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. In this chapter, we will take a detailed look at the filter chain, and develop several illustrative filter modules.

Before going into details, let's review a few basics. In Chapter 2, we saw that filters 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.

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

### 6.1: 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.

### 6.2: 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, that process document contents within a request. These are the filters most commonly relevant to applications programming.
• Protocol filters, that 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 that 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 contents 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 byteranges 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 (HTTP 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).

The examples presented in detail in this chapter are all content filters, of types AP_FTYPE_RESOURCE and AP_FTYPE_CONTENT_SET. The essential principles of writing a filter are no different for other filters, with a few exceptions:

6.2.1: Protocol

It is the responsibility of protocol filters to convert the input data from a byte stream to an HTTP request, and the output data back again. So the input protocol filter populates r->headers_in, while the output protocol filter converts r->headers_out to a byte stream.

6.2.2: Headers and Entities

Filters of types AP_FTYPE_RESOURCE and AP_FTYPE_CONTENT_SET will only ever see an HTTP request or response entity (body). The request and response headers may be accessed through the r->headers_in and r->headers_out tables respectively.

6.2.2.1: Caution

r->headers_out will be converted to a set of response headers the first time the output HTTP protocol filter is invoked. Any changes made later will have no effect!

By contrast, filters outside the protocol layer will not have r->headers_in and headers_out, but just a stream of bytes or lines. In fact they won't have a request_rec at all, and can't use fields from f->r.

6.2.3: Metadata Buckets

Inner filters will rarely see metadata buckets (except for EOS) and can commonly ignore them, although flush buckets should technically be flushed immediately. Outer filters should respect all metadata buckets.

6.3: 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 (8Kb) buffer. Our filter should process the data, and send (push) a chunk to the next filter in the chain.

Copyright © 2005-6 Nick Kew (niq@apache.org)
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.

### 6.3.1: 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. And to run 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 deal with that. Filters should always endeavour 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. Amongst 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.

### 6.4: 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.

Copyright © 2005-6 Nick Kew (niq@apache.org)
6.4.1: Output Filters

The callback prototype for output filters is

```c
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.

6.4.2: Input Filters

The input filter callback is a little more complex:

```c
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 our 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:

```c
typedef enum {
    AP_MODE_READBYTES, // The filter should return at most readbytes data. */
    AP_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).
    AP_MODE_EATCRLF,  // The filter should implicitly eat any CRLF pairs that it sees. */
    AP_MODE_SPECULATIVE, // The filter read should be treated as speculative and any
                         //   data should be stored for later retrieval in another mode. */
    AP_MODE_EXHAUSTIVE, // The filter read should be exhaustive and read until it can not
                         //   read any more.
                         //   Use this mode with extreme caution.
} apr_input_mode_t;
```

Copyright © 2005-6 Nick Kew (niq@apache.org)
NNTP or FTP over SSL for example.
*/

AP_MODE_INIT
}

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.

6.5: 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.
     */

Copyright © 2005-6 Nick Kew (niq@apache.org)
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:

```c
struct ap_filter_rec_t {
    const char *name;
    ap_filter_func filter_func;
    ap_init_filter_func filter_init_func;
    ap_filter_type ftype;
    struct ap_filter_rec_t *next;
};
```

Copyright © 2005-6 Nick Kew (niq@apache.org)
The name is just an identifier for filter configuration, which will be discussed in Chapter 8. The filter_func is the main callback we've 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.

### 6.6: 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.

The key concepts in managing data are the bucket and brigade. We have encountered them in chapter 3 and elsewhere. In this chapter, our examples will explore them in depth.

### 6.7: 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 in Apache 2.0 are:

- **SetOutputFilter**: Unconditionally insert a filter.
- **AddOutputFilter, RemoveOutputFilter**: Insert or remove a filter based on "extension".
- **AddOutputFilterByType**: Insert a filter based on Content Type. This is implemented in the `ap_set_content_type` function, and has problematic side-effects.

In the case of a proxy, extensions are meaningless, as we cannot know what conventions an origin server might adopt. Likewise, when the server generates content dynamically – or filters it dynamically with, for example, XSLT – it can be hard or impossible to configure the filter chain using the above. 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`.

### 6.7.1.1: 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 large 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.

### 6.7.2: 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.

6.7.3: Filter self-configuration

In addition to the standard filter configuration provided by the core and mod_filter, a filter may also be self-configuring:

6.7.3.1: The insert_filter hook

A hook for inserting filters is provided in the content handling phase of request processing, immediately before the content generator. This is used by mod_filter to insert the filter harness for dynamically-configured filters, and may be used by other modules. Here is mod_filter's hook, which inserts all entries in the FilterChain in order:

```c
static void filter_insert(request_rec *r)
{
    mod_filter_chain *p;
    ap_filter_rec_t *filter;
    mod_filter_cfg *cfg = ap_get_module_config(r->per_dir_config, &filter_module);

    int ranges = 1;
    mod_filter_ctx *ctx = apr_palloc(r->pool, sizeof(mod_filter_ctx));
    ap_set_module_config(r->request_config, &filter_module, ctx);

    for (p = cfg->chain; p; p = p->next) {
        filter = apr_hash_get(cfg->live_filters, p->fname, APR_HASH_KEY_STRING);
        ap_add_output_filter_handle(filter, NULL, r, r->connection);

        if (ranges && (filter->proto_flags
            & (AP_FILTER_PROTO_NO_BYTE_RANGE
                | AP_FILTER_PROTO_CHANGE_LENGTH))) {
            ctx->range = apr_table_get(r->headers_in, "Range");
            apr_table_unset(r->headers_in, "Range");
            ranges = 0;
        }
    }
    return;
}
```

which is hooked in using:

```c
static void filter_hooks(apr_pool_t *pool)
{
    ap_hook_insert_filter(filter_insert, NULL, NULL, APR_HOOK_MIDDLE);
}
```

When using this hook, modules should consider carefully where their filter should be inserted into the chain. They can explicitly run their filter_insert before or after mod_filter, to ensure they run before or after configured filters of the same type. For

Copyright © 2005-6 Nick Kew (niq@apache.org)
example, to run a filter before any dynamically-configured filter, you could either use
APR_HOOK_FIRST or an explicit ordering that puts mod_filter after your module:

```c
static void my_hooks(apr_pool_t *pool)
{
    static const char * const aszSucc[]={ "mod_filter.c", NULL };
    ap_hook_insert_filter(my_insert, NULL, aszSucc, APR_HOOK_MIDDLE);
}
```

### 6.7.3.2: Environment Variables and Notes

Another strategy open to filters is to examine the request details themselves to determine
whether to run or uninsert themselves. This was widely used in Apache 2.0 before
mod_filter became available, and may still be required where the configuration is more
complex than can be delegated to mod_filter.

An example is the compression filter in mod_deflate. Since it is older than mod_filter, it
provides explicitly for control by environment variables (which could either be set in
httpd.conf with SetEnv or similar or by another module). But it also provides some more
complex logic that is better handled internally than in httpd.conf. Relevant code from
mod_deflate is:

```c
/* only work on main request/no subrequests */
if (!ap_is_initial_req(r)) {
    ap_remove_output_filter(f);
    return ap_pass_brigade(f->next, bb);
}

/* some browsers might have problems, so set no-gzip
 * (with browsermatch) for them */
if (apr_table_get(r->subprocess_env, "no-gzip")) {
    ap_remove_output_filter(f);
    return ap_pass_brigade(f->next, bb);
}
/* Let's see what our current Content-Encoding is.
 * If it's already encoded, don't compress again.
 * (We could, but let's not.) */
encoding = apr_table_get(r->headers_out, "Content-Encoding");
/* CHOPPED for brevity */
/* Even if we don't accept this request based on it not having
 * the Accept-Encoding, we need to note that we were looking
 * for this header and downstream proxies should be aware of
 * that. */
apr_table_mergen(r->headers_out, "Vary", "Accept-Encoding");
/* force-gzip will just force it out regardless if the browser
 * can actually do anything with it. */
if (!apr_table_get(r->subprocess_env, "force-gzip")) {
    /* DELETED for brevity —

Copyright © 2005-6 Nick Kew (niq@apache.org)
* remove the filter if the browser doesn't accept gzip */

/* For a 304 or 204 response there is no entity included in
* the response and hence nothing to deflate. */
if (r->status == HTTP_NOT_MODIFIED
     || r->status == HTTP_NO_CONTENT) {
    ap_remove_output_filter(f);
    return ap_pass_brigade(f->next, bb);
}

/* if we pass all those checks, we will compress it */

6.7.4: 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 re-implement
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 proxy need to ensure they don't violate a Cache-
  Control: no-transform header from the backend.
- Filters may make responses uncacheable.

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 you handle all protocol considerations within
your filter, then it will work with any Apache 2.x. However, if you are not concerned
with backwards compatibility, you can dispense with this and leave it to mod_filter. If
you take advantage of this, please note that (at the time of writing) mod_filter's protocol
handling is considered experimental: you should 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

Copyright © 2005-6 Nick Kew (niq@apache.org)
AP_FILTER_PROTO_NO_BYTERANGE - filter requires complete input and cannot work on byteranges

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"

6.8: 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.

6.8.1: Bucket functions

Firstly 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.

```c
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 file;
}
```

Copyright © 2005-6 Nick Kew (niq@apache.org)
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.

```c
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 */
    }
}
```

### 6.8.2: 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:

```c
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_palloc(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. That means we're being error-tolerant.

    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 = strncspn(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) */

Copyright © 2005-6 Nick Kew (niq@apache.org)
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:

(a) This has to come after the main loop, to avoid the header itself getting parsed
    and HTML-escaped.

(b) It works because we can insert a bucket anywhere in the brigade, and in this case
    put it at the head.

(c) 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);
}

Copyright © 2005-6 Nick Kew (niq@apache.org)
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!

6.9: Complex Parsing

A slightly more complex task is to deal with parsing data where a pattern we want to match in the parse may span more than one bucket, or even more than one call to the filter function. Even a simple search-and-replace filter that matches words will need to save some context between calls, to avoid missing words that are split up. As an aside, this is a nontrivial task in general: witness the number of spam messages that get past spam filters by breaking up words that trigger detection.

The simplest way to deal with this is to collect the entire document body into memory. However, this is inefficient: it breaks Apache's pipelining architecture, and scales very badly as document size grows. We should avoid it wherever possible.

A module that faces exactly this task is mod_line_edit, a filter that provides text search-and-replace based on string or regular expression matching. This module works by rearranging its input into complete lines of text before editing it (the definition of a line is somewhat flexible, but it defaults to parsing normal lines of text). Let's look at it for some more advanced bucket manipulation. For the purposes of this discussion, we'll present a simplified version that only supports the unix “\n” lineend character. The guiding principle of this filter is to manipulate buckets and brigades at will (pointers are cheap), but only move or copy data where it's unavoidable. This demonstrates some new techniques:

• Creating new bucket brigades for our own purposes.
• Saving data between calls to the filter.
• Flattening data into a contiguous buffer.

This means rearranging any lines spanning more than one bucket, and saving any partial lines between calls to the filter.

    /* Filter to ensure we have no mid-line breaks that might be in the 
    * middle of a search string causing us to miss it! At the same 
    * time we split into lines to avoid pattern-matching over big 
    * chunks of memory. 
    */

    /* We're parsing into lines, so let's have a brigade to put them in */
    apr_bucket_brigade* bbline
        = apr_brigade_create(f->r->pool, f->c->bucket_alloc);

    /* We're saving any incomplete lines, so we store it on the filter 
    * ctx. We use a brigade so we don't have to touch the data 
    */
    line_edit_ctx* ctx = f->ctx;

Copyright © 2005-6 Nick Kew (niq@apache.org)
if (ctx == NULL) {
    ctx = f->ctx = apr_palloc(f->r->pool, sizeof(line_edit_ctx));
    ctx->bbsave = apr_brigade_create(f->r->pool, f->c->bucket_alloc);
}

/* Now the main loop over the input data */
b = APR_BUCKET_FIRST(bb);
while (b != APR_BRIGADE_SENTINEL(bb)) {
    if (!APR_BUCKET_IS_METADATA(b)) {
        if (apr_bucket_read(b, &buf, &bytes, APR_BLOCK_READ) == APR_SUCCESS) {
            /* parse loop */
            /* See if there's a lineend in the bucket (simplified)! */
            le = memchr(buf, '\n', bytes);
            if (le != NULL) {
                /* There is a lineend. Extract what's before it */
                offs = ((unsigned int)le - (unsigned int)buf)/sizeof(char) + 1;
                apr_bucket_split(b, offs);
                /* Increment pointers for when we iterate the parse loop */
                bytes -= offs;
                buf += offs;
                b1 = APR_BUCKET_NEXT(b);
            } else {
                /* no lineend found. Remember the dangling content */
                APR_BUCKET_REMOVE(b);
                APR_BUCKET_REMOVE(b);
                bytes = 0;
            }
        } else {
            /* no lineend found. Remember the dangling content */
            APR_BUCKET_REMOVE(b);
        }
    } /* parse loop: while bytes > 0 */
}

/* Is there any previous unterminated content? */
if (!APR_Brigade_EMPTY(ctx->bbsave)) {
    /* Assemble a complete line from the bits */
    rv = apr_brigade_pflatten(ctx->bbsave, &fbuf, &fbytes, f->r->pool);
    /* make b a new bucket of the flattened stuff */
    b = apr_bucket_pool_create(fbuf, fbytes, f->r->pool, f->r->connection->bucket_alloc);
    /* bbsave has been consumed, so clear it */
    apr_brigade_cleanup(ctx->bbsave);
} /* b now contains exactly one line */
/* Insert it into the lines brigade, and move the pointer */
APR_Brigade_INSERT_TAIL(bbline, b);
else {
    /* no lineend found. Remember the dangling content */
    APR_BUCKET_REMOVE(b);
    APR_Brigade_INSERT_TAIL(ctx->bbsave, b);
    bytes = 0;
}
} else if ( APR_BUCKET_IS_EOS(b) ) {
    /* If there's data to pass, send it in one bucket */
    if ( !APR_BRIGADE_EMPTY(ctx->bbsave) ) {
        rv = apr_brigade_pflatten(ctx->bbsave, &fbuf,
                                   &fbytes, f->r->pool);
        bl = apr_bucket_pool_create(fbuf, fbytes, f->r->pool,
                                     f->r->connection->bucket_alloc);
        APR_BRIGADE_INSERT_TAIL(bbline, bl);
    }
    apr_brigade_cleanup(ctx->bbsave);
    /* start again rather than segfault if a seriously buggy
     * filter in front of us sent a bogus EOS
     */
    f->ctx = NULL;
    /* move the EOS to the new brigade */
    APR_BUCKET_REMOVE(b);
    APR_BRIGADE_INSERT_TAIL(bbline, b);
} else { /* neither data nor EOS */
    /* chop flush or unknown metadata bucket types */
    apr_bucket_delete(b);
}
/* reset pointer to what's left (since we're not in a for-loop) */
b = APR_BRIGADE_FIRST(bb);
/* Now we have a bunch of complete lines in bbline.
 * If we saw an EOS, we also have that
 * and a possibly-unterminated last line
 */
/* we can either process them here, or pass them to another filter
 * that requires its input to be in complete lines
 */
/* now pass it down the chain */
rv = ap_pass_brigade(f->next, bbline);
/* if we have leftover data, don't risk it going out of scope */
for ( b = APR_BRIGADE_FIRST(ctx->bbsave);
      b != APR_BRIGADE_SENTINEL(ctx->bbsave);
      b = APR_BUCKET_NEXT(b)) {
    apr_bucket_setaside(b, f->r->pool);
}
return rv ;
}
6.10: Filtering through an existing parser

An alternative to parsing data ourselves is to feed it to an existing parser, typically from a third-party library. This author's various markup-aware modules, including his most popular module, mod_proxy_html, work like this; the filter simply reads each bucket and passes it to the library. This works well because the library itself supports processing data in arbitrary chunks, so we don't have to worry about troublesome breaks in the input data disrupting the parse. Here's an example from mod_xmlns, which uses the expat library to parse XML. The core filter here is very simple, so we'll give it in full:

```c
static int xmlns_filter(ap_filter_t* f, apr_bucket_brigade* bb) {
    apr_bucket* b ;
    const char* buf = 0 ;
    apr_size_t bytes = 0 ;

    xmlns_ctx* ctxt = (xmlns_ctx*)f->ctx ;
    if ( ! ctxt ) {
        xmlns_filter_init(f) ;
    }
    if ( ctxt , ! ctxt )
        return ap_pass_brigade(f->next, bb) ;

    for ( b = APR_BRIGADE_FIRST(bb) ;
            b != APR_BRIGADE_SENTINEL(bb) ;
            b = APR_BUCKET_NEXT(b) ) {
        if ( ( XML_Parse(ctxt->parser, buf, 0, 1) != XML_STATUS_OK ) ) {
            enum XML_Error err = XML_GetErrorCode(ctxt->parser) ;
            const XML_LChar* msg = XML_ErrorString(err) ;
            ap_log_rerror(APLOG_MARK, APLOG_ERR, 0, f->r,
                "Endparse Error %d: %s", err, msg) ;
        } else if ( APR_BUCKET_IS_FLUSH(b) ) {
            APR_BRIGADE_INSERT_TAIL(ctxt->public->bb,
                apr_bucket_flush_create(ctxt->public->bb->bucket_alloc) ) ;
            ap_pass_brigade(ctxt->public->f->next, ctxt->public->bb) ;
        } else if ( apr_bucket_read(b, &buf, &bytes, APR_BLOCK_READ)
                == APR_SUCCESS ) {
            if ( ( XML_Parse(ctxt->parser, buf, bytes, 0) != XML_STATUS_OK ) ) {
                enum XML_Error err = XML_GetErrorCode(ctxt->parser) ;
                const XML_LChar* msg = XML_ErrorString(err) ;
                ap_log_rerror(APLOG_MARK, APLOG_ERR, 0, f->r,
                    "Parse Error %d: %s", err, msg) ;
            } else {
                ap_log_rerror(APLOG_MARK, APLOG_ERR, 0, f->r,
                    "Error in bucket read") ;
            } } else {
            ap_log_rerror(APLOG_MARK, APLOG_ERR, 0, f->r,
                "Error in bucket read") ;
        }
    }
    return APR_SUCCESS ;
}
```

Copyright © 2005-6 Nick Kew (niq@apache.org)
As we can see, this takes the form of the now-familiar loop over input buckets, retrieving the bucket data (where applicable) into a buffer, and making a special case of EOS. But instead of parsing the data ourselves, we feed it to expat's chunk-parsing function XML_Parse. And we don't pass anything at all on to the next brigade! So how does that work?

When we use the XML parser here, we basically lose the input data altogether. Our module must of course set up handlers with the library, but these receive XML events such as startElement, endElement, characters, cdata and comment, rather than our input data. It has no option but to generate a new output stream from scratch. And of course the library has no notion of Apache concepts such as buckets, brigades, requests, or filters, so whatever we do has to be done from scratch.

Now, we could of course create a new bucket brigade, and populate it with new buckets for each XML event. But this is not an attractive option, for several reasons:

- Most markup events – for example, elements and attributes – involve generating just a few bytes of output per event. Creating new buckets for every few bytes becomes inefficient.
- We have no natural point at which to pass a brigade down to the next filter. We either have to break streaming, or do extra work to manage this ourselves.
- Creating buckets is an unduly awkward way to perform simple I/O.

This final reason alone could be considered compelling!

6.11: Stdio-Like Filter I/O

Fortunately, the filter API provides an alternative, stdio-like way to write data and pass it down the chain. We still need to create a bucket brigade for output, but all we need to do with it is to pass it to the stdio-like calls, along with the filter we're writing to, in the manner of a file descriptor. The stdio-like functions are defined in util_filter.h:

- `ap_fflush(f, bb)`
- `ap_fwrite(f, bb, buf, nbytes)`
- `ap_fputs(f, bb, str)`
- `ap_fputc(f, bb, c)`
- `ap_fputstrs(f, bb, ...) /* a NULL-terminated list of strings */`
- `ap_fprintf(f, bb, fmt, ...)`

Internally, the first time you use any of these calls, Apache will create a Heap bucket, normally of size 8Kb, and write your data to it. Subsequent writes append to the heap while there is sufficient space. When the heap space is exhausted, a second bucket of type transient is appended containing the data over and above the size of the heap bucket,
and the two are flushed down the chain. This is the same as the ap_rwrite/etc stdio-like API, and is the reason for the 8Kb default stream buffer size seen by many applications.

So, how does mod_xmlns uses stdio-like I/O? Since it's a SAX2 filter, it has to generate all output from the SAX2 event callbacks. Lets have a look at the essentials of some of the callbacks. Our examples use two define s, which are respectively the next filter in the chain, and our output bucket brigade:

```c
#define F ((xmlns_ctx*)ctx)->public->f->next
#define BB ((xmlns_ctx*)ctx)->public->bb
```

The simplest is the default callback, an expat callback that gets any data not passed to any other callback. Since we're registering callbacks for everything we need to process, anything passed to the default callback goes straight to the output:

```c
static void xdefault(void* ctx, const XML_Char* buf, int len) {
    ap_fwrite(F, BB, buf, len);
}
```

The most complex handler is that for the startElement event. We'll quote it in full to show use of the API to simplify a lot of small, fiddly writes.

```c
static void xstartElement(void* ctx, const XML_Char* name,
                          const XML_Char** atts) {
    const XML_Char** atts);
    parsedname name3; /* namespace, prefix, name */
    xmlns_active* ns;

    xmlns Parsename(name, &name3); /* parse the name expat gave us */

This next section is the core of the module. mod_xmlns exports an API for other modules to register handlers for namespace events, of which the most important is startElement. So lookup_ns will return a non-null value if and only if another module has registered a handler for the namespace and it is marked as active.

```c
/* If a handler for this namespace is active, we dispatch to it */
ns = lookup_ns((xmlns_ctx*)ctx, &name3);
if ( ns && ns->handler->StartElement ) {
    if ( ns->handler->StartElement((xmlns_public*)CTX->public, &name3,
    (const xmlns_attr_t*)atts) != DECLINED )
        return ;
} else return DECLINED;
```

The remainder of this function is just default behavior that reconstructs the element as-is when no handler has handled it. It serves to demonstrate filter stdio-style output.

```c
/* Default: either no handler, or it returned 0 to ask us to
 * produce default output */
ap_fputc(F, BB, '<');
if ( name3.nparts == 3 ) { /* it's prefix:element */
ap_fwrite(F, BB, name3.prefix, name3.prefixlen);
ap_fputc(F, BB, ':');
}
ap_fwrite(F, BB, name3.elt, name3.eltlen);
```

Copyright © 2005-6 Nick Kew (niq@apache.org)
if ( ns && ns->newns ) {
    if ( name3.nparts == 3 ) {
        ap_fputs(F, BB, " xmlns:")
        ap_fwrite(F, BB, name3.prefix, name3.prefixlen);
        ap_fputs(F, BB, "=");
        ap_fwrite(F, BB, name3.ns, name3.nslen);
        ap_fputc(F, BB, '"');
    } else if ( name3.nparts == 2 ) {
        ap_fputs(F, BB, " xmlns=");
        ap_fwrite(F, BB, name3.ns, name3.nslen);
        ap_fputc(F, BB, '"');
    }
    ns->newns = 0;
}

/* Now output any attributes */
if ( atts ) {
    const XML_Char** a;
    for ( a = atts; *a; a += 2 ) {
        parsedname a3;
        xmlns_parsename(*a, &a3);
        switch ( a3.nparts ) {
            case 1: /* simple name="value" */
                ap_fputstrs(F, BB, " ", a[0], "=" , a[1], " ", NULL);
                break;
            case 2: /* namespace-uri:name="value" */
                ap_fputc(F, BB, ' ');
                ap_fwrite(F, BB, a3.ns, a3.nslen);
                ap_fputc(F, BB, ':');
                ap_fwrite(F, BB, a3.elt, a3.eltlen);
                ap_fputstrs(F, BB, "=" , a[1], " ", NULL);
                break;
            case 3: /* prefix:name="value" */
                ap_fputc(F, BB, ' ');
                ap_fwrite(F, BB, a3.prefix, a3.prefixlen);
                ap_fputc(F, BB, ':');
                ap_fwrite(F, BB, a3.elt, a3.eltlen);
                ap_fputstrs(F, BB, "=" , a[1], " ", NULL);
                break;
        }
    }
    ap_fputc(F, BB, '>');
}

The advantage of this API here is clear. We have lots of writes of just a few bytes, so
direct manipulation of buckets would be insanely complex to write, as well as inefficient.
Classic buffered I/O is ideal. And we lose nothing, because there simply isn't an input stream we could pass through without any copying of data.
6.11.1: Warning

Note that mixing stdio-like I/O with direct bucket manipulation in the same filter is not advisable. The buffering in the stdio-like API will cause the data to reach the next filter in an unexpected order, and could cause data to be flushed at the wrong time. You would have to take great care to flush everything explicitly before switching mode, and effectively get the worst of both worlds. Hence, although the function xdefault above could explicitly create a new bucket (of type transient) to contain its data, we don't.

6.12: Input Filters and the Pull API

As already discussed above, the Input Filter API differs from the Output Filter API we've discussed above. As with output filters, the core of the filter is a callback function. But the role of the function is different. Where as the output filter accepts a chunk of data, processes it, and passes it to the next filter, the input filter requests data from the next filter in the chain, processes the data, and returns it to the caller. The basic form of an input filter can be demonstrated with a trivial do-nothing filter:

```c
int do_nothing_input_filter(ap_filter_t *f, apr_bucket_brigade *bb,
    ap_input_mode_t mode, apr_read_type_e block,
    apr_off_t readbytes) {
    int rv;
    rv = ap_get_brigade(f->next, bb, mode, block, nbytes);
    return rv;
}
```

We've already introduced the filter arguments. The first two are the same as for an output filter. The others will often be irrelevant to any particular filter, but are handled by Apache's core input filter and may be of use elsewhere.

6.12.1: Mode

Most filters will not want to support all input modes. For example, mod_deflate's input filter, which serves to uncompress input that arrives compressed at the server, is entirely inappropriate to line-mode data. The correct behaviour for an input filter called in an inappropriate mode is to pass the data straight through, or to remove itself from the chain. Hence mod_deflate uses:

```c
/* just get out of the way of things we don't want. */
if (mode != AP_MODE_READBYTES) {
    return ap_get_brigade(f->next, bb, mode, block, nbytes);
}
```

As a rule of thumb, a content filter will normally be called with AP_MODE_READBYTES. A connection filter will be called with AP_MODE_GETLINE until the HTTP headers are consumed by the protocol handler, and AP_MODE_READBYTES thereafter. But this behaviour may vary, and cannot be relied on: another filter or more commonly a content generator may use a different mode – hence the simple check in mod_deflate. MPMs may also use different input modes.

A module that supports multiple modes and modifies the data will typically need to use a
switch statement or similar construct.

### 6.12.2: Block

Blocking vs nonblocking reads are only relevant to bucket types such as socket where blocking is an issue. A filter should normally honour the block request and use the same value to retrieve data from the next filter, but with due caution may override it.

### 6.12.3: readbytes

This is relevant to AP_MODE_READBYTES. In principle, a filter should not return more data than this. In practice, it is often treated as advisory: it is honoured by the core input filter, but content filters sometimes ignore it. It may serve to optimise throughput of data by selecting a block size such as the widely-used 8Kb default.

A filter to which this may be highly relevant is mod_deflate, where the output data returned to the caller will often be many times greater than the input data from the next filter. mod_deflate deals with this by keeping a bucket brigade `proc_bb` in its filter context, and using the following logic:

```c
/* all the 'main business' of this filter only happens */
if (APR_BRIGADE_EMPTY(ctx->proc_bb)) {
    rv = ap_get_brigade(f->next, ctx->bb, mode, block, readbytes);

    /* Now inflate the data we just read into ctx->bb, */
    /* and put the inflated data into ctx->proc_bb */
}

/* at the end of the filter function, we partition ctx->proc_bb */
/* so it has at most readbytes bytes of data, which we then */
/* move to the caller's brigade bb. We then save any remainder. */
if (!APR_BRIGADE_EMPTY(ctx->proc_bb)) {
    apr_bucket_brigade *newbb;

    /* May return APR_INCOMPLETE which is fine by us. */
    apr_brigade_partition(ctx->proc_bb, readbytes, &bkt);

    newbb = apr_brigade_split(ctx->proc_bb, bkt);
    APR_BRIGADE_CONCAT(bb, ctx->proc_bb);
    APR_BRIGADE_CONCAT(ctx->proc_bb, newbb);
}
```

### 6.12.4: Input Filter Example

To conclude this chapter, let's present the mod_deflate input filter we've drawn on for the above illustrations, with additional comments where appropriate. I've slightly reduced
this by replacing some of the detail relevant to zlib (the compression library used) but not to Apache with comments.

/ * This is the deflate input filter (inflates). */
static apr_status_t deflate_in_filter(ap_filter_t *f,
     apr_bucket_brigade *bb,
     ap_input_mode_t mode,
     apr_read_type_e block,
     apr_off_t readbytes)
{
    apr_bucket *bkt;
    request_rec *r = f->r;
    deflate_ctx *ctx = f->ctx;
    int zRC;
    apr_status_t rv;
    deflate_filter_config *c;

    /* just get out of the way of things we don't want. */
    if (mode != AP_MODE_READBYTES) {
        return ap_get_brigade(f->next, bb, mode, block, readbytes);
    }

    c = ap_get_module_config(r->server->module_config, &deflate_module);
    if (!ctx) {
        int found = 0;
        char *token, deflate_hdr[10];
        const char *encoding;
        apr_size_t len;

        /* only work on main request/no subrequests */
        if (!ap_is_initial_req(r)) {
            ap_remove_input_filter(f);
            return ap_get_brigade(f->next, bb, mode, block, readbytes);
        }

        /* Let's see what our current Content-Encoding is. */
        encoding = apr_table_get(r->headers_in, "Content-Encoding");
        if (encoding) {
            const char *tmp = encoding;
            token = ap_get_token(r->pool, &tmp, 0);
            while (token && token[0]) {
                if (!strcasecmp(token, "gzip")) {
                    found = 1;
                    break;
                }
                /* Otherwise, skip token */
                tmp++;
                token = ap_get_token(r->pool, &tmp, 0);
            }
        }
    }

    /* It wasn't gzipped anyway, so there's nothing to do */

Copyright © 2005-6 Nick Kew (niq@apache.org)
if (found == 0) {
    ap_remove_input_filter(f);
    return ap_get_brigade(f->next, bb, mode, block, readbytes);
}

/* set up filter ctx */
ctx = apr_palloc(f->r->pool, sizeof(*ctx));
ctx->bb = apr_brigade_create(r->pool, f->c->bucket_alloc);
ctx->proc_bb = apr_brigade_create(r->pool, f->c->bucket_alloc);
ctx->buffer = apr_palloc(r->pool, c->bufferSize);

/* DELETED – get 10 bytes from upstream and check the gzip header */

ctx->stream.next_out = ctx->buffer;
ctx->stream.avail_out = c->bufferSize;

apr_brigade_cleanup(ctx->bb);

/* main business happens if we don’t already have data */
if (APR_BRIGADE_EMPTY(ctx->proc_bb)) {
    rv = ap_get_brigade(f->next, ctx->bb, mode, block, readbytes);

    if (rv != APR_SUCCESS) {
        /* What about APR_EAGAIN errors? */
        inflateEnd(&ctx->stream);
        return rv;
    }

    for (bkt = APR_BRIGADE_FIRST(ctx->bb);
        bkt != APR_BRIGADE_SENTINEL(ctx->bb);
        bkt = APR_BUCKET_NEXT(bkt))
    {
        const char *data;
        apr_size_t len;

        /* If we actually see the EOS, that means we screwed up! */
        if (APR_BUCKET_IS_EOS(bkt)) {
            inflateEnd(&ctx->stream);
            return APR_EGENERAL;
        }

        if (APR_BUCKET_IS_FLUSH(bkt)) {
            apr_bucket *tmp_heap;
            zRC = inflate(&(ctx->stream), Z_SYNC_FLUSH);
            if (zRC != Z_OK) {
                inflateEnd(&ctx->stream);
                return APR_EGENERAL;
            }

            ctx->stream.next_out = ctx->buffer;
            len = c->bufferSize - ctx->stream.avail_out;
        }
    }
}

Copyright © 2005-6 Nick Kew (niq@apache.org)
ctx->crc = crc32(ctx->crc, (const Bytef *)ctx->buffer, len);
tmp_heap = apr_bucket_heap_create((char *)ctx->buffer, len, NULL, f->c->bucket_alloc);
APR_BRIGADE_INSERT_TAIL(ctx->proc_bb, tmp_heap);
ctx->stream.avail_out = c->bufferSize;

/* Move everything to the returning brigade. */
/* Could cause us to return more than readbytes */
APR_BUCKET_REMOVE(bkt);
APR_BRIGADE_CONCAT(bb, ctx->bb);
break;

/* read */
apr_bucket_read(bkt, &data, &len, APR_BLOCK_READ);

/* pass through zlib inflate. */
/* DELETED for brevity
 * Inserts uncompressed data into ctx->proc_bb,
 * and inserts an EOS bucket when it hits the end
 * of the compressed input data stream.
 */
apr_brigade_cleanup(ctx->bb);

/* If we are about to return nothing for a 'blocking' read and
 * we have some data in our zlib buffer, flush it out so we can
 * return something.
 */
if (block == APR_BLOCK_READ &&
    APR_BRIGADE_EMPTY(ctx->proc_bb) &&
    ctx->stream.avail_out < c->bufferSize) {

    /* DELETED for brevity */
}
if (!APR_BRIGADE_EMPTY(ctx->proc_bb)) {
apr_bucket_brigade *newbb;

    /* May return APR_INCOMPLETE which is fine by us. */
apr_brigade_partition(ctx->proc_bb, readbytes, &bkt);

    newbb = apr_brigade_split(ctx->proc_bb, bkt);
    APR_BRIGADE_CONCAT(bb, ctx->proc_bb);
    APR_BRIGADE_CONCAT(ctx->proc_bb, newbb);
}
return APR_SUCCESS;
}
6.13: **Conclusion**

Filters are one of the most powerful and useful innovations in Apache 2, and are the single biggest architectural change that helps transform Apache from a mere webserver to a powerful applications platform. Programming filters is not straightforward, but is essential to application development with Apache. It therefore fully merits one of our longest chapters, including several examples.

With this chapter, we conclude the core introduction to request processing we started in Chapter 4. The following chapters move to more general module programming topics, with a brief return to request processing topics in Chapter 9.

Finally, readers should note that there is a section on filter debugging in Chapter 12.