Chapter 4: Writing a Content Generator

```
This chapter will focus entirely on developing a real module.

* Introduction
* A HelloWorld Module
* The module skeleton
* Using the request_rec
* Request, Response and Enviromnent
* Module I/O
```

In principle, one can do anything with CGI. But the range of problems for which CGI provides a good solution is much smaller.

The same is true of a content generator in Apache. It is the heart of processing a request, and of building a web application. And it can be extended to do anything permitted to the webserver by the underlying system. It is also the most basic kind of module in Apache.

All the main traditional applications normally work as content generators. For example, a PHP page, or an application server proxied by Apache. The caveat is that systems may interact with Apache in other ways, although this is not usual.

1 The HelloWorld Module.

In this chapter, we will develop a simple content generator. The customary HelloWorld, demonstrates the basic concepts of module programming, including the complete module structure, and use of the handler callback and `request_rec`.

By the end of the chapter, our HelloWorld module will be extended to report the full details of Request, Response, and Environment and any data posted to it.

The Module Skeleton

Every Apache module works by exporting a module data struct. In general, an Apache 2.x module takes the form:

```
module AP_MODULE_DECLARE_DATA some_module = {
      STANDARD20_MODULE_STUFF,
      some_dir_cfg,
      some_dir_merge,
      some_svr_cfg,
      some_svr_merge,
      some_cmds,
      some_hooks
} ;
```

Most of this is concerned with module configuration, and will be discussed in detail in Chapter 8. For the purposes of our HelloWorld module, we only need the hooks:

```
module AP_MODULE_DECLARE_DATA helloworld_module = {
      STANDARD20_MODULE_STUFF,
      NULL,
      NULL,
      NULL,
      NULL,
      NULL,
      helloworld_hooks
} ;
```

Having declared the module structure, we now need to instantiate the hooks function. This function will be run by Apache at server startup, and its purpose is to register our module's processing functions with the server core, so that our module will subsequently be invoked whenever it is appropriate. In the case of HelloWorld, we just need to register a simple content generator (handler):

```
static void helloworld_hooks(apr_pool_t* pool) {
  ap_hook_handler(helloworld_handler, NULL, NULL, APR_HOOK_MIDDLE) ;
}
```

Finally, we need to implement the `helloworld_handler`. This is a callback function that will be called by Apache at the appropriate point in processing an HTTP request. It may choose to handle or ignore a request. If it handles a request, it is responsible for sending a valid HTTP response to the client, and for ensuring that any data coming from the client gets read (or discarded). This is very similar to the responsibilities of a CGI script, or indeed of the webserver as a whole.

Our handler will start with a couple of basic sanity checks.

- First we check `r->handler`, to see if the request is for us. If the request is not for us, we ignore it by returning `DECLINED`. Apache will then pass control on to the next handler.
- Second, we only want to support the HTTP GET (and HEAD) methods. So we check for that, and return an HTTP error code for method not allowed if not. Returning an error code here will cause Apache to return an error page to the client.

The order of these checks is important. If we reversed them, our module might cause Apache to return an error page for, e.g., POST requests intended for another handler, such as a CGI script that accepts them.

Once we are satisfied that the request is OK and is meant for this handler, we generate the actual response: in this case, a trivial HTML page. Having done that, we return `OK` to tell Apache that we have dealt with this request, and no other handler should be called.

```
static int helloworld_handler(request_rec* r) {
  if ( !r->handler || strcmp(r->handler, "helloworld") ) {
    return DECLINED ;
  }
  if ( r->method_number != M_GET ) {
    return HTTP_METHOD_NOT_ALLOWED ;
  }
  ap_set_content_type(r, "text/html;charset=ascii") ;
  ap_rputs(
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01//EN\">\n", r) ;
  ap_rputs(
      "<html><head><title>Apache HelloWorld Module</title></head>", r) ;
  ap_rputs("<body><h1>Hello World!</h1>", r) ;
  ap_rputs("<p>This is the Apache HelloWorld module!</p>", r) ;
  ap_rputs("</body></html>", r) ;
  return OK ;
}
```

Return Values

Even this trivial handler has three different return values. This pattern is usual amongst modules: we can return

- `OK`, to indicate that this handler has fully and successfully dealt with the request. No further processing is necessary.
- `DECLINED`, to indicate that this handler takes no interest in the request and declines to process it. Apache will then try the next handler. The default handler, which simply returns a file from the local disc (or an error page if that fails), never returns `DECLINED`, so requests are always handled by someone.
- Any HTTP status code to indicate an error. The handler has taken responsibility for the request, but was unable or unwilling to complete it.

An HTTP status code diverts the entire processing chain within Apache. Normal processing of the request is aborted, and Apache sets up an internal redirect to an error document. Note that this can only work if Apache hasn't already started to send the response down the wire to the client: this can be an important design consideration in handling errors. To ensure correct behaviour, any such diversion must take place before writing any data (the first `ap_rputs` statements in our case).

Where possible, it is usually good practice to deal with errors earlier in the request processing cycle: this will be discussed in Chapter 5.

The Complete Module

Putting it all together and adding the required headers, we have a complete `mod_helloworld.c` source file:

```
/* The simplest HelloWorld module */

#include <httpd.h>
#include <http_protocol.h>

static int helloworld_handler(request_rec* r) {
  if ( !r->handler || strcmp(r->handler, "helloworld") ) {
    return DECLINED ;
  }
  if ( r->method_number != M_GET ) {
    return HTTP_METHOD_NOT_ALLOWED ;
  }
  ap_set_content_type(r, "text/html;charset=ascii") ;
  ap_rputs(
      "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01//EN\">\n", r) ;
  ap_rputs(
      "<html><head><title>Apache HelloWorld Module</title></head>", r) ;
  ap_rputs("<body><h1>Hello World!</h1>", r) ;
  ap_rputs("<p>This is the Apache HelloWorld module!</p>", r) ;
  ap_rputs("</body></html>", r) ;
  return OK ;
}
static void helloworld_hooks(apr_pool_t* pool) {
  ap_hook_handler(helloworld_handler, NULL, NULL, APR_HOOK_MIDDLE) ;
}
module AP_MODULE_DECLARE_DATA helloworld_module = {
        STANDARD20_MODULE_STUFF,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL,
        helloworld_hooks
} ;
```

And that's all we need! Now we can build the module and insert it into Apache:

```
apxs -cie mod_helloworld.c
```

and configure it as a handler in `httpd.conf` or in some `.htaccess`.

For example

```
LoadModule  helloworld_module modules/mod_helloworld.so
<Location /helloworld>
      SetHandler helloworld
</Location>
```

will cause requests to `/helloworld` on our server to invoke this module as its handler.

Note that both the `helloworld_hooks` and `helloworld_handler` functions are declared

static. This is usual, though not quite universal, in Apache modules. In general, only the module symbol itself is exported, and everything else is private to the module itself, so it is good practice to declare all functions as static. Exceptions arise when a module exports a service or API for other modules, as discussed in Chapter 7.

Using the request_rec

As we have just seen, the single argument to our handler function is the `request_rec`. This is equally true for all hooks involved in request processing.

The `request_rec` is a large data struct representing an HTTP request, and providing access to all the data involved in processing a request. It is also an argument to many lower-level API calls: in `helloworld_handler`, it serves as an argument to `ap_set_content_type` and as an io-descriptor-like argument to `ap_rputs`.

To take just one more example, suppose we want to serve a file from the local filesystem instead of a fixed HTML page. To do that we would use the `r->filename` argument to identify the file. But we also have file stat information we can use to optimise sending it. Instead of reading the file and sending its contents with `ap_rwrite`, we can send the file itself, allowing APR to take advantage of available system optimisations:

```
static int helloworld_handler(request_rec* r) {
  apr_file_t* fd ;
  apr_size_t sz ;
  apr_status_t rv ;

  /* "is it for us?" checks omitted for brevity */

  /* it's an error if r->filename and finfo haven't been set for us.
   * We could omit this check if we make certain assumptions concerning
   * use of our module, but if 'normal' processing is prevented by
   * some other module then r->filename might be null, and we don't
   * want to risk a segfault!
   */
  if ( ! r->filename || ! r->finfo || ! r->finfo.size ) {
    ap_log_rerror(APLOG_MARK, APLOG_ERR, 0, r,
            "Incomplete request_rec!") ;
    return HTTP_INTERNAL_SERVER_ERROR ;
  }

  ap_set_content_type(r, "text/html;charset=ascii") ;

  /* Now we can usefully set some additional headers from the file info
   * (1) Content-Length
   * (2) Last-Modified
   */
  ap_set_content_length(r, r->finfo.size) ;
  if ( r->finfo.mtime ) {
    char* datestring = apr_palloc(r->pool, APR_RFC822_DATE_LEN) ;
    apr_rfc822_date(datestring, r->finfo.mtime) ;
```

```
    apr_table_setn(r->headers_out, "Last-Modified", datestring) ;
  }

  rv = apr_file_open(&fd, r->filename,
      APR_READ|APR_SHARELOCK|APR_SENDFILE_ENABLED,
      APR_OS_DEFAULT, r->pool) ;
  if ( rv != APR_SUCCESS ) {
    ap_log_rerror(APLOG_MARK, APLOG_ERR, 0, r,
            "can't open %s", r->filename) ;
    return HTTP_NOT_FOUND ;
  }
  ap_send_fd(fd, r, 0, r->finfo.size, &sz) ;

  /* file_close here is purely optional.  If we omit it, APR will close
   * it for us when r is destroyed, because apr_file_open registered
   * a close on r->pool.
   */
  apr_file_close(fd) ;
  return OK ;
}
```

2 The Request, Response and Environment

Setting aside this little diversion into the filesystem, what else can a HelloWorld module usefully do?

Well, it can report general information, in the manner of programs such as the `printenv` CGI script bundled with Apache. Three of the most commonly used and useful sets of information in Apache modules are the Request headers, the Response headers, and the internal Environment variables. So, let's update the original HelloWorld to print them in the response page.

Each of these sets of information is held in an APR table that is part of the `request_rec`. We can iterate over the tables to print the full contents using `apr_table_do` and a callback. We'll use HTML tables to represent these Apache tables.

First, here's a callback to print a table entry as an HTML row. Of course, we need to escape the data for HTML:

```
static int printitem(void* rec, const char* key, const char* value) {
  /* rec is a userdata pointer.  We'll pass the request_rec in it */
  request_rec* r = rec ;
  ap_rprintf(r, "<tr><th scope=\"row\">%s</th><td>%s</td></tr>\n",
      ap_escape_html(r->pool, key), ap_escape_html(r->pool, value)) ;
  /* Zero would stop interating; any other return value continues */
  return 1 ;
}
```

Second, a function to use this to print an entire table:

```
static void printtable(request_rec* r, apr_table_t* t,
      const char* caption, const char* keyhead, const char* valhead) {

  /* print a table header */
  ap_rprintf(r, "<table><caption>%s</caption><thead>"
      "<tr><th scope=\"col\">%s</th><th scope=\"col\">%s"
      "</th></tr></thead><tbody>", caption, keyhead, valhead) ;

  /* Print the data: apr_table_do iterates over entries with our
callback */
  apr_table_do(printitem, r, t, NULL) ;

  /* and finish the table */
  ap_rputs("</tbody></table>\n", r) ;
}
```

Now we can simply wrap this in our HelloWorld handler:

```
static int helloworld_handler(request_rec* r) {
  if ( !r->handler || strcmp(r->handler, "helloworld") ) {
    return DECLINED ;
  }
  if ( r->method_number != M_GET ) {
    return HTTP_METHOD_NOT_ALLOWED ;
  }
  ap_set_content_type(r, "text/html;charset=ascii") ;
  ap_rputs("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01//EN\">\n"
      "<html><head><title>Apache HelloWorld Module</title></head>"
      "<body><h1>Hello World!</h1>"
      "<p>This is the Apache HelloWorld module!</p>", r) ;

  /* Print the tables */
  printtable(r, r->headers_in, "Request Headers", "Header", "Value") ;
  printtable(r, r->headers_out, "Response Headers", "Header", "Value") ;
  printtable(r, r->subprocess_env, "Environment", "Variable", "Value") ;

  ap_rputs("</body></html>", r) ;
  return OK ;
}
```

Module I/O

Our HelloWorld module generates output using a stdio-like family of functions: `ap_rputc`, `ap_rputs`, `ap_rwrite`, `ap_rvputs`, `ap_vrprintf`, `ap_rprintf`, `ap_rflush`. We have also seen the sendfile call `ap_send_file`. This is a very simple, high-level API inherited originally from earlier Apache versions, and suitable for many content generators. They are defined in `http_protocol.h`.

This simple high-level API is available to content generators only. Since the introduction of the filter chain, the underlying mechanism for generating output is based on buckets and brigades, as discussed in Chapters 3 and 6. Filter modules have different

mechanisms for generating output, and these are also available, and sometimes appropriate, to a content handler.

There are two fundamentally different ways to process or generate output in a filter:

- By direct manipulation of bucket and brigades.
- By another stdio-like API (which is in fact rather better than the `ap_r*` API, as back-compatibility wasn't an issue).

We will describe these in detail in Chapter 6. For now, we will simply look at the basic mechanics of using the filter-oriented I/O in a content generator.

There are three steps to using filter I/O for output:

(1) Create a bucket brigade

(2) Populate the brigade with the data we are writing

(3) Pass the brigade to the first output filter on the stack (`r->output_filters`)

These can be repeated as many times as needed, either by creating a new brigade or by reusing a single one. If a response is large and/or slow to generate, we may want to pass it down the filter chain in smaller chunks. The response can then be passed through the filters and to the client in chunks, giving us an efficient pipeline and avoiding the overhead of buffering the entire response. Working properly with the pipeline whenever possible is an extremely useful goal for filter modules.

For our HelloWorld module, all we need to do is to create the brigade, and then replace `ap_r*` family calls with the alternative stdio-like API defined in `util_filter.h`: `ap_fflush`, `ap_fwrite`, `ap_fputs`, `ap_fputc`, `ap_fputstrs`, `ap_fprintf`. These have a slightly different prototype: instead of passing the `request_rec` as a file descriptor, we have to pass both the destination filter rec we are writing to, and the bucket brigade. We'll see examples in Chapter 6.

Module input is slightly different. Again there is a legacy method inherited from Apache 1.x, but this is now treated as deprecated by most developers (though it is still supported). So we should normally prefer to use the input filter chain directly in new code:

(1) Create a bucket brigade

(2) Pull data in to the brigade from the first input filter (`r->input_filters`)

(3) Read the data in our buckets, and use it.

Both input methods are common in existing modules, including modules for Apache 2.x. So for familiarity's sake, let's introduce each in turn into our HelloWorld module. We'll update the module to support POSTs, and count the number of bytes POSTed (note that this will usually, but not necessarily, also be available in a “Content-Length” request header). We won't decode or display the actual data: although we could do, this is usually best handled by an input filter (or by a library such as `libapreq`). The functions we use here are documented in `http_protocol.h`:

```
#define BUFLEN 8192
static void check_postdata_old_method(request_rec* r) {
  char buf[BUFLEN] ;
  size_t bytes, count = 0 ;

  /* decide how to treat input. */
  if ( ap_setup_client_block(r, REQUEST_CHUNKED_DECHUNK) != OK ) {
    ap_log_rerror(APLOG_MARK, APLOG_ERR, 0, r, "Bad request body!") ;
    ap_rputs("<p>Bad request body.</p>\n", r) ;
    return ;
  }
  if ( ap_should_client_block(r) ) {
    for ( bytes = ap_get_client_block(r, buf, BUFLEN) ; bytes > 0 ;
          bytes = ap_get_client_block(r, buf, BUFLEN) ) {
      count += bytes ;
    }
    ap_rprintf(r, "<p>Got %d bytes of request body data.</p>\n",
      count) ;
  } else {
    ap_rputs("<p>No request body.</p>\n", r) ;
  }
}

static int helloworld_handler(request_rec* r) {
  if ( !r->handler || strcmp(r->handler, "helloworld") ) {
    return DECLINED ;
  }
  /* We could be just slightly sloppy and drop this altogether.
   * But it's good practice to reject anything that's not explictly
   * allowed.  It cuts off *potential* exploits for someone trying
   * to compromise the server.
   */
  if ( (r->method_number != M_GET) && (r->method_number != M_POST) ) {
    return HTTP_METHOD_NOT_ALLOWED ;
  }
  ap_set_content_type(r, "text/html;charset=ascii") ;
  ap_rputs("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01//EN\">\n"
      "<html><head><title>Apache HelloWorld Module</title></head>"
      "<body><h1>Hello World!</h1>"
      "<p>This is the Apache HelloWorld module!</p>", r) ;

  /* Print the tables */
  printtable(r, r->headers_in, "Request Headers", "Header", "Value") ;
  printtable(r, r->headers_out, "Response Headers", "Header", "Value") ;
  printtable(r, r->subprocess_env, "Environment", "Variable", "Value") ;

  check_postdata_old_method(r) ;

  ap_rputs("</body></html>", r) ;
  return OK ;
}
```

Here finally is `check_postdata` using the preferred method of direct access to the input filters, using functions documented in `util_filter.h`.

We create a brigade, then loop until EOS, filling it from the input filters. We will see this technique again in Chapter 6.

```
static void check_postdata_new_method(request_rec* r) {
  apr_status_t status ;
  int end = 0 ;
  apr_size_t bytes, count = 0 ;
  const char* buf ;
  apr_bucket* b ;
  apr_bucket_brigade* bb ;

  /* check whether there's any input to read */
  int has_input = 0 ;
  const char* hdr = apr_table_get(r->headers_in, "Content-Length") ;
  if ( hdr ) {
    has_input = 1 ;
  }
  hdr = apr_table_get(r->headers_in, "Transfer-Encoding") ;
  if ( hdr ) {
    if ( !strcasecmp(hdr, "chunked") ) {
      has_input = 1 ;
    } else {
      ap_rprintf("<p>Unsupported Transfer Encoding: %s</p>",
            ap_escape_html(r->pool, hdr)) ;
      return ;
    }
  }
  if ( ! has_input ) {
    ap_rputs("<p>No request body.</p>\n", r) ;
    return ;
  }

  /* OK, we have some input data.  Now read and count it */
  bb = apr_brigade_create(r->pool, r->connection->bucket_alloc) ;
  do {
    status = ap_get_brigade(r->input_filters, bb,
      AP_MODE_READBYTES, APR_BLOCK_READ, BUFLEN) ;
    if ( status == APR_SUCCESS ) {
      for ( b = APR_BRIGADE_FIRST(bb) ;
          b != APR_BRIGADE_SENTINEL(bb) ;
          b = APR_BUCKET_NEXT(b) ) {
        if ( APR_BUCKET_IS_EOS(b) ) {
          end = 1 ;
          break ;
        }

        /* to get the actual length, we need to read the data */
        status = apr_bucket_read(b, &buf, &bytes, APR_BLOCK_READ) ;
        count += bytes ;
      }
    }
    apr_brigade_cleanup(bb) ;
  } while ( ! end && status == APR_SUCCESS ) ;
```

```
  if ( status == APR_SUCCESS ) {
    ap_rprintf(r, "<p>Got %d bytes of request body data.</p>\n",
            count) ;
  } else {
    ap_rputs("<p>Error reading request body.</p>", r) ;
  }
}
```