BLOG TECH Faisal Memon of F5 July 23, 2015

A Guide to Caching with NGINX and NGINX Plus

caching

We all know that the performance of applications and web sites is a critical factor in their success. The process of making your application or web site perform better, however, is not always clear. Code quality and infrastructure are of course critical, but in many cases you can make vast improvements to the end user experience of your application by focusing on some very basic application delivery techniques. One such example is by implementing and optimizing caching in your application stack. This blog post covers techniques that can help both novice and advanced users see better performance from utilizing the content cache features included in NGINX and NGINX Plus.

Overview

A content cache sits in between a client and an "origin server", and saves copies of all the content it sees. If a client requests content that the cache has stored, it returns the content directly without contacting the origin server. This improves performance as the content cache is closer to the client, and more efficiently uses the application servers because they don't have to do the work of generating pages from scratch each time.

There are potentially multiple caches between the web browser and the application server: the client's browser cache, intermediary caches, content delivery networks (CDNs), and the load balancer or reverse proxy sitting in front of the application servers. Caching, even at the reverse proxy/load balancer level, can greatly improve performance.

As an example, last year I took on the task of performance tuning a website that was loading slowly. One of the first things I noticed was that it took over 1 second to generate the main home page. After some debugging, I discovered that because the page was marked as not cacheable, it was being dynamically generated in response to each request. The page itself was not changing very often and was not personalized, so this was not necessary. As an experiment, I marked the homepage to be cached for 5 seconds by the load balancer, and just doing that resulted in noticeable improvement. The time to first byte went down to a few milliseconds and the page loaded visibly faster.

NGINX is commonly deployed as a reverse proxy or load balancer in an application stack and has a full set of caching features. The next section discusses how to configure basic caching with NGINX.

How to Set Up and Configure Basic Caching

Only two directives are needed to enable basic caching: `proxy_cache_path` and `proxy_cache`. The `proxy_cache_path` directive sets the path and configuration of the cache, and the `proxy_cache` directive activates it.

Chat with Codey

```
                 inactive=60m use_temp_path=off;

server {
    # ...
    location / {
        proxy_cache my_cache;
        proxy_pass http://my_upstream;
    }
}
```

The parameters to the `proxy_cache_path` directive define the following settings:

- The local disk directory for the cache is called **/path/to/cache/**.
- `levels` sets up a two-level directory hierarchy under **/path/to/cache/**. Having a large number of files in a single directory can slow down file access, so we recommend a two-level directory hierarchy for most deployments. If the `levels` parameter is not included, NGINX puts all files in the same directory.
- `keys_zone` sets up a shared memory zone for storing the cache keys and metadata such as usage timers. Having a copy of the keys in memory enables NGINX to quickly determine if a request is a `HIT` or a `MISS` without having to go to disk, greatly speeding up the check. A 1-MB zone can store data for about 8,000 keys, so the 10-MB zone configured in the example can store data for about 80,000 keys.
- `max_size` sets the upper limit of the size of the cache (to 10 gigabytes in this example). It is optional; not specifying a value allows the cache to grow to use all available disk space. When the cache size reaches the limit, a process called the *cache manager* removes the files that were least recently used to bring the cache size back under the limit.
- `inactive` specifies how long an item can remain in the cache without being accessed. In this example, a file that has not been requested for 60 minutes is automatically deleted from the cache by the cache manager process, regardless of whether or not it has expired. The default value is 10 minutes (`10m`). Inactive content differs from expired content. NGINX does not automatically delete content that has expired as defined by a cache control header (`Cache-Control:max-age=120` for example). Expired (stale) content is deleted only when it has not been accessed for the time specified by `inactive`. When expired content is accessed, NGINX refreshes it from the origin server and resets the `inactive` timer.
- NGINX first writes files that are destined for the cache to a temporary storage area, and the `use_temp_path=off` directive instructs NGINX to write them to the same directories where they will be cached. We recommend that you set this parameter to `off` to avoid unnecessary copying of data between file systems. `use_temp_path` was introduced in NGINX version 1.7.10 and NGINX Plus R6.

And finally, the `proxy_cache` directive activates caching of all content that matches the URL of the parent `location` block (in the example, **/**). You can also include the `proxy_cache` directive in a `server` block; it applies to all `location` blocks for the server that don't have their own `proxy_cache` directive.

Chat with Codey

Delivering Cached Content When the Origin is Down

A powerful feature of NGINX content caching is that NGINX can be configured to deliver stale content from its cache when it can't get fresh content from the origin servers. This can happen if all the origin servers for a cached resource are down or temporarily busy. Rather than relay the error to the client, NGINX delivers the stale version of the file from its cache. This provides an extra level of directive:

```
location / {
    # ...
    proxy_cache_use_stale error timeout http_500 http_502 http_503
http_504;
}
```

With this sample configuration, if NGINX receives an `error`, `timeout`, or any of the specified `5xx` errors from the origin server and it has a stale version of the requested file in its cache, it delivers the stale file instead of relaying the error to the client.

Fine-Tuning the Cache and Improving Performance

NGINX has a wealth of optional settings for fine-tuning the performance of the cache. Here is an example that activates a few of them:

```
proxy_cache_path /path/to/cache levels=1:2 keys_zone=my_cache:10m
max_size=10g
                 inactive=60m use_temp_path=off;

server {
    # ...
    location / {
        proxy_cache my_cache;
        proxy_cache_revalidate on;
        proxy_cache_min_uses 3;
        proxy_cache_use_stale error timeout updating http_500 http_502
                              http_503 http_504;
        proxy_cache_background_update on;
        proxy_cache_lock on;

        proxy_pass http://my_upstream;
    }
}
```

Chat with Codey

These directives configure the following behavior:

- `proxy_cache_revalidate` instructs NGINX to use conditional `GET` requests when refreshing content from the origin servers. If a client requests an item that is cached but expired as defined by the cache control headers, NGINX includes the `If-Modified-Since` field in the header of the `GET` request it sends to the origin server. This saves on bandwidth, because the server sends the full item only if it has been modified since the time recorded in the `Last-Modified` header attached to the file when NGINX originally cached it.
- `proxy_cache_min_uses` sets the number of times an item must be requested by clients before NGINX caches it. This is useful if the cache is constantly filling up, as it ensures that only the most frequently accessed items are added to the cache. By default `proxy_cache_min_uses` is set to 1.
- The `updating` parameter to the `proxy_cache_use_stale` directive, combined with enabling the `proxy_cache_background_update` directive, instructs NGINX to deliver stale content when clients request an item that is expired or is in the process of being updated from the origin server. All updates will be done in the background. The stale file is returned for all requests until the updated file is fully downloaded.

wait for that request to be satisfied and then pull the file from the cache. Without `proxy_cache_lock` enabled, all requests that result in cache misses go straight to the origin server.

Splitting the Cache Across Multiple Hard Drives

If you have multiple hard drives, NGINX can be used to split the cache across them. Here is an example that splits clients evenly across two hard drives based on the request URI:

```
proxy_cache_path /path/to/hdd1 levels=1:2 keys_zone=my_cache_hdd1:10m
                 max_size=10g inactive=60m use_temp_path=off;
proxy_cache_path /path/to/hdd2 levels=1:2 keys_zone=my_cache_hdd2:10m
                 max_size=10g inactive=60m use_temp_path=off;

split_clients $request_uri $my_cache {
              50%          “my_cache_hdd1”;
              50%          “my_cache_hdd2”;
}

server {
    # ...
    location / {
        proxy_cache $my_cache;
        proxy_pass http://my_upstream;
    }
}
```

The two `proxy_cache_path` directives define two caches (`my_cache_hdd1` and `my_cache_hdd2`) on two different hard drives. The `split_clients` configuration block specifies that the results from half the requests (`50%`) are cached in `my_cache_hdd1` and the other half in `my_cache_hdd2`. The hash based on the `$request_uri` variable (the request URI) determines which cache is used for each request, the result being that requests for a given URI are always cached in the same cache.

Please note this approach is not a replacement for a RAID hard drive setup. If there is a hard drive failure this could lead to unpredictable behavior on the system, including users seeing 500 response codes for requests that were directed to the failed hard drive. A proper RAID hard drive setup can handle hard drive failures.

Chat with Codey

Frequently Asked Questions (FAQ)

This section answers some frequently asked questions about NGINX content caching.

Can the NGINX Cache Be Instrumented?

Yes, with the `add_header` directive:

```
add_header X-Cache-Status $upstream_cache_status;
```

This example adds an `X-Cache-Status` HTTP header in responses to clients. The following are the possible values for `$upstream_cache_status`:

- `MISS` – The response was not found in the cache and so was fetched from an origin server. The response might then have been cached.
- `BYPASS` – The response was fetched from the origin server instead of served from the cache because the request matched a `proxy_cache_bypass` directive (see

- EXPIRED – The entry in the cache has expired. The response contains fresh content from the origin server.
- STALE – The content is stale because the origin server is not responding correctly, and `proxy_cache_use_stale` was configured.
- UPDATING – The content is stale because the entry is currently being updated in response to a previous request, and `proxy_cache_use_stale updating` is configured.
- REVALIDATED – The `proxy_cache_revalidate` directive was enabled and NGINX verified that the current cached content was still valid (`If-Modified-Since` or `If-None-Match`).
- HIT – The response contains valid, fresh content direct from the cache.

How Does NGINX Determine Whether or Not to Cache Something?

By default, NGINX respects the `Cache-Control` headers from origin servers. It does not cache responses with `Cache-Control` set to `Private`, `No-Cache`, or `No-Store` or with `Set-Cookie` in the response header. NGINX only caches `GET` and `HEAD` client requests. You can override these defaults as described in the answers below.

NGINX does not cache responses if `proxy_buffering` is set to `off`. It is `on` by default.

Can `Cache-Control` Headers Be Ignored?

Yes, with the `proxy_ignore_headers` directive. For example, with this configuration:

```
location /images/ {
    proxy_cache my_cache;
    proxy_ignore_headers Cache-Control;
    proxy_cache_valid any 30m;
    # ...
}
```

NGINX ignores the `Cache-Control` header for everything under **/images/**. The `proxy_cache_valid` directive enforces an expiration for the cached data and is required if ignoring `Cache-Control` headers. NGINX does not cache files that have no expiration.

Can NGINX Cache Content with a `Set-Cookie` in the Header?

Yes, with the `proxy_ignore_headers` directive, as discussed in the previous answer.

Can NGINX Cache `POST` Requests?

Yes, with the `proxy_cache_methods` directive:

```
proxy_cache_methods GET HEAD POST;
```

This example enables caching of `POST` requests.

Can NGINX Cache Dynamic Content?

Yes, provided the `Cache-Control` header allows for it. Caching dynamic content for even a short period of time can reduce load on origin servers and databases, which improves time to first byte, as the page does not have to be regenerated for each request.

Can I Punch a Hole Through My Cache?

Yes, with the `proxy_cache_bypass` directive:

```
    proxy_cache_bypass $cookie_nocache $arg_nocache;
    # ...
}
```

The directive defines request types for which NGINX requests content from the origin server immediately instead of trying to find it in the cache first. This is sometimes referred to as "punching a hole" through the cache. In this example, NGINX does it for requests with a `nocache` cookie or argument, for example `http://www.example.com/?nocache=true`. NGINX can still cache the resulting response for future requests that aren't bypassed.

What Cache Key Does NGINX Use?

The default form of the keys that NGINX generates is similar to an MD5 hash of the following NGINX variables: `$scheme$proxy_host$request_uri`; the actual algorithm used is slightly more complicated.

```
proxy_cache_path /path/to/cache levels=1:2 keys_zone=my_cache:10m
max_size=10g
                 inactive=60m use_temp_path=off;

server {
    # ...
    location / {
        proxy_cache my_cache;
        proxy_pass http://my_upstream;
    }
}
```

For this sample configuration, the cache key for `http://www.example.org/my_image.jpg` is calculated as `md5("http://my_upstream:80/my_image.jpg")`.

Note that `$proxy_host` variable is used in the hashed value instead of the actual host name (`www.example.com`). `$proxy_host` is defined as the name and port of the proxied server as specified in the `proxy_pass` directive.

To change the variables (or other terms) used as the basis for the key, use the `proxy_cache_key` directive (see also the following question).

Chat with Codey

Can I Use a Cookie as Part of My Cache Key?

Yes, the cache key can be configured to be any arbitrary value, for example:

```
proxy_cache_key $proxy_host$request_uri$cookie_jessionid;
```

This example incorporates the value of the `JSESSIONID` cookie into the cache key. Items with the same URI but different `JSESSIONID` values are cached separately as unique items.

Does NGINX Use the `ETag` Header?

In NGINX 1.7.3 and NGINX Plus R5 and later, the `ETag` header is fully supported along with `If-None-Match`.

How Does NGINX Handle Byte Range Requests?

entire file from the origin server. If the request is for a single byte range, NGINX sends that range to the client as soon as it is encountered in the download stream. If the request specifies multiple byte ranges within the same file, NGINX delivers the entire file to the client when the download completes.

Once the download completes, NGINX moves the entire resource into the cache so that all future byte-range requests, whether for a single range or multiple ranges, are satisfied immediately from the cache.

Please note that the `upstream` server must support byte range requests for NGINX to honor byte range requests to that `upstream` server.

Does NGINX Support Cache Purging?

NGINX Plus supports selective purging of cached files. This is useful if a file has been updated on the origin server but is still valid in the NGINX Plus cache (the `Cache-Control:max-age` is still valid and the timeout set by the `inactive` parameter to the `proxy_cache_path` directive has not expired). With the cache-purge feature of NGINX Plus, this file can easily be deleted. For more details, see Purging Content from the Cache.

How Does NGINX Handle the `Pragma` Header?

The `Pragma:no-cache` header is added by clients to bypass all intermediary caches and go straight to the origin server for the requested content. NGINX does not honor the `Pragma` header by default, but you can configure the feature with the following `proxy_cache_bypass` directive:

```
location /images/ {
    proxy_cache my_cache;
    proxy_cache_bypass $http_pragma;
    # ...
}
```

Does NGINX Support the `stale-while-revalidate` and `stale-if-error` Extensions to the `Cache-Control` Header?

Yes, in NGINX Plus R12 and NGINX 1.11.10 and later. What these extensions do:

- The `stale-while-revalidate` extension of the `Cache-Control` HTTP header permits using a stale cached response if it is currently being updated.

  The `stale-if-error` extension of the `Cache-Control` HTTP header permits using a stale cached response in case of an error.

These headers have lower priority than the `proxy_cache_use_stale` directive described above.

Chat with Codey

Does NGINX Support the `Vary` Header?

Yes, in NGINX Plus R5 and NGINX 1.7.7 and later. Here is a good overview of the `Vary` header.

Further Reading

There are many more ways you can customize and tune NGINX caching. To learn even more about caching with NGINX, please take a look at the following resources:

- The ngx_http_proxy_module reference documentation contains all of the configuration options for content caching.

- The NGINX Plus Admin Guide has more configuration examples and information on tuning the NGINX cache.
- The Content Caching with NGINX Plus product page contains an overview on how to configure cache purging with NGINX Plus and provides other examples of cache customization.
- The High-Performance Caching with NGINX and NGINX Plus ebook provides a thorough deep-dive on NGINX content caching.

85 Comments NGINX Disqus' Privacy Policy 1 Login

Recommend 7 Tweet Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Aaron Kili K • 4 years ago

@Faisal Memon
Many thanks for this wonderful guide. It helped me understand caching in Nginx, it's just the perfect starting point for a beginner.

1 ^ | ˅ • Reply • Share ›

Aaron Kili K ➦ Aaron Kili K • 4 years ago

Hello,

Once again, we are using cookies to define sessions in our Adonisjs app, when we tried using the the directive proxy_ignore_headers Set-Cookie, it caused problems with user login sessions. When user1 logins in first, all other users share the session of user1.

How can we enable content caching and still send Set-Cookie header field to the client.

Chat with Codey