requests

What happens if you need to access the POSTed data more than once. For example, suppose you need to reuse it in subsequent handlers of the same request? POSTed data comes directly from the socket, and at the low level data can be read from a socket only once. You have to store it to make it available for reuse.

But what do you do with large multipart file uploads? Because POSTed data is not all read in one clump, it's a problem that's not easy to solve in a general way. A transparent way to do this is to switch the request method from POST to GET and store the POST data in the query string. The handler in Example A-1 does exactly that.

*Example A-1. Apache/POST2GET.pm*

```
package Apache::POST2GET;
use Apache::Constants qw(M_GET OK DECLINED);

sub handler {
    my $r = shift;
    return DECLINED unless $r->method eq "POST";
    $r->args(scalar $r->content);
    $r->method('GET');
    $r->method_number(M_GET);
    $r->headers_in->unset('Content-length');
    return OK;
}
1;
```

In *httpd.conf* add:

```
PerlInitHandler Apache::POST2GET
```

or even this:

```
<Limit POST>
    PerlInitHandler Apache::POST2GET
</Limit>
```

to save a few more cycles. This ensures that the handler will be called only for `POST` requests.

Be aware that this will work only if the `POST`ed data doesn't exceed the maximum allowed size for `GET` requests. The default maximum size is 8,190 bytes, but it can be lowered using the `LimitRequestLine` configuration directive.

Effectively, this trick turns the `POST` request into a `GET` request internally. Now when a module such as `CGI.pm` or `Apache::Request` parses the client data, it can do so more than once, since `$r->args` doesn't go away (unless you make it go away by resetting it).

If you are using `Apache::Request`, it solves this problem for you with its `instance( )` class method, which allows `Apache::Request` to be a singleton. This means that whenever you call `Apache::Request->instance( )` within a single request, you always get the same `Apache::Request` object back.

## Redirecting POST Requests

Under mod_cgi, it's not easy to redirect `POST` requests to another location. With mod_perl, however, you can easily redirect `POST` requests. All you have to do is read in the content, set the method to `GET`, populate `args( )` with the content to be forwarded, and finally do the redirect, as shown in Example A-2.

*Example A-2. redirect.pl*

```
use Apache::Constants qw(M_GET);
my $r = shift;
my $content = $r->content;
$r->method("GET");
$r->method_number(M_GET);
$r->headers_in->unset("Content-length");
$r->args($content);
$r->internal_redirect_handler("/new/url");
```

In this example we use `internal_redirect_handler()`, but you can use any other kind of redirect with this technique.

## Redirecting While Maintaining Environment Variables

Let's say you have a module that sets some environment variables. Redirecting most likely tells the web browser to fetch the new page. This makes it a totally new request, so no environment variables are preserved.

However, if you're using `internal_redirect()`, you can make the environment variables visible in the subprocess via `subprocess_env()`. The only nuance is that the `%ENV` keys will be prefixed with `REDIRECT_`. For example, `$ENV{CONTENT_LENGTH}` will become:

```
$r->subprocess_env->{REDIRECT_CONTENT_LENGTH};
```

## Handling Cookies

Unless you use a module such as `CGI::Cookie` or `Apache::Cookie`, you need to handle cookies yourself. Cookies are accessed via the `$ENV{HTTP_COOKIE}` environment variable. You can print the raw cookie string as `$ENV{HTTP_COOKIE}`. Here is a fairly well-known bit of code to take cookie values and put them into a hash:

```
sub get_cookies {
    # cookies are separated by a semicolon and a space, this will
    # split them and return a hash of cookies
    my @rawCookies = split /; /, $ENV{'HTTP_COOKIE'};
    my %cookies;

    foreach (@rawCookies){
        my($key, $val) = split /=/, $_;
        $cookies{$key} = $val;
    }

    return %cookies;
}
```

And here's a slimmer version:

```
sub get_cookies {
    map { split /=/, $_, 2 } split /; /, $ENV{'HTTP_COOKIE'};
}
```

## Sending Multiple Cookies with the mod_perl API

Given that you have prepared your cookies in `@cookies`, the following code will submit all the cookies:

```
for (@cookies) {
    $r->headers_out->add('Set-Cookie' => $_);
}
```

## Sending Cookies in REDIRECT Responses

You should use `err_headers_out()`, not `headers_out()`, when you want to send cookies in a `REDIRECT` response or in any other non-2XX response. The difference between `headers_out()` and `err_headers_out()` is that the latter prints even on error and persists across internal redirects (so the headers printed for `ErrorDocument` handlers will have them). Example A-3 shows a cookie being sent in a `REDIRECT`.

*Example A-3. redirect_cookie.pl*

```
use Apache::Constants qw(REDIRECT OK);
my $r = shift;
# prepare the cookie in $cookie
$r->err_headers_out->add('Set-Cookie' => $cookie);
$r->headers_out->set(Location => $location);
$r->status(REDIRECT);
$r->send_http_header;
return OK;
```

## CGI::params in the mod_perlish Way

Assuming that all your variables are single key-value pairs, you can retrieve request parameters in a way similar to using `CGI::params` with this technique:

```
my $r = shift;  # or $r = Apache->request
my %params = $r->method eq 'POST' ? $r->content : $r->args;
```

Also take a look at `Apache::Request`, which has the same API as `CGI.pm` for extracting and setting request parameters but is significantly faster, since it's implemented in C.

## Sending Email from mod_perl

There is nothing special about sending email from mod_perl, it's just that we do it a lot. There are a few important issues. The most widely used approach is starting a *sendmail* process and piping the headers and the body to it. The problem is that *sendmail* is a very heavy process, and it makes mod_perl processes less efficient.

If you don't want your process to wait until delivery is complete, you can tell *sendmail* not to deliver the email straight away, but to either do it in the background or just queue the job until the next queue run. This can significantly reduce the delay for the mod_perl process, which would otherwise have to wait for the *sendmail* process to complete. You can specify this for all deliveries in *sendmail.cf*, or for individual email messages on each invocation on the *sendmail* command line. Here are the options:

*-odb*
: Deliver in the background

*-odq*
: Queue only

*-odd*
: Queue, and also defer the DNS/NIS lookups

The current trend is to move away from `sendmail` and switch to using lighter mail delivery programs such as `qmail` or `postfix`. You should check the manpage of your favorite mailer application for equivalents to the configuration presented for *sendmail*.

Alternatively, you may want to use `Net::SMTP` to send your mail without calling an extra process. The main disadvantage of using `Net::SMTP` is that it might fail to deliver the mail because the destination peer server might be down. It can also be very slow, in which case the mod_perl application will do nothing while it waits for the mail to be sent.

## mod_rewrite in Perl

mod_rewrite provides virtually any functionality you can think of for manipulating URLs. Because of its highly generalized nature and use of complex regular expressions, it is not easy to use and has a high learning curve.

With the help of `PerlTransHandler`, which is invoked at the beginning of request processing, we can easily implement everything mod_rewrite does in Perl. For example, if we need to perform a redirect based on the query string and URI, we can use the following handler:

```
package Apache::MyRedirect;
use Apache::Constants qw(OK REDIRECT);
use constant DEFAULT_URI => 'http://www.example.org';
```

```
sub handler {
    my $r    = shift;
    my %args = $r->args;
    my $path = $r->uri;

    my $uri = (($args{'uri'}) ? $args{'uri'} : DEFAULT_URI) . $path;

    $r->header_out->add('Location' => $uri);
    $r->status(REDIRECT);
    $r->send_http_header;

    return OK;
}
1;
```

Set it up in *httpd.conf* as:

```
PerlTransHandler Apache::MyRedirect
```

The code consists of four parts: retrieving the request data, deciding what to do based on this data, setting the headers and the status code, and issuing the redirect.

So if a client submits the following request:

```
http://www.example.com/news/?uri=http://www2.example.com/
```

the `$uri` parameter is set to `http://www2.example.com/news/`, and the request will be redirected to that URI.

Let's look at another example. Suppose you want to make this translation before a content handler is invoked:

```
/articles/10/index.html  =>  /articles/index.html?id=10
```

The `TransHandler` shown in Example A-4 will do that for you.

*Example A-4. Book/Trans.pm*

```
package Book::Trans;
use Apache::Constants qw(:common);
sub handler {
    my $r = shift;
    my $uri = $r->uri;
    my($id) = ($uri =~ m|^/articles/(.*?)/|);
    $r->uri("/articles/index.html");
    $r->args("id=$id");
    return DECLINED;
}
1;
```

To configure this handler, add these lines to *httpd.conf*:

```
PerlModule Book::Trans
PerlTransHandler Book::Trans
```

The handler code retrieves the request object and the URI. Then it retrieves the *id*, using the regular expression. Finally, it sets the new value of the URI and the arguments string. The handler returns `DECLINED` so the default Apache `TransHandler` will take care of URI-to-filename remapping.

Notice the technique to set the arguments. By the time the Apache request object has been created, arguments are handled in a separate slot, so you cannot just push them into the original URI. Therefore, the `args( )` method should be used.

## Setting PerlHandler Based on MIME Type

It's very easy to implement a dispatching module based on the MIME type of the request—that is, for different content handlers to be called for different MIME types. Example A-5 shows such a dispatcher.

*Example A-5. Book/MimeTypeDispatch.pm*

```
package Book::MimeTypeDispatch;
use Apache::Constants qw(DECLINED);

my %mime_types = (
    'text/html'  => \&HTML::Template::handler,
    'text/plain' => \&Book::Text::handler,
);

sub handler {
    my $r = shift;
    if (my $h = $mime_types{$r->content_type}) {
        $r->push_handlers(PerlHandler => $h);
        $r->handler('perl-script');
    }
    return DECLINED;
}
1;
__END__
```

This should be done with `PerlFixupHandler`, so we add this line in *httpd.conf*:

```
PerlFixupHandler Book::MimeTypeDispatch
```

After declaring the package name and importing constants, we set a translation table of MIME types and the corresponding handlers to be called. Then comes the handler, where the request object is retrieved. If the request object's MIME type is found in our translation table, we set the handler that should handle this request; otherwise, we do nothing. At the end we return `DECLINED` so another fixup handler can take over.

## Singleton Database Handles

Let's say we have an object we want to be able to access anywhere in the code, without making it a global variable or passing it as an argument to functions. The singleton design pattern helps here. Rather than implementing this pattern from scratch, we will use `Class::Singleton`.

For example, if we have a class `Book::DBIHandle` that returns an instance of the opened database connection handle, we can use it in the `TransHandler` phase's handler (see Example A-6).

*Example A-6. Book/TransHandler.pm*

```
package Book::TransHandler;

use Book::DBIHandle;
use Apache::Constants qw(:common);

sub handler {
    my $r = shift;
    my $dbh = Book::DBIHandle->instance->dbh;
    $dbh->do("show tables");
    # ...
    return OK;
}
1;
```

We can then use the same database handle in the content-generation phase (see Example A-7).

*Example A-7. Book/ContentHandler.pm*

```
package Book::ContentHandler;

use Book::DBIHandle;
use Apache::Constants qw(:common);

sub handler {
    my $r = shift;
    my $dbh = Book::DBIHandle->instance->dbh;
    $dbh->do("select from foo...");
    # ...
    return OK;
}
1;
```

In *httpd.conf*, use the following setup for the `TransHandler` and content-generation phases:

```
PerlTransHandler +Book::TransHandler
<Location /dbihandle>
    SetHandler perl-script
```

```
        PerlHandler +Book::ContentHandler
    </Location>
```

This specifies that `Book::TransHandler` should be used as the `PerlTransHandler`, and `Book::ContentHandler` should be used as a content-generation handler. We use the `+` prefix to preload both modules at server startup, in order to improve memory sharing between the processes (as explained in Chapter 10).

`Book::DBIHandle`, shown in Example A-8, is a simple subclass of `Class::Singleton` that is used by both handlers.

*Example A-8. Book/DBIHandle.pm*

```
package Book::DBIHandle;

use strict;
use warnings;

use DBI;

use Class::Singleton;
@Book::DBIHandle::ISA = qw(Class::Singleton);

sub _new_instance {
    my($class, $args) = @_;

    my $self = DBI->connect($args->{dsn},    $args->{user},
                            $args->{passwd}, $args->{options})
        or die "Cannot connect to database: $DBI::errstr";

    return bless $self, $class;
}

sub dbh {
   my $self = shift;
   return $$self;
}
1;
```

`Book::DBIHandle` inherits the `instance()` method from `Class::Singleton` and overrides its `_new_instance()` method. `_new_instance()` accepts the `connect()` arguments and opens the connection using these arguments. The `_new_instance()` method will be called only the first time the `instance()` method is called.

We have used a reference to a scalar (`$dbh`) for the `Book::DBIHandle` objects. Therefore, we need to dereference the objects when we want to access the database handle in the code. The `dbh()` method does this for us.

Since each child process must have a unique database connection, we initialize the database handle during the `PerlChildInit` phase, similar to `DBI::connect_on_init()`. See Example A-9.

*Example A-9. Book/ChildInitHandler.pm*

```
package Book::ChildInitHandler;

use strict;
use Book::DBIHandle;
use Apache;

sub handler {
    my $s = Apache->server;

    my $dbh = Book::DBIHandle->instance(
        { dsn     => $s->dir_config('DATABASE_DSN'),
          user    => $s->dir_config('DATABASE_USER'),
          passwd  => $s->dir_config('DATABASE_PASSWD'),
          options => {
              AutoCommit => 0,
              RaiseError => 1,
              PrintError => 0,
              ChopBlanks => 1,
          },
        }
    );

    $s->log_error("$$: Book::DBIHandle object allocated, handle=$dbh");
}
1;
```

Here, the `instance( )` method is called for the first time, so its arguments are passed to the new `_new_instance( )` method. `_new_instance( )` initializes the database connection.

*httpd.conf* needs to be adjusted to enable the new `ChildInitHandler`:

```
PerlSetVar DATABASE_DSN    "DBI:mysql:test::localhost"
PerlSetVar DATABASE_USER   "foo"
PerlSetVar DATABASE_PASSWD "bar"

PerlChildInitHandler +Book::ChildInitHandler
```

## Terminating a Child Process on Request Completion

If you want to terminate the child process upon completion of processing the current request, use the `child_terminate( )` method anywhere in the code:

```
$r->child_terminate;
```

Apache won't actually terminate the child until everything it needs to do is done and the connection is closed.

# References

- *mod_perl Developer's Cookbook*, by Geoffrey Young, Paul Lindner, and Randy Kobes (Sams Publishing). Selected chapters and code examples available online from *http://www.modperlcookbook.org/*.
- For more information about signal handling, refer to the *perlipc* manpage
- `GET` and `POST` request methods are explained in section 9 of RFC 2068, "Hypertext Transfer Protocol—HTTP/1.1"
- Cookies
  - RFC 2965 specifies the HTTP State Management Mechanism, which describes three new headers, `Cookie`, `Cookie2`, and `Set-Cookie2`, that carry state information between participating origin servers and user agents
  - The cookie specification can be viewed at *http://home.netscape.com/newsref/std/cookie_spec.html*
  - BCP 44, RFC 2964, "Use of HTTP State Management," is an important adjunct to the cookie specification itself
  - Cookie Central (*http://www.cookiecentral.com/*) is another good resource for information about cookies
- "Design Patterns: Singletons," by Brian D. Foy (*The Perl Review*, Volume 0, Issue 1), available at *http://www.theperlreview.com/*.