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

**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](http://www.speedera.com/) and [Akamai](http://www.akamai.com/) 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.

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.

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.

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 no validator (an ETag or Last-Modified header) is present on a response, it will be considered uncacheable.
3. If the request is authenticated or secure, it won’t be cached.
4. 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.
   * If a browser cache has already seen the representation, and has been set to check once a session.
   * If a proxy 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.

1. If an 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.

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](http://www.mnot.net/cache_docs/#IMP-SERVER) 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 (which actually read the HTML), 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.

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.

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 OKDate: Fri, 30 Oct 1998 13:19:41 GMTServer: Apache/1.3.3 (Unix)Cache-Control: max-age=3600, must-revalidateExpires: Fri, 30 Oct 1998 14:19:41 GMTLast-Modified: Mon, 29 Jun 1998 02:28:12 GMTETag: "3e86-410-3596fbbc"Content-Length: 1040Content-Type: text/html

The HTML would follow these headers, separated by a blank line. See the [Implementation](http://www.mnot.net/cache_docs/#IMP-SERVER) 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 saw 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

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](http://www.ntp.org/) (NTP); talk to your local system administrator to find out more.

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.

**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 an 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 uncacheable.
* **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

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](http://www.mnot.net/cache_docs/#REF).

**Validators and Validation**

In [How Web Caches Work](http://www.mnot.net/cache_docs/#WORK), we said that validation is used by servers and caches to communicate when an 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 an 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 surer 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 in determining if an representation is fresh; 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](http://www.mnot.net/cache_docs/#SCRIPT).

**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 may 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 recognize 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.
* **Minimize use of SSL** — because encrypted pages are not stored by shared caches, use them only when you have to, and use images on SSL pages sparingly.
* **use the** [**Cacheability Engine**](http://www.mnot.net/cacheability/) — 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 reproducable 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 cacheble; 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.