Caching Guide

This document supplements the mod_cache, mod_cache_disk, mod_file_cache and htcacheclean reference documentation. It describes how to use the Apache HTTP Server's caching features to accelerate web and proxy serving, while avoiding common problems and misconfigurations.

Introduction

The Apache HTTP server offers a range of caching features that are designed to improve the performance of the server in various ways.

Three-state RFC2616 HTTP caching
: mod_cache and its provider modules mod_cache_disk provide intelligent, HTTP-aware caching. The content itself is stored in the cache, and mod_cache aims to honor all of the various HTTP headers and options that control the cacheability of content as described in Section 13 of RFC2616. mod_cache is aimed at both simple and complex caching configurations, where you are dealing with proxied content, dynamic local content or have a need to speed up access to local files on a potentially slow disk.

Two-state key/value shared object caching
: The shared object cache API (socache) and its provider modules provide a server wide key/value based shared object cache. These modules are designed to cache low level data such as SSL sessions and authentication credentials. Backends allow the data to be stored server wide in shared memory, or datacenter wide in a cache such as memcache or distcache.

Specialized file caching
: mod_file_cache offers the ability to pre-load files into memory on server startup, and can improve access times and save file handles on files that are accessed often, as there is no need to go to disk on each request.

To get the most from this document, you should be familiar with the basics of HTTP, and have read the Users' Guides to Mapping URLs to the Filesystem and Content negotiation.

Three-state RFC2616 HTTP caching

Related Modules	Related Directives
mod_cache	CacheEnable
mod_cache_disk	CacheDisable
	UseCanonicalName
	CacheNegotiatedDocs

The HTTP protocol contains built in support for an in-line caching mechanism described by section 13 of RFC2616, and the mod_cache module can be used to take advantage of this.

Unlike a simple two state key/value cache where the content disappears completely when no longer fresh, an HTTP cache includes a mechanism to retain stale content, and to ask the origin server whether this stale content has changed and if not, make it fresh again.

An entry in an HTTP cache exists in one of three states:

Fresh
: If the content is new enough (younger than its **freshness lifetime**), it is considered **fresh**. An HTTP cache is free to serve fresh content without making any calls to the origin server at all.

Stale

If the content is too old (older than its **freshness lifetime**), it is considered **stale**. An HTTP cache should contact the origin server and check whether the content is still fresh before serving stale content to a client. The origin server will either respond with replacement content if not still valid, or ideally, the origin server will respond with a code to tell the cache the content is still fresh, without the need to generate or send the content again. The content becomes fresh again and the cycle continues.

The HTTP protocol does allow the cache to serve stale data under certain circumstances, such as when an attempt to freshen the data with an origin server has failed with a 5xx error, or when another request is already in the process of freshening the given entry. In these cases a `Warning` header is added to the response.

Non Existent

If the cache gets full, it reserves the option to delete content from the cache to make space. Content can be deleted at any time, and can be stale or fresh. The htcacheclean tool can be run on a once off basis, or deployed as a daemon to keep the size of the cache within the given size, or the given number of inodes. The tool attempts to delete stale content before attempting to delete fresh content.

Full details of how HTTP caching works can be found in Section 13 of RFC2616.

Interaction with the Server

The `mod_cache` module hooks into the server in two possible places depending on the value of the `CacheQuickHandler` directive:

Quick handler phase

This phase happens very early on during the request processing, just after the request has been parsed. If the content is found within the cache, it is served immediately and almost all request processing is bypassed.

In this scenario, the cache behaves as if it has been "bolted on" to the front of the server.

This mode offers the best performance, as the majority of server processing is bypassed. This mode however also bypasses the authentication and authorization phases of server processing, so this mode should be chosen with care when this is important.

Requests with an "Authorization" header (for example, HTTP Basic Authentication) are neither cacheable nor served from the cache when `mod_cache` is running in this phase.

Normal handler phase

This phase happens late in the request processing, after all the request phases have completed.

In this scenario, the cache behaves as if it has been "bolted on" to the back of the server.

This mode offers the most flexibility, as the potential exists for caching to occur at a precisely controlled point in the filter chain, and cached content can be filtered or personalized before being sent to the client.

If the URL is not found within the cache, `mod_cache` will add a filter to the filter stack in order to record the response to the cache, and then stand down, allowing normal request processing to continue. If the content is determined to be cacheable, the content will be saved to the cache for future serving, otherwise the content will be ignored.

If the content found within the cache is stale, the mod_cache module converts the request into a **conditional request**. If the origin server responds with a normal response, the normal response is cached, replacing the content already cached. If the origin server responds with a 304 Not Modified response, the content is marked as fresh again, and the cached content is served by the filter instead of saving it.

Improving Cache Hits

When a virtual host is known by one of many different server aliases, ensuring that UseCanonicalName is set to On can dramatically improve the ratio of cache hits. This is because the hostname of the virtual-host serving the content is used within the cache key. With the setting set to On virtual-hosts with multiple server names or aliases will not produce differently cached entities, and instead content will be cached as per the canonical hostname.

Freshness Lifetime

Well formed content that is intended to be cached should declare an explicit freshness lifetime with the Cache-Control header's max-age or s-maxage fields, or by including an Expires header.

At the same time, the origin server defined freshness lifetime can be overridden by a client when the client presents their own Cache-Control header within the request. In this case, the lowest freshness lifetime between request and response wins.

When this freshness lifetime is missing from the request or the response, a default freshness lifetime is applied. The default freshness lifetime for cached entities is one hour, however this can be easily over-ridden by using the CacheDefaultExpire directive.

If a response does not include an Expires header but does include a Last-Modified header, mod_cache can infer a freshness lifetime based on a heuristic, which can be controlled through the use of the CacheLastModifiedFactor directive.

For local content, or for remote content that does not define its own Expires header, mod_expires may be used to fine-tune the freshness lifetime by adding max-age and Expires.

The maximum freshness lifetime may also be controlled by using the CacheMaxExpire.

A Brief Guide to Conditional Requests

When content expires from the cache and becomes stale, rather than pass on the original request, httpd will modify the request to make it conditional instead.

When an ETag header exists in the original cached response, mod_cache will add an If-None-Match header to the request to the origin server. When a Last-Modified header exists in the original cached response, mod_cache will add an If-Modified-Since header to the request to the origin server. Performing either of these actions makes the request **conditional**.

When a conditional request is received by an origin server, the origin server should check whether the ETag or the Last-Modified parameter has changed, as appropriate for the request. If not, the origin should respond with a terse "304 Not Modified" response. This signals to the cache that the stale content is still fresh should be used for subsequent requests until the content's new freshness lifetime is reached again.

If the content has changed, then the content is served as if the request were not conditional to begin with.

Conditional requests offer two benefits. Firstly, when making such a request to the origin server, if the content from the origin matches the content in the cache, this can be determined easily and without the overhead of transferring the entire resource.

Secondly, a well designed origin server will be designed in such a way that conditional requests will be significantly cheaper to produce than a full response. For static files, typically all that is involved is a call to `stat()` or similar system call, to see if the file has changed in size or modification time. As such, even local content may still be served faster from the cache if it has not changed.

Origin servers should make every effort to support conditional requests as is practical, however if conditional requests are not supported, the origin will respond as if the request was not conditional, and the cache will respond as if the content had changed and save the new content to the cache. In this case, the cache will behave like a simple two state cache, where content is effectively either fresh or deleted.

What Can be Cached?

The full definition of which responses can be cached by an HTTP cache is defined in RFC2616 Section 13.4 Response Cacheability, and can be summed up as follows:

1. Caching must be enabled for this URL. See the `CacheEnable` and `CacheDisable` directives.
2. If the response has an HTTP status code other than 200, 203, 300, 301 or 410 it must also specify an "Expires" or "Cache-Control" header.
3. The request must be a HTTP GET request.
4. If the response contains an "Authorization:" header, it must also contain an "s-maxage", "must-revalidate" or "public" option in the "Cache-Control:" header, or it won't be cached.
5. If the URL included a query string (e.g. from a HTML form GET method) it will not be cached unless the response specifies an explicit expiration by including an "Expires:" header or the max-age or s-maxage directive of the "Cache-Control:" header, as per RFC2616 sections 13.9 and 13.2.1.
6. If the response has a status of 200 (OK), the response must also include at least one of the "Etag", "Last-Modified" or the "Expires" headers, or the max-age or s-maxage directive of the "Cache-Control:" header, unless the `CacheIgnoreNoLastMod` directive has been used to require otherwise.
7. If the response includes the "private" option in a "Cache-Control:" header, it will not be stored unless the `CacheStorePrivate` has been used to require otherwise.
8. Likewise, if the response includes the "no-store" option in a "Cache-Control:" header, it will not be stored unless the `CacheStoreNoStore` has been used.
9. A response will not be stored if it includes a "Vary:" header containing the match-all "*".

What Should Not be Cached?

It should be up to the client creating the request, or the origin server constructing the response to decide whether or not the content should be cacheable or not by correctly setting the `Cache-Control` header, and `mod_cache` should be left alone to honor the wishes of the client or server as appropriate.

Content that is time sensitive, or which varies depending on the particulars of the request that are not covered by HTTP negotiation, should not be cached. This content should declare itself uncacheable using the `Cache-Control` header.

If content changes often, expressed by a freshness lifetime of minutes or seconds, the content can still be cached, however it is highly desirable that the origin server supports **conditional requests** correctly to ensure that full responses do not have to be generated on a regular basis.

Content that varies based on client provided request headers can be cached through intelligent use of the `Vary` response header.

Variable/Negotiated Content

When the origin server is designed to respond with different content based on the value of headers in the request, for example to serve multiple languages at the same URL, HTTP's caching mechanism makes it possible to cache multiple variants of the same page at the same URL.

This is done by the origin server adding a `Vary` header to indicate which headers must be taken into account by a cache when determining whether two variants are different from one another.

If for example, a response is received with a vary header such as;

```
Vary: negotiate,accept-language,accept-charset
```

mod_cache will only serve the cached content to requesters with accept-language and accept-charset headers matching those of the original request.

Multiple variants of the content can be cached side by side, mod_cache uses the `Vary` header and the corresponding values of the request headers listed by `Vary` to decide on which of many variants to return to the client.

Cache Setup Examples

Related Modules	Related Directives
mod_cache	CacheEnable
mod_cache_disk	CacheRoot
mod_cache_socache	CacheDirLevels
mod_socache_memcache	CacheDirLength
	CacheSocache

Caching to Disk

The mod_cache module relies on specific backend store implementations in order to manage the cache, and for caching to disk mod_cache_disk is provided to support this.

Typically the module will be configured as so;

```
CacheRoot   "/var/cache/apache/"
CacheEnable disk /
CacheDirLevels 2
CacheDirLength 1
```

Importantly, as the cached files are locally stored, operating system in-memory caching will typically be applied to their access also. So although the files are stored on disk, if they are frequently accessed it is likely the operating system will ensure that they are actually served from memory.

Understanding the Cache-Store

To store items in the cache, mod_cache_disk creates a 22 character hash of the URL being requested. This hash incorporates the hostname, protocol, port, path and any CGI arguments to the URL, as well as elements defined by the Vary header to ensure that multiple URLs do not collide with one another.

Each character may be any one of 64-different characters, which mean that overall there are 64^22 possible hashes. For example, a URL might be hashed to `xyTGxSMO2b68mBCykqkp1w`. This hash is used as a prefix for the naming of the files specific to that URL within the cache, however first it is split up into directories as per the `CacheDirLevels` and `CacheDirLength` directives.

`CacheDirLevels` specifies how many levels of subdirectory there should be, and `CacheDirLength` specifies how many characters should be in each directory. With the example settings given above, the hash would be turned into a filename prefix as `/var/cache/apache/x/y/TGxSMO2b68mBCykqkp1w`.

The overall aim of this technique is to reduce the number of subdirectories or files that may be in a particular directory, as most file-systems slow down as this number increases. With setting of "1" for `CacheDirLength` there can at most be 64 subdirectories at any particular level. With a setting of 2 there can be 64 * 64 subdirectories, and so on. Unless you have a good reason not to, using a setting of "1" for `CacheDirLength` is recommended.

Setting `CacheDirLevels` depends on how many files you anticipate to store in the cache. With the setting of "2" used in the above example, a grand total of 4096 subdirectories can ultimately be created. With 1 million files cached, this works out at roughly 245 cached URLs per directory.

Each URL uses at least two files in the cache-store. Typically there is a ".header" file, which includes meta-information about the URL, such as when it is due to expire and a ".data" file which is a verbatim copy of the content to be served.

In the case of a content negotiated via the "Vary" header, a ".vary" directory will be created for the URL in question. This directory will have multiple ".data" files corresponding to the differently negotiated content.

Maintaining the Disk Cache

The `mod_cache_disk` module makes no attempt to regulate the amount of disk space used by the cache, although it will gracefully stand down on any disk error and behave as if the cache was never present.

Instead, provided with httpd is the htcacheclean tool which allows you to clean the cache periodically. Determining how frequently to run htcacheclean and what target size to use for the cache is somewhat complex and trial and error may be needed to select optimal values.

htcacheclean has two modes of operation. It can be run as persistent daemon, or periodically from cron. htcacheclean can take up to an hour or more to process very large (tens of gigabytes) caches and if you are running it from cron it is recommended that you determine how long a typical run takes, to avoid running more than one instance at a time.

It is also recommended that an appropriate "nice" level is chosen for htcacheclean so that the tool does not cause excessive disk io while the server is running.

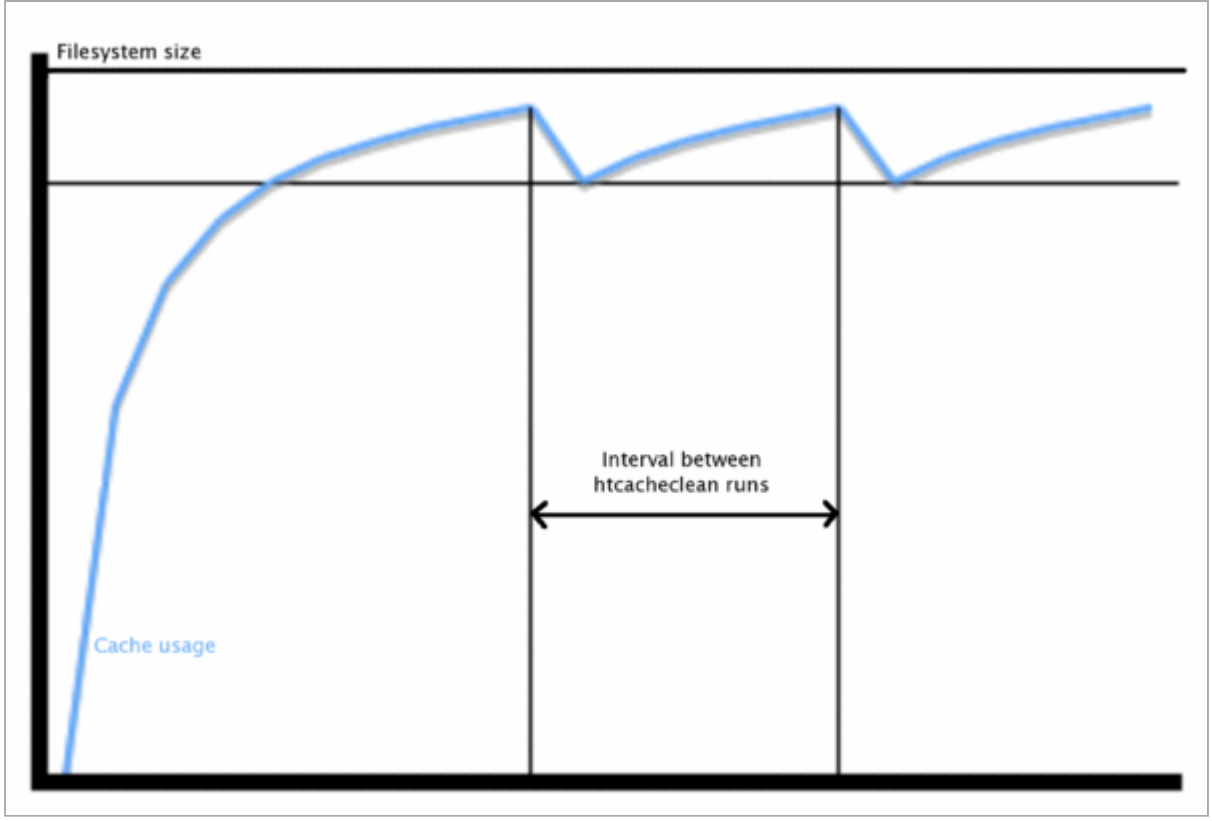

***Figure 1**: Typical cache growth / clean sequence.*

Because `mod_cache_disk` does not itself pay attention to how much space is used you should ensure that htcacheclean is configured to leave enough "grow room" following a clean.

Caching to memcached

Using the `mod_cache_socache` module, `mod_cache` can cache data from a variety of implementations (aka: "providers"). Using the `mod_socache_memcache` module, for example, one can specify that memcached is to be used as the the backend storage mechanism.

Typically the module will be configured as so:

```
CacheEnable socache /
CacheSocache memcache:memcd.example.com:11211
```

Additional `memcached` servers can be specified by appending them to the end of the `CacheSocache memcache:` line separated by commas:

```
CacheEnable socache /
CacheSocache memcache:mem1.example.com:11211,mem2.examp
```

This format is also used with the other various `mod_cache_socache` providers. For example:

```
CacheEnable socache /
CacheSocache shmcb:/path/to/datafile(512000)
```

```
CacheEnable socache /
CacheSocache dbm:/path/to/datafile
```

General Two-state Key/Value Shared Object Caching

Related Modules	Related Directives
`mod_authn_socache`	AuthnCacheSOCache
`mod_socache_dbm`	SSLSessionCache

mod_socache_dc	SSLStaplingCache
mod_socache_memcache	
mod_socache_shmcb	
mod_ssl	

The Apache HTTP server offers a low level shared object cache for caching information such as SSL sessions, or authentication credentials, within the socache interface.

Additional modules are provided for each implementation, offering the following backends:

mod_socache_dbm
: DBM based shared object cache.

mod_socache_dc
: Distcache based shared object cache.

mod_socache_memcache
: Memcache based shared object cache.

mod_socache_shmcb
: Shared memory based shared object cache.

Caching Authentication Credentials

Related Modules	Related Directives
mod_authn_socache	AuthnCacheSOCache

The mod_authn_socache module allows the result of authentication to be cached, relieving load on authentication backends.

Caching SSL Sessions

Related Modules	Related Directives
mod_ssl	SSLSessionCache
	SSLStaplingCache

The mod_ssl module uses the socache interface to provide a session cache and a stapling cache.

Specialized File Caching

Related Modules	Related Directives
mod_file_cache	CacheFile
	MMapFile

On platforms where a filesystem might be slow, or where file handles are expensive, the option exists to pre-load files into memory on startup.

On systems where opening files is slow, the option exists to open the file on startup and cache the file handle. These options can help on systems where access to static files is slow.

File-Handle Caching

The act of opening a file can itself be a source of delay, particularly on network filesystems. By maintaining a cache of open file descriptors for commonly served files, httpd can avoid this delay. Currently httpd provides one implementation of File-Handle Caching.

CacheFile

The most basic form of caching present in httpd is the file-handle caching provided by mod_file_cache. Rather than caching file-contents, this cache maintains a table of open file descriptors. Files to be cached in this manner are specified in the configuration file using the CacheFile directive.

The CacheFile directive instructs httpd to open the file when it is started and to re-use this file-handle for all subsequent access to this file.

```
CacheFile /usr/local/apache2/htdocs/index.html
```

If you intend to cache a large number of files in this manner, you must ensure that your operating system's limit for the number of open files is set appropriately.

Although using CacheFile does not cause the file-contents to be cached per-se, it does mean that if the file changes while httpd is running these changes will not be picked up. The file will be consistently served as it was when httpd was started.

If the file is removed while httpd is running, it will continue to maintain an open file descriptor and serve the file as it was when httpd was started. This usually also means that although the file will have been deleted, and not show up on the filesystem, extra free space will not be recovered until httpd is stopped and the file descriptor closed.

In-Memory Caching

Serving directly from system memory is universally the fastest method of serving content. Reading files from a disk controller or, even worse, from a remote network is orders of magnitude slower. Disk controllers usually involve physical processes, and network access is limited by your available bandwidth. Memory access on the other hand can take mere nano-seconds.

System memory isn't cheap though, byte for byte it's by far the most expensive type of storage and it's important to ensure that it is used efficiently. By caching files in memory you decrease the amount of memory available on the system. As we'll see, in the case of operating system caching, this is not so much of an issue, but when using httpd's own in-memory caching it is important to make sure that you do not allocate too much memory to a cache. Otherwise the system will be forced to swap out memory, which will likely degrade performance.

Operating System Caching

Almost all modern operating systems cache file-data in memory managed directly by the kernel. This is a powerful feature, and for the most part operating systems get it right. For example, on Linux, let's look at the difference in the time it takes to read a file for the first time and the second time;

```
colm@coroebus:~$ time cat testfile > /dev/null
real    0m0.065s
user    0m0.000s
sys     0m0.001s
colm@coroebus:~$ time cat testfile > /dev/null
real    0m0.003s
user    0m0.003s
sys     0m0.000s
```

Even for this small file, there is a huge difference in the amount of time it takes to read the file. This is because the kernel has cached the file contents in memory.

By ensuring there is "spare" memory on your system, you can ensure that more and more file-contents will be stored in this cache. This can be a very efficient means of in-memory caching, and involves no extra configuration of httpd at all.

Additionally, because the operating system knows when files are deleted or modified, it can automatically remove file contents from the cache when

necessary. This is a big advantage over httpd's in-memory caching which has no way of knowing when a file has changed.

Despite the performance and advantages of automatic operating system caching there are some circumstances in which in-memory caching may be better performed by httpd.

MMapFile Caching

mod_file_cache provides the MMapFile directive, which allows you to have httpd map a static file's contents into memory at start time (using the mmap system call). httpd will use the in-memory contents for all subsequent accesses to this file.

```
MMapFile /usr/local/apache2/htdocs/index.html
```

As with the CacheFile directive, any changes in these files will not be picked up by httpd after it has started.

The MMapFile directive does not keep track of how much memory it allocates, so you must ensure not to over-use the directive. Each httpd child process will replicate this memory, so it is critically important to ensure that the files mapped are not so large as to cause the system to swap memory.

Security Considerations

Authorization and Access Control

Using mod_cache in its default state where CacheQuickHandler is set to On is very much like having a caching reverse-proxy bolted to the front of the server. Requests will be served by the caching module unless it determines that the origin server should be queried just as an external cache would, and this drastically changes the security model of httpd.

As traversing a filesystem hierarchy to examine potential .htaccess files would be a very expensive operation, partially defeating the point of caching (to speed up requests), mod_cache makes no decision about whether a cached entity is authorised for serving. In other words; if mod_cache has cached some content, it will be served from the cache as long as that content has not expired.

If, for example, your configuration permits access to a resource by IP address you should ensure that this content is not cached. You can do this by using the CacheDisable directive, or mod_expires. Left unchecked, mod_cache - very much like a reverse proxy - would cache the content when served and then serve it to any client, on any IP address.

When the CacheQuickHandler directive is set to Off, the full set of request processing phases are executed and the security model remains unchanged.

Local exploits

As requests to end-users can be served from the cache, the cache itself can become a target for those wishing to deface or interfere with content. It is important to bear in mind that the cache must at all times be writable by the user which httpd is running as. This is in stark contrast to the usually recommended situation of maintaining all content unwritable by the Apache user.

If the Apache user is compromised, for example through a flaw in a CGI process, it is possible that the cache may be targeted. When using mod_cache_disk, it is relatively easy to insert or modify a cached entity.

This presents a somewhat elevated risk in comparison to the other types of attack it is possible to make as the Apache user. If you are using mod_cache_disk you should bear this in mind - ensure you upgrade httpd when security upgrades are

announced and run CGI processes as a non-Apache user using suEXEC if possible.

Cache Poisoning

When running httpd as a caching proxy server, there is also the potential for so-called cache poisoning. Cache Poisoning is a broad term for attacks in which an attacker causes the proxy server to retrieve incorrect (and usually undesirable) content from the origin server.

For example if the DNS servers used by your system running httpd are vulnerable to DNS cache poisoning, an attacker may be able to control where httpd connects to when requesting content from the origin server. Another example is so-called HTTP request-smuggling attacks.

This document is not the correct place for an in-depth discussion of HTTP request smuggling (instead, try your favourite search engine) however it is important to be aware that it is possible to make a series of requests, and to exploit a vulnerability on an origin webserver such that the attacker can entirely control the content retrieved by the proxy.

Denial of Service / Cachebusting

The Vary mechanism allows multiple variants of the same URL to be cached side by side. Depending on header values provided by the client, the cache will select the correct variant to return to the client. This mechanism can become a problem when an attempt is made to vary on a header that is known to contain a wide range of possible values under normal use, for example the `User-Agent` header. Depending on the popularity of the particular web site thousands or millions of duplicate cache entries could be created for the same URL, crowding out other entries in the cache.

In other cases, there may be a need to change the URL of a particular resource on every request, usually by adding a "cachebuster" string to the URL. If this content is declared cacheable by a server for a significant freshness lifetime, these entries can crowd out legitimate entries in a cache. While `mod_cache` provides a `CacheIgnoreURLSessionIdentifiers` directive, this directive should be used with care to ensure that downstream proxy or browser caches aren't subjected to the same denial of service issue.