1   Apache2::Filter - Perl API for Apache 2.0 Filtering

1.1 Synopsis

```
use Apache2::Filter ();

# filter attributes
my $c = $f->c;
my $r = $f->r;
my $frec = $f->frec();
my $next_f = $f->next;

my $ctx = $f->ctx;
$f->ctx($ctx);

# bucket brigade filtering API
$rc = $f->next->get_brigade($bb, $mode, $block, $readbytes);
$rc = $f->next->pass_brigade($bb);
$rc = $f->fflush($bb);

# streaming filtering API
while ($filter->read(my $buffer, $wanted)) {
    # transform $buffer here
    $filter->print($buffer);
}
if ($f->seen_eos) {
    $filter->print("filter signature");
}

# filter manipulations
$r->add_input_filter(\&callback);
$c->add_input_filter(\&callback);
$r->add_output_filter(\&callback);
$c->add_output_filter(\&callback);
$f->remove;
```

1.2 Description

`Apache2::Filter` provides Perl API for Apache 2.0 filtering framework.

Make sure to read the `Filtering tutorial|docs::2.0::user::handlers::filters`.

1.3 Common Filter API

The following methods can be called from any filter handler:

1.3.1 *c*

Get the current connection object from a connection or a request filter:

```
$c = $f->c;
```

- **obj: `$f ( Apache2::Filter object )`**
- **ret: `$c ( Apache2::Connection object )`**
- **since: 2.0.00**

1.3.2 *ctx*

Get/set the filter context data.

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

- **obj: `$f ( Apache2::Filter object )`**
- **opt arg2: `$ctx ( SCALAR )`**

    next context

- **ret: `$ctx ( SCALAR )`**

    current context

- **since: 2.0.00**

A filter context is created before the filter is called for the first time and it's destroyed at the end of the request. The context is preserved between filter invocations of the same request. So if a filter needs to store some data between invocations it should use the filter context for that. The filter context is initialized with the undef value.

The ctx method accepts a single SCALAR argument. Therefore if you want to store any other perl datas-tructure you should use a reference to it.

For example you can store a hash reference:

```
$f->ctx({ foo => 'bar' });
```

and then access it:

```
$foo = $f->ctx->{foo};
```

if you access the context more than once it's more efficient to copy it's value before using it:

```
my $ctx = $f->ctx;
$foo = $ctx->{foo};
```

to avoid redundant method calls. As of this writing $ctx is not a tied variable, so if you modify it need to store it at the end:

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

META: later we might make it a TIEd-variable interface, so it'll be stored automatically.

Besides its primary purpose of storing context data across multiple filter invocations, this method is also useful when used as a flag. For example here is how to ensure that something happens only once during the filter's life:

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

1.3.3 *frec*

Get/set the `Apache2::FilterRec` (filter record) object.

```
$frec = $f->frec();
```

- **obj: $f (Apache2::Filter object)**
- **ret: $frec (Apache2::FilterRec object)**
- **since: 2.0.00**

For example you can call `$frec->name` to get filter's name.

1.3.4 *next*

Return the `Apache2::Filter` object of the next filter in chain.

```
$next_f = $f->next;
```

- **obj: $f (Apache2::Filter object)**

  The current filter object

- **ret: $next_f (Apache2::Filter object)**

  The next filter object in chain

- **since: 2.0.00**

Since Apache inserts several core filters at the end of each chain, normally this method always returns an object. However if it's not a mod_perl filter handler, you can call only the following methods on it: `get_brigade`, `pass_brigade`, `c`, `r`, `frec` and `next`. If you call other methods the behavior is undefined.

The next filter can be a mod_perl one or not, it's easy to tell which one is that by calling `$f->frec->name`.

1.3.5 `r`

Inside an HTTP request filter retrieve the current request object:

```
$r = $f->r;
```

- **obj: `$f` (`Apache2::Filter object`)**
- **ret: `$r` (`Apache2::RequestRec object`)**
- **since: 2.0.00**

If a sub-request adds filters, then that sub-request object is associated with the filter.

1.3.6 `remove`

Remove the current filter from the filter chain (for the current request or connection).

```
$f->remove;
```

- **obj: `$f` (`Apache2::Filter object`)**
- **ret: no return value**
- **since: 2.0.00**

Notice that you should either complete the current filter invocation normally (by calling `get_brigade` or `pass_brigade` depending on the filter kind) or if nothing was done, return `Apache2::Const::DECLINED` and mod_perl will take care of passing the current bucket brigade through unmodified to the next filter in chain.

Note: calling remove() on the very top connection filter doesn't affect the filter chain due to a bug in Apache 2.0 (which may be fixed in 2.1). So don't use it with connection filters, till it gets fixed in Apache and then make sure to require the minimum Apache version if you rely on.

Remember that if the connection is `$c->keepalive`) and the connection filter is removed, it won't be added until the connection is closed. Which may happen after many HTTP requests. You may want to keep the filter in place and pass the data through unmodified, by returning `Apache2::Const::DECLINED`. If you need to reset the whole or parts of the filter context between requests, use the technique based on `$c->keepalives` counting.

This method works for native Apache (non-mod_perl) filters too.

1.4 Bucket Brigade Filter API

The following methods can be called from any filter, directly manipulating bucket brigades:

1.4.1 `fflush`

Flush a bucket brigade down the filter stack.

```
$rc = $f->fflush($bb);
```

- **obj: `$f ( Apache2::Filter object )`**

  The current filter

- **arg1: `$bb ( Apache2::Brigade object )`**

  The brigade to flush

- **ret: `$rc ( APR::Const status constant )`**

  Refer to the `pass_brigade()` entry.

- **excpt: `APR::Error`**

  Exceptions are thrown only when this function is called in the VOID context. Refer to the `get_brigade()` entry for details.

- **since: 2.0.00**

`fflush` is a shortcut method. So instead of doing:

```
my $b = APR::Bucket::flush_create($f->c->bucket_alloc);
$bb->insert_tail($b);
$f->pass_brigade($bb);
```

one can just write:

```
$f->fflush($bb);
```

1.4.2 `get_brigade`

This is a method to use in bucket brigade input filters. It acquires a bucket brigade from the upstream input filter.

```
$rc = $next_f->get_brigade($bb, $mode, $block, $readbytes);
$rc = $next_f->get_brigade($bb, $mode, $block);
$rc = $next_f->get_brigade($bb, $mode)
$rc = $next_f->get_brigade($bb);
```

- **obj: `$next_f ( Apache2::Filter object )`**

  The next filter in the filter chain.

Inside filter handlers it's usually $f->next. Inside protocol handlers: $c->input_filters.

- **arg1: $bb (`APR::Brigade object`)**

  The original bucket brigade passed to get_brigade(), which must be empty.

  Inside input filter handlers it's usually the second argument to the filter handler.

  Otherwise it should be created:

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

  On return it gets populated with the next bucket brigade. That brigade may contain nothing if there was no more data to read. The return status tells the outcome.

- **opt arg2: $mode (`Apache2::Const :input_mode constant`)**

  The filter mode in which the data should be read.

  If inside the filter handler, you should normally pass the same mode that was passed to the filter handler (the third argument).

  At the end of this section the available modes are presented.

  If the argument $mode is not passed, Apache2::Const::MODE_READBYTES is used as a default value.

- **opt arg3: $block (`APR::Const :read_type constant`)**

  You may ask the reading operation to be blocking: APR::Const::BLOCK_READ, or nonblocking: APR::Const::NONBLOCK_READ.

  If inside the filter handler, you should normally pass the same blocking mode argument that was passed to the filter handler (the forth argument).

  If the argument $block is not passed, APR::Const::BLOCK_READ is used as a default value.

- **opt arg4: $readbytes (integer)**

  How many bytes to read from the next filter.

  If inside the filter handler, you may want the same number of bytes, as the upstream filter, i.e. the argument that was passed to the filter handler (the fifth argument).

  If the argument $block is not passed, 8192 is used as a default value.

- **ret: $rc (`APR::Const status constant`)**

  On success, APR::Const::SUCCESS is returned and $bb is populated (see the $bb entry).

In case of a failure -- a failure code is returned, in which case normally it should be returned to the caller.

If the bottom-most filter doesn't read from the network, then `Apache2::NOBODY_READ` is returned (META: need to add this constant).

Inside protocol handlers the return code can also be `APR::Const::EOF`, which is success as well.

- **excpt: `APR::Error`**

  You don't have to ask for the return value. If this function is called in the VOID context, e.g.:

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

  mod_perl will do the error checking on your behalf, and if the return code is not `APR::Const::SUCCESS`, an `APR::Error  exception` will be thrown. The only time you want to do the error checking yourself, is when return codes besides `APR::Const::SUCCESS` are considered as successful and you want to manage them by yourself.

- **since: 2.0.00**

Available input filter modes (the optional second argument $mode) are:

- **`Apache2::Const::MODE_READBYTES`**

  The filter should return at most readbytes data

- **`Apache2::Const::MODE_GETLINE`**

  The filter should return at most one line of CRLF data. (If a potential line is too long or no CRLF is found, the filter may return partial data).

- **`Apache2::Const::MODE_EATCRLF`**

  The filter should implicitly eat any CRLF pairs that it sees.

- **`Apache2::Const::MODE_SPECULATIVE`**

  The filter read should be treated as speculative and any returned data should be stored for later retrieval in another mode.

- **`Apache2::Const::MODE_EXHAUSTIVE`**

  The filter read should be exhaustive and read until it can not read any more. Use this mode with extreme caution.

- **`Apache2::Const::MODE_INIT`**

  The filter should initialize the connection if needed, NNTP or FTP over SSL for example.

Either compile all these constants with:

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

But it's a bit more efficient to compile only those constants that you need.

Example:

Here is a fragment of a filter handler, that receives a bucket brigade from the upstream filter:

```
use Apache2::Filter ();
use APR::Const     -compile => qw(SUCCESS);
use Apache2::Const -compile => qw(OK);
sub filter {
    my ($f, $bb, $mode, $block, $readbytes) = @_;

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

    # ... process $bb

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

Usually arguments $mode, $block, $readbytes are the same as passed to the filter itself.

You can see that in case of a failure, the handler returns immediately with that failure code, which gets propagated to the downstream filter.

If you decide not check the return code, you can write it as:

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

    $f->next->get_brigade($bb, $mode, $block, $readbytes);

    # ... process $bb

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

and the error checking will be done on your behalf.

You will find many more examples in `the filter handlers|docs::2.0::user::handlers::filters` and `the protocol handlers|docs::2.0::user::handlers::protocols` tutorials.

1.4.3 *pass_brigade*

This is a method to use in bucket brigade output filters. It passes the current bucket brigade to the downstream output filter.

```
$rc = $next_f->pass_brigade($bb);
```

- **obj: $next_f (`Apache2::Filter object`)**

  The next filter in the filter chain.

  Inside output filter handlers it's usually `$f->next`. Inside protocol handlers: `$c->output_filters`.

- **arg1: $bb (`APR::Brigade object`)**

  The bucket brigade to pass.

  Inside output filter handlers it's usually the second argument to the filter handler (after potential manipulations).

- **ret: $rc (`APR::Const status constant`)**

  On success, `APR::Const::SUCCESS` is returned.

  In case of a failure -- a failure code is returned, in which case normally it should be returned to the caller.

  If the bottom-most filter doesn't write to the network, then `Apache2::NOBODY_WROTE` is returned (META: need to add this constant).

  Also refer to the `get_brigade()` entry to see how to avoid checking the errors explicitly.

- **excpt: `APR::Error`**

  Exceptions are thrown only when this function is called in the VOID context. Refer to the `get_brigade()` entry for details.

- **since: 2.0.00**

The caller relinquishes ownership of the brigade (i.e. it may get destroyed/overwritten/etc. by the callee).

Example:

Here is a fragment of a filter handler, that passes a bucket brigade to the downstream filter (after some potential processing of the buckets in the bucket brigade):

```
use Apache2::Filter ();
use APR::Const    -compile => qw(SUCCESS);
use Apache2::Const -compile => qw(OK);
sub filter {
    my ($f, $bb) = @_;

    # ... process $bb

    my $rc = $f->next->pass_brigade($bb);
```

```
    return $rc unless $rc == APR::Const::SUCCESS;

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

1.5 Streaming Filter API

The following methods can be called from any filter, which uses the simplified streaming functionality:

1.5.1 `print`

Send the contents of $buffer to the next filter in chain (via internal buffer).

```
$sent = $f->print($buffer);
```

- **obj: $f (Apache2::Filter object)**
- **arg1: $buffer (string)**

    The data to send.

- **ret: $sent (integer)**

    How many characters were sent. There is no need to check, since all should go through and if something goes work an exception will be thrown.

- **excpt: APR::Error**
- **since: 2.0.00**

This method should be used only in streaming filters.

1.5.2 `read`

Read data from the filter

```
$read = $f->read($buffer, $wanted);
```

- **obj: $f (Apache2::Filter object)**
- **arg1: $buffer (SCALAR)**

    The buffer to fill. All previous data will be lost.

- **opt arg2: $wanted (integer)**

    How many bytes to attempt to read.

    If this optional argument is not specified -- the default 8192 will be used.

- **ret: `$read` (integer)**

  How many bytes were actually read.

  `$buffer` gets populated with the string that is read. It will contain an empty string if there was nothing to read.

- **excpt: `APR::Error`**
- **since: 2.0.00**

Reads at most `$wanted` characters into `$buffer`. The returned value `$read` tells exactly how many were read, making it easy to use it in a while loop:

```
while ($filter->read(my $buffer, $wanted)) {
    # transform $buffer here
    $filter->print($buffer);
}
```

This is a streaming filter method, which acquires a single bucket brigade behind the scenes and reads data from all its buckets. Therefore it can only read from one bucket brigade per filter invocation.

If the EOS bucket is read, the `seen_eos` method will return a true value.

1.5.3 *seen_eos*

This methods returns a true value when the EOS bucket is seen by the `read` method.

```
$ok = $f->seen_eos;
```

- **obj: `$f ( Apache2::Filter object )`**

  The filter to remove

- **ret: `$ok` (boolean)**

  a true value if EOS has been seen, otherwise a false value

- **since: 2.0.00**

This method only works in streaming filters which exhaustively `$f->read` all the incoming data in a while loop, like so:

```
while ($f->read(my $buffer, $wanted)) {
    # do something with $buffer
}
if ($f->seen_eos) {
    # do something
}
```

The technique in this example is useful when a streaming filter wants to append something to the very end of data, or do something at the end of the last filter invocation. After the EOS bucket is read, the filter should expect not to be invoked again.

If an input streaming filter doesn't consume all data in the bucket brigade (or even in several bucket brigades), it has to generate the EOS event by itself. So when the filter is done it has to set the EOS flag:

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

when the filter handler returns, internally mod_perl will take care of creating and sending the EOS bucket to the upstream input filter.

A similar logic may apply for output filters.

In most other cases you shouldn't set this flag. When this flag is prematurely set (before the real EOS bucket has arrived) in the current filter invocation, instead of invoking the filter again, mod_perl will create and send the EOS bucket to the next filter, ignoring any other bucket brigades that may have left to consume. As mentioned earlier this special behavior is useful in writing special tests that test abnormal situations.

1.6 Other Filter-related API

Other methods which affect filters, but called on non-`Apache2::Filter` objects:

1.6.1 `add_input_filter`

Add `&callback` filter handler to input request filter chain.

```
$r->add_input_filter(\&callback);
```

Add `&callback` filter handler to input connection filter chain.

```
$c->add_input_filter(\&callback);
```

- **obj: `$c` (`Apache2::Connection object`) or `$r` (`Apache2::RequestRec object`)**
- **arg1: `&callback` (CODE ref)**
- **ret: no return value**
- **since: 2.0.00**

[META: It seems that you can't add a filter when another filter is called. I've tried to add an output connection filter from the input connection filter when it was called for the first time. It didn't have any affect for the first request (over keepalive connection). The only way I succeeded to do that is from that input connection filter's filter_init handler. In fact it does work if there is any filter additional filter of the same kind configured from httpd.conf or via filter_init. It looks like there is a bug in httpd, where it doesn't prepare the chain of 3rd party filter if none were inserted before the first filter was called.]

1.6.2 `add_output_filter`

Add `&callback` filter handler to output request filter chain.

```
$r->add_output_filter(\&callback);
```

Add `&callback` filter handler to output connection filter chain.

```
$c->add_output_filter(\&callback);
```

- **obj: `$c` (`Apache2::Connection object`) or `$r` (`Apache2::RequestRec object`)**
- **arg1: `&callback` (CODE ref)**
- **ret: no return value**
- **since: 2.0.00**

1.7 Filter Handler Attributes

Packages using filter attributes have to subclass `Apache2::Filter`:

```
package MyApache2::FilterCool;
use base qw(Apache2::Filter);
```

Attributes are parsed during the code compilation, by the function `MODIFY_CODE_ATTRIBUTES`, inherited from the `Apache2::Filter` package.

1.7.1 `FilterRequestHandler`

The `FilterRequestHandler` attribute tells mod_perl to insert the filter into an HTTP request filter chain.

For example, to configure an output request filter handler, use the `FilterRequestHandler` attribute in the handler subroutine's declaration:

```
package MyApache2::FilterOutputReq;
sub handler : FilterRequestHandler { ... }
```

and add the configuration entry:

```
PerlOutputFilterHandler MyApache2::FilterOutputReq
```

This is the default mode. So if you are writing an HTTP request filter, you don't have to specify this attribute.

The section HTTP Request vs. Connection Filters delves into more details.

1.7.2 `FilterConnectionHandler`

The `FilterConnectionHandler` attribute tells mod_perl to insert this filter into a connection filter chain.

For example, to configure an output connection filter handler, use the `FilterConnectionHandler` attribute in the handler subroutine's declaration:

```
package MyApache2::FilterOutputCon;
sub handler : FilterConnectionHandler { ... }
```

and add the configuration entry:

```
PerlOutputFilterHandler MyApache2::FilterOutputCon
```

The section HTTP Request vs. Connection Filters delves into more details.

1.7.3 `FilterInitHandler`

The attribute `FilterInitHandler` marks the function suitable to be used as a filter initialization callback, which is called immediately after a filter is inserted to the filter chain and before it's actually called.

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

In order to hook this filter callback, the real filter has to assign this callback using the `Filter-HasInitHandler` which accepts a reference to the callback function.

For further discussion and examples refer to the Filter Initialization Phase tutorial section.

1.7.4 `FilterHasInitHandler`

If a filter wants to run an initialization callback it can register such using the `FilterHasInitHandler` attribute. Similar to `push_handlers` the callback reference is expected, rather than a callback name. The used callback function has to 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;
}
```

For further discussion and examples refer to the Filter Initialization Phase tutorial section.

1.8 Configuration

mod_perl 2.0 filters configuration is explained in the filter handlers tutorial.

1.8.1 `PerlInputFilterHandler`

See `PerlInputFilterHandler`.

1.8.2 `PerlOutputFilterHandler`

See `PerlOutputFilterHandler`.

1.8.3 `PerlSetInputFilter`

See `PerlSetInputFilter`.

1.8.4 `PerlSetOutputFilter`

See `PerlSetInputFilter`.

1.9 TIE Interface

`Apache2::Filter` also implements a tied interface, so you can work with the `$f` object as a hash reference.

The TIE interface is mostly unimplemented and might be implemented post 2.0 release.

1.9.1 `TIEHANDLE`

```
$ret = TIEHANDLE($stashsv, $sv);
```

- **obj: `$stashsv` (SCALAR)**
- **arg1: `$sv` (SCALAR)**
- **ret: `$ret` (SCALAR)**
- **since: subject to change**

1.9.2 `PRINT`

```
$ret = PRINT(...);
```

- **obj: . . . (XXX)**
- **ret: `$ret` (integer)**
- **since: subject to change**

1.10 See Also

mod_perl 2.0 documentation.

1.11 Copyright

mod_perl 2.0 and its core modules are copyrighted under The Apache Software License, Version 2.0.

1.12 Authors

The mod_perl development team and numerous contributors.

Table of Contents: