Introduction from Equiinet

The following document is made publicly available by its author (see copyright notices below), and is supplied here in its entirety.

Of particular interest are the descriptions of HTTP headers, and tips for build a Cache-Aware site.

The same author has created a Cacheability Engine which can give an indication on how cacheable web pages are likely to be. This can be accessed via: http://www.cachepilot.com/news/Tools.asp

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 Web servers (or origin servers) and a client or many clients, and watches requests for HTML pages, images and files (collectively known as objects) come by, saving a copy for itself. Then, if there is another request for the same object, it will use the copy 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 the client to get the object and display it. This makes Web sites seem more responsive.
- To **reduce traffic** - Because each object is only gotten from the server once, it reduces the amount of bandwidth used by a client. This saves money if the client is paying by 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 browser (like Internet Explorer or Netscape), you'll probably notice a 'cache' setting. This lets you set aside a section of your computer's hard disk to store objects that you've seen, just for you. The browser cache works according to fairly simple rules. It will check to make sure that the objects are fresh, usually once a session (that is, the once in the current invocation of the browser).

This cache is useful when a client hits the 'back' button to go to a page they've already seen. Also, if you use the same navigation images throughout your site, they'll be served from the browser cache 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 ISP's often set them up on their firewalls.

Because proxy caches usually have a large number of users behind them, they are very good at reducing latency and traffic. That's because popular objects are requested only once, and served to a large number of clients.

Most proxy caches are deployed by large companies or ISPs that want to reduce the amount of Internet bandwidth that they use. Because the cache is shared by a large number of users, there are a large number of shared hits (objects that are requested by

a number of clients). Hit rates of 50% efficiency or greater are not uncommon. Proxy caches are a type of shared cache.

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 cache can 'hide' their users from them, making it difficult to see who's using the site.

Unfortunately for them, even if no Web caches were used, 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 document 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 document can show you how to configure your server to control this, while making it more cacheable.

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 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 an object 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 for a particular request (don't worry if you don't understand the details, it will be explained below):

1. If the object's headers tell the cache not to keep the object, it won't. Also, if no validator is present, most caches will mark the object as uncacheable.
2. If the object is authenticated or secure, it won't be cached.
3. A cached object 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 directive set, and is still within the fresh period.
    - If a browser cache has already seen the object, and has been set to check once a session.
    - If a proxy cache has seen the object recently, and it was modified relatively long ago.

    Fresh documents are served directly from the cache, without checking with the origin server.

4. If an object is stale, the origin server will be asked to validate the object, 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 object will be available instantly from the cache, while a validated object will avoid sending the entire object 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 the server 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 vs. 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 usually only honored by 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 slap a Pragma: no-cache meta tag on a home page, it won't necessarily cause it to be kept fresh, if it goes through a shared cache.

On the other hand, true HTTP headers give you a lot of control over how both browser caches and proxies handle your objects. 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.

- 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.

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 GMT
Server: Apache/1.3.3 (Unix)
Cache-Control: max-age=3600, must-revalidate
Expires: Fri, 30 Oct 1998 14:19:41 GMT
Last-Modified: Mon, 29 Jun 1998 02:28:12 GMT
ETag: "3e86-410-3596fbbc"
Content-Length: 1040
Content-Type: text/html
```

The HTML document would follow these headers, separated by a blank line.

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

Many people believe that assigning a Pragma: no-cache HTTP header to an object 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 the basic means of controlling caches; it tells all caches how long the object 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 client.

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 object (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 object 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 object 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
```

Cache-Control HTTP Headers

Although the Expires header is useful, it is still somewhat limited; there are many situations where content is cacheable, but the HTTP 1.0 protocol lacks methods of telling caches what it is, or how to work with it.

HTTP 1.1 introduces a new class of headers, the Cache-Control response headers, which allow Web publishers to define how pages should be handled by caches. They include directives to declare what should be cacheable, what may be stored by caches, modifications of the expiration mechanism, and revalidation and reload controls.

Interesting Cache-Control response headers include:

- **max-age=**[seconds] - specifies the maximum amount of time that an object will be considered fresh. Similar to Expires, this directive allows more flexibility. [seconds] is the number of seconds from the time of the request you wish the object to be fresh for.
- **s-maxage=**[seconds] - similar to max-age, except that it only applies to proxy (shared) caches.

- **public** - marks the response as cacheable, even if it would normally be uncacheable. For instance, if your pages are authenticated, the public directive makes them cacheable.
- **no-cache** - forces caches (both proxy and browser) 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 object freshness, without sacrificing all of the benefits of caching.
- **must-revalidate** - tells caches that they must obey any freshness information you give them about an object. The HTTP allows caches to take liberties with the freshness of objects; 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 the HTTP 1.1 draft; 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 an object has changed. By using it, caches avoid having to download the entire object 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, most caches will not store an object at all.

The most common validator is the time that the document last changed, the Last-Modified time. When a cache has an object stored that includes a Last-Modified header, it can use it to ask the server if the object 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 object 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 object really is the same.

Almost all caches use Last-Modified times in determining if an object is fresh; as more HTTP/1.1 caches come online, Etag headers will also be used.

Most modern Web servers will generate both ETag and Last-Modified validators for static content 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.

- **Refer to objects 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 specifying a far-away Expires header.
- **Make caches recognize regularly updated pages** by specifying an appropriate 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** - 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 (e.g., a Last-Modified or ETag HTTP 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;

- **If you have to use scripting, don't POST** unless it's appropriate. The POST method is (practically) impossible to cache; if you send information in the path or query (via GET), caches can store that information for the future. POST, on the other hand, is good for sending large amount of information to the server (which is why it won't be cached; it's very unlikely that the same exact POST will be made twice).
- **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 a client (whether a proxy or a browser) to request multiple objects 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 objects (especially images) and work with them first.

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

The most cacheable object is one with a long freshness time set. Validation does help reduce the time that it takes to see an object, 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 object 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.

I've got a page that is updated often. How do I keep caches from giving my users a stale copy?

The Expires header is the best way to do this. By setting the server to expire the document based on its modification time, you can automatically have caches mark it as stale a set amount of time after it is changed.

For example, if your site's home page changes every day at 8am, set the Expires header for 23 hours after the last modification time. This way, your users will always get a fresh copy of the page.

See also the Cache-Control: max-age header.

How can I see which HTTP headers are set for an object?

To see what the Expires and Last-Modified headers are, open the page with Netscape and select 'page info' from the View menu. This will give you a menu of the page and any objects (like images) associated with it, along with their details.

To see the full headers of an object, you'll need to manually connect to the Web server using a Telnet client. Depending on what program you use, you may need to type the

port into a separate field, or you may need to connect to www.myhost.com:80 or www.myhost.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 object. For instance, if you want to see the headers for http://www.myhost.com/foo.html, connect to www.myhost.com, port 80, and type:

```
GET /foo.html HTTP/1.1 [return]
Host: www.myhost.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 object. 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 marked private; they will not be cached by shared caches. However, you can mark authenticated pages public with a Cache-Control header; HTTP 1.1-compliant caches will then allow them to be cached.

If you'd like the 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 object from the cache.

Whether or not this is done, it's best to minimize use of authentication; for instance, 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 my users access my site through a cache?

SSL pages are not cached (or unencrypted) by proxy caches, so you don't have to worry about that. However, because caches store non-SSL requests and URLs fetched through them, you should be conscious of security on unsecured sites; an unscrupulous administrator could conceivably gather information about their users.

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 user 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 objects, the cached copy will be used until then.

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

If you want to reload an object from a specific cache, you can either force a reload (in Netscape, 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 object 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 provide 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 objects 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.

A Note About the HTTP

HTTP 1.1 compliance is mentioned several times in this document. As of the time it was written, the protocol is a work in progress. Because of this, it is virtually impossible for an application (whether a server, proxy or client) to be truly compliant. However, the protocol has been openly discussed for some time, and feature-frozen for enough time to allow developers to use the ideas contained in it, like Cache-

Control and ETags. When HTTP 1.1 is final, expect more vendors to openly state that their applications are compliant.

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 1.3

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. The modules we're looking for are mod_expires and mod_headers.

- 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 objects 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
modified
### (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.

Netscape Enterprise 3.6

Netscape 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 NES manual.

MS IIS 4.0

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.

Lotus Domino R5

Lotus' servers are notoriously difficult to cache; they don't provide any validators, so both browser and proxy caches can only use default mechanisms (i.e., once per session, and a few minutes of 'fresh' time, usually) to cache any content from them.

Even if this limitation is overcome, Notes' habit of referring to the same object by different URLs (depending on a variety of factors) bars any measurable gains. There is also no documented way to set an Expires, Cache-Control or other arbitrary HTTP header.

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 an Expires header, even if just for a few hours. 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_' appended to its name. So, if a client makes an If-Modified-Since request, it may show up like this:

```
HTTP_IF_MODIFIED_SINCE = Fri, 30 Oct 1998 14:19:41 GMT
```

See also the cgi_buffer library, which automatically handles ETag generation and validation, Content-Length generation and gzip Content-Encoding 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, objects 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;
  $ExpireString = "Expires: " . gmdate("D, d M Y
H:i:s", time() + $offset) . " GMT";
  Header($ExpireString);
?>
```

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. 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-Encoding for PHP scripts with a one-line include.

Cold Fusion 4.0

Cold Fusion, by Allaire is a commercial server-side scripting engine, with support for several Web servers on Windows and Solaris.

Cold Fusion makes setting arbitrary HTTP headers relatively easy, with the CFHEADER tag. Unfortunately, setting date-related functions in Cold Fusion isn't easy as Allaire's documentation leads you to believe; their example for setting an Expires header, as below, won't work.

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

It doesn't work 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.

Cold Fusion's date formatting functions make it difficult generate a date that is HTTP-valid; you'll need to either use a combination of `DateFormat`, `Hour`, `Minute` and `Second`, or roll your own. Of course, you can still use the CFHEADER tag to set Cache-Control: max-age and other headers.

Also, Remember that Web server headers are passed through with some implementations (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

Active Server Pages, build into IIS and now becoming available in other implementations, also allow you to set HTTP headers. For instance, to set an expiry time, use the properties of the Response object in your page, like this:

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

specifying the number of minutes from the request to expire the object. Likewise, absolute expiry time can be set like this (make sure you format HTTP date correctly):

```
<% Response.ExpiresAbsolute=#May 31,1996 13:30:15 GMT# %>
```

Cache-Control headers can be added like this:

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

- 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.
- Note that ASPs set a Cache-Control: private header by default, and must be declared public to be cacheable by HTTP 1.1 shared caches. While you're at it, consider giving them an Expires header as well.

References and Further Information

HTTP 1.1 Specification

http://www.w3.org/Protocols/
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 Overview

http://www.web-caching.com/
An excellent introduction to caching concepts, with links to other online resources.

Cache Now! Campaign

http://vancouver-webpages.com/CacheNow/
Cache Now! is a campaign to raise awareness of caching, from all perspectives.

On Interpreting Access Statistics

http://www.cranfield.ac.uk/docs/stats/
Jeff Goldberg's informative paper on why you shouldn't rely on access statistics and hit counters.

Cacheability Engine (See below)

http://www.mnot.net/cacheability/
Examines Web pages to determine how they will interact with Web caches, the Engine is a good debugging tool, and a companion to this tutorial.

cgi_buffer Library

http://www.mnot.net/cgi_buffer/

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 copy of this document can always be obtained from
http://www.mnot.net/cache_docs/

Version 1.5 - January 19, 2003

Cacheability Engine

To help you understand how Web Caches will treat a Web page, the To help you understand how Web Caches will treat a Web page, the Cacheability Engine will look at a URL (and optionally any images or objects associated with it), giving both specific cache-related data about it, and a general commentary on how cacheable the object is.

Remember, however, that it can't tell you everything about how cacheable your page is; it can only help you make an informed decision about your site.

For more information about cacheability, see the Tutorial.above.

The Cacheability Engine is free to use; see the bottom of this page for more information.

how do I use it?

The Engine can be accessed in a few ways;

1. as a public web Engine

- http://www.ircache.net/cgi-bin/cacheability.py

2. as an extension to your Web browser

By installing a browser extension, you can check any page's cacheability as you're surfing, with a single click. See the public Engine pages for more details.

3. by installing it locally

If you'll be using the Engine often, you can download it and install it on a computer of your choice. Once it's there, you can use it as a CGI script or a command-line utility. See the README file included for details.

how do I interpret the results?

The output of the Engine takes the form of a cascading list of objects. In the Web interface, the color of the list dot indicates how cachable the object is; a red dot is very uncacheable, a yellow one somewhat cacheable, and a green dot is quite cacheable. You can click on the object's URL to open a window to look at it.

Under the URL is a list of different HTTP headers and their values. For information about what each does, read the Tutorial.

If the object has a Last-Modified header, information about whether it supports validation will be written on that line. If the object has a validator, but returns a full response to a conditional request with the same object as before, the script will write "Validation not Supported".

Finally, each object will have a paragraph of commentary associated, which will point out the various ways which it will interact with a cache. For more information about the concepts involved, see the Tutorial.

In the Web interface, the object list will be followed by a list of links that were referenced by the URL. You can check the cacheability of each of these by following the links there.

Date headers and clock skew

In the course of using the Engine, you may get messages (either at the top of the Web interface, or in the commentary) about Date headers and clock skew. Briefly, if the clock of a Web server is inaccurate, it can cause problems with Caching, which is very date-dependant. If you get this message, you should synchronise the clock on the Web server; ask your System Administrator, or see www.ntp.org for more information.

If you see a message that the Web server didn't generate Date headers when it should have, you have a buggy Web server, and should upgrade.

limitations

While the Engine does its best to show you everything that may affect an object's cacheability, there are some things it can't do.

- **appropriate freshness** - The Engine does not suggest freshness values for objects; it only reports whether they are available. Only you can determine the proper values.
- **displayed values** - The date and other values displayed are parsed and reformatted; if you need to see the actual values served, look at the server output directly.
- **authentication** - HTTP authentication can't be entered yet. Note that it would never be a good idea to allow input of authentication information on a public service.
- **cookies** - The effects of cookies set from pages that aren't part of the query can't be gauged. Keep in mind that if a cookie is set to be sent for a directory/server combination that matches the object, it will be requested on every reference.
- **accelerators and transparent proxies** - If one of these devices is between the Engine and the Web server, it can have unpredictable effects, particularly on date-related fields and calculations (possibly including validation).
- **Web server farms** - If your site is served by a farm of servers, the Engine may get responses for different objects from different servers. This normally isn't a problem, unless the servers' contents, software version or clocks are out of sync.

- **proxies and dates** - Although the script can be configured to use a proxy, it is not recommended for the same reasons as above. If you must use one, use a tunnel (a proxy that doesn't modify or cache the connection).
- **object types** - Finally, some object types aren't fetched by the Engine upon reference, and will need to be checked directly. These include Java applets, image maps (client and server side) and URLs in Javascript or DHTML.
- **HTML and JavaScript refresh/redirect tags** - Some Web sites use META tags and/or JavaScript to redirect users to another page, rather than using HTTP redirect headers. Because there are a variety of ways that this can happen, the Engine can't automatically handle them. You'll have to go to their destinations manually.

will look at a URL (and optionally any images or objects associated with it), giving both specific cache-related data about it, and a general commentary on how cacheable the object is.

Remember, however, that it can't tell you everything about how cacheable your page is; it can only help you make an informed decision about your site.

For more information about cacheability, see the Tutorial.

The Cacheability Engine is free to use; see the bottom of this page for more information.

how do I use it?

The Engine can be accessed in a few ways;

1. as a public web Engine

The following sites host public copies:

- IRCache [host site]
- web-caching.com [host site]

If you'd like to make your own public page available, please contact me.

2. as an extension to your Web browser

By installing a browser extension, you can check any page's cacheability as you're surfing, with a single click. See the public Engine pages for more details.

3. by installing it locally

If you'll be using the Engine often, you can download it and install it on a computer of your choice. Once it's there, you can use it as a CGI script or a command-line utility. See the README file included for details.

how do I interpret the results?

The output of the Engine takes the form of a cascading list of objects. In the Web interface, the color of the list dot indicates how cachable the object is; a red dot is very uncacheable, a yellow one somewhat cacheable, and a green dot is quite cacheable. You can click on the object's URL to open a window to look at it.

Under the URL is a list of different HTTP headers and their values. For information about what each does, read the Tutorial.

If the object has a Last-Modified header, information about whether it supports validation will be written on that line. If the object has a validator, but returns a full response to a conditional request with the same object as before, the script will write "Validation not Supported".

Finally, each object will have a paragraph of commentary associated, which will point out the various ways which it will interact with a cache. For more information about the concepts involved, see the Tutorial.

In the Web interface, the object list will be followed by a list of links that were referenced by the URL. You can check the cacheability of each of these by following the links there.

Date headers and clock skew

In the course of using the Engine, you may get messages (either at the top of the Web interface, or in the commentary) about Date headers and clock skew. Breifly, if the clock of a Web server is inaccurate, it can cause problems with Caching, which is very date-dependant. If you get this message, you should synchronise the clock on the Web server; ask your System Administrator, or see www.ntp.org for more information.

If you see a message that the Web server didn't generate Date headers when it should have, you have a buggy Web server, and should upgrade.

limitations

While the Engine does its best to show you everything that may affect an object's cacheability, there are some things it can't do.

- **appropriate freshness** - The Engine does not suggest freshness values for objects; it only reports whether they are available. Only you can determine the proper values.
- **displayed values** - The date and other values displayed are parsed and reformatted; if you need to see the actual values served, look at the server output directly.
- **authentication** - HTTP authentication can't be entered yet. Note that it would never be a good idea to allow input of authentication information on a public service.

- **cookies** - The effects of cookies set from pages that aren't part of the query can't be gauged. Keep in mind that if a cookie is set to be sent for a directory/server combination that matches the object, it will be requested on every reference.
- **accelerators and transparent proxies** - If one of these devices is between the Engine and the Web server, it can have unpredictable effects, particularly on date-related fields and calculations (possibly including validation).
- **Web server farms** - If your site is served by a farm of servers, the Engine may get responses for different objects from different servers. This normally isn't a problem, unless the servers' contents, software version or clocks are out of sync.
- **proxies and dates** - Alhthough the script can be configured to use a proxy, it is not recommended for the same reasons as above. If you must use one, use a tunnel (a proxy that doesn't modify or cache the connection).
- **object types** - Finally, some object types aren't fetched by the Engine upon reference, and will need to be checked directly. These include Java applets, imagemaps (client and server side) and URLs in Javascript or DHTML.
- **HTML and JavaScript refresh/redirect tags** - Some Web sites use META tags and/or JavaScript to redirect users to another page, rather than using HTTP redirect headers. Because there are a variety of ways that this can happen, the Engine can't automatically handle them. You'll have to go to their destinations manually.