CHAPTER 16

HTTP Headers for Optimal Performance

Header composition is often neglected in the CGI world. Dynamic content is dynamic, after all, so why would anybody care about HTTP headers? Because pages are generated dynamically, one might expect that pages without a `Last-Modified` header are fine, and that an `If-Modified-Since` header in the client's request can be ignored. This laissez-faire attitude is a disadvantage when you're trying to create a server that is entirely driven by dynamic components and the number of hits is significant.

If the number of hits on your server is not significant and is never going to be, then it is safe to skip this chapter. But if keeping up with the number of requests is important, learning what cache-friendliness means and how to cooperate with caches to increase the performance of the site can provide significant benefits. If Squid or mod_proxy is used in *httpd* accelerator mode (as discussed in Chapter 12), it is crucial to learn how best to cooperate with it.

In this chapter, when we refer to a section in the HTTP standard, we are using HTTP standard 1.1, which is documented in RFC 2616. The HTTP standard describes many headers. In this chapter, we discuss only the headers most relevant to caching. We divide them into three sets: date headers, content headers, and the special `Vary` header.

Date-Related Headers

The various headers related to when a document was created, when it was last modified, and when it should be considered stale are discussed in the following sections.

Date Header

Section 14.18 of the HTTP standard deals with the circumstances under which we must or must not send a `Date` header. For almost everything a normal mod_perl user does, a `Date` header needs to be generated. But the mod_perl programmer doesn't have to worry about this header, since the Apache server guarantees that it is always sent.

In *http_protocol.c*, the `Date` header is set according to `$r->request_time`. A mod_perl script can read, but not change, `$r->request_time`.

Last-Modified Header

Section 14.29 of the HTTP standard covers the `Last-Modified` header, which is mostly used as a *weak validator*. Here is an excerpt from the HTTP specification:

```
A validator that does not always change when the resource changes is a "weak
validator."

One can think of a strong validator as one that changes whenever the bits of an
entity changes, while a weak value changes whenever the meaning of an entity changes.
```

What this means is that we must decide for ourselves when a page has changed enough to warrant the `Last-Modified` header being updated. Suppose, for example that we have a page that contains text with a white background. If we change the background to light gray then clearly the page has changed, but if the text remains the same we would consider the semantics (meaning) of the page to be unchanged. On the other hand, if we changed the text, the semantics may well be changed. For some pages it is not quite so straightforward to decide whether the semantics have changed or not. This may be because each page comprises several components, or it might be because the page itself allows interaction that affects how it appears. In all cases, we must determine the moment in time when the semantics changed and use that moment for the `Last-Modified` header.

Consider for example a page that provides a text-to-GIF renderer that takes as input a font to use, background and foreground colors, and a string to render. The images embedded in the resultant page are generated on the fly, but the structure of the page is constant. Should the page be considered unchanged so long as the underlying script is unchanged, or should the page be considered to have changed with each new request?

Actually, a few more things are relevant: the semantics also change a little when we update one of the fonts that may be used or when we update the `ImageMagick` or equivalent image-generating program. All the factors that affect the output should be considered if we want to get it right.

In the case of a page comprised of several components, we must check when the semantics of each component last changed. Then we pick the most recent of these times. Of course, the determination of the moment of change for each component may be easy or it may be subtle.

mod_perl provides two convenient methods to deal with this header: `update_mtime()` and `set_last_modified()`. These methods and several others are unavailable in the standard mod_perl environment but are silently imported when we use `Apache::File`. Refer to the `Apache::File` manpage for more information.

The `update_mtime()` function takes Unix's `time(2)` (in Perl the equivalent is also the `time()` function) as its argument and sets Apache's request structure `finfo.st_mtime` to this value. It does so only when the argument is greater than the previously stored `finfo.st_mtime`.

The `set_last_modified()` function sets the outgoing `Last-Modified` header to the string that corresponds to the stored `finfo.st_mtime`. When passing a Unix `time(2)` to `set_last_modified()`, mod_perl calls `update_mtime()` with this argument first.

The following code is an example of setting the `Last-Modified` header by retrieving the last-modified time from a Revision Control System (RCS)–style of date tag.

```
use Apache::File;
use Date::Parse;
$Mtime ||= Date::Parse::str2time(
    substr q$Date: 2003/05/09 21:34:23 $, 6);
$r->set_last_modified($Mtime);
```

Normally we would use the `Apache::Util::parsedate` function, but since it doesn't parse the RCS format, we have used the `Date::Parse` module instead.

Expires and Cache-Control Headers

Section 14.21 of the HTTP standard deals with the `Expires` header. The purpose of the `Expires` header is to determine a point in time after which the document should be considered out of date (stale). Don't confuse this with the very different meaning of the `Last-Modified` header. The `Expires` header is useful to avoid unnecessary validation from now until the document expires, and it helps the recipients to clean up their stored documents. Here's an excerpt from the HTTP standard:

```
The presence of an Expires field does not imply that the original resource will
change or cease to exist at, before, or after that time.
```

Think carefully before setting up a time when a resource should be regarded as stale. Most of the time we can determine an expected lifetime from "now" (that is, the time of the request). We do not recommend hardcoding the expiration date, because when we forget that we did it, and the date arrives, we will serve already expired documents that cannot be cached. If a resource really will never expire, make sure to follow the advice given by the HTTP specification:

```
To mark a response as "never expires," an origin server sends an Expires date
approximately one year from the time the response is sent.  HTTP/1.1 servers SHOULD
NOT send Expires dates more than one year in the future.
```

For example, to expire a document half a year from now, use the following code:

```
$r->header_out('Expires',
               HTTP::Date::time2str(time + 180*24*60*60));
```

or:

```
$r->header_out('Expires',
               Apache::Util::ht_time(time + 180*24*60*60));
```

The latter method should be faster, but it's available only under mod_perl.

A very handy alternative to this computation is available in the HTTP/1.1 cache-control mechanism. Instead of setting the `Expires` header, we can specify a delta value in a `Cache-Control` header. For example:

```
$r->header_out('Cache-Control', "max-age=" . 180*24*60*60);
```

This is much more processor-economical than the previous example because Perl computes the value only once, at compile time, and optimizes it into a constant.

As this alternative is available only in HTTP/1.1 and old cache servers may not understand this header, it may be advisable to send both headers. In this case the `Cache-Control` header takes precedence, so the `Expires` header is ignored by HTTP/1.1-compliant clients. Or we could use an `if...else` clause:

```
if ($r->protocol =~ /(\d\.\d)/ && $1 >= 1.1) {
    $r->header_out('Cache-Control', "max-age=" . 180*24*60*60);
}
else {
    $r->header_out('Expires',
                   HTTP::Date::time2str(time + 180*24*60*60));
}
```

Again, use the `Apache::Util::ht_time( )` alternative instead of `HTTP::Date::time2str( )` if possible.

If the Apache server is restarted regularly (e.g., for log rotation), it might be beneficial to save the `Expires` header in a global variable to save the runtime computation overhead.

To avoid caching altogether, call:

```
$r->no_cache(1);
```

which sets the headers:

```
Pragma: no-cache
Cache-control: no-cache
```

This should work in most browsers.

Don't set `Expires` with `$r->header_out` if you use `$r->no_cache`, because `header_out( )` takes precedence. The problem that remains is that there are broken browsers that ignore `Expires` headers.

Content Headers

The following sections describe the HTTP headers that specify the type and length of the content, and the version of the content being sent. Note that in this section we often use the term *message*. This term is used to describe the data that comprises the HTTP headers along with their associated content; the content is the actual page, image, file, etc.

Content-Type Header

Most CGI programmers are familiar with Content-Type. Sections 3.7, 7.2.1, and 14.17 of the HTTP specification cover the details. mod_perl has a content_type( ) method to deal with this header:

```
$r->content_type("image/png");
```

Content-Type *should* be included in every set of headers, according to the standard, and Apache will generate one if your code doesn't. It will be whatever is specified in the relevant DefaultType configuration directive, or text/plain if none is active.

Content-Length Header

According to section 14.13 of the HTTP specification, the Content-Length header is the number of octets (8-bit bytes) in the body of a message. If the length can be determined prior to sending, it can be very useful to include it. The most important reason is that KeepAlive requests (when the same connection is used to fetch more than one object from the web server) work only with responses that contain a Content-Length header. In mod_perl we can write:

```
$r->header_out('Content-Length', $length);
```

When using Apache::File, the additional set_content_length( ) method, which is slightly more efficient than the above, becomes available to the Apache class. In this case we can write:

```
$r->set_content_length($length);
```

The Content-Length header can have a significant impact on caches by invalidating cache entries, as the following extract from the specification explains:

```
The response to a HEAD request MAY be cacheable in the sense that the information
contained in the response MAY be used to update a previously cached entity from that
resource.  If the new field values indicate that the cached entity differs from the
current entity (as would be indicated by a change in Content-Length, Content-MD5,
ETag or Last-Modified), then the cache MUST treat the cache entry as stale.
```

It is important not to send an erroneous Content-Length header in a response to either a GET or a HEAD request.

Entity Tags

An entity tag (ETag) is a validator that can be used instead of, or in addition to, the Last-Modified header; it is a quoted string that can be used to identify different versions of a particular resource. An entity tag can be added to the response headers like this:

```
$r->header_out("ETag","\"$VERSION\"");
```

mod_perl offers the `$r->set_etag( )` method if we have `use( )`ed `Apache::File`. However, we strongly recommend that you don't use the `set_etag( )` method! `set_etag( )` is meant to be used in conjunction with a static request for a file on disk that has been `stat( )`ed in the course of the current request. It is inappropriate and dangerous to use it for dynamic content.

By sending an entity tag we are promising the recipient that we will not send the same `ETag` for the same resource again unless the content is "equal" to what we are sending now.

The pros and cons of using entity tags are discussed in section 13.3 of the HTTP specification. For mod_perl programmers, that discussion can be summed up as follows.

There are strong and weak validators. Strong validators change whenever a single bit changes in the response; i.e., when anything changes, even if the meaning is unchanged. Weak validators change only when the meaning of the response changes. Strong validators are needed for caches to allow for sub-range requests. Weak validators allow more efficient caching of equivalent objects. Algorithms such as MD5 or SHA are good strong validators, but what is usually required when we want to take advantage of caching is a good weak validator.

A `Last-Modified` time, when used as a validator in a request, can be strong or weak, depending on a couple of rules described in section 13.3.3 of the HTTP standard. This is mostly relevant for range requests, as this quote from section 14.27 explains:

```
If the client has no entity tag for an entity, but does have a Last-Modified date, it
MAY use that date in an If-Range header.
```

But it is not limited to range requests. As section 13.3.1 states, the value of the `Last-Modified` header can also be used as a cache validator.

The fact that a `Last-Modified` date may be used as a strong validator can be pretty disturbing if we are in fact changing our output slightly without changing its semantics. To prevent this kind of misunderstanding between us and the cache servers in the response chain, we can send a weak validator in an `ETag` header. This is possible because the specification states:

```
If a client wishes to perform a sub-range retrieval on a value for which it has only
a Last-Modified time and no opaque validator, it MAY do this only if the Last-
Modified time is strong in the sense described here.
```

In other words, by sending an `ETag` that is marked as weak, we prevent the cache server from using the `Last-Modified` header as a strong validator.

An `ETag` value is marked as a weak validator by prepending the string `W/` to the quoted string; otherwise, it is strong. In Perl this would mean something like this:

```
$r->header_out('ETag',"W/\"$VERSION\"");
```

HTTP Range Requests

It is possible in web clients to interrupt the connection before the data transfer has finished. As a result, the client may have partial documents or images loaded into its memory. If the page is reentered later, it is useful to be able to request the server to return just the missing portion of the document, instead of retransferring the entire file.

There are also a number of web applications that benefit from being able to request the server to give a byte range of a document. As an example, a PDF viewer would need to be able to access individual pages by byte range—the table that defines those ranges is located at the end of the PDF file.

In practice, most of the data on the Web is represented as a byte stream and can be addressed with a byte range to retrieve a desired portion of it.

For such an exchange to happen, the server needs to let the client know that it can support byte ranges, which it does by sending the `Accept-Ranges` header:

```
Accept-Ranges: bytes
```

The server will send this header only for documents for which it will be able to satisfy the byte-range request—e.g., for PDF documents or images that are only partially cached and can be partially reloaded if the user interrupts the page load.

The client requests a byte range using the `Range` header:

```
Range: bytes=0-500,5000-
```

Because of the architecture of the byte-range request and response, the client is not limited to attempting to use byte ranges only when this header is present. If a server does not support the `Range` header, it will simply ignore it and send the entire document as a response.

Consider carefully which string is chosen to act as a validator. We are on our own with this decision:

```
... only the service author knows the semantics of a resource well enough to select
an appropriate cache validation mechanism, and the specification of any validator
comparison function more complex than byte-equality would open up a can of worms.
Thus, comparisons of any other headers (except Last-Modified, for compatibility with
HTTP/1.0) are never used for purposes of validating a cache entry.
```

If we are composing a message from multiple components, it may be necessary to combine some kind of version information for all these components into a single string.

If we are producing relatively large documents, or content that does not change frequently, then a strong entity tag will probably be preferred, since this will give caches a chance to transfer the document in chunks.

Content Negotiation

Content negotiation is a wonderful feature that was introduced with HTTP/1.1. Unfortunately it is not yet widely supported. Probably the most popular usage scenario for content negotiation is language negotiation for multilingual sites. Users specify in their browsers' preferences the languages they can read and order them according to their ability. When the browser sends a request to the server, among the headers it sends it also includes an `Accept-Language` header. The server uses the `Accept-Language` header to determine which of the available representations of the document best fits the user's preferences. But content negotiation is not limited to language. Quoting the specification:

```
HTTP/1.1 includes the following request-header fields for enabling server-driven
negotiation through description of user agent capabilities and user preferences:
Accept (section 14.1), Accept-Charset (section 14.2), Accept-Encoding (section 14.3),
Accept-Language (section 14.4), and User-Agent (section 14.43). However, an origin
server is not limited to these dimensions and MAY vary the response based on any
aspect of the request, including information outside the request-header fields or
within extension header fields not defined by this specification.
```

The Vary Header

To signal to the recipient that content negotiation has been used to determine the best available representation for a given request, the server must include a `Vary` header. This tells the recipient which request headers have been used to determine the representation that is used. So an answer may be generated like this:

```
$r->header_out('Vary', join ", ",
               qw(accept accept-language accept-encoding user-agent));
```

The header of a very cool page may greet the user with something like this:

```
Hallo Harri, Dein NutScrape versteht zwar PNG aber leider kein GZIP.
```

However, this header has the side effect of being expensive for a caching proxy. As of this writing, Squid (Version 2.3.STABLE4) does not cache resources that come with a `Vary` header at all. So without a clever workaround, the Squid accelerator is of no use for these documents.

HTTP Requests

Section 13.11 of the specification states that the only two cacheable methods are `GET` and `HEAD`. Responses to `POST` requests are not cacheable, as you'll see in a moment.

GET Requests

Most mod_perl programs are written to service GET requests. The server passes the request to the mod_perl code, which composes and sends back the headers and the content body.

But there is a certain situation that needs a workaround to achieve better cacheability. We need to deal with the "?" in the relative path part of the requested URI. Section 13.9 specifies that:

```
... caches MUST NOT treat responses to such URIs as fresh unless the server provides
an explicit expiration time.  This specifically means that responses from HTTP/1.0
servers for such URIs SHOULD NOT be taken from a cache.
```

Although it is tempting to imagine that if we are using HTTP/1.1 and send an explicit expiration time we are safe, the reality is unfortunately somewhat different. It has been common for quite a long time to misconfigure cache servers so that they treat all GET requests containing a question mark as uncacheable. People even used to mark anything that contained the string “cgi-bin” as uncacheable.

To work around this bug in HEAD requests, we have stopped calling CGI directories *cgi-bin* and we have written the following handler, which lets us work with CGI-like query strings without rewriting the software (e.g., Apache::Request and CGI.pm) that deals with them:

```
sub handler {
    my $r = shift;
    my $uri = $r->uri;
    if ( my($u1,$u2) = $uri =~ / ^ ([^?]+?) ; ([^?]*) $ /x ) {
        $r->uri($u1);
        $r->args($u2);
    }
    elsif ( my ($u1,$u2) = $uri =~ m/^(.*?)%3[Bb](.*)$/ ) {
        # protect against old proxies that escape volens nolens
        # (see HTTP standard section 5.1.2)
        $r->uri($u1);
        $u2 =~ s/%3[Bb]/;/g;
        $u2 =~ s/%26/;/g; # &
        $u2 =~ s/%3[Dd]/=/g;
        $r->args($u2);
    }
    DECLINED;
}
```

This handler must be installed as a PerlPostReadRequestHandler.

The handler takes any request that contains one or more semicolons but *no* question mark and changes it so that the first semicolon is interpreted as a question mark and everything after that as the query string. So now we can replace the request:

```
http://example.com/query?BGCOLOR=blue;FGCOLOR=red
```

with:

```
http://example.com/query;BGCOLOR=blue;FGCOLOR=red
```

This allows the coexistence of queries from ordinary forms that are being processed by a browser alongside predefined requests for the same resource. It has one minor bug: Apache doesn't allow percent-escaped slashes in such a query string. So instead of:

```
http://example.com/query;BGCOLOR=blue;FGCOLOR=red;FONT=%2Ffont%2Fpath
```

we must use:

```
http://example.com/query;BGCOLOR=blue;FGCOLOR=red;FONT=/font/path
```

To unescape the escaped characters, use the following code:

```
s/%([0-9A-Fa-f]{2})/chr hex $1/ge;
```

Conditional GET Requests

A rather challenging request that may be received is the conditional `GET`, which typically means a request with an `If-Modified-Since` header. The HTTP specification has this to say:

```
The semantics of the GET method change to a "conditional GET" if the request message
includes an If-Modified-Since, If-Unmodified-Since, If-Match, If-None-Match, or If-
Range header field.  A conditional GET method requests that the entity be transferred
only under the circumstances described by the conditional header field(s). The
conditional GET method is intended to reduce unnecessary network usage by allowing
cached entities to be refreshed without requiring multiple requests or transferring
data already held by the client.
```

So how can we reduce the unnecessary network usage in such a case? mod_perl makes it easy by providing access to Apache's `meets_conditions()` function (which lives in `Apache::File`). The `Last-Modified` (and possibly `ETag`) headers must be set up before calling this method. If the return value of this method is anything other than `OK`, then this value is the one that should be returned from the handler when we have finished. Apache handles the rest for us. For example:

```
if ((my $result = $r->meets_conditions) != OK) {
    return $result;
}
#else ... go and send the response body ...
```

If we have a Squid accelerator running, it will often handle the conditionals for us, and we can enjoy its extremely fast responses for such requests by reading the *access.log* file. Just *grep* for `TCP_IMS_HIT/304`. However, there are circumstances under which Squid may not be allowed to use its cache. That is why the origin server (which is the server we are programming) needs to handle conditional `GET`s as well, even if a Squid accelerator is running.

HEAD Requests

Among the headers described thus far, the date-related ones (Date, Last-Modified, and Expires/Cache-Control) are usually easy to produce and thus should be computed for HEAD requests just the same as for GET requests.

The Content-Type and Content-Length headers should be exactly the same as would be supplied to the corresponding GET request. But since it may be expensive to compute them, they can easily be omitted, since there is nothing in the specification that requires them to be sent.

What is important is that the response to a HEAD request *must not* contain a message-body. The code in a mod_perl handler might look like this:

```
# compute the headers that are easy to compute
# currently equivalent to $r->method eq "HEAD"
if ( $r->header_only ) {
    $r->send_http_header;
    return OK;
}
```

If a Squid accelerator is being used, it will be able to handle the whole HEAD request by itself, but under some circumstances it may not be allowed to do so.

POST Requests

The response to a POST request is not cacheable, due to an underspecification in the HTTP standards. Section 13.4 does not forbid caching of responses to POST requests, but no other part of the HTTP standard explains how the caching of POST requests could be implemented, so we are in a vacuum. No existing caching servers implement the caching of POST requests (although some browsers with more aggressive caching implement their own caching of POST requests). However, this may change if someone does the groundwork of defining the semantics for cache operations on POST requests.

Note that if a Squid accelerator is being used, you should be aware that it accelerates outgoing traffic but does not bundle incoming traffic. Squid is of no benefit at all on POST requests, which could be a problem if the site receives a lot of long POST requests. Using GET instead of POST means that requests can be cached, so the possibility of using GETs should always be considered. However, unlike with POSTs, there are size limits and visibility issues that apply to GETs, so they may not be suitable in every case.

Avoiding Dealing with Headers

There is another approach to dynamic content that is possible with mod_perl. This approach is appropriate if the content changes relatively infrequently, if we expect lots of requests to retrieve the same content before it changes again, and if it is much cheaper to test whether the content needs refreshing than it is to refresh it.

In this situation, a `PerlFixupHandler` can be installed for the relevant location. This handler must test whether the content is up to date or not, returning `DECLINED` so that the Apache core can serve the content from a file if it is up to date. If the content has expired, the handler should regenerate the content into the file, update the `$r->finfo` status and *still* return `DECLINED`, which will force Apache to serve the now updated file. Updating `$r->finfo` can be achieved by calling:

```
$r->filename($file); # force update of the finfo structure
```

even if this seems redundant because the filename is the same as `$file`. This is important because otherwise Apache would use the out-of-date *finfo* when generating the response header.

References

- "Hypertext Transfer Protocol—HTTP/1.0," RFC 1945T, by T. Berners-Lee, *et al.*: *http://www.w3.org/Protocols/rfc1945/rfc1945/*
- "Hypertext Transfer Protocol—HTTP/1.1," RFC 2616, by R. Fielding, *et al.*: *http://www.w3.org/Protocols/rfc2616/rfc2616/*
- "Cachebusting—Cause and Prevention, by Martin Hamilton. *draft-hamilton-cachebusting-01*. Also available online at *http://vancouver-webpages.com/CacheNow/*.
- *Writing Apache Modules with Perl and C*, by Lincoln Stein and Doug MacEachern (O'Reilly). Selected chapters available online at *http://www.modperl.com/*.
- *mod_perl Developer's Cookbook*, by Geoffrey Young, Paul Lindner, and Randy Kobes (Sams Publishing). Selected chapters and code examples available online at *http://www.modperlcookbook.org/*.
- Prevent the browser from caching a page *http://www.pacificnet.net/~johnr/meta.html*.

  This page is an explanation of how to use the `Meta` HTML tag to prevent caching, by browser or proxy, of an individual page wherein the page in question has data that may be of a sensitive nature (as in a "form page for submittal") and the creator of the page wants to make sure that the page does not get submitted twice.