Chapter 6: Filter Modules

In terms of application development, the most important innovation in Apache 2 is the filter architecture, and the ability to chain multiple different data processing operations at will. This chaptertakes a detailed look at the filter chain, and offers several illustrative filter modules.

Before going into details, let's review a few basics. Chapter 2 described howfilters operate on a "data" axis, orthogonal to the processing axis familiar from Apache 1 and other webservers. But this is not the whole story. Strictly speaking, it is only really accurate for content filters: that is, those filters that operate on the body of an HTTP request or response. If your application is not concerned directly with processing HTTP requests, you may need to use filters that are not so clearly associated with the content generator.

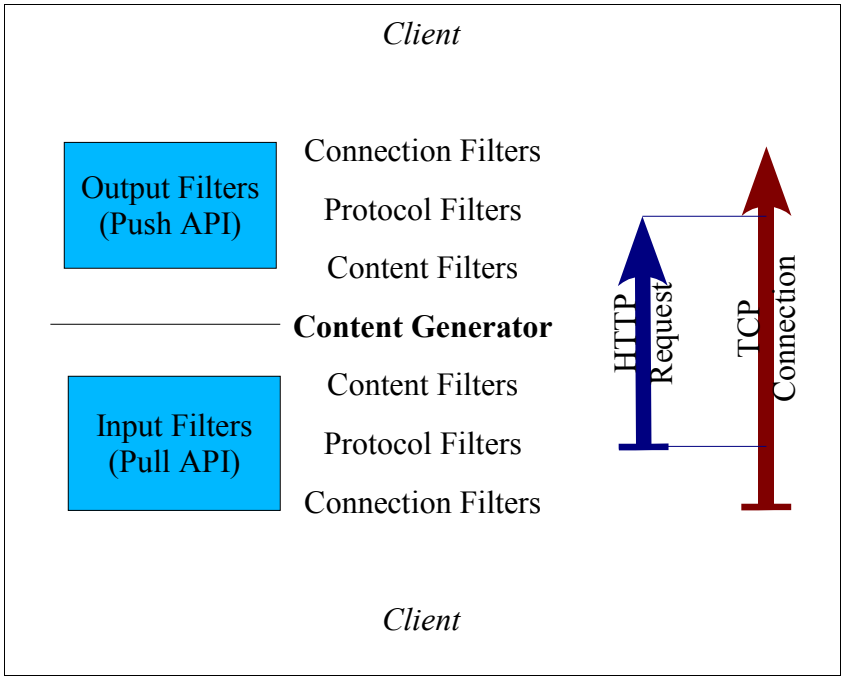
Types of Filter

So let's take a closer look at the filter chain. Filters are classified in two ways:

1. Input and Output Filters

Filters that process request data coming from a client are known as input filters. Filters that process response data as it is sent out to the client are known as output filters.

We will deal with the APIs for input and output filters in detail below.



2. Content, Protocol and Connection Filters

Each filter chain (input and output) passes through predefined stages. Thus the same filter architecture can be used for various different kinds of operation. In brief, from the content generator to the client, we have:

- Content filters, which process document contents within a request. These are the

filters most commonly relevant to applications programming.

- Protocol filters, which deal with details of the protocol but treat the contents as opaque. These are concerned with translating between HTTP data (as defined in RFC2616) and Apache's internal representation in the `request_rec` and associated structures.
- Connection filters, which process a TCP connection without reference to HTTP (either the protocol or contents). These are concerned with interfacing apache with the network, and operate entirely outside the scope of HTTP or of any `request_rec`.

Although the function of these filters is very different, moving from an applications level in the inner layers to a system level further out, the API is the same throughout. There is just one important difference: the inner filters, working on HTTP, have a a valid `request_rec` object, whereas connection-level filters have none. All filters have a `conn_rec` for the TCP connection.

In more detail, the output chain comprises the following stages in an enumeration in `util_filter.h` (the input chain is an exact mirror-image of this, and uses the same definitions).

`AP_FTYPE_RESOURCE` is for content filters. These are the first to see content as it is produced by the content generator, and they serve to examine, modify, or even completely rewrite it. This is the most common form of applications filter, and encompasses markup processing (such as SSI or XML filtering), image processing, or content assembly/aggregation. Resource filters may completely change the nature of the contents: for example, an XSLT filter might change the contents from XML to HTML or PDF.

`AP_FTYPE_CONTENT_SET` is a second stage of content filtering. It is intended for operations concerned with packaging the contents, such as `mod_deflate` (which applies gzip compression).

Filters of type `RESOURCE` or `CONTENT_SET` operate on an HTTP Response Entity, the body contents being returned to the client. The HTTP headers don't pass through these filters. The headers can be accessed in exactly the same way as in a content generator, via the headers tables in the `request_rec`.

`AP_FTYPE_PROTOCOL` is the third layer of filtering. The normal function here is to insert the HTTP headers ahead of the data emerging from the content filters. This is dealt with by a core filter `HTTP_HEADER` (function `ap_http_header_filter`), so applications can normally ignore it. Apache also handles byte ranges requests using a protocol filter.

`AP_FTYPE_TRANSCODE` is for transport-level packaging. Apache implements HTTP chunking (where applicable) at this level.

`AP_FTYPE_CONNECTION` filters operate on connections, at the TCP level (below the HTTP level, so "requests" no longer exist). Apache (`mod_ssl`) uses it for SSL

encoding. Another application is throttling and bandwidth control.

`AP_FTYPE_NETWORK` is the final layer, and deals with the connection to the client itself. This is normally dealt with by Apache's "CORE" output filter (function `ap_core_output_filter`).

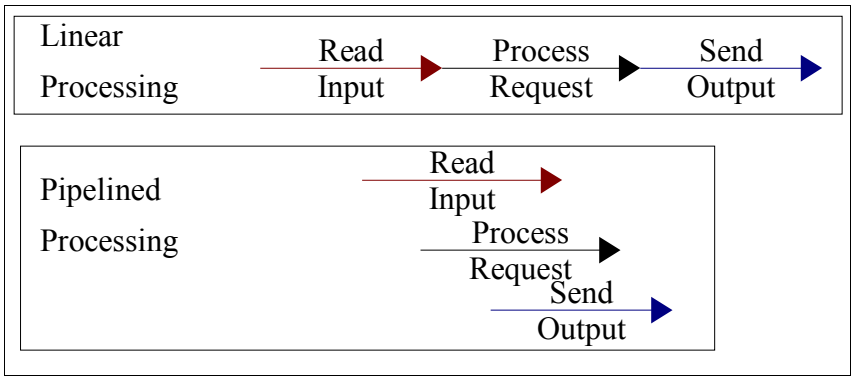
Anatomy of a Filter

The heart of a filter module is a callback function. How this is called differs between input and output filters:

- The input filter chain runs whenever the handler requests data from the client. Apache will call our callback function to request (pull) a chunk of data from it. Our filter must in turn pull a chunk of data from the next filter in the chain, process it, and return the requested data to the caller.
- The output filter chain runs whenever the handler sends a chunk of data to the client. This may be triggered explicitly by the handler (with `ap_pass_brigade`), or implicitly when a handler using the stdio-like APIs has filled a default (8Kbit) buffer. Our filter should process the data, and send (push) a chunk to the next filter in the chain.

Apart from the callback, there is an optional initialisation function, and filter modules may of course independently use other parts of the Apache API where necessary.

Pipelining



The basic principle of pipelining is that we should not have to wait for one stage of processing to complete before starting on the next. In the context of a webserver, where I/O commonly takes far more time than processing a request, this is an important performance issue.

In the Apache 2.x filter architecture, we don't just have the three stages to processing data. Every filter is itself a stage. So there is still more to be gained by pipelining. As far as possible, we want to run the filters in parallel.

To run pipelined filters on large documents without introducing scalability problems, we must avoid having to load an entire document into memory at once. Apache's filters therefore work on chunks of data rather than entire documents. Any general-purpose filter must allow for this. Filters should always seek to cooperate with this pipelining: ideally, a filter should always process a chunk of data and pass it on before the callback returns. Sometimes this is not possible, and a filter needs to buffer data over more than

one call: for example, running an XSLT transform requires that the entire document be parsed into an in-memory structure, so an XSLT filter can't avoid breaking the pipeline.

Pipelining can be an important consideration when designing a module. If you are planning to use an external library, it's worth reviewing how well it will work with the pipeline. In the case of an input filter, that's usually straightforward: it can just pull in more data from the pipeline on demand. But for an output filter, you need to look for an API that can accept arbitrary chunks of data. This author has written a number of XML- and HTML-parsing filters, and working with the Apache pipeline has a profound effect on the choice of a parser. Among markup processing libraries, Expat and libxml2 have `parseChunk` APIs and work well with Apache, but Tidy, OpenSP and Xerces-C have no such APIs, and so cannot be used without breaking the pipeline.

The Filter API and Objects

We have already introduced the filter callback function. This differs between input and output filters. So let's deal with each in turn.

Output Filters

The callback prototype for output filters is

```
apr_status_t my_output_filter_func(ap_filter_t* f,
       apr_bucket_brigade* bb)
```

Here `f` is the filter object, and `bb` a bucket brigade containing an arbitrary chunk (zero or more bytes) of data in APR buckets. The filter `func` should process the data in `bb`, then pass the processed data to the next filter in the chain, `f->next`. We will see how to do this when we develop filter examples later in this chapter.

Input Filters

The input filter callback is a little more complex:

```
apr_status_t my_input_filter_func(
                        ap_filter_t* f,
                        apr_bucket_brigade* bb,
                        ap_input_mode_t mode,
                        apr_read_type_e block,
                        apr_off_t readbytes)
```

The first two arguments are the same as the output filter arguments, although the usage differs. This is a pull API, and the function is responsible for fetching a chunk of data from the next filter in the input chain, putting that data into the bucket brigade, and returning to the caller. The other arguments are as follows:

mode is one of an enum:

```
typedef enum {
    /** The filter should return at most readbytes data. */
    AP_MODE_READBYTES,
    /** 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).
     */
    AP_MODE_GETLINE,
    /** The filter should implicitly eat any CRLF pairs that it sees. */
    AP_MODE_EATCRLF,
    /** The filter read should be treated as speculative and any
returned
     *  data should be stored for later retrieval in another mode. */
    AP_MODE_SPECULATIVE,
    /** The filter read should be exhaustive and read until it can not
     *  read any more.
     *  Use this mode with extreme caution.
     */
    AP_MODE_EXHAUSTIVE,
    /** The filter should initialize the connection if needed,
     *  NNTP or FTP over SSL for example.
     */
    AP_MODE_INIT
} ap_input_mode_t;
```

Clearly not all of these are relevant to every filter. A filter that cannot support the mode it is called with is inappropriate, and should remove itself from the filter chain. A filter may often call the next filter using the same mode it was called with, but this is not always appropriate, and a filter is free to do otherwise.

The **block** argument is one of APR_BLOCK_READ or APR_NONBLOCK_READ, and determines whether the filter should block if necessary. **readbytes** is an indication of the number of bytes the filter should read. It is not a hard limit, as a filter that increases data size may read the same volume of data but have more to return.

Filter Objects

The filter object (along with others discussed in this chapter) is defined in util_filter.h. Note that the final comment is not strictly accurate - it should read connection-level filtering for input filtering:

```
/**
 * The representation of a filter chain.  Each request has a list
 * of these structures which are called in turn to filter the data.  Sub
 * requests get an exact copy of the main requests filter chain.
 */
struct ap_filter_t {
    /** The internal representation of this filter.  This includes
```

```
     *  the filter's name, type, and the actual function pointer.
     */
    ap_filter_rec_t *frec;

    /** A place to store any data associated with the current filter */
    void *ctx;

    /** The next filter in the chain */
    ap_filter_t *next;

    /** The request_rec associated with the current filter.  If a sub-
request
     *  adds filters, then the sub-request is the request associated
with the
     *  filter.
     */
    request_rec *r;

    /** The conn_rec associated with the current filter.  This is
analogous
     *  to the request_rec, except that it is used for input filtering.
     */
    conn_rec *c;
};
```

The fields that most filter modules will use here are `ctx`, to store application data for the filter between calls, and the `request_rec` to access all the normal request data (in the case of connection-level filters, there is no valid request_rec field, and the conn_rec serves a similar purpose). Also the next field will be used to push data to the next filter in the output chain, or pull data from the next filter in the input chain.

The `frec` field can normally be treated as opaque by applications, but is necessary to our understanding of filter internals.

Here it is:

```
/**
 * This structure is used for recording information about the
 * registered filters. It associates a name with the filter's callback
 * and filter type.
 *
 * At the moment, these are simply linked in a chain, so a ->next
pointer
 * is available.
 *
 * It is used for any filter that can be inserted in the filter chain.
 * This may be either a httpd-2.0 filter or a mod_filter harness.
 * In the latter case it contains provider and protocol information.
 * In the former case, the new fields (from providers) are ignored.
 */
struct ap_filter_rec_t {
    /** The registered name for this filter */
    const char *name;
```

```
    /** The function to call when this filter is invoked. */
    ap_filter_func filter_func;

    /** The function to call before the handlers are invoked. Notice
     * that this function is called only for filters participating in
     * the http protocol. Filters for other protocols are to be
     * initialized by the protocols themselves.
     */
    ap_init_filter_func filter_init_func;

    /** The type of filter, either AP_FTYPE_CONTENT or
AP_FTYPE_CONNECTION.
     * An AP_FTYPE_CONTENT filter modifies the data based on information
     * found in the content.  An AP_FTYPE_CONNECTION filter modifies the
     * data based on the type of connection.
     */
    ap_filter_type ftype;

    /** The next filter_rec in the list */
    struct ap_filter_rec_t *next;

    /** Providers for this filter */
    ap_filter_provider_t *providers;

    /** Trace level for this filter */
    int debug;

    /** Protocol flags for this filter */
    unsigned int proto_flags;
};
```

The name is just an identifier for filter configuration, which will be discussed in Chapter 8. The `filter_func` is the main callback already introduced, and the `filter_init_func` is a seldom-used initialisation function called when the filter is inserted and before the first data are available.

The final three fields were introduced with the smart filtering architecture in Apache 2.1.

Filter I/O

Data passes through the filter chain on the bucket brigade. There are several strategies for dealing with the data in a filter:

- If the filter merely looks at the data but doesn't change anything, it can pass the brigade on as-is.
- If the filter makes changes but preserve content length (e.g. a case filter for ASCII text), it can replace bytes in-place.
- A filter that passes through most of the data intact but makes some changes can

edit the data by direct bucket manipulation.

- A filter that completely transforms the data will often need to replace the data completely, by creating an entirely new brigade and populating it. It can do that either directly, or using stdio-like functions. There are two families of stdio-like functions. APR provides apr_brigade_puts/`apr_brigade_write`/etc, while `util_filter` provides `ap_fwrite`/`ap_fputs`/`ap_fprintf`/etc.

Managing I/O is at the heart of filtering, and will be demonstrated at length when we develop example filters later in this chapter.

Smart Filtering in Apache 2.1

The original Apache 2.0 filter architecture presents problems when used in with unknown content; either in a proxy or with a local handler that generates different content types to order. The basic difficulty lies in the Apache configuration. Content filters need to be applied conditionally: for example, we don't want to pass images through an HTML filter. The generic configuration directives for filters are:

- `SetOutputFilter` (unconditionally inserts a filter)
- `AddOutputFilter`, `RemoveOutputFilter` (inserts or removes a filter based on "extension")
- `AddOutputFilterByType` (inserts a filter based on Content Type)

In the case of a proxy, extensions are meaningless, as we cannot know what conventions an origin server might adopt. Neither is `AddOutputFilterByType` (nor its hypothetical siblings such as `AddOutputFilterByEncoding` or `AddOutputFilterByLanguage`) any use, because the response headers from the proxy are unknown at the time the filter is inserted and initialised. So we have to resort to the unsatisfactory hack of inserting a filter unconditionally, checking the response headers from the proxy, and then having the filter remove itself where appropriate. Examples of filters that will do this are `mod_deflate`, `mod_xmlns`, `mod_accessibility` and `mod_proxy_html`.

Pre- and Post-Processing

As with an origin server, it may be necessary to preprocess data before the main content-transforming filter, and/or postprocess afterwards. For example, when dealing with gzipped content, we need to uncompress it for processing and re-compress the processed data. Similarly, in an image-processing filter, we need to decode the original image format and re-encode the processed data.

This may involve more than one phase. For example, when filtering text, we may need to

both to uncompress gzipped data and transcode the character set before the main filter.

So, potentially we have a multiplicity of filters: transformation filters, together with pre- and post-processing for different content types and encodings. To repeat the hack of having each filter inserted and determining whether to run or remove itself in such a setup goes beyond simple inelegance and into the absurd. An alternative architecture is required.

mod_filter

The solution to this is implemented in Apache 2.1 in `mod_filter`. `mod_filter` works by introducing indirection into the filter chain. Instead of inserting filters in the chain, we insert a filter harness, which in turn dispatches conditionally to a filter provider. Any content filter may be used as a provider to `mod_filter`; no change to existing filter modules is required (although it may be possible to simplify them). There can be multiple providers for one filter, but no more than one provider will run for any single request.

A filter chain comprises any number of instances of the filter harness, each of which may have any number of providers. A special case is that of a single provider with unconditional dispatch: this is equivalent to inserting the provider filter directly into the chain.

`mod_filter` is only implemented for output filters: the configuration problems it deals with are not relevant to the input chain. And although it can be applied anywhere in the output filter chain, it is only really relevant to content (application) filters. Neither the old nor the new filter configuration directives are generally used for the outer filters: for example, SSL (both input and output) is activated by `mod_ssl`'s own configuration directives instead.

Protocol Handling

In Apache 2.0, each filter is responsible for ensuring that whatever changes it makes are correctly represented in the HTTP response headers, and that it does not run when it would make an illegal change. This imposes a burden on filter authors to reimplement some common functionality in every filter. For example,

- Many filters will change the content, invalidating existing content tags, checksums, hashes, and lengths.
- Filters that require an entire, unbroken response in input need to ensure they don't get byteranges from a backend.
- Filters that transform output in a filter need to ensure they don't violate a Cache-Control: no-transform header from the backend.
- Filters may make responses uncachable.

`mod_filter` aims to offer generic handling of these details of filter implementation, reducing the complexity required of content filter modules. At the same time, mod_filter should not interfere with a filter that wants to handle all aspects of the protocol. By

default (i.e. in the absence of any explicit instructions), mod_filter will leave the headers untouched.

Thus, filter developers have two options. If we handle all protocol considerations within our filter, then it will work with any Apache 2.x. However, if we are not concerned with backwards compatibility, we can dispense with this and leave it to `mod_filter`. One danger with this approach, however, is that (at the time of writing) `mod_filter`'s protocol handling is considered experimental: we must be prepared to test it, and if necessary to maintain it in future.

The API for filter protocol handling is simple. The protocol is defined in a bitfield (unsigned int), which is passed as an argument when the filter is registered (in function `ap_register_output_filter_protocol`), or later in function `ap_filter_protocol`.

Currently supported bits are:

- **`AP_FILTER_PROTO_CHANGE`** - filter changes the contents (thus invalidating checksums, etc.)
- **`AP_FILTER_PROTO_CHANGE_LENGTH`** - filter changes the length of the contents
- **`AP_FILTER_PROTO_NO_BYTERANGE`** - filter requires complete input and cannot work on byte ranges
- **`AP_FILTER_PROTO_NO_PROXY`** - filter cannot run in a proxy (e.g. Makes changes that would violate mandatory HTTP requirements in a proxy)
- **`AP_FILTER_PROTO_NO_CACHE`** - filter output is non-cacheable, even if the input was cacheable
- **`AP_FILTER_PROTO_TRANSFORM`** - filter is incompatible with "Cache-Control: no-transform"

Example: Filtering text by direct manipulation of buckets.

Our first filter example is a simple filter that manipulates buckets directly. It passes data straight through, but transforms it by manipulating pointers.

The purpose of this module is to display plain text files as HTML, prettified and having a site header and footer. So what the module has to do is:

- add a header at the top
- add a footer at the bottom
- escape the text as required by HTML.

The header and footer are files specified by the system administrator responsible for the site.

Bucket functions

First, we introduce two functions to deal with the data insertions: one for the files, one for the simple entity replacements.

Creating a File bucket requires an open filehandle and a byte range within the file. Since we're transmitting the entire file, we just stat its size to set the byte range. We open it with a shared lock and with sendfile enabled for maximum performance.

```
static apr_bucket* txt_file_bucket(request_rec* r, const char* fname) {
  apr_file_t* file = NULL ;
  apr_finfo_t finfo ;
  if ( apr_stat(&finfo, fname, APR_FINFO_SIZE, r->pool)
            != APR_SUCCESS ) {
    return NULL ;
  }
  if ( apr_file_open(&file, fname,
            APR_READ|APR_SHARELOCK|APR_SENDFILE_ENABLED,
            APR_OS_DEFAULT, r->pool ) != APR_SUCCESS ) {
    return NULL ;
  }
  if ( ! file ) {
    return NULL ;
  }
  return apr_bucket_file_create(file, 0, finfo.size, r->pool,
        r->connection->bucket_alloc) ;
}
```

Creating the simple text replacements, we can just make a bucket of a string. By making the strings static, we avoid having to worry about their lifetime.

```
static apr_bucket* txt_esc(char c, apr_bucket_alloc_t* alloc ) {
  static const char* lt = "&lt;" ;
  static const char* gt = "&gt;" ;
  static const char* amp = "&amp;" ;
  static const char* quot = "&quot;" ;
  switch (c) {
    case '<': return apr_bucket_immortal_create(lt, 4, alloc) ;
    case '>': return apr_bucket_immortal_create(gt, 4, alloc) ;
    case '&': return apr_bucket_immortal_create(amp, 5, alloc) ;
    case '"': return apr_bucket_immortal_create(quot, 6, alloc) ;
    default: return NULL ;      /* shut compilers up */
  }
}
```

The Filter

Now the main filter itself is broadly straightforward, but there are a number of interesting and unexpected points to consider. Since this is a little longer than the above utility

functions, we'll comment it inline instead.

The `txt_cfg` struct used here is the module's configuration, and just contains filenames for the header and footer. Since that may be used concurrently by many threads, we access it as read-only and use a second - private - `txt_ctxt` object to maintain our own state:

```
typedef struct txt_cfg {
  const char* head ;
  const char* foot ;
} txt_cfg ;
typedef struct txt_ctxt {
  int state ;
  const char* head ;
  const char* foot ;
} txt_ctxt ;

static int txt_filter(ap_filter_t* f, apr_bucket_brigade* bb) {
  apr_bucket* b ;
  txt_ctxt* ctxt = f->ctx ;

  if ( ctxt == NULL ) {
    txt_cfg* cfg = ap_get_module_config(r->per_dir_config, &txt_module)
;
    ctxt = f->ctx = apr_pcalloc(f->r->pool, sizeof(txt_ctxt)) ;
    ctxt->head = cfg->head ;
    ctxt->foot = cfg->foot ;
  }
```

Main Loop: This construct is typical for iterating over the incoming data

```
  for ( b = APR_BRIGADE_FIRST(bb);
        b != APR_BRIGADE_SENTINEL(bb);
        b = APR_BUCKET_NEXT(b) ) {

    const char* buf ;
    size_t bytes ;
```

As in any filter, we need to check for EOS. When we encounter it, we insert the footer in front of it. We shouldn't get more than one EOS, but just in case we do we'll note having inserted the footer. This creates error-tolerance.

```
    if ( APR_BUCKET_IS_EOS(b) ) {
      /* end of input file - insert footer if any */
      if ( ctxt->foot && ! (ctxt->state & TXT_FOOT ) ) {
        ctxt->state |= TXT_FOOT ;
        APR_BUCKET_INSERT_BEFORE(b, txt_file_bucket(r, ctxt->foot));
      }
```

The main case is a bucket containing data, We can get it as a simple buffer with its size

in bytes:

```
    } else if ( apr_bucket_read(b, &buf, &bytes, APR_BLOCK_READ)
        == APR_SUCCESS ) {
      /* We have a bucket full of text.  Just escape it
       * where necessary
       */
      size_t count = 0 ;
      const char* p = buf ;
```

Now we can search for characters that need replacing, and replace them

```
      while ( count < bytes ) {
        size_t sz = strcspn(p, "<>&\"") ;
        count += sz ;
```

Here comes the tricky bit: replacing a single character inline.

```
        if ( count < bytes ) {

          /* split off buffer at the character */
          apr_bucket_split(b, sz) ;

          /* skip over the before-buffer (where nothing changes) */
          b = APR_BUCKET_NEXT(b) ;

          /* insert the replacement for the character */
          APR_BUCKET_INSERT_BEFORE(b, txt_esc(p[sz],
                f->r->connection->bucket_alloc)) ;

          /* split off the char we just replaced */
          apr_bucket_split(b, 1) ;

          /* ... and remove it */
          APR_BUCKET_REMOVE(b) ;

          /* Move cursor on to what-remains, so it stays
           * in sequence with the main loop.
           */
          b = APR_BUCKET_NEXT(b) ;
          /* Finally, increment our counters */
          count += 1 ;
          p += sz + 1 ;
        }
      }
    }
  }
```

Now we insert the Header if it hasn't already been inserted.

Note that:

- this has to come after the main loop – otherwise, the header itself would get parsed and HTML-escaped
- it works because we can insert a bucket anywhere in the brigade, and in this case put it at the head
- as with the footer, we save state to avoid inserting it more than once.

```
  if ( ctxt->head && ! (ctxt->state & TXT_HEAD ) ) {
    ctxt->state |= TXT_HEAD ;
    APR_BRIGADE_INSERT_HEAD(bb, txt_file_bucket(r, ctxt->head));
  }
```

Note that we created a new bucket every time we replaced a character. Couldn't we have prepared four buckets in advance - one for each of the characters to be replaced - and then re-used them whenever the character occurred?

The problem here is that each bucket is linked to its neighbours. So if we re-use the same bucket, we lose the links, so that the brigade now jumps over any data between the two instances of it. Hence we do need a new bucket every time. That means this technique becomes inefficient when a high proportion of input data has to be changed.

Now we've finished manipulating data, we just pass it down the filter chain.

```
  return ap_pass_brigade(f->next, bb) ;
}
```

`mod_txt` was written one idle afternoon, after someone had asked on IRC whether such a module existed. It seemed such an obvious thing to do, and a great example to use here. Working with buckets and brigades is one of the hardest parts of the Apache API, and it needs such a simple demonstrator module!

Example 2: an input filter and the PULL API

Example 3: Using external libraries and the stdio-like API