Finally, to activate the `Cookbook::EmailUploads` handler, install the module in your mod_perl Perl library directory, add the following directives to your `httpd.conf`, and restart your server.

```
PerlModule Cookbook::Mail
PerlModule Cookbook::EmailUploads

<Location /email-uploads>
  SetHandler perl-script
  PerlHandler Cookbook::EmailUploads
</Location>
```

15.4. Filtered Content Generation

You want to filter the output of one `PerlHandler` into another, allowing each handler to process the output of the previous handler before sending content to the client.

Technique

Use the `Apache::Filter` module, available from CPAN.

```
PerlModule Apache::Filter
PerlModule Apache::Compress

PerlModule Cookbook::Clean

Alias /clean /usr/local/apache/htdocs
<Location /clean>
  SetHandler perl-script
  PerlHandler Cookbook::Clean Apache::Compress
  PerlSetVar  Filter On
</Location>
```

Comments

One of the classic problems of the Apache 1.3 architecture is that passing the output of one content handler to another is impossible. A good example is the inability to use mod_cgi to output HTML with embedded SSI tags for mod_include to process. Although recent advances in the Apache 2.0 architecture have opened up this ability to the rest of the Apache programming world, for mod_perl developers this ability has been available for years.

Through the magic of Perl's TIEHANDLE interface and some third-party CPAN modules, mod_perl can offer filtered content generation—the ability for any number of stacked PerlHandlers to read data from a previous handler, process it, and pass the new data to another PerlHandler on the stack. Historically, there have been a few different approaches to output filtering in the mod_perl community, but Apache::Filter is the only implementation that is actively maintained, and as such it has become the standard.

As explained in more detail in the next recipe, mod_perl tie()s STDOUT to the Apache class, which steals away calls to print() to do some custom processing before passing the data to Apache's output routines. Recipe 6.10 showed that it is possible to re-tie() STDOUT to a class other than Apache in order to redirect output to a variable. Apache::Filter does something similar; by implementing a complete TIEHANDLE interface, as well as some other acrobatics, it stores away content generated by one PerlHandler and makes it available to the next handler in the PerlHandler stack. When the last PerlHandler in the chain is run, Apache::Filter tie()s STDOUT back to the Apache class and the final output is sent to the browser.

The capability to chain together content handlers is an incredibly powerful technique. Not only does it provide a functionality that has long been coveted by the C module world, but it also makes it possible to modularize PerlHandlers into separately maintainable components that can be swapped in and out of your configuration at will. As an illustration, we can modify the Cookbook::Clean module from Recipe 7.10 to use Apache::Filter and show how few changes are needed to accommodate Apache::Filter.

```
sub handler {

  my $r = shift;

  return DECLINED unless $r->content_type eq 'text/html';

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

  my $fh = undef;

  if (lc $r->dir_config('Filter') eq 'on') {
    # Register ourselves with Apache::Filter so
    # later filters can see our output.
    $r = $r->filter_register;

    # Get any output from previous filters in the chain.
```

```
    ($fh, my $status) = $r->filter_input;
    return $status unless $status == OK
  }
  else {
    # We are not part of a filter chain, so just process as normal.
    $fh = Apache::File->new($r->filename);
    return DECLINED unless $fh;
  }

  # Slurp the file.
  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});

  $h->strip($cfg->{_options});

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

  return OK;
}
```

As you can see, the effort that's required for each individual `PerlHandler` in the output chain is minimal. `Apache::Filter` really only requires one change from the way you would ordinarily program a content handler. To start the process, the handler needs to register itself with `Apache::Filter` so it knows how many filters are in the `PerlHandler` chain and can do its behind-the-scenes magic. This is done by calling `$r->register_filter()`, which returns a new request object `tie()`d to `Apache::Filter` instead of the `Apache` class. `register_filter()` is *added* to the `Apache` class directly instead of via the traditional subclassing mechanism, which is unusual. However, don't let the details of the implementation bog you down too much: Remember that `Apache::Filter` is trying to make the hard things easy, but doing so requires a fair amount of wizardry. After `$r` is redefined as an `Apache::Filter` object, all handler output normally sent to the browser is diverted to the next `PerlHandler` instead.

Unless a `PerlHandler` is designed to be the first content handler in the chain, it will want to read the data from the prior `PerlHandler`, process it, and pass the new data

down the chain. Reading data from the previous `PerlHandler` is accomplished by the `filter_input()` method, also added to the `Apache` class, which returns a filehandle and a status. Most of the time, if the status is other than `OK`, you would want to propagate the status back to Apache, but there might be circumstances when your handler does not care that there is no content to manipulate. The choice of whether to return the `Apache::Filter` status value depends on what you are trying to accomplish.

The returned filehandle, tied to the `Apache::Filter` class, can be treated in exactly the same way as any other filehandle. In fact, if the call to `$r->filter_input()` is made from the first handler in the chain, `Apache::Filter` opens the filehandle on `$r->filename`, making it interchangeable with a call to `Apache::File->new($r->filename)`. This works to our advantage, allowing modules to be written in such a way to work both in and out of the `Apache::Filter` framework, which is what our modified `Cookbook::Clean` class has done. It also allows each handler to retrieve input and send output in exactly the same manner, regardless of the handler's position in the chain.

The determining factor for whether our code chooses to use the filehandle from `Apache::Filter` or `Apache::File` is the presence of `PerlSetVar Filter On` in our configuration. Although you can use whichever trigger you want for your own `PerlHandlers`, `Filter On` is a convention used by the `Apache::Filter` "aware" modules on CPAN. Following this convention allows our handler to be used alone or in conjunction with the other filtering modules on CPAN to creative and powerful ends. The result is that `Cookbook::Clean` can now be used as a standalone `PerlHandler`, which simply cleans up individual requests, or as any part of an `Apache::Filter` chain.

To demonstrate the power of this approach, we can couple our new `Cookbook::Clean` code with `Apache::Compress`. `Apache::Compress` is available from CPAN and also is capable of being used within the `Apache::Filter` framework. To activate `Cookbook::Clean` and `Apache::Compress`, we use the solution configuration at the start of this recipe. When a request is received `Cookbook::Clean` runs first, obtains the requested resource via `$r->filter_input`, and cleans the HTML. Instead of passing its data to the browser, however, `Cookbook::Clean` hands it off to `Apache::Compress` for further processing. `Apache::Compress` checks for the `Accept-Encoding: gzip` header and, if present, runs the input it receives through a `Compress::Zlib` routine to compress the content using the `gzip` encoding scheme. The results would ordinarily be sent to the next `PerlHandler` in the chain, but because `Apache::Compress` is the final configured `PerlHandler`, `STDOUT` has been magically restored and the browser receives the clean, compressed output.

The interesting point to note about this entire setup is that, with the exception of the `filter_register()` and `filter_input()` logic, the process is practically transparent.

`Apache::Filter` intercepts calls to `$r->print`, `send_fd()`, `send_http_header()`, and just about everything you can do to `STDOUT`, all of which simplify the process significantly for module developers.

We chose to retrofit `Cookbook::Clean` to be `Apache::Filter` aware for a specific reason—the combination of a module like `Cookbook::Clean` coupled with the compression available via `Apache::Compress` can be an extremely powerful combination. In simple illustration of benefits of chaining together these two modules, we ran the English version of the Apache installation test page, `index.html.en`, through a few variations of `Cookbook::Clean` and `Apache::Compress`, as shown in the following table.

PerlHandler Combination	Total Bytes Sent	% Reduction
`default-handler`	1310	-
`Cookbook::Clean`	1177	10%
`Apache::Compress`	751	42%
`Cookbook::Clean Apache::Compress`	668	49%

Even though that additional 7% reduction in using `Apache::Compress` only over coupling it with `Cookbook::Clean` might not seem like much, for some people *any* ability to reduce bandwidth transfer is a benefit, especially for where bandwidth is not as plentiful as in the United States.

If this particular combination of filters does not interest you, there are many other applications of filtered content generation, including joining `Apache::RegistryFilter` (distributed with `Apache::Filter`) with `Apache::SSI` to overcome the classic problem we mentioned at the start of this recipe. In all, though, it should be clear to see how using `Apache::Filter` allows for a highly maintainable, modular application model.

15.5. Preventing Cross-Site Scripting Attacks

You want to protect your Web site from malicious user generated content, such as hacked input fields and cross-site scripting attacks.

Technique

First, verify all user inputs from form fields, URLs, or query strings. Enforce this checking by enabling mod_perl's `PerlTaintCheck` option in your `httpd.conf`. Then,