Caching Tutorial for Web Authors and Webmasters

This is an informational document. Although technical in nature, it attempts to make the concepts involved understandable and applicable in real-world situations. Because of this, some aspects of the material are simplified or omitted, for the sake of clarity. If you are interested in the minutia of the subject, please explore the References and Further Information at the end.

What's a Web Cache? Why do people use them?

A *Web cache* sits between one or more Web servers (also known as *origin servers*) and a client or many clients, and watches requests come by, saving copies of the responses — like HTML pages, images and files (collectively known as *representations*) — for itself. Then, if there is another request for the same URL, it can use the response that it has, instead of asking the origin server for it again.

There are two main reasons that Web caches are used:

- To **reduce latency** — Because the request is satisfied from the cache (which is closer to the client) instead of the origin server, it takes less time for it to get the representation and display it. This makes the Web seem more responsive.
- To **reduce network traffic** — Because representations are reused, it reduces the amount of bandwidth used by a client. This saves money if the client is paying for traffic, and keeps their bandwidth requirements lower and more manageable.

Kinds of Web Caches

Browser Caches

If you examine the preferences dialog of any modern Web browser (like Internet Explorer, Safari or Mozilla), you'll probably notice a "cache" setting. This lets you set aside a section of your computer's hard disk to store representations that you've seen, just for you. The browser cache works according to fairly simple rules. It will check to make sure that the representations are fresh, usually once a session (that is, the once in the current invocation of the browser).

This cache is especially useful when users hit the "back" button or click a link to see a page they've just looked at. Also, if you use the same navigation images throughout your site, they'll be served from browsers' caches almost instantaneously.

Proxy Caches

Web proxy caches work on the same principle, but a much larger scale. Proxies serve hundreds or thousands of users in the same way; large corporations and ISPs often set them up on their firewalls, or as standalone devices (also known as *intermediaries*).

Because proxy caches aren't part of the client or the origin server, but instead are out on the network, requests have to be routed to them somehow. One way to do this is to use your browser's proxy setting to manually tell it what proxy to use; another is using interception. *Interception proxies* have Web requests redirected to them by the underlying network itself, so that clients don't need to be configured for them, or even know about them.

Proxy caches are a type of *shared cache*; rather than just having one person using them, they usually have a large number of users, and because of this they are very good at reducing latency and network traffic. That's because popular representations are reused a number of times.

Gateway Caches

Also known as "reverse proxy caches" or "surrogate caches," gateway caches are also intermediaries, but instead of being deployed by network administrators to save bandwidth, they're typically deployed by Webmasters themselves, to make their sites more scalable, reliable and better performing.

Requests can be routed to gateway caches by a number of methods, but typically some form of load balancer is used to make one or more of them look like the origin server to clients.

Content delivery networks (CDNs) distribute gateway caches throughout the Internet (or a part of it) and sell caching to interested Web sites. Speedera and Akamai are examples of CDNs.

This tutorial focuses mostly on browser and proxy caches, although some of the information is suitable for those interested in gateway caches as well.

Aren't Web Caches bad for me? Why should I help them?

Web caching is one of the most misunderstood technologies on the Internet. Webmasters in particular fear losing control of their site, because a proxy cache can "hide" their users from them, making it difficult to see who's using the site.

Unfortunately for them, even if Web caches didn't exist, there are too many variables on the Internet to assure that they'll be able to get an accurate picture of how users see their site. If this is a big concern for you, this tutorial will teach you how to get the statistics you need without making your site cache-unfriendly.

Another concern is that caches can serve content that is out of date, or *stale*. However, this tutorial can show you how to configure your server to control how your content is cached.

On the other hand, if you plan your site well, caches can help your Web site load faster, and save load on your server and Internet link. The difference can be dramatic; a site that is difficult to cache may take several seconds to load, while one that takes advantage of caching can seem instantaneous in comparison. Users will appreciate a fast-loading site, and will visit more often.

Think of it this way; many large Internet companies are spending millions of dollars setting up farms of servers around the world to replicate their content, in order to make it as fast to access as possible for their users. Caches do the same for you, and they're even closer to the end user. Best of all, you don't have to pay for them.

CDNs are an interesting development, because unlike many proxy caches, their gateway caches are aligned with the interests of the Web site being cached, so that these problems aren't seen. However, even when you use a CDN, you still have to consider that there will be proxy and browser caches downstream.

The fact is that proxy and browser caches will be used whether you like it or not. If you don't configure your site to be cached correctly, it will be cached using whatever defaults the cache's administrator decides upon.

How Web Caches Work

All caches have a set of rules that they use to determine when to serve a representation from the cache, if it's available. Some of these rules are set in the protocols (HTTP 1.0 and 1.1), and some are set by the administrator of the cache (either the user of the browser cache, or the proxy administrator).

Generally speaking, these are the most common rules that are followed (don't worry if you don't understand the details, it will be explained below):

1. If the response's headers tell the cache not to keep it, it won't.
2. If the request is authenticated or secure (i.e., HTTPS), it won't be cached by shared caches.

3. A cached representation is considered *fresh* (that is, able to be sent to a client without checking with the origin server) if:
   - It has an expiry time or other age-controlling header set, and is still within the fresh period, or
   - If the cache has seen the representation recently, and it was modified relatively long ago.

   Fresh representations are served directly from the cache, without checking with the origin server.
4. If a representation is stale, the origin server will be asked to *validate* it, or tell the cache whether the copy that it has is still good.
5. Under certain circumstances — for example, when it's disconnected from a network — a cache can serve stale responses without checking with the origin server.

If no validator (an `ETag` or `Last-Modified` header) is present on a response, *and* it doesn't have any explicit freshness information, it will usually — but not always — be considered uncacheable.

Together, *freshness* and *validation* are the most important ways that a cache works with content. A fresh representation will be available instantly from the cache, while a validated representation will avoid sending the entire representation over again if it hasn't changed.

How (and how not) to Control Caches

There are several tools that Web designers and Webmasters can use to fine-tune how caches will treat their sites. It may require getting your hands a little dirty with your server's configuration, but the results are worth it. For details on how to use these tools with your server, see the Implementation sections below.

HTML Meta Tags and HTTP Headers

HTML authors can put tags in a document's <HEAD> section that describe its attributes. These *meta tags* are often used in the belief that they can mark a document as uncacheable, or expire it at a certain time.

Meta tags are easy to use, but aren't very effective. That's because they're only honored by a few browser caches, not proxy caches (which almost never read the HTML in the document). While it may be tempting to put a Pragma: no-cache meta tag into a Web page, it won't necessarily cause it to be kept fresh.

On the other hand, true *HTTP headers* give you a lot of control over how both browser caches and proxies handle your representations. They can't be seen in the HTML, and are usually automatically generated by the Web server. However, you can control them to some degree, depending on the server you use. In the following sections, you'll see what HTTP headers are interesting, and how to apply them to your site.

HTTP headers are sent by the server before the HTML, and only seen by the browser and any intermediate caches. Typical HTTP 1.1 response headers might look like this:

```
HTTP/1.1 200 OK
Date: Fri, 30 Oct 1998 13:19:41 GM
Server: Apache/1.3.3 (Unix)
Cache-Control: max-age=3600, must-
Expires: Fri, 30 Oct 1998 14:19:41
Last-Modified: Mon, 29 Jun 1998 02
ETag: "3e86-410-3596fbbc"
Content-Length: 1040
Content-Type: text/html
```

If your site is hosted at an ISP or hosting farm and they don't give you the ability to set arbitrary HTTP headers (like `Expires` and `Cache-Control`), complain loudly; these are tools necessary for doing your job.

The HTML would follow these headers, separated by a blank line. See the Implementation sections for information about how to set HTTP headers.

Pragma HTTP Headers (and why they don't work)

Many people believe that assigning a `Pragma: no-cache` HTTP header to a representation will make it uncacheable. This is not necessarily true; the HTTP specification does not set any guidelines for Pragma response headers; instead, Pragma request headers (the headers that a browser sends to a server) are discussed. Although a few caches may honor this header, the majority won't, and it won't have any effect. Use the headers below instead.

Controlling Freshness with the Expires HTTP Header

The `Expires` HTTP header is a basic means of controlling caches; it tells all caches how long the associated representation is fresh for. After that time, caches will always check back with the origin server to see if a document is changed. `Expires` headers are supported by practically every cache.

Most Web servers allow you to set `Expires` response headers in a number of ways. Commonly, they will allow setting an absolute time to expire, a time based on the last time that the client retrieved the representation (last *access time*), or a time based on the last time the document changed on your server (last *modification time*).

`Expires` headers are especially good for making static images (like navigation bars and buttons) cacheable. Because they don't change much, you can set extremely long expiry time on them, making your site appear much more responsive to your users. They're also useful for controlling caching of a page that is regularly changed. For instance, if you update a news page once a day at 6am, you can set the representation to expire at that time, so caches will know when to get a fresh copy, without users having to hit 'reload'.

The **only** value valid in an `Expires` header is a HTTP date; anything else will most likely be interpreted as 'in the past', so that the representation is uncacheable. Also, remember that the time in a HTTP date is Greenwich Mean Time (GMT), not local time.

For example:

```
Expires: Fri, 30 Oct 1998 14:19:41 GMT
```

Although the `Expires` header is useful, it has some limitations. First, because there's a date involved, the clocks on the Web server and the cache must be synchronised; if they have a different idea of the time, the intended results won't be achieved, and caches might wrongly consider stale content as fresh.

Another problem with `Expires` is that it's easy to forget that you've set some content to expire at a particular time. If you don't update an `Expires` time before it passes, each and every request will go back to your Web server, increasing load and latency.

It's important to make sure that your Web server's clock is accurate if you use the `Expires` header. One way to do this is using the Network Time Protocol (NTP); talk to your local system administrator to find out more.

Cache-Control HTTP Headers

HTTP 1.1 introduced a new class of headers, `Cache-Control` response headers, to give Web publishers more control over their content, and to address the limitations of `Expires` .

Useful `Cache-Control` response headers include:

- **`max-age=`** [seconds] — specifies the maximum amount of time that a representation will be considered fresh. Similar to `Expires` , this directive is relative to the time of the request, rather than absolute. [seconds] is the number of seconds from the time of the request you wish the representation to be fresh for.
- **`s-maxage=`** [seconds] — similar to `max-age` , except that it only applies to shared (e.g., proxy) caches.
- **`public`** — marks authenticated responses as cacheable; normally, if HTTP authentication is required, responses are automatically private.
- **`private`** — allows caches that are specific to one user (e.g., in a browser) to store the response; shared caches (e.g., in a proxy) may not.
- **`no-cache`** — forces caches to submit the request to the origin server for validation before releasing a cached copy, every time. This is useful to assure that authentication is respected (in combination with public), or to maintain rigid freshness, without sacrificing all of the benefits of caching.
- **`no-store`** — instructs caches not to keep a copy of the representation under any conditions.
- **`must-revalidate`** — tells caches that they must obey any freshness information you give them about a representation. HTTP allows caches to serve stale representations under special conditions; by specifying this header, you're telling the cache that you want it to strictly follow your rules.
- **`proxy-revalidate`** — similar to `must-revalidate` , except that it only applies to proxy caches.

For example:

```
Cache-Control: max-age=3600, must-revalidate
```

When both `Cache-Control` and `Expires` are present, `Cache-Control` takes precedence. If you plan to use the `Cache-Control` headers, you should have a look at the excellent documentation in HTTP 1.1; see References and Further Information.

Validators and Validation

In How Web Caches Work, we said that validation is used by servers and caches to communicate when a representation has changed. By using it, caches avoid having to download the entire representation when they already have a copy locally, but they're not sure if it's still fresh.

Validators are very important; if one isn't present, and there isn't any freshness information (`Expires` or `Cache-Control`) available, caches will not store a representation at all.

The most common validator is the time that the document last changed, as communicated in `Last-Modified` header. When a cache has a representation stored that includes a `Last-Modified` header, it can use it to ask the server if the representation has changed since the last time it was seen, with an `If-Modified-Since` request.

HTTP 1.1 introduced a new kind of validator called the *ETag*. ETags are unique identifiers that are generated by the server and changed every time the representation does. Because the server controls how the ETag is generated, caches can be sure that if the ETag matches when they make a `If-None-Match` request, the representation really is the same.

Almost all caches use Last-Modified times as validators; ETag validation is also becoming prevalent.

Most modern Web servers will generate both `ETag` and `Last-Modified` headers to use as validators for static content (i.e., files) automatically; you won't have to do anything. However, they don't know enough about dynamic content (like CGI, ASP or database sites) to generate them; see Writing Cache-Aware Scripts.

Tips for Building a Cache-Aware Site

Besides using freshness information and validation, there are a number of other things you can do to make your site more cache-friendly.

- **Use URLs consistently** — this is the golden rule of caching. If you serve the same content on different pages, to different users, or from different sites, it should use the same URL. This is the easiest and most effective way to make your site cache-friendly. For example, if you use "/index.html" in your HTML as a reference once, always use it that way.
- **Use a common library of images** and other elements and refer back to them from different places.
- **Make caches store images and pages that don't change often** by using a `Cache-Control: max-age` header with a large value.

- **Make caches recognise regularly updated pages** by specifying an appropriate max-age or expiration time.
- **If a resource (especially a downloadable file) changes, change its name.** That way, you can make it expire far in the future, and still guarantee that the correct version is served; the page that links to it is the only one that will need a short expiry time.
- **Don't change files unnecessarily.** If you do, everything will have a falsely young `Last-Modified` date. For instance, when updating your site, don't copy over the entire site; just move the files that you've changed.
- **Use cookies only where necessary** — cookies are difficult to cache, and aren't needed in most situations. If you must use a cookie, limit its use to dynamic pages.
- **Check your pages with REDbot** — it can help you apply many of the concepts in this tutorial.

Writing Cache-Aware Scripts

By default, most scripts won't return a validator (a `Last-Modified` or `ETag` response header) or freshness information (`Expires` or `Cache-Control`). While some scripts really are dynamic (meaning that they return a different response for every request), many (like search engines and database-driven sites) can benefit from being cache-friendly.

Generally speaking, if a script produces output that is reproducible with the same request at a later time (whether it be minutes or days later), it should be cacheable. If the content of the script changes only depending on what's in the URL, it is cacheable; if the output depends on a cookie, authentication information or other external criteria, it probably isn't.

- The best way to make a script cache-friendly (as well as perform better) is to dump its content to a plain file whenever it changes. The Web server can then treat it like any other Web page, generating and using validators, which makes your life easier. Remember to only write files that have changed, so the `Last-Modified` times are preserved.
- Another way to make a script cacheable in a limited fashion is to set an age-related header for as far in the future as practical. Although this can be done with `Expires` , it's probably easiest to do so with `Cache-Control: max-age` , which will make the request fresh for an amount of time after the request.
- If you can't do that, you'll need to make the script generate a validator, and then respond to `If-Modified-Since` and/or `If-None-Match` requests. This can be done by parsing the HTTP headers, and then responding with `304 Not Modified` when appropriate. Unfortunately, this is not a trival task.

Some other tips;

- **Don't use POST** unless it's appropriate. Responses to the POST method aren't kept by most caches; if you send information in the path or query (via GET), caches can store that information for the future.
- **Don't embed user-specific information in the URL** unless the content generated is completely unique to that user.

- **Don't count on all requests from a user coming from the same host**, because caches often work together.
- **Generate `Content-Length` response headers.** It's easy to do, and it will allow the response of your script to be used in a *persistent connection*. This allows clients to request multiple representations on one TCP/IP connection, instead of setting up a connection for every request. It makes your site seem much faster.

See the Implementation Notes for more specific information.

Frequently Asked Questions

What are the most important things to make cacheable?

A good strategy is to identify the most popular, largest representations (especially images) and work with them first.

How can I make my pages as fast as possible with caches?

The most cacheable representation is one with a long freshness time set. Validation does help reduce the time that it takes to see a representation, but the cache still has to contact the origin server to see if it's fresh. If the cache already knows it's fresh, it will be served directly.

I understand that caching is good, but I need to keep statistics on how many people visit my page!

If you must know every time a page is accessed, select ONE small item on a page (or the page itself), and make it uncacheable, by giving it a suitable headers. For example, you could refer to a 1x1 transparent uncacheable image from each page. The `Referer` header will contain information about what page called it.

Be aware that even this will not give truly accurate statistics about your users, and is unfriendly to the Internet and your users; it generates unnecessary traffic, and forces people to wait for that uncached item to be downloaded. For more information about this, see On Interpreting Access Statistics in the references.

How can I see a representation's HTTP headers?

Many Web browsers let you see the `Expires` and `Last-Modified` headers are in a "page info" or similar interface. If available, this will give you a menu of the page and any representations (like images) associated with it, along with their details.

To see the full headers of a representation, you can manually connect to the Web server using a Telnet client.

To do so, you may need to type the port (be default, 80) into a separate field, or you may need to connect to `www.example.com:80` or `www.example.com 80` (note the space). Consult your Telnet client's documentation.

Once you've opened a connection to the site, type a request for the representation. For instance, if you want to see the headers for `http://www.example.com/foo.html` , connect to

`www.example.com`, port `80`, and type:

```
GET /foo.html HTTP/1.1 [return]
Host: www.example.com [return][return]
```

Press the Return key every time you see `[return]`; make sure to press it twice at the end. This will print the headers, and then the full representation. To see the headers only, substitute HEAD for GET.

My pages are password-protected; how do proxy caches deal with them?

By default, pages protected with HTTP authentication are considered private; they will not be kept by shared caches. However, you can make authenticated pages public with a Cache-Control: public header; HTTP 1.1-compliant caches will then allow them to be cached.

If you'd like such pages to be cacheable, but still authenticated for every user, combine the `Cache-Control: public` and `no-cache` headers. This tells the cache that it must submit the new client's authentication information to the origin server before releasing the representation from the cache. This would look like:

```
Cache-Control: public, no-cache
```

Whether or not this is done, it's best to minimize use of authentication; for example, if your images are not sensitive, put them in a separate directory and configure your server not to force authentication for it. That way, those images will be naturally cacheable.

Should I worry about security if people access my site through a cache?

`https://` pages are not cached (or decrypted) by proxy caches, so you don't have to worry about that. However, because caches store `http://` responses and URLs fetched through them, you should be conscious about unsecured sites; an unscrupulous administrator could conceivably gather information about their users, especially in the URL.

In fact, any administrator on the network between your server and your clients could gather this type of information. One particular problem is when CGI scripts put usernames and passwords in the URL itself; this makes it trivial for others to find and use their login.

If you're aware of the issues surrounding Web security in general, you shouldn't have any surprises from proxy caches.

I'm looking for an integrated Web publishing solution. Which ones are cache-aware?

It varies. Generally speaking, the more complex a solution is, the more difficult it is to cache. The worst are ones which dynamically generate all content and don't provide validators; they may not be cacheable at all. Speak with your vendor's technical staff for more information, and see the Implementation notes below.

My images expire a month from now, but I need to change them in the caches now!

The Expires header can't be circumvented; unless the cache (either browser or proxy) runs out of room and has to delete the representations, the cached copy will be used until then.

The most effective solution is to change any links to them; that way, completely new representations will be loaded fresh from the origin server. Remember that any page that refers to these representations will be cached as well. Because of this, it's best to make static images and similar representations very cacheable, while keeping the HTML pages that refer to them on a tight leash.

If you want to reload a representation from a specific cache, you can either force a reload (in Firefox, holding down shift while pressing 'reload' will do this by issuing a `Pragma: no-cache` request header) while using the cache. Or, you can have the cache administrator delete the representation through their interface.

I run a Web Hosting service. How can I let my users publish cache-friendly pages?

If you're using Apache, consider allowing them to use .htaccess files and providing appropriate documentation.

Otherwise, you can establish predetermined areas for various caching attributes in each virtual server. For instance, you could specify a directory /cache-1m that will be cached for one month after access, and a /no-cache area that will be served with headers instructing caches not to store representations from it.

Whatever you are able to do, it is best to work with your largest customers first on caching. Most of the savings (in bandwidth and in load on your servers) will be realized from high-volume sites.

I've marked my pages as cacheable, but my browser keeps requesting them on every request. How do I force the cache to keep representations of them?

Caches aren't required to keep a representation and reuse it; they're only required to **not** keep or use them under some conditions. All caches make decisions about which representations to keep based upon their size, type (e.g., image vs. html), or by how much space they have left to keep local copies. Yours may not be considered worth keeping around, compared to more popular or larger representations.

Some caches do allow their administrators to prioritize what kinds of representations are kept, and some allow representations to be "pinned" in cache, so that they're always available.

Implementation Notes — Web Servers

Generally speaking, it's best to use the latest version of whatever Web server you've chosen to deploy. Not only will they likely contain more cache-friendly features, new versions also usually have important security and performance improvements.

Apache HTTP Server

Apache uses optional modules to include headers, including both Expires and Cache-Control. Both modules are available in the 1.2 or greater distribution.

The modules need to be built into Apache; although they are included in the distribution, they are not turned on by default. To find out if the modules are enabled in your server, find the httpd binary and run `httpd -l` ; this should print a list of the available modules (note that this only lists compiled-in modules; on later versions of Apache, use `httpd -M` to include dynamically loaded modules as well). The modules we're looking for are expires_module and headers_module.

- If they aren't available, and you have administrative access, you can recompile Apache to include them. This can be done either by uncommenting the appropriate lines in the Configuration file, or using the `-enable-module=expires` and `-enable-module=headers` arguments to configure (1.3 or greater). Consult the INSTALL file found with the Apache distribution.

Once you have an Apache with the appropriate modules, you can use mod_expires to specify when representations should expire, either in .htaccess files or in the server's access.conf file. You can specify expiry from either access or modification time, and apply it to a file type or as a default. See the module documentation for more information, and speak with your local Apache guru if you have trouble.

To apply `Cache-Control` headers, you'll need to use the mod_headers module, which allows you to specify arbitrary HTTP headers for a resource. See the mod_headers documentation.

Here's an example .htaccess file that demonstrates the use of some headers.

- .htaccess files allow web publishers to use commands normally only found in configuration files. They affect the content of the directory they're in and their subdirectories. Talk to your server administrator to find out if they're enabled.

```
### activate mod_expires
ExpiresActive On
### Expire .gif's 1 month from when they're accessed
ExpiresByType image/gif A2592000
### Expire everything else 1 day from when it's last modi
### (this uses the Alternative syntax)
ExpiresDefault "modification plus 1 day"
### Apply a Cache-Control header to index.html
<Files index.html>
Header append Cache-Control "public, must-revalidate"
</Files>
```

- Note that mod_expires automatically calculates and inserts a `Cache-Control:max-age` header as appropriate.

Apache 2's configuration is very similar to that of 1.3; see the 2.2 mod_expires and mod_headers documentation for more information.

Microsoft IIS

Microsoft's Internet Information Server makes it very easy to set headers in a somewhat flexible way. Note that this is only possible in version 4 of the server, which will run only on NT Server.

To specify headers for an area of a site, select it in the `Administration Tools` interface, and bring up its properties. After selecting the `HTTP Headers` tab, you should see two interesting areas; `Enable Content Expiration` and `Custom HTTP headers`. The first should be self-explanatory, and the second can be used to apply Cache-Control headers.

See the ASP section below for information about setting headers in Active Server Pages. It is also possible to set headers from ISAPI modules; refer to MSDN for details.

Netscape/iPlanet Enterprise Server

As of version 3.6, Enterprise Server does not provide any obvious way to set Expires headers. However, it has supported HTTP 1.1 features since version 3.0. This means that HTTP 1.1 caches (proxy and browser) will be able to take advantage of Cache-Control settings you make.

To use Cache-Control headers, choose `Content Management | Cache Control Directives` in the administration server. Then, using the Resource Picker, choose the directory where you want to set the headers. After setting the headers, click 'OK'. For more information, see the NES manual.

Implementation Notes — Server-Side Scripting

Because the emphasis in server-side scripting is on dynamic content, it doesn't make for very cacheable pages, even when the content could be cached. If your content changes often, but not on every page hit, consider setting a Cache-Control: max-age header; most users access pages again in a relatively short period of time. For instance, when users hit the 'back' button, if there isn't any validator or freshness information available, they'll have to wait until the page is re-downloaded from the server to see it.

> **One thing to keep in mind is that it may be easier to set HTTP headers with your Web server rather than in the scripting language. Try both.**

CGI

CGI scripts are one of the most popular ways to generate content. You can easily append HTTP response headers by adding them before you send the body; Most CGI implementations already require you to do this for the `Content-Type` header. For instance, in Perl;

```
#!/usr/bin/perl
print "Content-type: text/html\n";
print "Expires: Thu, 29 Oct 1998 17:04:19 GMT\n";
print "\n";
### the content body follows...
```

Since it's all text, you can easily generate `Expires` and other date-related headers with in-built functions. It's even easier if you use `Cache-Control: max-age`;

```
print "Cache-Control: max-age=600\n";
```

This will make the script cacheable for 10 minutes after the request, so that if the user hits the 'back' button, they won't be resubmitting the request.

The CGI specification also makes request headers that the client sends available in the environment of the script; each header has 'HTTP_' prepended to its name. So, if a client makes an `If-Modified-Since` request, it will show up as `HTTP_IF_MODIFIED_SINCE`.

See also the cgi_buffer library, which automatically handles ETag generation and validation, `Content-Length` generation and gzip content-coding for Perl and Python CGI scripts with a one-line include. The Python version can also be used to wrap arbitrary CGI scripts with.

Server Side Includes

SSI (often used with the extension .shtml) is one of the first ways that Web publishers were able to get dynamic content into pages. By using special tags in the pages, a limited form of in-HTML scripting was available.

Most implementations of SSI do not set validators, and as such are not cacheable. However, Apache's implementation does allow users to specify which SSI files can be cached, by setting the group execute permissions on the appropriate files, combined with the `XbitHack full` directive. For more information, see the mod_include documentation.

PHP

PHP is a server-side scripting language that, when built into the server, can be used to embed scripts inside a page's HTML, much like SSI, but with a far larger number of options. PHP can be used as a CGI script on any Web server (Unix or Windows), or as an Apache module.

By default, representations processed by PHP are not assigned validators, and are therefore uncacheable. However, developers can set HTTP headers by using the `Header()` function.

For example, this will create a Cache-Control header, as well as an Expires header three days in the future:

```
<?php
 Header("Cache-Control: must-revalidate");

 $offset = 60 * 60 * 24 * 3;
 $ExpStr = "Expires: " . gmdate("D, d M Y H:i:s", time() -
 Header($ExpStr);
?>
```

Remember that the `Header()` function MUST come before any other output.

As you can see, you'll have to create the HTTP date for an `Expires` header by hand; PHP doesn't provide a function to do it for you (although recent versions have made it easier; see the PHP's date documentation). Of course, it's easy to set a `Cache-Control: max-age` header, which is just as good for most situations.

For more information, see the manual entry for header.

See also the cgi_buffer library, which automatically handles `ETag` generation and validation, `Content-Length` generation and gzip content-coding for PHP scripts with a one-line include.

Cold Fusion

Cold Fusion, by Macromedia is a commercial server-side scripting engine, with support for several Web servers on Windows, Linux and several flavors of Unix.

Cold Fusion makes setting arbitrary HTTP headers relatively easy, with the `CFHEADER` tag. Unfortunately, their example for setting an `Expires` header, as below, is a bit misleading.

```
<CFHEADER NAME="Expires" VALUE="#Now()#">
```

It doesn't work like you might think, because the time (in this case, when the request is made) doesn't get converted to a HTTP-valid date; instead, it just gets printed as a representation of Cold Fusion's Date/Time object. Most clients will either ignore such a value, or convert it to a default, like January 1, 1970.

However, Cold Fusion does provide a date formatting function that will do the job; `GetHttpTimeString`. In combination with `DateAdd`, it's easy to set Expires dates; here, we set a header to declare that representations of the page expire in one month;

```
<cfheader name="Expires"
  value="#GetHttpTimeString(DateAdd('m', 1, Now()))#">
```

You can also use the `CFHEADER` tag to set `Cache-Control: max-age` and other headers.

Remember that Web server headers are passed through in some deployments of Cold Fusion (such as CGI); check yours to determine whether you can use this to your advantage, by setting headers on the server instead of in Cold Fusion.

ASP and ASP.NET

Active Server Pages, built into IIS and also available for other Web servers, also allows you to set HTTP headers. For instance, to set an expiry time, you can use the properties of the `Response` object;

```
<% Response.Expires=1440 %>
```

specifying the number of minutes from the request to expire the representation. `Cache-Control` headers can be added like this:

```
<% Response.CacheControl="public"
```

In ASP.NET, `Response.Expires` is deprecated; the proper way to set cache-related headers is with `Response.Cache`;

```
Response.Cache.SetExpires ( DateTi
Response.Cache.SetCacheability ( P
```

When setting HTTP headers from ASPs, make sure you either place the Response method calls before any HTML generation, or use `Response.Buffer` to buffer the output. Also, note that some versions of IIS set a `Cache-Control: private` header on ASPs by default, and must be declared public to be cacheable by shared caches.

References and Further Information

HTTP 1.1 Specification

The HTTP 1.1 spec has many extensions for making pages cacheable, and is the authoritative guide to implementing the protocol. See sections 13, 14.9, 14.21, and 14.25.

Web-Caching.com

An excellent introduction to caching concepts, with links to other online resources.

On Interpreting Access Statistics

Jeff Goldberg's informative rant on why you shouldn't rely on access statistics and hit counters.

REDbot

Examines HTTP resources to determine how they will interact with Web caches, and generally how well they use the protocol.

cgi_buffer Library

One-line include in Perl CGI, Python CGI and PHP scripts automatically handles ETag generation and validation, Content-Length generation and gzip Content-Encoding — correctly. The Python version can also be used as a wrapper around arbitrary CGI scripts.

About This Document

The latest revision of this document can always be obtained from https://www.mnot.net/cache_docs/

Translations are available in: Chinese, Czech, German, and French.

6 May, 2013