1 Issuing Correct HTTP Headers

1.1 Description

To make caching of dynamic documents possible, which can give you a considerable performance gain, setting a number of HTTP headers is of a vital importance. This document explains which headers you need to pay attention to, and how to work with them.

As there is always more than one way to do it, I'm tempted to believe one must be the best. Hardly ever am I right.

1.2 The Origin of this Chapter

This chapter has been contributed to the documentation by Andreas Koenig. You will find the references and other related info at the bottom of this page. It was previously distributed from CPAN, but this documentation is now its official resting-place.

If you have any questions regarding this specific document only, please refer to Andreas, since he is the guru on this subject. On any other matter please contact the mod_perl mailing list.

1.3 Why Headers

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

If the number of hits is not significant, don't bother to read this document.

If the number of hits is significant, you might want to consider what cache-friendliness means (you may also want to read [4]) and how you can cooperate with caches to increase the performance of your site. Especially if you use Squid in accelerator mode (helpful hints for Squid, see [1]), you will have a strong motivation to cooperate with it. This document may help you to do it correctly.

1.4 Which Headers

The HTTP standard (v 1.1 is specified in [3], v 1.0 in [2]) describes lots of headers. In this document, we only discuss those headers which are most relevant to caching.

I have grouped the headers into three groups: date headers, content headers, and the special Vary header.

1.4.1 Date Related Headers

1.4.1.1 Date

Section 14.18 of the HTTP standard deals with the circumstances under which you must or must not send a `Date` header. For almost everything a normal mod_perl user is doing, 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 this header is 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`.

1.4.1.2 Last-Modified

Section 14.29 of the HTTP standard deals with this. The `Last-Modified` header is mostly used as a so-called weak validator. Here are two sentences from the HTTP specs:

```
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.
```

This tells us that we should consider the semantics of the page we are generating and not the date when we are running. The question is, when did the **meaning** of this page change last time? Let's imagine the document in question is a text-to-gif renderer that takes as input a font to use, background and foreground colours, and a string to render. Although the actual image is created on-the-fly, the semantics of the page are determined when the script was last changed, right?

Actually, a few more things are relevant: the semantics also change a little when you update one of the fonts that may be used or when you update your `ImageMagick` or equivalent program. It's something you should consider, if you want to get it right.

If you have a page which comprises several components, you should ask all the components when they changed their semantic behaviour last time. Then pick the oldest of those times.

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

update_mtime() takes a UNIX time 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 a previously stored `finfo.st_mtime`.

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

```
use Apache::File;
use Date::Parse;
# Date::Parse parses RCS format, Apache::Util::parsedate doesn't
$Mtime ||=
  Date::Parse::str2time(substr q$Date: 2002/07/31 14:41:49 $, 6);
$r->set_last_modified($Mtime);
```

1.4.1.3 Expires and Cache-Control

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 on until the document expires and it helps the recipients to clean up their stored documents. A sentence 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.
```

So think before you set up a time when you believe a resource should be regarded as stale. Most of the time I can determine an expected lifetime from "now", that is the time of the request. I would not recommend hardcoding the date of Expiry, because when you forget that you did it, and the date arrives, you will serve "already expired" documents that cannot be cached at all by anybody. If you believe a resource will never expire, read this quote from the HTTP specs:

```
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.
```

Now the code for the mod_perl programmer who wants to expire a document half a year from now:

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

A very handy alternative to this computation is available in HTTP 1.1, the cache control mechanism. Instead of setting the `Expires` header you can specify a delta value in a `Cache-Control` header. You can do that by executing just:

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

which is, of course much cheaper than the first example because perl computes the value only once at compile time and optimizes it into a constant.

As this alternative is only available in HTTP 1.1 and old cache servers may not understand this header, it is advisable to send both headers. In this case the `Cache-Control` header takes precedence, so the `Expires` header is ignored on HTTP 1.1 compliant servers. Or you could go with 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));
}
```

If you restart your Apache server regularly, I'd save the `Expires` header in a global variable. Oh, well, this is probably over-engineered now.

To avoid caching altogether call:

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

which sets the headers:

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

which should work in major 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 which ignore `Expires` headers.

1.4.2 Content Related Headers

1.4.2.1 Content-Type

You are most probably familiar with `Content-Type`. Sections 3.7, 7.2.1 and 14.17 of the HTTP specs cover the details. mod_perl has the `content_type()` method to deal with this header, for example:

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

`Content-Type` *should* be included in all messages according to the specs, and Apache will generate one if you don't. It will be whatever is specified in the relevant `DefaultType` configuration directive or `text/plain` if none is active.

1.4.2.2 Content-Length

According to section 14.13 of the HTTP specifications, the `Content-Length` header is the number of octets in the body of a message. If it can be determined prior to sending, it can be very useful for several reasons to include it. The most important reason why it is good to include it is that keepalive requests only work with responses that contain a `Content-Length` header. In mod_perl you can say

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

If you use `Apache::File`, you get the additional `set_content_length()` method for the Apache class which is a bit more efficient than the above. You can then say:

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

The `Content-Length` header can have an important 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.
```

So be careful never to send a wrong `Content-Length`, either in a GET or in a HEAD request.

1.4.2.3 Entity Tags

An `Entity Tag` is a validator which can be used instead of, or in addition to, the `Last-Modified` header. An entity tag is a quoted string which can be used to identify different versions of a particular resource. An entity tag can be added to the response headers like so:

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

Note: mod_perl offers the `Apache::set_etag()` method if you have loaded `Apache::File`. It is strongly recommended that you *do not* use this method unless you know what you are doing. `set_etag()` is expecting 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 you promise the recipient that you will not send the same `ETag` for the same resource again unless the content is *'equal'* to what you are sending now (see below for what equality means).

The pros and cons of using entity tags are discussed in section 13.3 of the HTTP specs. For us 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. Weak validators change when the meaning of the response changes. Strong validators are needed for caches to allow for sub-range requests. Weak validators allow a more efficient caching of equivalent objects. Algorithms like MD5 or SHA are good strong validators, but what we usually want, 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. Please refer to section 13.3.3 of the HTTP standard to understand these rules. This is mostly relevant for range requests as this citation of 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 a If-Range header.
```

But it is not limited to range requests. Section 13.3.1 succinctly states that:

```
The Last-Modified entity-header field value is often 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 the semantics of the output. To prevent these kinds 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 specs say:

```
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 them an `ETag` that is marked as weak we prevent them from using the Last-Modified header as a strong validator.

An `ETag` value is marked as a weak validator by preceding 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\"");
```

Consider carefully which string you choose to act as a validator. You are on your own with this decision because...

```
... 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 you 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 you are producing relatively large documents, or content that does not change frequently, you most likely will prefer a strong entity tag, thus giving caches a chance to transfer the document in chunks. (Anybody in the mood to add a chapter about ranges to this document?)

1.4.3 Content Negotiation

Content negotiation is a particularly wonderful feature that was introduced with HTTP 1.1. Unfortunately it is not yet widely supported. Probably the most popular usage scenario of content negotiation is language negotiation. A user specifies in the browser preferences the languages they understand and how well they understand them. The browser includes these settings in an `Accept-Language` header when it sends the request to the server and the server then chooses from several available representations of the document the one that best fits the user's preferences. Content negotiation is not limited to language. Citing the specs:

```
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.
```

1.4.3.1 Vary

In order 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 it. 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

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

but it has the side effect of being expensive for a caching proxy. As of this writing, Squid (version 2.1PATCH2) does not cache resources that come with a Vary header at all. So unless you find a clever workaround, you won't enjoy your Squid accelerator for these documents :-(

1.5 Requests

Section 13.11 of the specifications states that the only two cacheable methods are `GET` and `HEAD`.

1.5.1 HEAD

Among the above recommended headers, 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 as it can be expensive to compute them, they can just as well be omitted, since there is nothing in the specs that forces you to compute them.

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

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

If you are running a Squid accelerator, it will be able to handle the whole HEAD request for you, but under some circumstances it may not be allowed to do so.

1.5.2 POST

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 caching of POST requests could be implemented, so we are in a vacuum here and all existing caching servers therefore refuse to implement caching of POST requests. This may change if somebody does the groundwork of defining the semantics for cache operations on POST. Note that some browsers with their more aggressive caching do implement caching of POST requests.

Note: If you are running a Squid accelerator, you should be aware that it accelerates outgoing traffic, but does not bundle incoming traffic. If you have long POST requests, Squid doesn't buy you anything. So always consider using a GET instead of a POST if possible.

1.5.3 GET

A normal GET is what we usually write our mod_perl programs for. Nothing special about it. We send our headers followed by the body.

But there is a certain case that needs a workaround to achieve better cacheability. We need to deal with the "?" in the rel_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.
```

You're tempted to believe that if we are using HTTP 1.1 and send an explicit expiration time we're on the safe side? Unfortunately reality is a little bit different. It has been a bad habit for quite a long time to misconfigure cache servers such that they treat all GET requests containing a question mark as uncacheable. People even used to mark everything as uncacheable that contained the string cgi-bin.

To work around this bug in the HEAD requests, I have stopped calling my CGI directories cgi-bin and I have written the following handler that lets me work with CGI-like query strings without rewriting the software (such as Apache::Request and CGI.pm) that deals with them.

```
sub handler {
  my($r) = @_;
  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/%3B/;/gi;
    $u2 =~ s/%26/;/gi; # &
```

```
    $u2 =~ s/%3D/=/gi;
    $r->args($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 such that the first semicolon is interpreted as a question mark and everything after that as the query string. You can now exchange the request:

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

with:

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

Thus it allows the co-existence of queries from ordinary forms that are being processed by a browser and 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%2Fbla
```

you have to use:

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

1.5.4 Conditional GET

A rather challenging request mod_perl programmers can get is the conditional `GET`, which typically means a request with an If-Modified-Since header. The HTTP specifications have 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 for you by offering Apache's `meets_conditions()`. You have to set up your `Last-Modified` (and possibly `ETag`) header before calling this method. If the return value of this method is anything other than `OK`, you should return that value from your handler and you're done. Apache handles the rest for you. The following example is taken from [5]:

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

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

1.6 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 you 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 case a `PerlFixupHandler` can be installed for the relevant location. It tests whether the content is up to date. If so, it returns `DECLINED` and lets the Apache core serve the content from a file. Otherwise, it regenerates the content into the file, updates the `$r->finfo` status and again returns `DECLINED` so that Apache serves the updated file. Updating `$r->finfo` can be achieved by calling

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

even if this seems redundant because the filename is already equal to `$file`. Setting the filename has the side effect of doing a `stat()` on the file. This is important because otherwise Apache would use the out of date `finfo` when generating the response header.

1.7 References

1.7.1 [1]

Stas Bekman: mod_perl Guide

1.7.2 [2]

T. Berners-Lee et al.: Hypertext Transfer Protocol -- HTTP/1.0, RFC 1945.

1.7.3 [3]

R. Fielding et al.: Hypertext Transfer Protocol -- HTTP/1.1, RFC 2616.

1.7.4 [4]

Martin Hamilton: Cachebusting - cause and prevention, draft-hamilton-cachebusting-01. Also available online at http://vancouver-webpages.com/CacheNow/

1.7.5 [5]

Lincoln Stein, Doug MacEachern: Writing Apache Modules with Perl and C, O'Reilly, 1-56592-567-X. Selected chapters available online at http://www.modperl.com/ .

1.8 Other resources

- Prevent the browser from Caching a page http://www.pacificnet.net/~johnr/meta.html

  This page is an explanation of using the Meta 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. Please notice that some of the information on this page is a little bit outdated, but it's still a good resource for those who cannot generate their own HTTP headers.

- Web Caching and Content Delivery Resources http://www.web-caching.com/

1.9 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

- Stas Bekman <stas (at) stason.org>

1.10 Authors

- Andreas Koenig <andreas.koenig (at) anima.de>

Only the major authors are listed above. For contributors see the Changes file.

Table of Contents: