Chapter 5: Writing a Metadata handler

Before returning contents to the client, Apache needs to examine the HTTP request with reference to the server configuration. Much of the Apache standard code is concerned with this, and sometimes we may need to write a new module to support different behaviour. Such modules work by hooking in to the early parts of request processing, before any content generator is invoked, and sometimes by diverting or aborting the entire request.

This chapter looks at the metadata sent to the server in an HTTP request, and describes how the standard modules in Apache deal with this in handling a request. Finally, we will develop a couple of new modules that use Apache's hooks to deal with metadata and influence request processing.

Note that there is no universally agreed nomenclature here. Modules directly relevant to this chapter fall under three separate categories in the Apache distribution:

- aaa (access and authentication)
- mappers (mapping from a Request URL to the internal structure of the server and/or the filesystem)
- metadata (modules that explicitly manipulate HTTP headers and/or Apache's internal state)

It is also worth noting that there is a lot of folklore concerning the uses of metadata, prevalent among web developers with little understanding of HTTP. Some common wisdom is sound, but some is seriously flawed and often leads to applications that are fundamentally broken, and can never be made to work as intended outside a limited (and often unexpected) range of circumstances. This is common not only in areas popular with semi-programmers, such as CGI, PHP and ASP, but even in top-end (seriously expensive) professional environments. So in discussing use of HTTP headers, it is useful to be aware of some of the more common misconceptions.

1 HTTP

To discuss HTTP request processing, we first need an overview of some basics of the HTTP protocol. HTTP is one of a broad family of networking protocols for passing messages, dating back to the early days of the Internet (the oldest still in general use today is SMTP, the e-mail standard known as RFC822 and dated 1983). The protocol of the web is HTTP, and is specified in RFC1945 (HTTP 1.0) and RFC2616 (HTTP 1.1 - the current protocol version). These protocols share some common overall characteristics, designed for exchange of messages.

Envelopes, Cover Notes, Letters, Enclosures

Before the Internet, there were other means of non-face-to-face communication. For these to work, they need two different things.First, there's the contents of the communication: the letter, telegram, fax, or telephone conversation. Then there is addressing information: the envelope, the phone number, the fax number and cover sheet. This ensures that the contents can be directed and delivered to the intended recipient.

When the Internet messaging protocols were designed, a similar approach was adopted. A modern Internet message comprises envelope, cover note, and message contents. The contents may be a single letter, a letter with enclosures, or empty.

Metadata, Data

Adopting the letter metaphor for the Internet, we speak of "data" (content) and "metadata" (information about the data – roughly speaking, the "envelope").

HTTP metadata can be quite extensive. This chapter includes examples of it, but we will not present a detailed or thorough overview. The authoritative specification is RFC2616, which is included as Appendix ??.

Request and Response

One important characteristic of the RFC822 family of Internet protocols, including all versions of HTTP, is that all messages are two-way. Every transaction comprises both a Request from the client to the server, and a Response from the server back to the client. That is to say, a complete message comprising metadata and (optionally) data passes each way.

Anatomy of an HTTP Request

The first thing Apache must do on receiving a request is to check the cover sheet (metadata) and decide how to deal with the request. The server configuration, together with HTTP rules, will determine what to do.

In HTTP, we have two sets of request metadata to deal with: the request line, and the request headers. The first is a single line, which serves to specify the request method and the resource requested. The headers provide supplementary metadata that may be of relevance to the server in generating a response, or in secondary tasks such as logging and analysing usage patterns.

Consider a hypothetical request:

```
GET /index.html HTTP/1.1
Accept: text/html,application/xhtml+xml
Accept-Encoding: gzip
Accept-Language: en
Authorization: DWB/2xgwF9e9
Cookie: prefs=laid-back
Host: www.example.com
If-Modified-Since: Sun Apr 24 11:12:15 GMT 2005
User-Agent: The Universal Proxy (Mozilla 7.2; Compatible)
X-foo: bar;wibble
```

The first line indicates a GET request for "/index.html" on the server. Combined with the "Host" header (which is mandatory in HTTP 1.1, and has also been supported in HTTP 1.0 browsers since 1995), it identified the requested resource as http://www.example.com/index.html (which is necessary if and only if the server is running more than one virtual host on the IP address and port the request came on).

The remaining headers are optional, and illustrate the kind of metadata Apache may wish to deal with. No single module is likely to be concerned with all request headers, but many modules are concerned with some part of the above:

- Mapping "`/index.html`" to the filesystem or to a custom handler from the server configuration for www.example.com
- Selecting a response acceptable to the browser based on the various `Accept-*` headers
- Checking whether the user is authorised to access the resource requested (`Authorization` header).
- Checking whether, from the information supplied, the client already has an up-to-date copy in cache (`If-Modified-Since`). If so, we just confirm that with a "`304 Not Modified`" response and save the bandwidth of returning the entire response body.
- Private application data passed between this server and this user (`Cookie` and `X-anything` headers).
- Logging data (`User-Agent`).

To deal effectively with all these issues, Apache implements several request processing phases before content generation. Modules can hook in to any of these phases to adjust, or take full control of, different aspects of request processing.

2 Request Processing in Apache

We have already introduced the request processing cycle. Standard phases are:

- `post_read_request`
- `translate_name`
- `map_to_storage`
- `header_parser`
- `access_checker`
- `check_user_id`
- `auth_checker`
- `type_checker`
- `fixups`
- `handler`
- `logger`

Let's consider how the standard modules in Apache deal with our request.

The first task we identified was to map "`/index.html`" to the filesystem. By default, the Apache core will do this by appending the request in the path to the "document root" (a configuration setting) at the end of the `translate_name` phase. So if our document root for www.example.com is `/var/www/example.com`, then the default is to map that to a file `/var/www/example.com/index.html`. A second default handler, at the end of `map_to_storage`, cross-references the file to <Directory> and <Files> configuration, and if `.htaccess` files are enabled, merges them into the configuration.

A standard module that can change the default behaviour is `mod_alias`. The "Alias" configuration directive is used to specify a different mapping to the filesystem for selected request paths. Alias uses a `translate_name` hook that replaces the default. Subsequent processing, including the default `map_to_storage` handler, is unchanged.

Our second task was to select a response acceptable to the browser, according to the `Accept-*` headers sent.

```
Accept: text/html,application/xhtml+xml
Accept-Encoding: gzip
Accept-Language: en
```

These may simply be ignored, and will be if we simply use default processing with multiviews (content negotiation provided by mod_negotiation) disabled. It may also be handled in different ways: for dynamic contents, gzip encoding is determined by an output filter (`mod_deflate`), not by a metadata handler. Likewise, an XSLT output filter could deal with selection of content types. But a regular case we should consider (not least because it's a wheel that's been reinvented badly by many organisations that should know better) is standard content negotiation, in which Apache selects an appropriate static file from several available options.

The module concerned here is `mod_negotiation`, and typical uses are:

- For multilingual sites, to select a language specified in the user's browser preferences.
- For sites supporting different devices, to select between HTML and XML variants, or between SVG and bitmap images.

The simplest use of `mod_negotiation` is just to create a choice of resources. For example, if we supply files

- `index.html.en` (English)
- `index.html.fr` (French)
- `index.html.de` (German)
- `index.html.it` (Italian)

together with appropriate AddLanguage directives and MultiViews, then `mod_negotiation` will select one of the above files according to the `Accept-Language` header sent by the browser (and configured by the user in a preferences menu, or as supplied to the user localised by an ISP or other network administrator). `mod_negotiation` uses a `type_checker` handler to map `index.html` to one of the available variants. When a variant is chosen, it overwrites the request state with one that is identical except in that a new file has been selected.

The third task was to check that the user is authorised to access the resource. The Access and Authentication phases of request processing check the client's credentials (if any) supplied in the request headers, together with the client's IP address, against any policies for the requested resource defined in `httpd.conf` or an applicable `.htaccess` file. We will demonstrate an authentication module at the end of this chapter.

The fourth task was to check when the resource was last modified, so we don't have to re-send it if it's older than the version the client has cached, as indicated in the `If-Modified-Since` header. This is one of several HTTP headers concerned with caching and efficiency. By default, this is only dealt with in the handler/content generator phase, but a module that is not concerned with the possible effects of another module's fixups could run this earlier.

The fifth task was to deal with application-specific data, including cookies and any private HTTP extensions (X-anything headers). This is entirely application-specific, and cannot be generalised. Applications should always be sure to implement fallback behaviour for clients that don't supply a cookie, or any private header. A major blunder sometimes seen on websites is to redirect any client without a cookie to a page that sets one, and send them into an infinite loop.

The final task is to deal with logging the request. This is the only use that should ever be made of a User-Agent string. One of the commonest errors on the web today is to try to infer client characteristics or capabilities from a User-Agent string. This is wrong for many reasons. Firstly, many browsers spoof their User-Agent strings to avoid being excluded from MSIE-only sites (ironically, MSIE itself still uses the "Mozilla" keyword, which was introduced originally to avoid MSIE users being excluded from Netscape-enhanced sites in the mid-1990s). Secondly, it fails when different browsers share a common cache server, unless it breaks caching completely. Thirdly, and most importantly, it is at best a poor reinvention of HTTP content negotiation, based on the preferences and capabilities stated in the Accept-* headers.

3 Diverting a Request: Internal Redirect

The request processing cycle we have looked at so far may be diverted at any point using a mechanism known as an internal redirect. This replaces the current request with a new request for the new (redirected) resource.

The internal redirect emulates HTTP redirection (such as an HTTP 302 response), but without requiring an additional request from the browser. This mirrors the distinction in the dual nature of the CGI `Location` header:

```
Location: http://www.example.com/foo/
```

causes Apache to send an HTTP redirection to the browser, including of course the HTTP Location header.

By contrast, a relative URL - which is illegal in HTTP - is allowed in CGI:

```
Location: /foo/
```

generates an internal redirect, without involving the browser.

A module can explicitly divert request processing using the `ap_internal_redirect` or `ap_internal_fast_redirect` calls, documented in `http_request.h`. But the most usual way to generate an internal redirect is by returning an HTTP status code (or indeed any value other than OK or DECLINED) from a handler for any phase in request processing. This causes Apache to divert processing to an error document.

4 Error Documents

As mentioned in Chapter 4, if a handler returns an HTTP status code, this diverts the entire request processing into an error path. Any function implementing an earlier hook in the request cycle may likewise return an HTTP status code. What actually then happens is that Apache sets up an internal redirect to the "ErrorDocument" for the HTTP status code in question.

An ErrorDocument is, by default, simply a predefined document that presents the user with a brief explanation of the error. A server administrator can change this using the `ErrorDocument` directive. Because an error document is treated internally as a different request, it can be served by any handler (such as a CGI or PHP script). But - to avoid going into an error loop - this is not recursive: an error document cannot divert to another error document. If that happens, it generates a predefined server error.

A special case of this is ErrorDocuments for HTTP 3xx status codes. These are not errors, but redirections and similar messages. So as well as an internal redirection, they generate an HTTP redirection. This is straightforward and perfectly normal.

5 Malformed and Malicious Requests

A basic principle of security on the web is always to exercise caution in what you accept from any unknown source. That includes HTTP requests coming from anywhere on the web. Most of these will be legitimate: human-driven browsers, spiders such as googlebot, proxy cache agents, QA tools such as Site Valet, etc. But a significant number are attempts to exploit security vulnerabilities. Traces of some rather old IIS worms (Nimda/Code Red) are routinely seen in Apache logs, using IIS bugs to take control of Windows servers. Although Apache has yet to suffer any comparable attack, it is every module developer's business to keep Apache clean! The basic rule is to determine what inputs - or pattern of inputs - an application will accept, and reject any request that fails to match an acceptable pattern.

Based on what we have now seen, we have a ready-made solution for any module to deal with bad requests. Simply abort by returning HTTP status code 400 (Bad Request), or where applicable a more specific HTTP 4xx status code, as soon as you encounter the bad inputs. Don't even try to deal with it directly: that way lie complexity and security vulnerabilities.

6 Gathering Information: Subrequests

A second form of diversion from normal request processing is the subrequest. This is a diversion to a new request, but unlike the internal redirection, processing returns to the original request after the subrequest completes.

Subrequests were an important tool in Apache 1.x, where they could be used to improvise a primitive form of filtering, where a module sets up a handler to run another handler in a subrequest, and intercept incoming and/or outgoing data. In Apache 2.x this is no longer necessary, but the subrequest has a new role: it may be abbreviated to run a fast partial request. For example, `mod_autoindex` runs a subrequest to each file in a directory, and lists only those that are accessible to the server. Of course, at the system level it could do that with a simple stat, but running a subrequest means it can also ascertain whether the server configuration permits access.

The subrequest API in Apache2 comprises four methods to create a subrequest from a request:

- `ap_sub_req_lookup_uri`
- `ap_sub_req_lookup_file`

- `ap_sub_req_lookup_dirent`
- `ap_sub_req_method_uri`

together with a method to run it:

- `ap_run_sub_req`

and one to dispose of it when done:

- `ap_destroy_sub_req`

Running a subrequest is optional: a fast subrequest (as used in `mod_autoindex`) is a lookup only. Destroying the subrequest is always required, whether or not it has been run.

A module that demonstrates both forms of subrequest is `mod_include`. The SSI `<!--#include virtual="..."-->` directive is implemented by a full subrequest to the included resource, whereas other directives such as `<!--#fsize ...-->` and `<!--#flastmod ...-->` use only a lookup to find information about the resource (metadata), without actually serving the resource to the client.

7 Developing a Module

We now have a brief overview of the earlier phases of request processing. In the remainder of this chapter, we will develop a couple of modules to demonstrate. Firstly, we will present a particular problem and its solution. Secondly, we will develop an authentication module - historically the most common form of Apache module (judging by the number of them registered at modules.apache.org).

Selecting different variants of a document

The author's Site Valet product includes a facility for users to publish reports to the server. These reports are part of a QA/audit process, and as such are important but will be accessed infrequently, so system performance is not a major issue.

Reports are generated and stored on the server in an XML format used within Site Valet. But the reports need to be accessible in other formats: HTML for web browsers and human readers, and EARL (RDF) for the Semantic Web. This is accomplished using by applying an XSLT transformation on the fly when a document is served to anyone other than the Valet tools. The XSLT transformation will be performed by `mod_transform`, which is a prerequisite for our module.

The problem is, in a sense, the opposite of content negotiation. Instead of selecting one of many static resources according to the request headers, we respond to the user's explicit request for a different URL.

Explicitly, if an XML report is stored at `{DOCUMENT_ROOT}/reports/example`, then we want to map request URLs as:

- `http://server/reports/example` --> XML (unchanged)
- `http://server/reports/example.html` --> HTML
- `http://server/reports/example.rdf` --> EARL

For additional flexibility, our module enables users to define other formats, by introducing their own XSLT stylesheets. Let's call this module `mod_choices`.

The core of `mod_choices` is a `type_checker` hook. To set the scene for it, we need to define the relevant data structs. Firstly the module configuration:

```
typedef struct choices_cfg {
  int choices ;              /* flag to turn this module on/off */
  apr_hash_t* transforms ;   /* table of "extensions" known to
                              * this server.
                              */
} choices_cfg ;
```

Then, each "extension" is a record describing how to handle itself:

```
typedef struct choices_transform {
  const char* ctype ;        /* Content-Type for this extension */
  const char* xslt ;         /* Name of XSLT stylesheet for
                              * this extension.
                              */
} choices_transform ;
```

To implement "`.html`" and "`.rdf`" as described above, we will set up entries in the transforms table. The configuration will look something like:

```
<Directory /var/www/reports/>
      # set "choices" in choices_cfg
      Choices On

      # Set up choices_transform entries for HTML and RDF (EARL)
      ChoicesTransform html text/html;charset=utf-8 transforms/html.xslt
      ChoicesTransform rdf application/rdf+xml transforms/earl.xslt
</Directory>
```

Now we can present the main function:

```
static int choices_select(request_rec* r) {

  /* first, look up our module configuration */
  choices_cfg* cfg =
      ap_get_module_config(r->per_dir_config, &choices_module) ;

  if ( ! cfg->choices ) {
    /* This request has nothing to do with this module */
    return DECLINED ;
  }
```

```
if ( r->method_number != M_GET ) {
  /* Other methods are allowed, but this hook isn't interested. */
  return DECLINED ;
}

if ( ! r->filename ) {
  /* This can't happen.  But if it does (e.g. a buggy third-party
   * module has messed up our request), a server error is better
   * than a server crash when we dereference a null pointer :-)
   */
  return HTTP_INTERNAL_SERVER_ERROR ;
}

/* Our request has been mapped to the filesystem, but it may not
 * match anything that's really there.  We can stat it to find out.
 */
if ( apr_stat(&r->finfo, r->filename, APR_FINFO_SIZE, r->pool)
    == APR_SUCCESS ) {
  /* The request maps directly to a file.  We don't need to
   * do anything except serve it as XML
   */
  ap_set_content_type(r, "application/xml;charset=utf-8") ;
} else {
  /* The request doesn't map to a file.  We need to check whether we
   * can map to a file by stripping the "extension" off.
   *
   * First, we split the filename.
   */
  char* ext = strrchr(r->filename, '.') ;
  if ( ext ) {
    *ext++ = 0 ;
  } else {
    /* No such file and not a name we can parse as an extension */
    return HTTP_NOT_FOUND ; /* (1) */
  }
  /* Now we can see whether we have a file we can map */
  if ( apr_stat(&r->finfo, r->filename, APR_FINFO_SIZE, r->pool)
    == APR_SUCCESS ) {
    /* OK, it's there.  Now check whether it's an extension we know */
    choices_transform* fmt = apr_table_get(cfg->transforms, ext) ;
    if ( fmt ) {
      /* OK, we have a transform for this extension
       * Now we set request properties accordingly
       */
      ap_set_content_type(r, fmt->ctype) ;

      /* this function is exported by mod_transform, and selects
       * an XSLT transform to run for this request
       */
      mod_transform_set_XSLT(r, fmt->name) ;

      /* Finally, we insert mod_transform in the output chain
       * The filter name is also exported by mod_transform
```

```
         */
        ap_add_output_filter(XSLT_FILTER_NAME, NULL, r, r->connection) ;
      } else {
        /* We don't know this extension, so we can't serve it
         * If this was a negotiated resource, we'd return NOT_ACCEPTABLE
         * (HTTP 406) here.  Since it isn't, we return NOT_FOUND.
         */
        return HTTP_NOT_FOUND ;     /* (2) */
      }
    } else {
      /* apr_stat failed - there's no underlying file to serve */
      return HTTP_NOT_FOUND ; /* (3) */
    }
  }

  /* OK, we've finished configuring this request */
  return OK ;
}
```

Now, all we need to do is to hook in the handler, together with its configuration. We'll omit details of the configuration for the time being, and return to it in Chapter 8 when we discuss the subject.

```
static void choices_hooks(apr_pool_t* pool) {
  ap_hook_type_checker(choices_select, NULL, NULL, APR_HOOK_FIRST) ;
}
module AP_MODULE_DECLARE_DATA choices_module = {
        STANDARD20_MODULE_STUFF,
        choices_cr_cfg ,      /* omitted for brevity until Chapter 8 */
        choices_merge_cfg ,   /* omitted for brevity until Chapter 8 */
        NULL ,
        NULL ,
        choices_cmds ,        /* omitted for brevity until Chapter 8 */
        choices_hooks
} ;
```

Error Handling and Reusability

Now, the above is adequate for our application. Anyone using it will be using a URL generated by the application, so how we deal with failure to find a resource is unimportant. But if we want to generalise our module a little, we could make some changes. There's no problem with re-using it as-is; it's just rather specific to a single project, and not very likely to be useful more generally.

There are three points at which we return HTTP status 404 (Not Found), thereby diverting processing into an ErrorDocument subrequest. Numbers (1) and (3) are where we failed to find any resource. Number (2) is where we found the file but don't know what to do with the extension.

Now, `mod_choices` implements one scheme for dealing with variants on a resource.

mod_negotiation implements another scheme, and another third-party module might provide an altogether different mapping. At points (1) and (3), we could return DECLINED instead of HTTP_NOT_FOUND to enable those modules to work alongside mod_choices. (At (3) we'd also need to restore r->filename to its original value first.) If we do that, we can let Apache apply several different schemes until it finds one that works (or gives up).

We could do the same at (2). But this failure is probably down to a server configuration that doesn't quite meet the client's expectations: e.g. if the client requested

```
http://server/reports/example.html
```

but the ChoicesTransform line for html was missing from httpd.conf. So another option is to return HTTP_MULTIPLE_CHOICES. This will divert us into an ErrorDocument. To be useful, our ErrorDocument will need to tell the client what options are available, so we'll have to write our own handler for it, too. The handler needs the mod_choices configuration, so it'll be easiest to implement it in the same module.

Here's a minimal implementation:

```
static int choices_errordoc(request_rec* r) {

  /* This is an error handler we can use if we return
   * HTTP_MULTIPLE_CHOICES instead of HTTP_NOT_FOUND at (2) above
   */

  choices_cfg* cfg ;

  /* Ignore the request if we're not in an internal redirect */
  if ( ! r->prev || ! r->prev->filename ) {
    return DECLINED ;
  }

  /* Insist on being configured before we do anything */
  if ( strcmp(r->handler, "choices-errordoc") ) {
    return DECLINED ;
  }

  cfg = ap_get_module_config(r->prev->per_dir_config, &choices_module) ;

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

  /* Now we can print an error page, listing the 'base' (XML)
   * document and other available variants.
   *
   * The base name is in r->prev->filename, since we already
   * stripped the "extension".
   */
  ap_rprintf(r, "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01//EN\">\n"
      "<html><head><title>No such format</title></head>"
      "<body><h1>Format not supported</h1>"
      "<p>mod_chocies on this server is not configured to support "
      "the requested document format.  Available options are:</p>"
      "<table><thead><tr>"
      "<th scope=\"col\">Document Type</th>"
      "<th scope=\"col\">URL</th>"
      "</tr></thead><tbody>"
      "<tr><td>application/xml</td>"
      "<td><a href=\"%s\" type=\"application/xml\">%s</a></td></tr>",
      r->prev->filename, r->prev->filename) ;

  /* We already saw how to iterate over a table in Chapter 4,
   * so we leave implementation of choices_show as an exercise
   * for the reader.
   */
  apr_table_do(choices_show, r, cfg->transforms, NULL) ;
  ap_rputs("</tbody></table></body></html>" , r) ;
  return OK ;
}
```

Now we just need to add this to our hooks:

```
static void choices_hooks(apr_pool_t* pool) {
  ap_hook_type_checker(choices_select, NULL, NULL, APR_HOOK_FIRST) ;
  ap_hook_handler(choices_errordoc, NULL, NULL, APR_HOOK_MIDDLE) ;
}
```

and configure it:

```
<Directory /var/www/reports/>
      # set "choices" in choices_cfg
      Choices On

      # Set up choices_transform entries for HTML and RDF (EARL)
      ChoicesTransform html text/html;charset=utf-8 transforms/html.xslt
      ChoicesTransform rdf application/rdf+xml transforms/earl.xslt

      # Set up ErrorDoc handling for Multiple Choices
      ErrorDocument 300 /reports/error300
      <Files      error300>
            SetHandler  choices-errordoc
            Choices     Off
      </Files>
</Directory>
```