Filters in Apache 2.0 - Perl.com

By Geoffrey Young on April 17, 2003 12:00 AM

Not too long ago, despite a relative dearth of free tuits, I decided that I had put off my investigation of mod_perl 2.0 for too long - it was time to really start kicking the tires and tinkering with all the new stuff I had been hearing about. What I found was that the new mod_perl API is full of interesting features, yet discovering and using them was tedious, frustrating, enlightening, and fun all at the same time. Hopefully, I can help ease some of the growing pains you are likely to encounter by experiencing the pain myself first, then sharing some of the lessons through a series of articles. Consider this and future articles to be our voyage together into the rocky but exciting new mod_perl frontier.

One of the more interesting and practical features to come out of the Apache 2.0 redesign effort is output filters. While in Apache 2.0 there are all kinds of filters, including input and connection filters, it's output filters that are most interesting to me - mostly because 2.0 discussions make a point of saying that it's impossible (well, really, really hard) to filter output content in Apache 1.3, despite the fact that mod_perl users have been able filter content (to some degree) for years. Thus, when I began to play around with mod_perl 2.0 it seemed only logical that my first task would be to port the instructional yet useful `Apache::Clean`, a content filter for mod_perl 1.0, over to the new architecture.

What we will be examining here is a preliminary implementation of `Apache::Clean` using the mod_perl 2.0 API. Because mod_perl 2.0 is still being tweaked daily, if you want to follow along on your own box, then you would need the current version of mod_perl from CVS, or a recent snapshot - the latest versions shipped with Linux distributions like RedHat, or even the latest version on CPAN (1.99_08), are far too out of date for what we will be doing. The most current version of Apache 2.0, as well as Perl 5.8.0, will also be helpful. Keep in mind that many of the more interesting features in mod_perl 2.0 are not entirely stable yet, so do not be surprised if things work just a bit differently six months from now.

What Are Output Filters Anyway?

Go ahead, admit it. At some point, you wrote a CGI script that generated HTML with embedded Server Side Include tags. The impetus behind the idea was a simple one: You had hopes that the embedded SSI tags would save you from the extra work of, say, adding a canned footer to the bottom of your otherwise dynamic page. Sounds reasonable, right? Seeing those SSI tags left unprocessed in the resulting page must have been shocking.

As it turns out, whether you knew it or not, in Apache-speak you were trying to filter your content, or pass the output of one process (the CGI script) into another (Apache's SSI engine) for subsequent processing. Content filtering is a simple idea, and one that feels natural to us as programmers. After all, Apache is supposed to be modular, and piping modular components together - `cat yachts.txt | wc -l` - is something we do on the Unix command line all the time. Wanting the same functionality in our Web server

of choice seems not only logical, but almost required in the interests of efficient application programming.

While the idea is certainly sound, the above experiment exposes a limitation of the Apache 1.3 server itself, namely that by design you cannot have more than one content handler for a given request - you can use either mod_cgi to process and CGI script, or mod_include to parse an SSI document, but not both.

With Apache 2.0, the idea of output filters were introduced, which provide an official way to intercept and manipulate data on its way from the content handler to the browser. In the case of our SSI example, mod_include has been implemented as an output filter in Apache 2.0, giving it the ability to post-process either static files (served by the default Apache content handler) or dynamically generated scripts (such as those generated by mod_cgi, mod_perl, or mod_php). True to its goal of exposing the entire Apache API to Perl, mod_perl allows you to plug into the Apache filter API and create your own output filters in Perl, which is what we will be doing with `Apache::Clean`.

`HTML::Clean` and `Apache::Clean`

Let's take a moment to look at `HTML::Clean` before delving into `Apache::Clean`, which is basically just a mod_perl wrapper that takes `HTML::Clean` and turns it into an output filter. `HTML::Clean` is a nifty little module that reduces the size of an HTML page using a number of different but simple techniques, such as removing unnecessary white space, replacing longer HTML tags with shorter equivalents, and so on. The end result is a page that, while still valid HTML and easily rendered by a browser, is relatively compact. If reducing bandwidth is important in your environment, then using `HTML::Clean` to tidy up static pages offline is a quick and easy way to save some bytes.

Here is a simple example of `HTML::Clean` in action.

```
use HTML::Clean ();

use strict;

my $dirty = q!<strong>&quot;helm's alee&quot;</strong>!;

my $h = HTML::Clean->new(\$dirty);

$h->strip({ shortertags => 1, entities => 1 });

print ${$h->data};
```

As you can see, the interface for `HTML::Clean` is object-oriented and fairly straightforward. Things begin

by calling the `new()` constructor to create an `HTML::Clean` object. `new()` accepts either a filename to clean or a reference to a string containing some HTML. Deciding exactly which aspects of the HTML to tidy is determined in one of two ways: either using the `level()` method to set an optimization level, or by passing the `strip()` method any number of options from a rich set. In either case, `strip()` is used to actually clean the HTML. After that, calling the `data()` method returns a reference to a string containing the HTML, polished to a Perly white. In our sample code, the original HTML has been changed to

```
<b>"helm's alee"</b>
```

which is half the size of our original string yet displayed the same way by browsers.

Depending on the size of your site, using `HTML::Clean` can lead to a significant reduction in the number of bytes sent over the wire - for instance, the front page of the current `mod_perl project homepage` becomes 70% of it's original size when scrubbed with `$h->level(9)`. However, while spending the time to tidy static HTML might make sense, the number of static pages on any given site seems to be diminishing daily. What about dynamically generated HTML?

One way to handle dynamic HTML would be to add `HTML::Clean` routines to each dynamic component of your application, a process that really is neither scalable nor maintainable. A better solution would be to have Apache inject `HTML::Clean` processing directly into the server response wherever we wanted it, to create a pluggable module that we could configure to post-process requests to any given URI. Enter `Apache::Clean`.

`Apache::Clean` provides a basic interface into `HTML::Clean` but it works as an output filter. As briefly mentioned, `Apache::Clean` already exists for mod_perl 1.0, but over in Apache 1.3 land it was limited in that it could only post-process responses generated by mod_perl, and that only after sufficient magic. We are not going to get into how that all worked in mod_perl 1.0 - for a detailed explanation see Recipe 15.4 in the *mod_perl Developer's Cookbook* or the original `Apache::Clean` manpage. With Apache 2.0 and the advent of output filters, we can now code `Apache::Clean` as a genuine part of Apache's request processing, allowing us to clean responses on their way to the browser entirely independent of who generates the content.

New Directives

Here is a look at a possible configuration for Apache 2.0, one that takes output of a CGI script, post-processes it for SSI tags, then cleans it with our `Apache::Clean` output filter.

```
Alias /cgi-bin /usr/local/apache2/cgi-bin
```

```
<Location /cgi-bin>
  SetHandler cgi-script

  SetOutputFilter INCLUDES
  PerlOutputFilterHandler Apache::Clean

  PerlSetVar CleanOption shortertags
  PerlAddVar CleanOption whitespace

  Options +ExecCGI +Includes
</Location>
```

As with Apache 1.3, mod_cgi is still enabled the same way - in our case via the `SetHandler cgi-script` directive, although this is not the only way and the familiar `ScriptAlias` directive is still supported. What is different in this `httpd.conf` snippet is the configuration of the SSI engine, mod_include. As already mentioned, mod_include was implemented as an output filter in Apache 2.0, and output filters bring with them a new directive. The `SetOutputFilter` directive activates the SSI engine - the `INCLUDES` filter - within our container. This means that requests to `cgi-bin/`, no matter who handles the actual generation of content, will be parsed by mod_include. See the mod_include documentation for other possible SSI configurations and options.

With the generic Apache bits out of the way, we can move on to the mod_perl part, which isn't all that complex. While the `PerlSetVar` and `PerlAddVar` directives are exactly the same as they were in mod_perl 1.0, mod_perl 2.0 introduces a new directive - `PerlOutputFilterHandler` - which specifies the Perl output filter for the request. In our sample `httpd.conf`, the `Apache::Clean` output filter will be added after mod_include, which inserts SSI processing after mod_cgi. The really cool part about filters is that everything happens without any tricks or magic - getting all these independent modules to work in harmony in creating the server response is all perfectly normal, which is a huge improvement over Apache 1.3.

In the interests of safety, one thing that you should note about our sample configuration is that it does not include the `entities` option. Because we're cleaning dynamic content, reducing entity tags (such as changing `&quot;` to `"`) would inadvertently remove any protection against Cross Site Scripting introduced by the generating script. For more information about Cross Site Scripting and how to protect against it, a good overview is provided in this `perl.com` article.

Introducing mod_perl 2.0

mod_perl actually offers two different APIs for coding the Perl output filter. We are going to be using the simpler, streaming API, which hides the raw Apache API a bit. Of course, if you are feeling bold and want

to manipulate the Apache bucket brigades directly, you are more than welcome, but it is a more complex process so we are not going to talk about it here. Instead, here is our new `Apache::Clean` handler, ported to mod_perl 2.0 using the streaming filter API.

```
package Apache::Clean;

use 5.008;

use Apache::Filter ();      # $f
use Apache::RequestRec ();  # $r
use Apache::RequestUtil (); # $r->dir_config()
use Apache::Log ();         # $log->info()
use APR::Table ();          # dir_config->get() and headers_out->get()

use Apache::Const -compile => qw(OK DECLINED);

use HTML::Clean ();

use strict;

sub handler {

  my $f   = shift;

  my $r   = $f->r;

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

  # we only process HTML documents
  unless ($r->content_type =~ m!text/html!i) {
    $log->info('skipping request to ', $r->uri, ' (not an HTML document)');

    return Apache::DECLINED;
  }

  my $context;

  unless ($f->ctx) {
    # these are things we only want to do once no matter how
    # many times our filter is invoked per request
```

```
  # parse the configuration options
  my $level = $r->dir_config->get('CleanLevel') || 1;

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

  # store the configuration
  $context = { level   => $level,
               options => \%options,
               extra   => undef };

  # output filters that alter content are responsible for removing
  # the Content-Length header, but we only need to do this once.
  $r->headers_out->unset('Content-Length');
}

# retrieve the filter context, which was set up on the first invocation
$context ||= $f->ctx;

# now, filter the content
while ($f->read(my $buffer, 1024)) {

  # prepend any tags leftover from the last buffer or invocation
  $buffer = $context->{extra} . $buffer if $context->{extra};

  # if our buffer ends in a split tag ('<strong' for example)
  # save processing the tag for later
  if (($context->{extra}) = $buffer =~ m/(<[^>]*)$/) {
    $buffer = substr($buffer, 0, - length($context->{extra}));
  }

  my $h = HTML::Clean->new(\$buffer);

  $h->level($context->{level});

  $h->strip($context->{options});

  $f->print(${$h->data});
}

if ($f->seen_eos) {
```

```
    # we've seen the end of the data stream

    # print any leftover data
    $f->print($context->{extra}) if $context->{extra};
  }
  else {
    # there's more data to come

    # store the filter context, including any leftover data
    # in the 'extra' key
    $f->ctx($context);
  }

  return Apache::OK;
}

1;
```

If you can dismiss the mod_perl specific bits for a moment, you will see the `HTML::Clean` logic embedded in the middle of the handler, which is not very different from the isolated code we used to illustrate `HTML::Clean` by itself. One of the things we need to do differently, however, is determine which options to pass to the `level()` and `options()` methods of `HTML::Clean`. Here, we use `$r->dir_config()` to gather whatever `httpd.conf` options we specified through our `PerlSetVar` and `PerlAddVar` configurations.

```
my $level = $r->dir_config->get('CleanLevel') || 1;

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

This use of `dir_config()` is in fact no different than how we would have coded it in mod_perl 1.0. Similarly, later methods like `r->content_type()`, `$r->server->log->info()`, and `$r->uri()` also behave the same as they did in mod_perl 1.0, which should offer some degree of comfort. For instance, the block

```
unless ($r->content_type =~ m!text/html!i) {
  $log->info('skipping request to ', $r->uri, ' (not an HTML document)');
```

```
  return Apache::DECLINED;
}
```

looks almost exactly the same as it would have been in mod_perl 1.0, save the use of `Apache::DECLINED`. The new `Apache::Const` class provides access to all constants you will need in your handlers, albeit through a slightly different interface than before - when using the `-compile` option, constants are imported into the `Apache::` namespace. If you want the constants in your own namespace, mimicking the `OK` of yore, you can just `use Apache::Const` by itself without the without the `-compile` option.

Some of the other minor differences you will notice are the addition of a bunch of `use` statements at the top of the handler. Whereas with mod_perl 1.0 just about every class was magically present when you needed it, with mod_perl 2.0 you need to be very specific about the classes you will be using in your handler, and almost nothing is available by default.

In general, the most important class is `Apache::RequestRec`, which provides access to all the elements in the Apache C `request_rec` structure. Methods originating from the request object, `$r`, but not operating on the actual `request_rec` slots, such as `$r->dir_config()`, are defined in `Apache::RequestUtil`. This is a nice separation, and can help you think about mod_perl more in terms of access to the underlying Apache guts than just a box of black magic.

If you recall from 1.0, `$r->dir_config()` returned an `Apache::Table` object, which corresponded to an Apache table and allowed things like headers to be stored in a case-insensitive, multi-keyed manner. In 2.0, Apache tables are accessed through the APR (Apache Portable Runtime) layer, so any API that accesses tables needs to `use APR::Table`. This includes the `get()` and `set()` methods used on tables like `headers_out` and `dir_config`.

Besides `Apache::RequestRec`, `Apache::RequestUtil`, and `APR::Table`, our handler also needs access to the `Apache::Log` and `Apache::Filter` classes. `Apache::Log` works no differently than it did under mod_perl 1.0, while `Apache::Filter` is entirely new and will be discussed shortly.

From experience, I can tell you that determining which module you need to `use` in order to access the functionality you require is maddening. In the old days (just weeks before this article was written) developers needed to plow through code examples in the mod_perl test suite in order to discern which modules they needed. But no more. Recently introduced was the `ModPerl::MethodLookup` package, which contains the `lookup_method()` function - just pass it the name of the method you are looking for and you will get back a list of modules likely to suit your needs. See the `MethodLookup` documentation for more details.

With basic housekeeping out of the way, we can focus on the guts of our output filter and the

`Apache::Filter` streaming API. You will notice that the first argument passed to the `handler()` subroutine is an `Apache::Filter` object, not the typical `$r` you might have been expecting. mod_perl 2.0 has stepped up it's DWIM factor a bit in an attempt to make writing filters a bit more intuitive - in fact, it is possible to write an output filter without ever needing to access `$r`, so mod_perl gives you what you will primarily need. In order to access `$r` we call the (aptly named) `r()` method on our `$f`, then use `$r` as the gateway to per-request attributes, such as the MIME type of the response. Note that our filter genuinely declines to process the request if the content is not HTML. No fancy footwork, just processing like it should be.

Beyond the typical handler initializations is where things really start to get unfamiliar, starting with the notion of filter context, or `$f->ctx()`. Unlike with mod_perl 1.0, where every handler is called only once per request, output filters can be (and generally are) called multiple times per request. There can be several reasons for this, but for our purposes it is sufficient to understand that we need to adjust our normal handler logic and compensate for some of the subtle behaviors that can arise by being called several times.

So, the first thing we do is isolate parts of the request that only need to happen once per request. `$f->ctx()`, which stores the filter context, will return `undef` on the first request, so we can use it as an indicator of the initial invocation of our filter. Since we only really need to parse our `httpd.conf` configuration once, we use the initial invocation to get our `PerlSetVar` options and store them in a hash for later - because `$f->ctx()` can store a scalar for us, we store our hash as a reference in `$context`. We also set aside space in the hash for the `extra` element, which will become important later.

Another thing we need to do only once per request is to remove the `Content-Length` header from the response. Apache 2.0 has taken great steps to make sure that all requests are as RFC compliant as possible, while at the same time trying to make the developer's life easier. As it turns out, part of this was the addition of the Content-Length filter, which calculates the length of the response and adds the `Content-Length` header if Apache deems it appropriate. If you are writing a handler that alters content to the point where the length of the response is different (which is probably true in most cases) you are responsible for removing the `Content-Length` header from the outgoing headers table. A simple call to `$r->headers_out->unset()` is all we need to accomplish this, which is again the same as it would have been in mod_perl 1.0. And don't worry, if the `Content-Length` is missing Apache takes other steps, such as using a chunked transfer encoding, to ensure that the request is HTTP compliant.

That about wraps up all the processing which should only happen once per request. If you do not like seeing the informational `"skipping..."` message on every non-HTML filter invocation, feel free to add logic that tests against `$f->ctx()` there as well.

Once we have taken care of one-time-only processing, we can move on to the heart of our output filter. The actual `Apache::Filter` streaming API is fairly straight forward. For the most part, we simply call `$f->read()` to read incoming data in chunks, in our case `1K` at a time. Sending the processed data down the

line requires only that we call `$f->print()`. All in all, the basis of the streaming API couldn't be any simpler. Where it begins to get complex stems from the nature of our particular filter.

The idea behind `HTML::Clean` is that it can, in part, make HTML more compact. However, since HTML is tag based, and those tags often come in pairs, we need to take special steps to make sure that our tags remain balanced after `Apache::Clean` has run. Because we are reading and processing data in chunks, there is the possibility that a tag might be stranded between chunks. For instance, if the HTML looked like

```
[1019 bytes of stuff] <strong>Bold!</strong> [more stuff]
```

the first chunk of data that `Apache::Clean` would see is

```
[1019 bytes of stuff] <str
```

Because `<str` is not a valid HTML tag, `HTML::Clean` leaves it unaltered. When the next chunk of data is read from the filter, it comes across as

```
ong>Bold!</strong> [more stuff]
```

and `HTML::Clean` again leaves the unrecognized `ong>` unprocessed. However, it does catch the closing `</strong>` tag. The end result, as you can probably see now, would be

```
[1020 bytes of stuff] <strong>Bold!</b> [more stuff]
```

which is definitely undesirable. Our matching regex and `extra` manipulations make certain that any dangling tags are prepended to the front of the next buffer, safeguarding against this particular problem. Of course, this kind of logic is not required of all filters. Just remember to keep in mind the complexity that operating on data in pieces adds when you implement your own filter.

Once we are finished processing all the data from the current filter invocation we come to a critical junction - determining whether we have seen all of the data from the actual response. If our filter will not be called again for this request, `$f->seen_eos()` will return true, meaning that we have reached the end of the

data stream. Because we may have leftover tag fragments stored in `$context->{extra}`, we need to send those along before exiting our filter. On the other hand, if there is more data to come, then we would need to store the current filter context so it can be used on the next invocation.

Voila!

So there you have it, output filtering made easy with mod_perl 2.0. All in all, it is a bit different than what you might be used to with mod_perl 1.0, but it's not that difficult once you get your head around it. And it does allow for some pretty amazing things. For instance, not only can we now use Perl to code interesting handlers like `Apache::Clean`, but the overall filter mechanism makes it possible to use Perl to manipulate *any and all* content originating from the server - just a simple line like

```
PerlOutputFilterHandler My::Cool::Stuff
```

on the server level of your `httpd.conf` (such as next to the `DocumentRoot` directive) will allow you to post-process *every* request. Cool.

Of course, that's not the end of the story, and what's left over has both positive and negative sides. What we've seen here is really just scratching the surface of filters: There are still input and connection filters to talk about, as well as raw access to the Apache filtering API via bucket brigades. The downside is that constructing a filter that handles conditional `GET` requests properly isn't really as good as it could be due to the (current) incompleteness of the mod_perl 2.0 filtering API. However, these things aside, what we've accomplished here is already pretty impressive, and I hope it gets your creativity flowing and you start tinkering with the very cool new world of mod_perl 2.0.

If you want to try the code from this article, then it is available as a distribution from my Web site. Other good sources of information are the (growing) mod_perl 2.0 docs, particularly the section on filtering, which has been (very) recently updated with the latest information.

Stay Tuned ...

How did I know that my code here actually worked and did what I expected it to? I wrote a test suite for `Clean.pm` using the new `Apache::Test` framework and put my code against a live Apache server under every situation I could think of. `Apache::Test` is probably the single best thing to come out of the mod_perl 2.0 redesign effort, and what I will share with you next time.

Thanks

Many thanks to Stas Bekman, who reviewed this article, even though it meant having to deal with both my questions, suggestions and API gripes.