

1 Input and Output Filters

1.1 Description

This chapter discusses `mod_perl`'s input and output filter handlers.

If all you need is to lookup the filtering API proceed directly to the `Apache2::Filter` and `Apache2::FilterRec` manpages.

1.2 Introducing Filters

You certainly already know how filters work, because you encounter filters so often in real life. If you are unfortunate to live in smog-filled cities like Saigon or Bangkok you are probably used to wear a dust filter mask:



If you are smoker, chances are that you smoke cigarettes with filters:



If you are a coffee gourmand, you have certainly tried a filter coffee:



The shower that you use, may have a water filter:



When the sun is too bright, you protect your eyes by wearing sun goggles with UV filter:



If are a photographer you can't go a step without using filter lenses:



If you love music, you might be unaware of it, but your super-modern audio system is literally loaded with various electronic filters:



There are many more places in our lives where filters are used. The purpose of all filters is to apply some transformation to what's coming into the filter, letting something different out of the filter. Certainly in some cases it's possible to modify the source itself, but that makes things unflexible, and but most of the time we have no control over the source. The advantage of using filters to modify something is that they can be replaced when requirements change. Filters also can be stacked, which allows us to make each filter do simple transformations. For example by combining several different filters, we can apply multiple transformations. In certain situations combining several filters of the same kind let's us achieve a better quality output.

The `mod_perl` filters are not any different, they receive some data, modify it and send it out. In the case of filtering the output of the response handler, we could certainly change the response handler's logic to do something different, since we control the response handler. But this may make the code unnecessary complex. If we can apply transformations to the response handler's output, it certainly gives us more flexibility and simplifies things. For example if a response needs to be compressed before sent out, it'd be very inconvenient and inefficient to code in the response handler itself. Using a filter for that purpose is a perfect solution. Similarly, in certain cases, using an input filter to transform the incoming request data is the most wise solution. Think of the same example of having the incoming data coming compressed.

Just like with real life filters, you can pipe several filters to modify each other's output. You can also customize a selection of different filters at run time.

Without much further ado, let's write a simple but useful obfuscation filter for our HTML documents.

We are going to use a very simple obfuscation -- turn an HTML document into a one liner, which will make it harder to read its source without a special processing. To accomplish that we are going to remove characters `\012 (\n)` and `\015 (\r)`, which depending on the platform alone or as a combination represent the end of line and a carriage return.

And here is the filter handler code:

```
#file:MyApache2/FilterObfuscate.pm
#-----
package MyApache2::FilterObfuscate;

use strict;
use warnings;
```

```

use Apache2::Filter ();
use Apache2::RequestRec ();
use APR::Table ();

use Apache2::Const -compile => qw(OK);

use constant BUFF_LEN => 1024;

sub handler {
    my $f = shift;

    unless ($f->ctx) {
        $f->r->headers_out->unset('Content-Length');
        $f->ctx(1);
    }

    while ($f->read(my $buffer, BUFF_LEN)) {
        $buffer =~ s/[\r\n]//g;
        $f->print($buffer);
    }

    return Apache2::Const::OK;
}
1;

```

The directives below configure Apache to apply the `MyApache2::FilterObfuscate` filter to all requests that get mapped to files with an `.html` extension:

```

<Files ~ "\.html">
    PerlOutputFilterHandler MyApache2::FilterObfuscate
</Files>

```

Filters are expected to return `Apache2::Const::OK` or `Apache2::Const::DECLINED`. But instead of receiving `$r` (the request object) as the first argument, they receive `$f` (the filter object). The filter object is described later in this chapter.

The filter starts by unsetting the `Content-Length` response header, because it modifies the length of the response body (shrinks it). If the response handler sets the `Content-Length` header and the filter doesn't unset it, the client may have problems receiving the response since it will expect more data than it was sent. *Setting the Content-Length Header* below describes how to set the `Content-Length` header if you need to.

The core of this filter is a read-modify-print expression in a while loop. The logic is very simple: read at most `BUFF_LEN` characters of data into `$buffer`, apply the regex to remove any occurrences of `\n` and `\r` in it, and print the resulting data out. The input data may come from a response handler, or from an upstream filter. The output data goes to the next filter in the output chain. Even though in this example we haven't configured any more filters, internally Apache itself uses several core filters to manipulate the data and send it out to the client.

As we are going to explain in detail in the following sections, the same filter may be called many times during a single request, every time receiving a subsequent chunk of data. For example if the POSTed request data is 64k long, an input filter could be invoked 8 times, each time receiving 8k of data. The same may happen during the response phase, where an upstream filter may split 64k of output in 8, 8k chunks. The while loop that we just saw is going to read each of these 8k in 8 calls, since it requests 1k on every `read()` call.

Since it's enough to unset the `Content-Length` header when the filter is called the first time, we need to have some flag telling us whether we have done the job. The method `ctx()` provides this functionality:

```
unless ($f->ctx) {
    $f->r->headers_out->unset('Content-Length');
    $f->ctx(1);
}
```

The `unset()` call will be made only on the first filter call for each request. You can store any kind of a Perl data structure in `$f->ctx` and retrieve it later in subsequent filter invocations of the same request. There are several examples using this method in the following sections.

To be truly useful, the `MyApache2::FilterObfuscate` filter logic should take into account situations where removing new line characters will make the document render incorrectly in the browser. As we mentioned above, this is the case if there are multi-line `<pre>...</pre>` entries. Since this increases the complexity of the filter, we will disregard this requirement for now.

A positive side-effect of this obfuscation algorithm is that it reduces the amount of the data sent to the client. The `Apache::Clean` module, available from the CPAN, provides a production-ready implementation of this technique which takes into account the HTML markup specifics.

`mod_perl` I/O filtering follows the Perl principle of making simple things easy and difficult things possible. You have seen that it's trivial to write simple filters. As you read through this chapter you will see that much more difficult things are possible, and that the code is more elaborate.

1.3 I/O Filtering Concepts

Before introducing the APIs, `mod_perl` provides for Apache Filtering, there are several important concepts to understand.

1.3.1 Two Methods for Manipulating Data

`mod_perl` provides two interfaces to filtering: a direct bucket brigades manipulation interface and a simpler, stream-oriented interface. Apache 2.0 considers all incoming and outgoing data as chunks of information, disregarding their kind and source or storage methods. These data chunks are stored in *buckets*, which form bucket brigades. Input and output filters massage the data in these *bucket brigades*. Response and protocol handlers also receive and send data using bucket brigades, though in most cases this is hidden behind wrappers, such as `read()` and `print()`.

mod_perl 2.0 filters can directly manipulate the bucket brigades or use the simplified streaming interface where the filter object acts similar to a filehandle, which can be read from and printed to.

Even though you don't use bucket brigades directly when you use the streaming filter interface (which works on bucket brigades behind the scenes), it's still important to understand bucket brigades. For example you need to know that an output filter will be invoked as many times as the number of bucket brigades sent from an upstream filter or a content handler. Or you need to know that the end of stream indicator (EOS) is sometimes sent in a separate bucket brigade, so it shouldn't be a surprise that the filter was invoked even though no real data went through. As we delve into the filter details you will see that understanding bucket brigades, will help to understand how filters work.

Moreover you will need to understand bucket brigades if you plan to implement protocol modules.

1.3.2 HTTP Request Versus Connection Filters

HTTP request filters are applied when Apache serves an HTTP request. HTTP request input filters get invoked on the body of the HTTP request only if the body is consumed by the content handler. HTTP request headers are not passed through the HTTP request input filters. HTTP response output filters get invoked on the body of the HTTP response if the content handler has generated one. HTTP response headers are not passed through the HTTP response output filters.

It is also possible to apply filters at the connection level. A connection may be configured to serve one or more HTTP requests, or handle other protocols. Connection filters see all the incoming and outgoing data. If an HTTP request is served, connection filters can modify the HTTP headers and the body of request and response. If a different protocol is served over the connection (e.g., IMAP), the data could have a completely different pattern than the HTTP protocol (headers + body). Thus, the only difference between connection filters and request filters is that connection filters see everything from the request, i.e., the headers and the body, whereas request filters see only the body.

mod_perl 2.0 may support several other Apache filter types in the future.

1.3.3 Multiple Invocations of Filter Handlers

Unlike other Apache handlers, filter handlers may get invoked more than once during the same request. Filters get invoked as many times as the number of bucket brigades sent from an upstream filter or a content provider.

For example, a content generation handler may send a string, then force a flush, and then send more data:

```
# assuming buffered STDOUT ($|==0)
$r->print("foo");
$r->rflush;
$r->print("bar");
```

In this case, Apache will generate one bucket brigade with two buckets. There are several types of buckets which contain data; in this example, the data type is *transient*:

```

bucket type      data
-----
1st   transient  foo
2nd   flush

```

Apache sends this bucket brigade to the filter chain. Then, assuming no more data is sent after `print("bar")`, it will create a last bucket brigade, with one bucket, containing data:

```

bucket type      data
-----
1st   transient  bar

```

and send it to the filter chain. Finally it will send yet another bucket brigade with the EOS bucket indicating that there will be no more data sent:

```

bucket type      data
-----
1st   eos

```

EOS buckets are valid for request filters. For connection filters, you will get one only in the response filters and only at the end of the connection. You can see a sample workaround for this situation in the module `Apache2::Filter::HTTPHeadersFixup` available on the CPAN.

Note that the EOS bucket may come attached to the last bucket brigade with data, instead of coming in its own bucket brigade. The location depends on the other Apache modules manipulating the buckets and can vary. Filters should never assume that the EOS bucket is arriving alone in a bucket brigade. Therefore the first output filter will be invoked two or three times (three times if EOS is coming in its own brigade), depending on the number of bucket brigades sent by the response handler.

An upstream filter can modify the bucket brigades, by inserting extra bucket brigades or even by collecting the data from multiple bucket brigades and sending it along in just one brigade. Therefore, when coding a filter, never assume that the filter is always going to be invoked once, or any fixed number of times. Neither can you assume how the data is going to come in. To accommodate these situations, a typical filter handler may need to split its logic in three parts.

To illustrate, below is some pseudo-code that represents all three parts, i.e., initialization, processing, and finalization. This is a typical stream-oriented filter handler.

```

sub handler {
    my $f = shift;

    # runs on first invocation
    unless ($f->ctx) {
        init($f);
        $f->ctx(1);
    }

    # runs on all invocations
    process($f);

    # runs on the last invocation
    if ($f->seen_eos) {
        finalize($f);
    }
}

```

```

    }

    return Apache2::Const::OK;
}
sub init      { ... }
sub process  { ... }
sub finalize { ... }

```

The following diagram depicts all three parts:

filter flow logic

Let's explain each part using this pseudo-filter.

1. Initialization

During initialization, the filter runs code that you want executed only once, even if there are multiple invocations of the filter (this is during a single request). The filter context (`$f->ctx`) is used as a flag to accomplish this task. For each new request the filter context is created before the filter is called for the first time, and it is destroyed at the end of the request.

```

unless ($f->ctx) {
    init($f);
    $f->ctx(1);
}

```

When the filter is invoked for the first time `$f->ctx` returns `undef` and the custom function `init()` is called. This function could, for example, retrieve some configuration data set in *httpd.conf* or initialize some data structure to a default value.

To make sure that `init()` won't be called on the following invocations, we must set the filter context before the first invocation is completed:

```
$f->ctx(1);
```

In practice, the context is not just used as a flag, but to store real data. You can use it to hold any data structure and pass it between successive filter invocations. For example, the following filter handler counts the number of times it was invoked during a single request:

```

sub handler {
    my $f = shift;

    my $ctx = $f->ctx;
    $ctx->{invoked}++;
    $f->ctx($ctx);
    warn "filter was invoked $ctx->{invoked} times\n";

    return Apache2::Const::DECLINED;
}

```

Since this filter handler doesn't consume the data from the upstream filter, it's important that this handler return `Apache2::Const::DECLINED`, in which case `mod_perl` passes the current bucket brigade to the next filter. If this handler returns `Apache2::Const::OK`, the data will be lost, and if that data included a special EOS token, this may cause problems.

Unsetting the `Content-Length` header for filters that modify the response body length is a good example of code to run in the initialization phase:

```
unless ($f->ctx) {
    $f->r->headers_out->unset('Content-Length');
    $f->ctx(1);
}
```

We will see more initialization examples later in this chapter.

2. Processing

The next part:

```
process($f);
```

is unconditionally invoked on every filter invocation. That's where the incoming data is read, modified and sent out to the next filter in the filter chain. Here is an example that lowers the case of the characters passing through:

```
use constant READ_SIZE => 1024;
sub process {
    my $f = shift;
    while ($f->read(my $data, READ_SIZE)) {
        $f->print(lc $data);
    }
}
```

Here the filter operates only on a single bucket brigade. Since it manipulates every character separately the logic is simple.

In more complicated situations, a filter may need to buffer data before the transformation can be applied. For example, if the filter operates on HTML tokens (e.g., ``), it's possible that one brigade will include the beginning of the token (`<img`) and the remainder of the token (`src="me.jpg">`) will come in the next bucket brigade (on the next filter invocation). To operate on the token as a whole, you would need to capture each piece over several invocations. To do so, you can store the unprocessed data in the filter context and then access it again on the next invocation.

Another good example of the need to buffer data is a filter that performs data compression, because compression is usually effective only when applied to relatively big chunks of data. If a single bucket brigade doesn't contain enough data, the filter may need to buffer the data in the filter context until it collects enough to compress it.

3. Finalization

Finally, some filters need to know when they are invoked for the last time, in order to perform various cleanups and/or flush any remaining data. As mentioned earlier, Apache indicates this event by a special end of stream "token", represented by a bucket of type EOS. If the filter is using the streaming interface, rather than manipulating the bucket brigades directly, and it was calling `read()` in a while loop, it can check for the EOS token using the `$f->seen_eos` method:

```
if ($f->seen_eos) {
    finalize($f);
}
```

This check should be done at the end of the filter handler because the EOS token can come attached to the tail of some data or all alone such that the last invocation gets only the EOS token. If this test is performed at the beginning of the handler and the EOS bucket was sent in together with the data, the EOS event may be missed and the filter won't function properly.

Filters that directly manipulate bucket brigades must manually look for a bucket whose type is EOS. There are examples of this method later in the chapter.

While not all filters need to perform all of these steps, this is a good model to keep in mind while working on your filter handlers. Since filters are called multiple times per request, you will likely use these steps, with initialization, processing, and finishing, on all but the simplest filters.

1.3.4 Blocking Calls

All filters (excluding the core filter that reads from the network and the core filter that writes to it) block at least once when invoked. Depending on whether this is an input or an output filter, the blocking happens when the bucket brigade is requested from the upstream filter or when the bucket brigade is passed to the downstream filter.

Input and output filters differ in the ways they acquire the bucket brigades, and thus in how blocking is handled. Each type is described separately below. Although you can't see the difference when using the streaming API, it's important to understand how things work underneath. Therefore the examples below are transparent filters, passing data through them unmodified. Instead of reading the data in and printing it out, the bucket brigades are passed as is. This makes it easier to observe the blocking behavior.

The first example is a transparent input filter:

```
#file:MyApache2/FilterTransparent.pm (first part)
#-----
package MyApache2::FilterTransparent;

use Apache2::Filter ();

use Apache2::Const -compile => qw(OK);
use APR::Const     -compile => ':common';

sub in {
    my ($f, $bb, $mode, $block, $readbytes) = @_;

    my $rv = $f->next->get_brigade($bb, $mode, $block, $readbytes);
```

```

    return $rv unless $rv == APR::Const::SUCCESS;

    return Apache2::Const::OK;
}

```

When the input filter *in()* is invoked, it first asks the upstream filter for the next bucket brigade (using the `get_brigade()` call). That upstream filter is in turn going to ask for the bucket brigade from the next upstream filter and so on up the chain, until the last filter (called `core_in`), the one that reads from the network, is reached. The `core_in` filter reads, using a socket, a portion of the incoming data from the network, processes it, and sends it to its downstream filter. That filter processes the data and send it to its downstream filter, etc., until it reaches the first filter that requested the data. (In reality some other handler triggers the request for the bucket brigade, such as an HTTP response handler or a protocol module, but for this discussion it's sufficient to assume that it's the first filter that issues the `get_brigade()` call.)

The following diagram depicts a typical input filter data flow in addition to the program control flow.

input filter data flow

The black- and white-headed arrows show when the control is passed from one filter to another. In addition, the black-headed arrows show the actual data flow. The diagram includes some pseudo-code, in Perl for the `mod_perl` filters and in C for the internal Apache filters. You don't have to understand C to understand this diagram. What's important to understand is that when input filters are invoked, they first call each other via the `get_brigade()` call and then block (notice the brick wall on the diagram), waiting for the call to return. When this call returns, all upstream filters have already completed their filtering task on the bucket brigade.

As mentioned earlier, the streaming interface hides the details, but the first `$f->read()` call will block as the layer under it performs the `get_brigade()` call.

The diagram shows only part of the actual input filter chain for an HTTP request. The `...` indicates that there are more filters in between the `mod_perl` filter and `http_in`.

Now let's look at what happens in the output filters chain. Here the first filter acquires the bucket brigades containing the response data from the content handler (or another protocol handler if we aren't talking HTTP). It may then make some modification and pass the data to the next filter (using the `pass_brigade()` call), which in turn applies its modifications and sends the bucket brigade to the next filter, etc. This continues all the way down to the last filter (called `core`) which writes the data to the network via the socket the client is listening to.

Even though the output filters don't have to wait to acquire the bucket brigade (since the upstream filter passes it to them as an argument), they still block in a similar fashion to input filters, since they have to wait for the `pass_brigade()` call to return. In this case, they are waiting to pass the data along rather than waiting to receive it.

Here is an example of a transparent output filter:

```
#file:MyApache2/FilterTransparent.pm (continued)
#-----
sub out {
    my ($f, $bb) = @_;

    my $rv = $f->next->pass_brigade($bb);
    return $rv unless $rv == APR::Const::SUCCESS;

    return Apache2::Const::OK;
}
1;
```

The *out()* filter passes *\$bb* to the downstream filter unmodified. If you add print statements before and after the *pass_brigade()* call and configure the same filter twice, the print will show the blocking call.

The following diagram depicts a typical output filter data flow in addition to the program control flow:

output filter data flow

Similar to the input filters chain diagram, the arrows show the program control flow and in addition the black-headed arrows show the data flow. Again, it uses Perl pseudo-code for the *mod_perl* filter and C pseudo-code for the Apache filters and the brick walls represent the waiting. The diagram shows only part of the real HTTP response filters chain, where . . . stands for the omitted filters.

1.4 mod_perl Filters Declaration and Configuration

Now that we have laid out some basic concepts involved in filter use, we can look at how *mod_perl* filters are declared and configured.

1.4.1 Filter Priority Types

When Apache filters are configured they are inserted into the filters chain according to their priority type. In most cases when using one or two filters things will just work, however if you find that the order of filter invocation is wrong, you should consider the filter priority type. Unfortunately this information is available only in the Apache source code, unless it's documented in the module man pages. Numerical definitions of priority types, such as *AP_FTYPE_CONTENT_SET* and *AP_FTYPE_RESOURCE*, can be found in the Apache source distribution in *include/util_filter.h*.

As of this writing Apache comes with two core filters: *DEFLATE* and *INCLUDES*. Regardless of your configuration directives, e.g.,:

```
SetOutputFilter DEFLATE
SetOutputFilter INCLUDES
```

the *INCLUDES* filter will be inserted in the filters chain before the *DEFLATE* filter, even though it was configured after it. This is because the *DEFLATE* filter is of type *AP_FTYPE_CONTENT_SET* (20), whereas the *INCLUDES* filter is of type *AP_FTYPE_RESOURCE* (10).

As of this writing `mod_perl` provides two kind of filters with fixed priority type (the type is defined by the filter handler's attribute):

Handler's Attribute	Priority	Value
FilterRequestHandler	AP_FTYPE_RESOURCE	10
FilterConnectionHandler	AP_FTYPE_PROTOCOL	30

Therefore `FilterRequestHandler` filters (10) will always be invoked before the `DEFLATE` filter (20), whereas `FilterConnectionHandler` filters (30) will be invoked after it. When two filters have the same priority (e.g., the `INCLUDES` filter (10) has the same priority as `FilterRequestHandler` filters (10)), they are run in the order they are configured. Therefore filters are inserted according to the configuration order when `PerlSetOutputFilter` or `PerlSetInputFilter` are used.

1.4.2 *PerlInputFilterHandler*

The `PerlInputFilterHandler` directive registers a filter, and inserts it into the relevant input filters chain.

This handler is of type `VOID`.

The handler's configuration scope is `DIR`.

Arguments

`PerlInputFilterHandler` handlers are passed two arguments: an `Apache2::Filter` object; and an `APR::Brigade` object;

See the examples that follow in this chapter for further explanation.

Return

Filters are expected to return `Apache2::Const::OK` or `Apache2::Const::DECLINED`. See the examples that follow in this chapter for further explanation.

Examples

`PerlInputFilterHandler` handlers are automatically `AutoLoaded`, since they need to be compiled before the filter attributes can be accessed. Therefore if the filter handler subroutine is not called handler, you must preload the module containing the filter subroutine at server startup. A filter handler can be configured not to be `AutoLoaded`, using the `-` prefix. For example:

```
PerlInputFilterHandler -MyApache2::FilterTest::lc
```

The following sections include several examples that use the `PerlInputFilterHandler` handler.

1.4.3 PerlOutputFilterHandler

The PerlOutputFilterHandler directive registers a filter, and inserts it into the relevant output filters chain.

This handler is of type VOID.

The handler's configuration scope is DIR.

Arguments

PerlOutputFilterHandler handlers are passed five arguments: an Apache2::Filter object; an APR::Brigade object; an Apache2::Const :input_mode constant; an APR::Const :read_type constant; and the number of bytes to read.

See the examples that follow in this chapter for further explanation.

Return

Examples

The following sections include several examples that use the PerlOutputFilterHandler handler.

Similar to PerlInputFilterHandler, PerlOutputFilterHandler handlers are automatically AutoLoaded.

1.4.4 PerlSetInputFilter

The SetInputFilter directive, documented at <http://httpd.apache.org/docs-2.0/mod/core.html#setinputfilter>, sets the filter or filters which will process client requests and POST input when they are received by the server (in addition to any filters configured earlier).

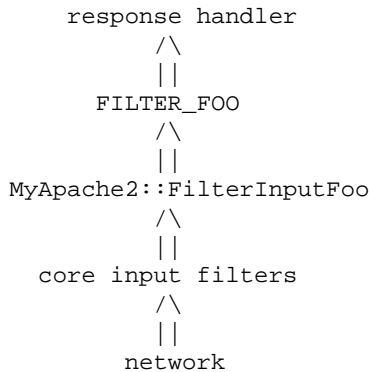
To mix mod_perl and non-mod_perl input filters of the same priority nothing special should be done. For example if we have an imaginary Apache filter FILTER_FOO and mod_perl filter MyApache2::FilterInputFoo, this configuration:

```
SetInputFilter FILTER_FOO
PerlInputFilterHandler MyApache2::FilterInputFoo
```

will add both filters. However the order of their invocation might not be as you expect. To make the invocation order the same as the insertion order, replace SetInputFilter with PerlSetInputFilter, like so:

```
PerlSetInputFilter FILTER_FOO
PerlInputFilterHandler MyApache2::FilterInputFoo
```

Now the `FILTER_FOO` filter will always be executed before the `MyApache2::FilterInputFoo` filter, since it was configured before `MyApache2::FilterInputFoo` (i.e., it'll apply its transformations on the incoming data last). The diagram below shows the input filters chain and the data flow from the network to the response handler for the presented configuration:



As explained in the section `Filter Priority Types` this directive won't affect filters of different priority. For example assuming that `MyApache2::FilterInputFoo` is a `FilterRequestHandler` filter, the configurations:

```

PerlInputFilterHandler MyApache2::FilterInputFoo
PerlSetInputFilter DEFLATE

```

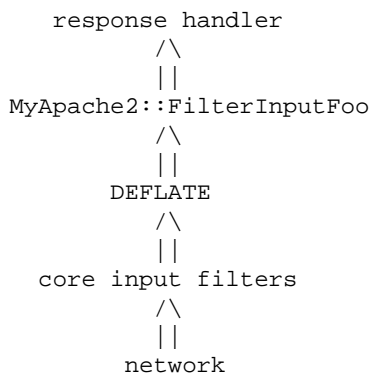
and

```

PerlSetInputFilter DEFLATE
PerlInputFilterHandler MyApache2::FilterInputFoo

```

are equivalent, because `mod_deflate`'s `DEFLATE` filter has a higher priority than `MyApache2::FilterInputFoo`. Therefore, it will always be inserted into the filter chain after `MyApache2::FilterInputFoo`, (i.e. the `DEFLATE` filter will apply its transformations on the incoming data first). The diagram below shows the input filters chain and the data flow from the network to the response handler for the presented configuration:



SetInputFilter's ; semantics are supported as well. For example, in the following configuration:

```
PerlInputFilterHandler MyApache2::FilterInputFoo
PerlSetInputFilter FILTER_FOO;FILTER_BAR
```

MyApache2::FilterOutputFoo will be executed first, followed by FILTER_FOO and finally by FILTER_BAR (again, assuming that all three filters have the same priority).

1.4.5 PerlSetOutputFilter

The SetOutputFilter directive, documented at <http://httpd.apache.org/docs-2.0/mod/core.html#setoutputfilter> sets the filters which will process responses from the server before they are sent to the client (in addition to any filters configured earlier).

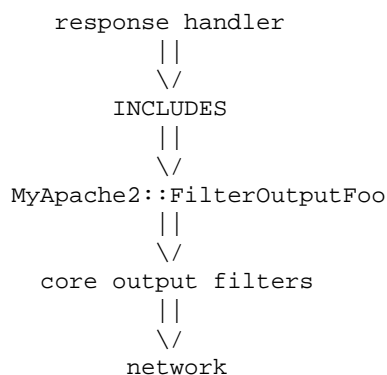
To mix mod_perl and non-mod_perl output filters of the same priority nothing special should be done. This configuration:

```
SetOutputFilter INCLUDES
PerlOutputFilterHandler MyApache2::FilterOutputFoo
```

As with input filters, to preserve the insertion order replace SetOutputFilter with PerlSetOutputFilter, like so:

```
PerlSetOutputFilter INCLUDES
PerlOutputFilterHandler MyApache2::FilterOutputFoo
```

Now mod_include's INCLUDES filter will always be executed before the MyApache2::FilterOutputFoo filter. The diagram below shows the output filters chain and the data flow from the response handler to the network for the presented configuration:



SetOutputFilter's ; semantics are supported as well. For example, in the following configuration:

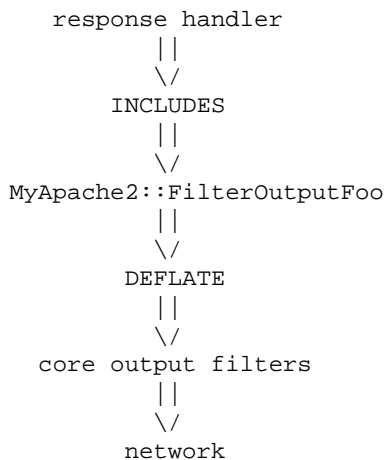
```
PerlOutputFilterHandler MyApache2::FilterOutputFoo
PerlSetOutputFilter INCLUDES;FILTER_FOO
```

MyApache2::FilterOutputFoo will be executed first, followed by INCLUDES and finally by FILTER_FOO (again, assuming that all three filters have the same priority).

As explained in the `PerlSetInputFilter` section, if filters have different priorities, the insertion order might be different. For example in the following configuration:

```
PerlSetOutputFilter DEFLATE
PerlSetOutputFilter INCLUDES
PerlOutputFilterHandler MyApache2::FilterOutputFoo
```

`mod_include`'s `INCLUDES` filter will be always executed before the `MyApache2::FilterOutputFoo` filter. The latter will be followed by `mod_deflate`'s `DEFLATE` filter, even though it was configured before the other two filters. This is because it has a higher priority. And the corresponding diagram looks like so:



1.4.6 Adding OutputFilters Dynamically

If you have the need to add output filters dynamically during the request, `mod_perl` 2.0 offers you the possibility to push filter callbacks at request time. For example here is how to add an output filter during the `Fixup` phase:

```
<Files *.html >
  PerlFixupHandler MyApache2::AddFilterDyn
</Files>
```

And the corresponding module is:

```
#file:MyApache2/AddFilterDyn.pm
#-----
package MyApache2::AddFilterDyn;

use Apache2::Filter;
use MyApache2::FilterObfuscate;

use Apache2::Const -compile => qw(OK);

sub handler {
    my $r = shift;

    $r->add_output_filter(\&MyApache2::FilterObfuscate::handler);
}
```

```

        return Apache2::Const::OK;
    }

    1;

```

You can also add connection filters dynamically. For more information refer to the `Apache2::Filter` manpages `add_input_filter` and `add_output_filter`.

1.4.7 HTTP Request vs. Connection Filters

`mod_perl 2.0` supports connection and HTTP request filtering. `mod_perl` filter handlers specify the type of the filter using the method attributes.

HTTP request filter handlers are declared using the `FilterRequestHandler` attribute. Consider the following request input and output filters skeleton:

```

package MyApache2::FilterRequestFoo;
use base qw(Apache2::Filter);

sub input : FilterRequestHandler {
    my ($f, $bb, $mode, $block, $readbytes) = @_;
    #...
}

sub output : FilterRequestHandler {
    my ($f, $bb) = @_;
    #...
}

1;

```

If the attribute is not specified, the default `FilterRequestHandler` attribute is assumed. Filters specifying subroutine attributes must subclass `Apache2::Filter`, others only need to:

```

use Apache2::Filter ();

```

Request filters are usually configured in the `<Location>` or equivalent sections:

```

PerlModule MyApache2::FilterRequestFoo
PerlModule MyApache2::NiceResponse
<Location /filter_foo>
    SetHandler modperl
    PerlResponseHandler      MyApache2::NiceResponse
    PerlInputFilterHandler   MyApache2::FilterRequestFoo::input
    PerlOutputFilterHandler  MyApache2::FilterRequestFoo::output
</Location>

```

Now we have the request input and output filters configured.

The connection filter handler uses the `FilterConnectionHandler` attribute. Here is a similar example for the connection input and output filters.

```

package MyApache2::FilterConnectionBar;
use base qw(Apache2::Filter);

sub input : FilterConnectionHandler {
    my ($f, $bb, $mode, $block, $readbytes) = @_;
    #...
}

sub output : FilterConnectionHandler {
    my ($f, $bb) = @_;
    #...
}

1;

```

With connection filters, unlike the request filters, the configuration must be done outside the `<Location>` or equivalent sections, usually within the `<VirtualHost>` or the global server configuration:

```

Listen 8005
<VirtualHost _default_:8005>
    PerlModule MyApache2::FilterConnectionBar
    PerlModule MyApache2::NiceResponse

    PerlInputFilterHandler MyApache2::FilterConnectionBar::input
    PerlOutputFilterHandler MyApache2::FilterConnectionBar::output
    <Location />
        SetHandler modperl
        PerlResponseHandler MyApache2::NiceResponse
    </Location>

</VirtualHost>

```

This configures the connection input and output filters.

As can be seen from the above examples, the only difference between connection filters and request filters is that connection filters see everything from the request, i.e., the headers and the body, whereas request filters see only the body.

1.4.8 Filter Initialization Phase

There is one more callback in the filter framework. And that's `FilterInitHandler`. This *init* callback runs immediately after the filter handler is inserted into the filter chain, before it is invoked for the first time. Here is a skeleton of an init handler:

```

sub init : FilterInitHandler {
    my $f = shift;
    #...
    return Apache2::Const::OK;
}

```

The attribute `FilterInitHandler` marks the Perl function as suitable to be used as a filter initialization callback.

For example you may decide to dynamically remove a filter before it had a chance to run, if some condition is true:

```
sub init : FilterInitHandler {
    my $f = shift;
    $f->remove() if should_remove_filter();
    return Apache2::Const::OK;
}
```

Not all `Apache2::Filter` methods can be used in the init handler, because it's not a filter. Hence you can use methods that operate on the filter itself, such as `remove()` and `ctx()` or retrieve request information, such as `r()` and `c()`. You cannot use methods that operate on data, such as `read()` and `print()`.

In order to hook an init filter handler, the real filter has to assign this callback using the `FilterHasInitHandler` function which accepts a reference to the callback function, similar to `push_handlers()`. The callback function referred to must have the `FilterInitHandler` attribute. For example:

```
package MyApache2::FilterBar;
use base qw(Apache2::Filter);
sub init : FilterInitHandler { ... }
sub filter : FilterRequestHandler FilterHasInitHandler(\&init) {
    my ($f, $bb) = @_;
    # ...
    return Apache2::Const::OK;
}
```

While attributes are parsed during compilation (it's really a sort of source filter), the argument to the `FilterHasInitHandler()` attribute is compiled at a later stage once the module is compiled.

The argument to `FilterHasInitHandler()` can be any Perl code which when `eval()`'ed returns a code reference. For example:

```
package MyApache2::OtherFilter;
use base qw(Apache2::Filter);
sub init : FilterInitHandler { ... }

package MyApache2::FilterBar;
use MyApache2::OtherFilter;
use base qw(Apache2::Filter);
sub get_pre_handler { \&MyApache2::OtherFilter::init }
sub filter : FilterHasInitHandler(get_pre_handler()) { ... }
```

Here the `MyApache2::FilterBar::filter` handler is configured to run the `MyApache2::OtherFilter::init` init handler.

Notice that the argument to `FilterHasInitHandler()` is always `eval()`'ed in the package of the real filter handler (not the init handler). So the above code leads to the following evaluation:

```
$init_sub = eval "package MyApache2::FilterBar; get_pre_handler()";
```

This part is actually done in C, using the `eval_pv()` C call.

Currently only one initialization callback can be registered per filter handler.

1.5 All-in-One Filter

Before we delve into the details of how to write filters that do something with the data, let's first write a simple filter that does nothing but snooping on the data that goes through it. We are going to develop the `MyApache2::FilterSnoop` handler which can snoop on request and connection filters, in input and output modes.

First we create a simple response handler that dumps the request's *args* and *content* as strings:

```
#file:MyApache2/Dump.pm
#-----
package MyApache2::Dump;

use strict;
use warnings;

use Apache2::RequestRec ();
use Apache2::RequestIO ();
use Apache2::Filter ();
use APR::Brigade ();
use APR::Bucket ();

use Apache2::Const -compile => qw(OK M_POST);

sub handler {
    my $r = shift;
    $r->content_type('text/plain');

    $r->print("args:\n", $r->args, "\n");

    if ($r->method_number == Apache2::Const::M_POST) {
        my $data = content($r);
        $r->print("content:\n$data\n");
    }

    return Apache2::Const::OK;
}

use Apache2::Connection ();

use Apache2::Const -compile => qw(MODE_READBYTES);
use APR::Const -compile => qw(SUCCESS BLOCK_READ);

use constant IOBUFSIZE => 8192;
```

```

sub content {
    my $r = shift;

    my $bb = APR::Brigade->new($r->pool, $r->connection->bucket_alloc);

    my $data = '';
    my $seen_eos = 0;
    do {
        $r->input_filters->get_brigade($bb, Apache2::Const::MODE_READBYTES,
                                       APR::Const::BLOCK_READ, IOBUFSIZE);

        for (my $b = $bb->first; $b; $b = $bb->next($b)) {
            if ($b->is_eos) {
                $seen_eos++;
                last;
            }

            if ($b->read(my $buf)) {
                $data .= $buf;
            }

            $b->remove; # optimization to reuse memory
        }
    } while (!$seen_eos);

    $bb->destroy;

    return $data;
}
1;

```

which is configured as:

```

PerlModule MyApache2::Dump
<Location /dump>
    SetHandler modperl
    PerlResponseHandler MyApache2::Dump
</Location>

```

If we issue the following request:

```
% echo "mod_perl rules" | POST 'http://localhost:8002/dump?foo=1&bar=2'
```

the response will be:

```

args:
foo=1&bar=2
content:
mod_perl rules

```

As you can see it simply dumped the query string and the posted data.

Now let's write the snooping filter:

```
#file:MyApache2/FilterSnoop.pm
#-----
package MyApache2::FilterSnoop;

use strict;
use warnings;

use base qw(Apache2::Filter);
use Apache2::FilterRec ();
use APR::Brigade ();
use APR::Bucket ();
use APR::BucketType ();

use Apache2::Const -compile => qw(OK DECLINED);
use APR::Const      -compile => ':common';

sub connection : FilterConnectionHandler { snoop("connection", @_) }
sub request    : FilterRequestHandler   { snoop("request",    @_) }

sub snoop {
    my $type = shift;
    my ($f, $bb, $mode, $block, $readbytes) = @_; # filter args

    # $mode, $block, $readbytes are passed only for input filters
    my $stream = defined $mode ? "input" : "output";

    # read the data and pass-through the bucket brigades unchanged
    if (defined $mode) {
        # input filter
        my $rv = $f->next->get_brigade($bb, $mode, $block, $readbytes);
        return $rv unless $rv == APR::Const::SUCCESS;
        bb_dump($type, $stream, $bb);
    }
    else {
        # output filter
        bb_dump($type, $stream, $bb);
        my $rv = $f->next->pass_brigade($bb);
        return $rv unless $rv == APR::Const::SUCCESS;
    }

    return Apache2::Const::OK;
}

sub bb_dump {
    my ($type, $stream, $bb) = @_;

    my @data;
    for (my $b = $bb->first; $b; $b = $bb->next($b)) {
        $b->read(my $bdata);
        push @data, $b->type->name, $bdata;
    }

    # send the sniffed info to STDERR so not to interfere with normal
    # output
    my $direction = $stream eq 'output' ? ">>>" : "<<<";
}
```

```

print STDERR "\n$direction $type $stream filter\n";

my $c = 1;
while (my ($btype, $data) = splice @data, 0, 2) {
    print STDERR "    o bucket $c: $btype\n";
    print STDERR "[$data]\n";
    $c++;
}
1;

```

Recall that there are two types of two filter handlers, one for connection and another for request filtering:

```

sub connection : FilterConnectionHandler { snoop("connection", @_) }
sub request    : FilterRequestHandler   { snoop("request",    @_) }

```

Both handlers forward their arguments to the `snoop()` function, which does the real work. These two subroutines are added in order to assign the two different attributes. In addition, the functions pass the filter type to `snoop()` as the first argument, which gets shifted off `@_`. The rest of `@_` are the arguments that were originally passed to the filter handler.

It's easy to know whether a filter handler is running in the input or the output mode. Although the arguments `$f` and `$bb` are always passed, the arguments `$mode`, `$block`, and `$readbytes` are passed only to input filter handlers.

If we are in input mode, in the same call we retrieve the bucket brigade from the previous filter on the input filters stack and immediately link it to the `$bb` variable which makes the bucket brigade available to the next input filter when the filter handler returns. If we forget to perform this linking our filter will become a black hole into which data simply disappears. Next we call `bb_dump()` which dumps the type of the filter and the contents of the bucket brigade to `STDERR`, without influencing the normal data flow.

If we are in output mode, the `$bb` variable already points to the current bucket brigade. Therefore we can read the contents of the brigade right away, and then we pass the brigade to the next filter.

Let's snoop on connection and request filter levels in both directions by applying the following configuration:

```

Listen 8008
<VirtualHost _default_:8008>
    PerlModule MyApache2::FilterSnoop
    PerlModule MyApache2::Dump

    # Connection filters
    PerlInputFilterHandler  MyApache2::FilterSnoop::connection
    PerlOutputFilterHandler MyApache2::FilterSnoop::connection

    <Location /dump>
        SetHandler modperl
        PerlResponseHandler MyApache2::Dump
        # Request filters
        PerlInputFilterHandler  MyApache2::FilterSnoop::request

```

```

        PerlOutputFilterHandler MyApache2::FilterSnoop::request
    </Location>

</VirtualHost>

```

Notice that we use a virtual host because we want to install connection filters.

If we issue the following request:

```
% echo "mod_perl rules" | POST 'http://localhost:8008/dump?foo=1&bar=2'
```

we get the same response as before we installed `MyApache2::FilterSnoop` because our snooping filter didn't change anything. The output didn't change, but there was some new information printed to the *error_log*. We present it all here, in order to understand how filters work.

First we can see the connection input filter at work, as it processes the HTTP headers. We can see that for this request each header is put into a separate brigade with a single bucket. The data is conveniently enclosed by [] so you can see the new line characters as well.

```

<<< connection input filter
    o bucket 1: HEAP
[POST /dump?foo=1&bar=2 HTTP/1.1
]

<<< connection input filter
    o bucket 1: HEAP
[TE: deflate,gzip;q=0.3
]

<<< connection input filter
    o bucket 1: HEAP
[Connection: TE, close
]

<<< connection input filter
    o bucket 1: HEAP
[Host: localhost:8008
]

<<< connection input filter
    o bucket 1: HEAP
[User-Agent: lwp-request/2.01
]

<<< connection input filter
    o bucket 1: HEAP
[Content-Length: 14
]

<<< connection input filter
    o bucket 1: HEAP
[Content-Type: application/x-www-form-urlencoded
]

```

```
<<< connection input filter
    o bucket 1: HEAP
[
]
```

Here the HTTP header has been terminated by a double new line. So far all the buckets were of the *HEAP* type, meaning that they were allocated from the heap memory. Notice that the HTTP request input filters will never see the bucket brigades with HTTP headers because they are consumed by the last core connection filter.

The following two entries are generated when `MyApache2::Dump::handler` reads the POSTed content:

```
<<< connection input filter
    o bucket 1: HEAP
[mod_perl rules]

<<< request input filter
    o bucket 1: HEAP
[mod_perl rules]
    o bucket 2: EOS
[]
```

As shown earlier, the connection input filter is run before the request input filter. Since our connection input filter was passing the data through unmodified and no other custom connection input filter was configured, the request input filter sees the same data. The last bucket in the brigade received by the request input filter is of type *EOS*, meaning that all the input data from the current request has been received.

Next we can see that `MyApache2::Dump::handler` has generated its response. However we can see that only the request output filter gets run at this point:

```
>>> request output filter
    o bucket 1: TRANSIENT
[args:
foo=1&bar=2
content:
mod_perl rules
]
```

This happens because Apache hasn't yet sent the response HTTP headers to the client. The request filter sees a bucket brigade with a single bucket of type *TRANSIENT* which is allocated from the stack memory.

The moment the first bucket brigade of the response body has entered the connection output filters, Apache injects a bucket brigade with the HTTP headers. Therefore we can see that the connection output filter is filtering the brigade with HTTP headers (notice that the request output filters don't see it):

```

>>> connection output filter
      o bucket 1: HEAP
[HTTP/1.1 200 OK
Date: Tue, 07 Mar 2006 10:59:08 GMT
Server: Apache/2.0.55 (Unix) mod_perl/2.000002
Perl/v5.8.4 mod_ssl/2.0.55 OpenSSL/0.9.7c DAV/2
Connection: close
Transfer-Encoding: chunked
Content-Type: text/plain; charset=ISO-8859-1

]

```

This is followed by the first response body's brigade:

```

>>> connection output filter
      o bucket 1: TRANSIENT
[2b
]
      o bucket 2: TRANSIENT
[args:
foo=1&bar=2
content:
mod_perl rules

]
      o bucket 3: IMMORTAL
[
]

```

If the response is large, the request and connection filters will filter chunks of the response one by one. These chunks are typically 8k in size, but this size can vary.

Finally, Apache sends a series of bucket brigades to finish off the response, including the end of stream meta-bucket to tell filters that they shouldn't expect any more data, and flush buckets to flush the data, to make sure that any buffered output is sent to the client:

```

>>> connection output filter
      o bucket 1: IMMORTAL
[0

]
      o bucket 2: EOS
[]

>>> connection output filter
      o bucket 1: FLUSH
[]

>>> connection output filter
      o bucket 1: FLUSH
[]

```

This module helps to illustrate that each filter handler can be called many times during each request and connection. It is called for each bucket brigade. Also it is important to mention that HTTP request input filters are invoked only if there is some POSTed data to read and it's consumed by a content handler.

1.6 Input Filters

mod_perl supports Connection and HTTP Request input filters. In the following sections we will look at each of these in turn.

1.6.1 Connection Input Filters

Let's say that we want to test how our handlers behave when they are requested as HEAD requests, rather than GET requests. We can alter the request headers at the incoming connection level with the alteration transparent to all handlers.

This example's filter handler looks for data like:

```
GET /perl/test.pl HTTP/1.1
```

and turns it into:

```
HEAD /perl/test.pl HTTP/1.1
```

The following input filter handler does that by directly manipulating the bucket brigades:

```
#file:MyApache2/InputFilterGET2HEAD.pm
#-----
package MyApache2::InputFilterGET2HEAD;

use strict;
use warnings;

use base qw(Apache2::Filter);

use APR::Brigade ();
use APR::Bucket ();

use Apache2::Const -compile => qw(OK DECLINED);
use APR::Const      -compile => ':common';

sub handler : FilterConnectionHandler {
    my ($f, $bb, $mode, $block, $readbytes) = @_;

    return Apache2::Const::DECLINED if $f->ctx;

    my $rv = $f->next->get_brigade($bb, $mode, $block, $readbytes);
    return $rv unless $rv == APR::Const::SUCCESS;

    for (my $b = $bb->first; $b; $b = $bb->next($b)) {
        $b->read(my $data);
        warn("data: $data\n");
    }
}
```

```

        if ($data and $data =~ s|^GET|HEAD|) {
            my $nb = APR::Bucket->new($bb->bucket_alloc, $data);
            $b->insert_after($nb);
            $b->remove; # no longer needed
            $f->ctx(1); # flag that that we have done the job
                        last;
        }
    }

    return Apache2::Const::OK;
}
1;

```

The filter handler is called for each bucket brigade, which then includes buckets with data. The gist of any input filter handler is to request the bucket brigade from the upstream filter, and return it to the downstream filter using the second argument `$bb`. It's important to remember that you can call methods on this argument, but you shouldn't assign to this argument, or the chain will be broken.

There are two techniques to choose from to retrieve-modify-return bucket brigades:

1. Create a new empty bucket brigade `$ctx_bb`, pass it to the upstream filter via `get_brigade()` and wait for this call to return. When it returns, `$ctx_bb` will be populated with buckets. Now the filter should move the bucket from `$ctx_bb` to `$bb`, on the way modifying the buckets if needed. Once the buckets are moved, and the filter returns, the downstream filter will receive the populated bucket brigade.
2. Pass `$bb` to the upstream filter using `get_brigade()` so it will be populated with buckets. Once `get_brigade()` returns, the filter can go through the buckets and modify them in place, or it can do nothing and just return (in which case, the downstream filter will receive the bucket brigade unmodified).

Both techniques allow addition and removal of buckets. Though the second technique is more efficient since it doesn't have the overhead of create the new brigade and moving the bucket from one brigade to another. In this example we have chosen to use the second technique, in the next example we will see the first technique.

Our filter has to perform the substitution of only one HTTP header (which normally resides in one bucket), so we have to make sure that no other data gets mangled (e.g. there could be POSTED data and it may match `/^GET/` in one of the buckets). We use `$f->ctx` as a flag here. When it's undefined the filter knows that it hasn't done the required substitution, though once it completes the job it sets the context to 1.

Using the information stored in the context, the filter can immediately return `Apache2::Const::DECLINED` when it's invoked after the substitution job has been done:

```
return Apache2::Const::DECLINED if $f->ctx;
```

In that case `mod_perl` will call `get_brigade()` internally which will pass the bucket brigade to the downstream filter. Alternatively the filter could do:

```
my $rv = $f->next->get_brigade($bb, $mode, $block, $readbytes);
return $rv unless $rv == APR::Const::SUCCESS;
return Apache2::Const::OK if $f->ctx;
```

but this is a bit less efficient.

[META: the most efficient thing to do is to remove the filter itself once the job is done, so it won't be even invoked after the job has been done.

```
if ($f->ctx) {
    $f->remove;
    return Apache2::Const::DECLINED;
}
```

However, this can't be used with Apache 2.0.49 and lower, since it has a bug when trying to remove the edge connection filter (it doesn't remove it). Most likely that problem will be not fixed in the 2.0 series due to design flows. I don't know if it's going to be fixed in 2.1 series.]

If the job wasn't done yet, the filter calls `get_brigade`, which populates the `$bb` bucket brigade. Next, the filter steps through the buckets looking for the bucket that matches the regex: `/^GET/`. If a match is found, a new bucket is created with the modified data (`s/^GET/HEAD/`). Now it has to be inserted in place of the old bucket. In our example we insert the new bucket after the bucket that we have just modified and immediately remove that bucket that we don't need anymore:

```
$b->insert_after($nb);
$b->remove; # no longer needed
```

Finally we set the context to 1, so we know not to apply the substitution on the following data, and break from the *for* loop.

The handler returns `Apache2::Const::OK` indicating that everything was fine. The downstream filter will receive the bucket brigade with one bucket modified.

Now let's check that the handler works properly. For example, consider the following response handler:

```
#file:MyApache2/RequestType.pm
#-----
package MyApache2::RequestType;

use strict;
use warnings;

use Apache2::RequestIO ();
use Apache2::RequestRec ();
use Apache2::Response ();

use Apache2::Const -compile => 'OK';

sub handler {
    my $r = shift;

    $r->content_type('text/plain');
    my $response = "the request type was " . $r->method;
```

```

    $r->set_content_length(length $response);
    $r->print($response);

    return Apache2::Const::OK;
}

1;

```

This handler returns to the client the request type it has issued. For a HEAD request Apache will discard the response body, but it will still set the correct Content-Length header, which will be 24 for a GET request and 25 for a HEAD request. Therefore, if this response handler is configured as:

```

Listen 8005
<VirtualHost _default_:8005>
  <Location />
    SetHandler modperl
    PerlResponseHandler +MyApache2::RequestType
  </Location>
</VirtualHost>

```

and a GET request is issued to /:

```

panic% perl -MLWP::UserAgent -le \
'$r = LWP::UserAgent->new()->get("http://localhost:8005/"); \
print $r->headers->content_length . ": " . $r->content'
24: the request type was GET

```

the response body will be:

```

the request type was GET

```

and the Content-Length header will be set to 24. This is what we would expect since the request was processed normally. However, if we enable the MyApache2::InputFilterGET2HEAD input connection filter:

```

Listen 8005
<VirtualHost _default_:8005>
  PerlInputFilterHandler +MyApache2::InputFilterGET2HEAD

  <Location />
    SetHandler modperl
    PerlResponseHandler +MyApache2::RequestType
  </Location>
</VirtualHost>

```

and issue the same GET request, we get only:

```

25:

```

This means the body was discarded by Apache, because our filter turned the GET request into a HEAD request. If Apache wasn't discarding the body on HEAD, the response would be:

the request type was HEAD

That's why the content length is reported as 25 and not 24 as in the real GET request. So the content length of 25 and lack of body text in the response confirm that our filter is acting as we expected.

1.6.2 HTTP Request Input Filters

Request filters are really non-different from connection filters, other than that they are working on request and response bodies and have an access to a request object.

1.6.3 Bucket Brigade-based Input Filters

As we have seen, filters can be either bucket brigade based, or stream oriented. Here we look at a request input filter that lowercases the request's body by directly manipulating the bucket brigade: `MyApache2::InputRequestFilterLC`.

```
#file:MyApache2/InputRequestFilterLC.pm
#-----
package MyApache2::InputRequestFilterLC;

use strict;
use warnings;

use base qw(Apache2::Filter);

use Apache2::Connection ();
use APR::Brigade ();
use APR::Bucket ();

use Apache2::Const -compile => 'OK';
use APR::Const      -compile => ':common';

sub handler : FilterRequestHandler {
    my ($f, $bb, $mode, $block, $readbytes) = @_;

    my $c = $f->c;
    my $bb_ctx = APR::Brigade->new($c->pool, $c->bucket_alloc);
    my $rv = $f->next->get_brigade($bb_ctx, $mode, $block, $readbytes);
    return $rv unless $rv == APR::Const::SUCCESS;

    while (!$bb_ctx->is_empty) {
        my $b = $bb_ctx->first;

        if ($b->is_eos) {
            $bb->insert_tail($b);
            last;
        }

        my $len = $b->read(my $data);
        $b = APR::Bucket->new($bb->bucket_alloc, lc $data) if $len;

        $b->remove;
    }
}
```

```

        $bb->insert_tail($b);
    }

    return Apache2::Const::OK;

}

1;

```

As promised, in this filter handler we have used the first technique of bucket brigade modification. The handler creates a temporary bucket brigade (`ctx_bb`), populates it with data using `get_brigade()`, and then moves buckets from it to the bucket brigade `$bb`. This bucket brigade is then retrieved by the downstream filter when our handler returns.

This filter doesn't need to know whether it was invoked for the first time or whether it has already done something. It's a stateless handler, since it has to lower case everything that passes through it. Notice that this filter can't be used as the connection filter for HTTP requests, since it will invalidate the incoming request headers. For example the first header line:

```
GET /perl/TEST.pl HTTP/1.1
```

becomes:

```
get /perl/test.pl http/1.1
```

which invalidates the request method, the URL and the protocol.

To test, we can use the `MyApache2::Dump` response handler, presented earlier, which dumps the query string and the content body as a response. Configure the server as follows:

```
<Location /lc_input>
    SetHandler modperl
    PerlResponseHandler +MyApache2::Dump
    PerlInputFilterHandler +MyApache2::InputRequestFilterLC
</Location>
```

Now when issuing a POST request:

```
% echo "mOd_pErL RuLeS" | POST 'http://localhost:8002/lc_input?FoO=1&BAR=2'
```

we get a response:

```
args:
FoO=1&BAR=2
content:
mod_perl rules
```

We can see that our filter has lowercased the POSTed body before the content handler received it. And you can see that the query string wasn't changed.

We have devoted so much attention to bucket brigade filters, even though they are simple to manipulate, because it is important to understand how the filters work underneath. This understanding is essential when you need to debug filters or to optimize them. There are cases when a bucket brigade filter may be more efficient than the stream-oriented version. For example if the filter applies a transformation to selected buckets, certain buckets may contain open filehandles or pipes, rather than real data. When you call `read()`, as shown above, the buckets will be forced to read that data in. But if you didn't want to modify these buckets you could pass them as they are and let Apache perform faster techniques for sending data from the file handles or pipes.

The call to `$b->read()`, or any other operation that internally forces the bucket to read the information into the memory (like the `length()` op), makes the data handling less efficient because it creates more work. Therefore care should be taken so not to read the data in unless it's really necessary, and sometimes you can gain this efficiency only by working with the bucket brigades.

1.6.4 Stream-oriented Input Filters

Let's now look at the same filter implemented using the stream-oriented API.

```
#file:MyApache2/InputRequestFilterLC2.pm
#-----
package MyApache2::InputRequestFilterLC2;

use strict;
use warnings;

use base qw(Apache2::Filter);

use Apache2::Const -compile => 'OK';

use constant BUFF_LEN => 1024;

sub handler : FilterRequestHandler {
    my $f = shift;

    while ($f->read(my $buffer, BUFF_LEN)) {
        $f->print(lc $buffer);
    }

    Apache2::Const::OK;
}
1;
```

The logic is very simple here. The filter reads in a loop and prints the modified data, which at some point will be sent to the next filter. The data transmission is triggered every time the internal `mod_perl` buffer is filled or when the filter returns.

`read()` populates `$buffer` to a maximum of `BUFF_LEN` characters (1024 in our example). Assuming that the current bucket brigade contains 2050 chars, `read()` will get the first 1024 characters, then 1024 characters more and finally the remaining 2 characters. Note that even though the response handler may have sent more than 2050 characters, every filter invocation operates on a single bucket brigade so you have to wait for the next invocation to get more input. Earlier we showed that you can force the generation

of several bucket brigades in the content handler by using `rflush()`. For example:

```
$r->print("string");
$r->rflush();
$r->print("another string");
```

It's only possible to get more than one bucket brigade from the same filter handler invocation if the filter is not using the streaming interface. In that case you can call `get_brigade()` as many times as needed or until EOS is received.

The configuration section is nearly identical for the two types of filters:

```
<Location /lc_input2>
    SetHandler modperl
    PerlResponseHandler +MyApache2::Dump
    PerlInputFilterHandler +MyApache2::InputRequestFilterLC2
</Location>
```

When issuing a POST request:

```
% echo "mOd_pErL RuLeS" | POST 'http://localhost:8002/lc_input2?FoO=1&BAR=2'
```

we get the response:

```
args:
FoO=1&BAR=2
content:
mod_perl rules
```

As before, we see that our filter has lowercased the POSTed body before the content handler received it and the query string wasn't changed.

1.7 Output Filters

As discussed above in the section HTTP Request vs. Connection Filters, `mod_perl` supports Connection and HTTP Request output filters. In the following sections we will look at each of these in turn.

`mod_perl` supports Connection and HTTP Request output filters. The differences between connection filters and HTTP request filters are described above in the section HTTP Request vs. Connection Filters.

1.7.1 Connection Output Filters

Connection filters filter **all** the data that is going through the server. Therefore if the connection is of the HTTP request type, connection output filters see the headers and the body of the response, whereas request output filters see only the response body.

META: for now see the request output filter explanations and examples, connection output filter examples will be added soon. Interesting ideas for such filters are welcome (possible ideas: mangling output headers for HTTP requests, pretty much anything for protocol modules).

1.7.2 HTTP Request Output Filters

As mentioned earlier, output filters can be written using the bucket brigades manipulation or the simplified stream-oriented interface. This section will show examples of both.

In order to generate output that can be manipulated by the two types of output filters, we will first develop a response handler that sends two lines of output: numerals 1234567890 and the English alphabet in a single string:

```
#file:MyApache2/SendAlphaNum.pm
#-----
package MyApache2::SendAlphaNum;

use strict;
use warnings;

use Apache2::RequestRec ();
use Apache2::RequestIO ();

use Apache2::Const -compile => qw(OK);

sub handler {
    my $r = shift;

    $r->content_type('text/plain');

    $r->print(1..9, "0\n");
    $r->print('a'..'z', "\n");

    return Apache2::Const::OK;
}
1;
```

In the examples below, we'll create a filter handler to reverse every line of the response body, preserving the new line characters in their places. Since we want to reverse characters only in the response body, without breaking the HTTP headers, we will use the HTTP request output filter rather than a connection output filter.

1.7.2.1 Stream-oriented Output Filters

The first filter implementation uses the stream-oriented filtering API:

```
#file:MyApache2/FilterReverse1.pm
#-----
package MyApache2::FilterReverse1;

use strict;
use warnings;

use base qw(Apache2::Filter);

use Apache2::Const -compile => qw(OK);
```

```

use constant BUFF_LEN => 1024;

sub handler : FilterRequestHandler {
    my $f = shift;

    while ($f->read(my $buffer, BUFF_LEN)) {
        for (split "\n", $buffer) {
            $f->print(scalar reverse $_);
            $f->print("\n");
        }
    }

    return Apache2::Const::OK;
}
1;

```

Next, we add the following configuration to *httpd.conf*:

```

PerlModule MyApache2::FilterReverse1
PerlModule MyApache2::SendAlphaNum
<Location /reverse1>
    SetHandler modperl
    PerlResponseHandler MyApache2::SendAlphaNum
    PerlOutputFilterHandler MyApache2::FilterReverse1
</Location>

```

Now when a request to */reverse1* is sent, the response handler `MyApache2::SendAlphaNum::handler()` sends:

```

1234567890
abcdefghijklmnopqrstuvwxy

```

as a response. The output filter handler `MyApache2::FilterReverse1::handler` reverses the lines, so the client gets:

```

0987654321
zyxwvutsrqponmlkjihgfedcba

```

The `Apache2::Filter` module loads the `read()` and `print()` methods which encapsulate the stream-oriented filtering interface.

The reversing filter is quite simple: in the loop it reads the data in the *readline()* mode in chunks up to the buffer length (1024 in our example), and then prints each line reversed while preserving the new line control characters at the end of each line. Behind the scenes `$f->read()` retrieves the incoming brigade and gets the data from it, and `$f->print()` appends to the new brigade which is then sent to the next filter in the stack. `read()` breaks the *while* loop when the brigade is emptied or the end of stream is received.

While this code is simple and easy to explain, there are cases it won't handle correctly. For example, it will have problems if the input lines are longer than 1,024 characters. It also doesn't account for the different line terminators on different platforms (e.g., "\n", "\r", or "\r\n"). Moreover a single line may be split across two or even more bucket brigades, so we have to store the unprocessed string in the filter context so it can be used on the following invocations. Below is an example of a more complete handler, which takes

care of these issues:

```

sub handler {
    my $f = shift;

    my $leftover = $f->ctx;
    while ($f->read(my $buffer, BUFF_LEN)) {
        $buffer = $leftover . $buffer if defined $leftover;
        $leftover = undef;
        while ($buffer =~ /([\r\n]*)([\r\n]*)/g) {
            $leftover = $1, last unless $2;
            $f->print(scalar(reverse $1), $2);
        }
    }

    if ($f->seen_eos) {
        $f->print(scalar reverse $leftover) if defined $leftover;
    }
    else {
        $f->ctx($leftover) if defined $leftover;
    }

    return Apache2::Const::OK;
}

```

The handler uses the `$leftover` variable to store unprocessed data as long as it fails to assemble a complete line or there is an incomplete line following the new line token. On the next handler invocation this data is then prepended to the next chunk that is read. When the filter is invoked for the last time, signaled by the `$f->seen_eos` method, it unconditionally reverses and sends the data down the stream, which is then flushed down to the client.

1.7.2.2 Bucket Brigade-based Output Filters

The following filter implementation uses the bucket brigades API to accomplish exactly the same task as the first filter.

```

#file:MyApache2/FilterReverse2.pm
#-----
package MyApache2::FilterReverse2;

use strict;
use warnings;

use base qw(Apache2::Filter);

use APR::Brigade ();
use APR::Bucket ();

use Apache2::Const -compile => 'OK';
use APR::Const      -compile => ':common';

sub handler : FilterRequestHandler {
    my ($f, $bb) = @_;

    my $bb_ctx = APR::Brigade->new($f->c->pool, $f->c->bucket_alloc);

```

```

while (!$bb->is_empty) {
    my $b = $bb->first;

    $b->remove;

    if ($b->is_eos) {
        $bb_ctx->insert_tail($b);
        last;
    }

    if ($b->read(my $data)) {
        $data = join "",
            map {scalar(reverse $_), "\n"} split "\n", $data;
        $b = APR::Bucket->new($bb->bucket_alloc, $data);
    }

    $bb_ctx->insert_tail($b);
}

my $rv = $f->next->pass_brigade($bb_ctx);

return $rv unless $rv == APR::Const::SUCCESS;

return Apache2::Const::OK;
}
1;

```

Below is the corresponding configuration for *httpd.conf*:

```

PerlModule MyApache2::FilterReverse2
PerlModule MyApache2::SendAlphaNum
<Location /reverse2>
    SetHandler modperl
    PerlResponseHandler      MyApache2::SendAlphaNum
    PerlOutputFilterHandler  MyApache2::FilterReverse2
</Location>

```

Now when a request to */reverse2* is made, the client gets:

```

0987654321
zyxwvutsrqponmlkjihgfedcba

```

as expected.

The bucket brigades output filter version is just a bit more complicated than the stream-oriented one. The handler receives the incoming bucket brigade *\$bb* as its second argument. When the handler finishes it must pass a brigade to the next filter in the stack, so we create a new bucket brigade into which we put the modified buckets and which eventually we pass to the next filter.

The core of the handler removes buckets from the head of the bucket brigade *\$bb*, while buckets are available, reads the data from the buckets, then reverses and puts the data into a newly created bucket. The new bucket is then inserted on the end of the new bucket brigade using the *insert_tail()* method. If we see a bucket which designates the end of stream, we insert that bucket on the tail of the new bucket

brigade and break the loop. Finally we pass the created brigade with modified data to the next filter and return.

Similar to the original version of `MyApache2::FilterReverse1::handler`, this filter is not smart enough to handle incomplete lines. The filter could be made more foolproof by building a better matching rule, and using the `$leftover` buffer as demonstrated in the previous section. This left as an exercise for the reader.

1.8 Filter Applications

The following sections provide various filter applications and their implementation.

1.8.1 Handling Data Underruns

Sometimes filters need to read at least N bytes before they can apply their transformation. It's quite possible that reading one bucket brigade is not enough, but that two or more are needed. This situation is sometimes referred to as an *underrun*.

Let's take an input filter as an example. When the filter realizes that it doesn't have enough data in the current bucket brigade, it can store the read data in the filter context, and wait for the next invocation of itself, which may or may not satisfy its needs. While it is gathering the data from the bucket brigades, it must return an empty bucket brigade to the upstream input filter. However, this is not the most efficient technique to resolve underruns.

Instead of returning an empty bucket brigade, the input filter can request extra bucket brigades until the underrun condition gets resolved. Note that this solution is transparent to any filters before or after the current filter.

Consider this HTTP request:

```
% perl -MLWP::UserAgent -le ' \
  $r = LWP::UserAgent->new()->post("http://localhost:8011/", \
    [content => "x" x (40 * 1024 + 7)]); \
  print $r->is_success ? $r->content : "failed: " . $r->code'
read 40975 chars
```

This client POSTs just a little bit more than 40kb of data to the server. Normally Apache splits incoming POSTed data into 8kb chunks, putting each chunk into a separate bucket brigade. Therefore we expect to get 5 brigades of 8kb, and one brigade with just a few bytes (a total of 6 bucket brigades).

Now let's assume our example filter needs to have $1024 * 16 + 5$ bytes to have a complete token before it can start its processing. The extra 5 bytes are just so we don't perfectly fit into 8kb bucket brigades, making the example closer to real situations. Having 40,975 bytes of input and a token size of 16,389 bytes, we will have 2 full tokens and a remainder of 8,197 bytes.

Before showing any code, let's look at the filter debug output to better explain what we expect to happen:

```

filter called
asking for a bb
asking for a bb
asking for a bb
storing the remainder: 7611 bytes

filter called
asking for a bb
asking for a bb
storing the remainder: 7222 bytes

filter called
asking for a bb
seen eos, flushing the remaining: 8197 bytes

```

We can see that the filter was invoked three times. The first time it has consumed three bucket brigades, collecting one full token of 16,389 bytes with a remainder of 7,611 bytes to be processed on the next invocation. The second time it needed only two more bucket brigades and this time, after completing the second token, 7,222 bytes remained. Finally on the third invocation it consumed the last bucket brigade for a total of six, just as we expected. However, it didn't have enough for the third token and since EOS has been seen (no more data expected), it has flushed the remaining 8,197 bytes as we calculated earlier.

It is clear from the debugging output that the filter was invoked only three times, instead of six times (there were six bucket brigades). Notice that the upstream input filter, if there is one, isn't aware that there were six bucket brigades, since it saw only three. Our example filter didn't do much with those tokens, so it has only repackaged data from 8kb per bucket brigade, to 16,389 bytes per bucket brigade. But of course in a real implementation some transformation would be applied on these tokens.

Now let's look at the implementation details. First let's look at the `response()` handler, which is the first part of the module:

```

#file:MyApache2/Underrun.pm
#-----
package MyApache2::Underrun;

use strict;
use warnings;

use constant IOBUFSIZE => 8192;

use Apache2::Const -compile => qw(MODE_READBYTES OK M_POST);
use APR::Const      -compile => qw(SUCCESS BLOCK_READ);

sub response {
    my $r = shift;

    $r->content_type('text/plain');

    if ($r->method_number == Apache2::Const::M_POST) {
        my $data = read_post($r);
        #warn "HANDLER READ: $data\n";
        my $length = length $data;
        $r->print("read $length chars");
    }
}

```

```

    return Apache2::Const::OK;
}

sub read_post {
    my $r = shift;

    my $bb = APR::Brigade->new($r->pool, $r->connection->bucket_alloc);

    my $data = '';
    my $seen_eos = 0;
    do {
        $r->input_filters->get_brigade($bb, Apache2::Const::MODE_READBYTES,
                                       APR::Const::BLOCK_READ, IOBUFSIZE);

        for (my $b = $bb->first; $b; $b = $bb->next($b)) {
            if ($b->is_eos) {

                $seen_eos++;
                last;
            }

            if ($b->read(my $buf)) {
                $data .= $buf;
            }

            $b->remove; # optimization to reuse memory
        }
    } while (!$seen_eos);

    $bb->destroy;

    return $data;
}

```

The `response()` handler is trivial -- it reads the POSTed data and prints how many bytes it has read. `read_post()` sucks in all POSTed data without parsing it.

Now comes the filter (which lives in the same package):

```

#file:MyApache2/Underrun.pm (continued)
#-----
use Apache2::Filter ();

use Apache2::Const -compile => qw(OK M_POST);

use constant TOKEN_SIZE => 1024*16 + 5; # ~16k

sub filter {
    my ($f, $bb, $mode, $block, $readbytes) = @_;
    my $ba = $f->r->connection->bucket_alloc;
    my $ctx = $f->ctx;
    my $buffer = defined $ctx ? $ctx : '';
    $ctx = ''; # reset
    my $seen_eos = 0;
    my $data;

```

```

warn "\nfilter called\n";

# fetch and consume bucket brigades until we have at least TOKEN_SIZE
# bytes to work with
do {
    my $tbb = APR::Brigade->new($f->r->pool, $ba);
    my $rv = $f->next->get_brigade($tbb, $mode, $block, $readbytes);
    warn "asking for a bb\n";
    ($data, $seen_eos) = flatten_bb($tbb);
    $tbb->destroy;
    $buffer .= $data;
} while (!$seen_eos && length($buffer) < TOKEN_SIZE);

# now create a bucket per chunk of TOKEN_SIZE size and put the remainder
# in ctx
for (split_buffer($buffer)) {
    if (length($_) == TOKEN_SIZE) {
        $bb->insert_tail(APR::Bucket->new($ba, $_));
    }
    else {
        $ctx .= $_;
    }
}

my $len = length($ctx);
if ($seen_eos) {
    # flush the remainder
    $bb->insert_tail(APR::Bucket->new($ba, $ctx));
    $bb->insert_tail(APR::Bucket::eos_create($ba));
    warn "seen eos, flushing the remaining: $len bytes\n";
}
else {
    # will re-use the remainder on the next invocation
    $f->ctx($ctx);
    warn "storing the remainder: $len bytes\n";
}

return Apache2::Const::OK;
}

# split a string into tokens of TOKEN_SIZE bytes and a remainder
sub split_buffer {
    my $buffer = shift;
    if ($] < 5.007) {
        my @tokens = $buffer =~ /(.{[TOKEN_SIZE]}|.+)/g;
        return @tokens;
    }
    else {
        # available only since 5.7.x+
        return unpack "(A" . TOKEN_SIZE . ")*", $buffer;
    }
}

sub flatten_bb {
    my ($bb) = shift;

    my $seen_eos = 0;

```

```

my @data;
for (my $b = $bb->first; $b; $b = $bb->next($b)) {
    $seen_eos++, last if $b->is_eos;
    $b->read(my $bdata);
    push @data, $bdata;
}

return (join('', @data), $seen_eos);
}

1;

```

The filter calls `get_brigade()` in a do-while loop until it reads enough data or sees EOS. Notice that it may get underruns several times, and then suddenly receive a lot of data at once, which will be enough for more than one minimal size token, so we have to take this into an account. Once the underrun condition is satisfied (we have at least one complete token) the tokens are put into a bucket brigade and returned to the upstream filter for processing, keeping any remainders in the filter context for the next invocations or flushing all the remaining data if EOS is seen.

Note that this example cannot be implemented with streaming filters because each invocation gives the filter exactly one bucket brigade to work with. The streaming interface does not currently provide a facility to fetch extra brigades.

Here is the Apache configuration for this example:

```

PerlModule MyApache2::Underrun
<Location />
    PerlInputFilterHandler MyApache2::Underrun::filter
    SetHandler modperl
    PerlResponseHandler MyApache2::Underrun::response
</Location>

```

1.8.2 Setting the Content-Length Header in Request Output Filters

Earlier we have stated that a filter that modifies the content's length must unset the Content-Length HTTP header. However sometimes it's desirable to have this header set, for example when dealing with proxies.

Since the headers are sent before the data, all the data must first be buffered and processed. You cannot accomplish this task with the streaming filter API since it passes FLUSH buckets through. As soon as the FLUSH bucket is received by the core filter that sends the headers, it generates the headers and sends those out. Therefore the bucket brigade API must be used here to have a complete control over what's going through. Here is a possible implementation:

```

#file:MyApache2/FilterChangeLength.pm
#-----
package MyApache2::FilterChangeLength;

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

use Apache2::RequestRec ();

```

```

use APR::Table ();
use APR::Bucket ();
use APR::Brigade ();

use base qw(Apache2::Filter);

use Apache2::Const -compile => qw(OK);
use APR::Const     -compile => ':common';

sub handler {
    my ($filter, $bb) = @_;

    my $ctx = $filter->ctx;

    # no need to unset the C-L header, since this filter makes sure to
    # correct it before any headers go out.
    #unless ($ctx) {
    #    $filter->r->headers_out->unset('Content-Length');
    #}

    my $data = exists $ctx->{data} ? $ctx->{data} : '';
    $ctx->{invoked}++;
    my ($bdata, $seen_eos) = flatten_bb($bb);
    $bdata =~ s/--/g;
    $data .= $bdata if $bdata;

    if ($seen_eos) {
        my $len = length $data;
        $filter->r->headers_out->set('Content-Length', $len);
        $filter->print($data) if $data;
    }

    else {
        # store context for all but the last invocation
        $ctx->{data} = $data;
        $filter->ctx($ctx);
    }

    return Apache2::Const::OK;
}

sub flatten_bb {
    my ($bb) = shift;

    my $seen_eos = 0;

    my @data;
    for (my $b = $bb->first; $b; $b = $bb->next($b)) {
        $seen_eos++, last if $b->is_eos;
        $b->read(my $bdata);
        push @data, $bdata;
    }
    return (join('', @data), $seen_eos);
}

1;

```

In this module we use `flatten_bb()` to read the data from the buckets and signal when the EOS is received. The filter simply collects the data, storing it in the filter context. When it receives EOS it sets the `Content-Length` header and sends the data out.

The configuration is straightforward:

```
PerlOutputFilterHandler MyApache2::FilterChangeLength
```

1.9 Filter Tips and Tricks

Various tips to use in filters.

1.9.1 Altering the Content-Type Response Header

Let's say that you want to modify the `Content-Type` header in the request output filter:

```
sub handler : FilterRequestHandler {
    my $f = shift;
    ...
    $f->r->content_type("text/html; charset=$charset");
    ...
}
```

Request filters have an access to the request object, so we simply modify it.

1.10 Writing Well-Behaving Filters

Filter writers must follow the following rules:

1.10.1 Connection Filters over KeepAlive Connections

Whenever a new HTTP request is processed, request filters get their context (`$f->ctx`) reset. This is also true for the connection filter context, as long as the connection is not a `keepalive` connection. When the connection is kept alive, there could be many requests processed during a single connection and the same filter context will persist through all of them, until the maximum number of `KeepAlive` requests over the same connection is reached or until the client breaks the connection.

Sometimes it's desirable to reset the whole context or parts of it before a HTTP request is processed. For example `Apache2::Filter::HTTPHeadersFixup` needs to know when it should start and stop processing HTTP headers. It keeps the state in the filter's context. The problem is that whenever a new HTTP request is coming in, it needs to be able to reset the state machine. If it doesn't, it will process the HTTP headers of the first request and miss the rest of the requests.

So let's say we have a hypothetical module `MyApache2::Filter::StateMachine` which implements an input connection filter and it processes incoming data as long as the `state` flag is down. Once that flag goes up, the filter switches to the pass-through-unmodified mode. Here is a skeleton of the module:

1.10.1 Connection Filters over KeepAlive Connections

```
#file:MyApache2/Filter/StateMachine.pm
#-----
package MyApache2::Filter::StateMachine;

use base qw(Apache2::Filter);
use Apache2::Connection ();

use Apache2::Const -compile => qw(OK DECLINED CONN_KEEPAALIVE);

sub handler : FilterConnectionHandler {
    my ($f, $bb, $mode, $block, $readbytes) = @_;

    my $ctx = context($f);

    # pass through unmodified
    return Apache2::Const::DECLINED if $ctx->{state};

    # get data, do some processing, send it out
    process(); # your code comes here

    # change the state if some condition is reached
    $ctx->{state}++ if $done_condition;

    return Apache2::Const::OK;
}

sub context {
    my ($f) = shift;

    my $ctx = $f->ctx;
    unless ($ctx) {
        $ctx = {
            state => 0,
        };

        $f->ctx($ctx);
    }

    return $ctx;
}

1;
```

To make this module work properly over KeepAlive connections, we want to reset the *state* flag at the very beginning of the new request. To accomplish this, all we need to do is to change the context wrapper to be:

```
sub context {
    my ($f) = shift;

    my $ctx = $f->ctx;
    unless ($ctx) {
        $ctx = {
            state          => 0,
            keepalives    => $f->c->keepalives,
        };
    }
}
```

```

    $f->ctx($ctx);
    return $ctx;
}

my $c = $f->c;
if ($c->keepalive == Apache2::Const::CONN_KEEPALIVE &&
    $ctx->{state} && $c->keepalives > $ctx->{keepalives}) {

    $ctx->{state} = 0;
    $ctx->{keepalives} = $c->keepalives;
}

return $ctx;
}

```

The only difference from the previous implementation is that we maintain one more state, which stores the number of requests served over the current connection. When Apache reports more served requests than we have in the context that means that we have a new request coming in. So we reset the *state* flag and store the new value of the served connections.

For a more complete real-world implementation, see: <http://search.cpan.org/dist/Apache-Filter-HTTP-HeadersFixup/>

1.10.2 Adjusting HTTP Headers

The following information is relevant for HTTP filters

- **Unsetting the Content-Length header**

HTTP response filters modifying the length of the body they process must unset the `Content-Length` header. For example, a compression filter modifies the body length, whereas a lowercasing filter doesn't; therefore the former has to unset the header, and the latter doesn't have to.

The header must be unset before any output is sent from the filter. If this rule is not followed, an HTTP response header with incorrect `Content-Length` value might be sent.

Since you want to run this code once during the multiple filter invocations, use the `ctx()` method to set the flag:

```

unless ($f->ctx) {
    $f->r->headers_out->unset('Content-Length');
    $f->ctx(1);
}

```

- **META:** Same goes for last-modified/etags, which may need to be unset, "vary" might need to be added if you want caching to work properly (depending on what your filter does).

1.10.3 Other issues

META: to be written. Meanwhile collecting important inputs from various sources.

[

If a filter desires to store the incoming buckets for post processing. It must check whether the bucket type is transient. If it is -- the data must be copied away. If not -- the buckets may contain corrupted data when used later. The right thing is accomplished transparently by `apr_bucket_setaside`, for which we need to provide a perl glue.

]

[

This one will be expanded by Geoff at some point:

HTTP output filter developers are ought to handle conditional GETs properly... (mostly for the reason of efficiency?)

]

[

talk about issues like not losing meta-buckets. e.g. if the filter runs a switch statement and propagates buckets types that were known at the time of writing, it may drop buckets of new types which may be added later, so it's important to ensure that there is a default cause where the bucket is passed as is.

of course mention the fact where things like EOS buckets must be passed, or the whole chain will be broken. Or if some filter decides to inject an EOS bucket by itself, it should probably consume and destroy the rest of the incoming bb. need to check on this issue.

]

[

Need to document somewhere (concepts?) that the buckets should never be modified directly, because the filter can't know ho else could be referencing it at the same time. (shared mem/cache/memory mapped files are examples on where you don't want to modify the data). Instead the data should be moved into a new bucket.

Also it looks like we need to `$b->destroy` (need to add the API) in addition to `$b->remove`. Which can be done in one stroke using `$b->delete` (need to add the API).

]

[

Mention `mod_bucketeer` as filter debugging tool (in addition to `FilterSnoop`)

]

1.11 Writing Efficient Filters

As of this writing, the Apache network input filter reads in 8000B chunks (not 8192B) and makes each bucket 8000B in size. Based on this, the most efficient reading technique is:

```
use constant BUFF_LEN => 8000;
while ($f->read(my $buffer, BUFF_LEN)) {
    # manip $buffer
    $f->print($buffer);
}
```

however if there is some filter in between, it may change the size of the buckets. Also this number may change in the future.

Hmm, I've also seen it read in 7819 chunks. I suppose this is not very reliable. But it's probably a good idea to ask at least 8k, so if a bucket brigade has < 8k, nothing will need to be stored in the internal buffer. i.e. `read()` will return less than asked for.

]

[

Bucket Brigades are used to make the data flow between filters and handlers more efficient. e.g. a file handle can be put in a bucket and the read from the file can be postponed to the very moment when the data is sent to the client, thus saving a lot of memory and CPU cycles. though filters writers should be aware that if they call `$b->read()`, or any other operation that internally forces the bucket to read the information into the memory (like the `length()` op) and thus making the data handling inefficient. therefore a care should be taken so not to read the data in, unless it's really necessary.

]

1.12 CPAN Modules

Several modules are available on the CPAN that implement `mod_perl 2.0` filters. As with all code on the CPAN, the source code is fully available, so you can download these modules and see exactly how they work.

- **Apache::Clean - Interface into HTML::Clean for mod_perl 2.0**

<http://search.cpan.org/dist/Apache-Clean/>

- **Apache::Filter::HTTPHeadersFixup - Manipulate Apache 2 HTTP Headers**

<http://search.cpan.org/dist/Apache-Filter-HTTPHeadersFixup/>

1.13 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Stas Bekman [<http://stason.org/>]

1.14 Authors

-

Only the major authors are listed above. For contributors see the Changes file.

Table of Contents:

1	Input and Output Filters	1
1.1	Description	2
1.2	Introducing Filters	2
1.3	I/O Filtering Concepts	6
1.3.1	Two Methods for Manipulating Data	6
1.3.2	HTTP Request Versus Connection Filters	7
1.3.3	Multiple Invocations of Filter Handlers	7
1.3.4	Blocking Calls	11
1.4	mod_perl Filters Declaration and Configuration	13
1.4.1	Filter Priority Types	13
1.4.2	PerlInputFilterHandler	14
1.4.3	PerlOutputFilterHandler	15
1.4.4	PerlSetInputFilter	15
1.4.5	PerlSetOutputFilter	17
1.4.6	Adding OutputFilters Dynamically	18
1.4.7	HTTP Request vs. Connection Filters	19
1.4.8	Filter Initialization Phase	20
1.5	All-in-One Filter	22
1.6	Input Filters	29
1.6.1	Connection Input Filters	29
1.6.2	HTTP Request Input Filters	33
1.6.3	Bucket Brigade-based Input Filters	33
1.6.4	Stream-oriented Input Filters	35
1.7	Output Filters	36
1.7.1	Connection Output Filters	36
1.7.2	HTTP Request Output Filters	37
1.7.2.1	Stream-oriented Output Filters	37
1.7.2.2	Bucket Brigade-based Output Filters	39
1.8	Filter Applications	41
1.8.1	Handling Data Underruns	41
1.8.2	Setting the Content-Length Header in Request Output Filters	45
1.9	Filter Tips and Tricks	47
1.9.1	Altering the Content-Type Response Header	47
1.10	Writing Well-Behaving Filters	47
1.10.1	Connection Filters over KeepAlive Connections	47
1.10.2	Adjusting HTTP Headers	49
1.10.3	Other issues	50
1.11	Writing Efficient Filters	51
1.12	CPAN Modules	51
1.13	Maintainers	52
1.14	Authors	52