**Performance Tuning Parameters**

TCP Parameters

net.core.**somaxconn** – The maximum number of connections that can be queued for acceptance by NGINX. The default is often very low and that’s usually acceptable because NGINX accepts connections very quickly, but it can be worth increasing it if your website experiences heavy traffic. If error messages in the kernel log indicate that the value is too small, increase it until the errors stop.

**Note:** If you set this to a value greater than 512, change the backlog parameter to the NGINX [listen](http://nginx.org/en/docs/http/ngx_http_core_module.html?&_ga=2.200530793.693525960.1494049036-668064378.1492784715#listen) directive to match.

net.core.**netdev\_max\_backlog** – The rate at which packets are buffered by the network card before being handed off to the CPU. Increasing the value can improve performance on machines with a high amount of bandwidth. Check the kernel log for errors related to this setting, and consult the network card documentation for advice on changing it.

## **HTTP and TCP Optimizations**

### Keep Alive

Keep alive allows for fewer reconnections from the browser.

* keepalive\_timeout and keepalive\_requests control the keep alive settings.
* sendfile optimizes serving static files from the file system, like logos.
* tcp\_nodelay allows nginx to make TCP send multiple buffers as individual packets.
* tcp\_nopush optimizes the amount of data sent down the wire at once by activating the TCP\_CORK option within the TCP stack. TCP\_CORK blocks the data until the packet reaches the MSS, which is equal to the MTU minus the 40 or 60 bytes of the IP header.

**/etc/nginx/nginx.conf**

|  |  |
| --- | --- |
| 1  2  3  4  5 | keepalive\_timeout 65;  keepalive\_requests 100000;  sendfile on;  tcp\_nopush on;  tcp\_nodelay on; |

### Timeouts

Timeouts can also drastically improve performance.

* client\_body\_timeout sends directives for the time a server will wait for a **body** to be sent.
* client\_header\_timeout sends directives for the time a server will wait for a **header** body to be sent. These directives are responsible for the time a server will wait for a client body or client header to be sent after request. If neither a body or header is sent, the server will issue a 408 error or Request time out.
* send\_timeout specifies the response timeout to the client. This timeout does not apply to the entire transfer but, rather, only between two subsequent client-read operations. Thus, if the client has not read any data for this amount of time, then nginx shuts down the connection.

**/etc/nginx/nginx.conf**

|  |  |
| --- | --- |
| 1  2  3 | client\_header\_timeout 3m;  client\_body\_timeout 3m;  send\_timeout 3m; |

### Scale TCP Window

The TCP window scale option is an option to increase the receive window size allowed in Transmission Control Protocol above its former maximum value of 65,535 bytes. This TCP option, along with several others, is defined in IETF RFC 1323 which deals with long fat networks. It can be defined with the net.ipv4.tcp\_window\_scaling = 1 tag.

### Backlog Packets Before Drop

The net.ipv4.tcp\_max\_syn\_backlog determines a number of packets to keep in the backlog before the kernel starts dropping them. A sane value is net.ipv4.tcp\_max\_syn\_backlog = 3240000.

### Close connection on Missing Client Response

reset\_timedout\_connection on; allows the server to close the connection after a client stops responding. This frees up socket-associated memory.

Static Asset Serving

If your site serves static assets (such as CSS/JavaScript/images), nginx can cache these files for a short period of time. Adding this within your configuration block tells nginx to cache 1000 files for 30 seconds, excluding any files that haven’t been accessed in 20 seconds, and only files that have been accessed at least 5 times in that timeframe. If you aren’t deploying frequently you can safely bump up these numbers higher.

**/etc/nginx/nginx.conf**

|  |  |
| --- | --- |
| 1  2  3  4 | open\_file\_cache max=1000 inactive=20s;  open\_file\_cache\_valid 30s;  open\_file\_cache\_min\_uses 5;  open\_file\_cache\_errors off; |

### Buffer Size

Making tweaks to the buffer size can be advantageous. If the buffer sizes are too low, then nginx will write to a temporary file. This will cause for excessive disk I/O.

* client\_body\_buffer\_size handles the client buffer size. Most client buffers are coming from POST method form submissions. 128k is normally a good choice for this setting.
* client\_max\_body\_size sets the max body buffer size. If the size in a request exceeds the configured value, the 413 (Request Entity Too Large) error is returned to the client. For reference, browsers cannot correctly display 413 errors. Setting size to 0 disables checking of client request body size.
* client\_header\_buffer\_size handles the client header size. 1k is usually a sane choice for this by default.
* large\_client\_header\_buffers shows the maximum number and size of buffers for large client headers. 4 headers with 4k buffers should be sufficient here.
* output\_buffers sets the number and size of the buffers used for reading a response from a disk. If possible, the transmission of client data will be postponed until nginx has at least the set size of bytes of data to send. The zero value disables postponing data transmission.

**/etc/nginx/nginx.conf**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | client\_body\_buffer\_size 128k;  client\_max\_body\_size 10m;  client\_header\_buffer\_size 1k;  large\_client\_header\_buffers 4 4k;  output\_buffers 1 32k;  postpone\_output 1460; |

Ephemeral Ports

When NGINX is acting as a proxy, each connection to an upstream server uses a temporary, or *ephemeral*, port. You might want to change this setting:

* net.ipv4.**ip\_local\_port\_range** – The start and end of the range of port values. If you see that you are running out of ports, increase the range. A common setting is ports 1024 to 65000.

File Descriptors

File descriptors are operating system resources used to represent connections and open files, among other things. NGINX can use up to two file descriptors per connection. For example, if NGINX is proxying, it generally uses one file descriptor for the client connection and another for the connection to the proxied server, though this ratio is much lower if HTTP keepalives are used. For a system serving a large number of connections, the following settings might need to be adjusted:

* sys.fs.**file\_max** – The system-wide limit for file descriptors
* **nofile** – The user file descriptor limit, set in the **/etc/security/limits.conf** file

Keepalive Connections

[Keepalive connections](https://www.nginx.com/blog/http-keepalives-and-web-performance/) can have a major impact on performance by reducing the CPU and network overhead needed to open and close connections. NGINX terminates all client connections and creates separate and independent connections to the upstream servers. NGINX supports keepalives for both clients and upstream servers. The following directives relate to client keepalives:

* [keepalive\_requests](http://nginx.org/r/keepalive_requests) – The number of requests a client can make over a single keepalive connection. The default is 100, but a much higher value can be especially useful for testing with a load-generation tool, which generally sends a large number of requests from a single client
* [keepalive\_timeout](http://nginx.org/r/keepalive_timeout) – How long an idle keepalive connection remains open.

The following directive relates to upstream keepalives:

* [keepalive](http://nginx.org/r/keepalive) – The number of idle keepalive connections to an upstream server that remain open for each worker process. There is no default value.

To enable keepalive connections to upstream servers you must also include the following directives in the configuration:

[proxy\_http\_version](http://nginx.org/r/proxy_http_version) 1.1;

[proxy\_set\_header](http://nginx.org/r/proxy_set_header) Connection "";

### Access Logging

Logging every request consumes both CPU and I/O cycles, and one way to reduce the impact is to enable access-log buffering. With buffering, instead of performing a separate write operation for each log entry, NGINX buffers a series of entries and writes them to the file together in a single operation.

To enable access-log buffering, include the buffer=size parameter to the [access\_log](http://nginx.org/r/access_log" \t "_blank) directive; NGINX writes the buffer contents to the log when the buffer reaches the size value. To have NGINX write the buffer after a specified amount of time, include the flush=time parameter. When both parameters are set, NGINX writes entries to the log file when the next log entry will not fit into the buffer or the entries in the buffer are older than the specified time, respectively. Log entries are also written when a worker process is reopening its log files or shutting down. To disable access logging completely, include the off parameter to the access\_log directive.

### Sendfile

The operating system’s sendfile() system call copies data from one file descriptor to another, often achieving zero-copy, which can speed up TCP data transfers. To enable NGINX to use it, include the [sendfile](http://nginx.org/r/sendfile" \t "_blank) directive in the http context or a server or location context. NGINX can then write cached or on-disk content down a socket without any context switching to user space, making the write extremely fast and consuming fewer CPU cycles. Note, however, that because data copied with sendfile() bypasses user space, it is not subject to the regular NGINX processing chain and filters that change content, such as [gzip](http://nginx.org/en/docs/http/ngx_http_gzip_module.html" \t "_blank). When a configuration context includes both the sendfile directive and directives that activate a content-changing filter, NGINX automatically disables sendfile for that context.

Limits

You can set various limits that help prevent clients from consuming too many resources, which can adversely the performance of your system as well as user experience and security. The following are some of the relevant directives:

* [limit\_conn](http://nginx.org/r/limit_conn) and [limit\_conn\_zone](http://nginx.org/r/limit_conn_zone" \t "_blank) – Limit the number of client connections NGINX accepts, for example from a single IP address. Setting them can help prevent individual clients from opening too many connections and consuming more than their share of resources.
* [limit\_rate](http://nginx.org/r/limit_rate) – Limits the rate at which responses are transmitted to a client, per connection (so clients that open multiple connections can consume this amount of bandwidth for each connection). Setting a limit can prevent the system from being overloaded by certain clients, ensuring more even quality of service for all clients.
* [limit\_req](http://nginx.org/r/limit_req) and [limit\_req\_zone](http://nginx.org/r/limit_req_zone" \t "_blank) – Limit the rate of requests being processed by NGINX, which has the same benefits as setting limit\_rate. They can also improve security, especially for login pages, by limiting the request rate to a value reasonable for human users but too slow for programs trying to overwhelm your application with requests (such as bots in a [DDoS attack](https://www.nginx.com/blog/mitigating-ddos-attacks-with-nginx-and-nginx-plus/)).
* [max\_conns](http://nginx.org/en/docs/http/ngx_http_upstream_module.html#max_conns) parameter to the server directive in an upstream configuration block – Sets the maximum number of simultaneous connections accepted by a server in an upstream group. Imposing a limit can help prevent the upstream servers from being overloaded. Setting the value to 0 (zero, the default) means there is no limit.
* [queue](http://nginx.org/r/queue) (NGINX Plus) – Creates a queue in which requests are placed when all the available servers in the upstream group have reached their max\_conns limit. This directive sets the maximum number of requests in the queue and, optionally, the maximum time they wait (60 seconds by default) before an error is returned. Requests are not queued if you omit this directive.

**Caching Parameters**

## **Enabling the Caching of Responses**

To enable caching, include the [proxy\_cache\_path](http://nginx.org/en/docs/http/ngx_http_proxy_module.html" \l "proxy_cache_path) directive in the top-level http context. The mandatory first parameter is the local filesystem path for cached content, and the mandatory keys\_zone parameter defines the name and size of the shared memory zone that is used to store metadata about cached items:

http {

...

proxy\_cache\_path /data/nginx/cache **keys\_zone**=**one**:10m;

}

Then include the [proxy\_cache](http://nginx.org/en/docs/http/ngx_http_proxy_module.html" \l "proxy_cache) directive in the context (protocol type, virtual server, or location) for which you want to cache server responses, specifying the zone name defined by the keys\_zoneparameter to the proxy\_cache\_path directive (in this case, one):

http {

...

proxy\_cache\_path /data/nginx/cache keys\_zone=one:10m;

server {

proxy\_cache **one**;

location / {

proxy\_pass http://localhost:8000;

}

}

}

Note that the size defined by the keys\_zone parameter does not limit the total amount of cached response data. Cached responses themselves are stored with a copy of the metadata in specific files on the filesystem. To limit the amount of cached response data, include the max\_size parameter to the proxy\_cache\_path directive. (But note that the amount of cached data can temporarily exceed this limit, as described in the following section.)

## **NGINX Processes Involved in Caching**

There are two additional NGINX processes involved in caching:

* The cache manager is activated periodically to check the state of the cache. If the cache size exceeds the limit set by the max\_size parameter to the proxy\_cache\_path directive, the cache manager removes the data that was accessed least recently. As previously mentioned, the amount of cached data can temporarily exceed the limit during the time between cache manager activations.
* The cache loader runs only once, right after NGINX starts. It loads metadata about previously cached data into the shared memory zone. Loading the whole cache at once could consume sufficient resources to slow NGINX performance during the first few minutes after startup. To avoid this, configure iterative loading of the cache by including the following parameters to the proxy\_cache\_path directive:
  + loader\_threshold – Duration of an iteration, in milliseconds (by default, 200)
  + loader\_files – Maximum number of items loaded during one iteration (by default, 100)
  + loader\_sleeps – Delay between iterations, in milliseconds (by default, 50)

In the following example, iterations last 300 milliseconds or until 200 items have been loaded:

proxy\_cache\_path /data/nginx/cache keys\_zone=one:10m loader\_threshold=300 loader\_files=200;

## **Specifying Which Requests to Cache**

By default, NGINX Plus caches all responses to requests made with the HTTP GET and HEAD methods the first time such responses are received from a proxied server. As the key (identifier) for a request, NGINX Plus uses the request string. If a request has the same key as a cached response, NGINX Plus sends the cached response to the client. You can include various directives in the http, server, or location context to control which responses are cached.

To change the request characteristics used in calculating the key, include the [proxy\_cache\_key](http://nginx.org/en/docs/http/ngx_http_proxy_module.html" \l "proxy_cache_key)directive:

proxy\_cache\_key "$host$request\_uri$cookie\_user";

To define the minimum number of times that a request with the same key must be made before the response is cached, include the [proxy\_cache\_min\_uses](http://nginx.org/en/docs/http/ngx_http_proxy_module.html" \l "proxy_cache_min_uses) directive:

proxy\_cache\_min\_uses 5;

To cache responses to requests with methods other than GET and HEAD, list them along with GET and HEAD as parameters to the [proxy\_cache\_methods](http://nginx.org/en/docs/http/ngx_http_proxy_module.html" \l "proxy_cache_methods) directive:

proxy\_cache\_methods GET HEAD POST;

## **Limiting or Bypassing Caching**

By default, responses remain in the cache indefinitely. They are removed only when the cache exceeds the maximum configured size, and then in order by length of time since they were last requested. You can set how long cached responses are considered valid, or even whether they are used at all, by including directives in the http, server, or location context:

To limit how long cached responses with specific status codes are considered valid, include the [proxy\_cache\_valid](http://nginx.org/en/docs/http/ngx_http_proxy_module.html" \l "proxy_cache_valid) directive:

proxy\_cache\_valid 200 302 10m;

proxy\_cache\_valid 404 1m;

In this example, responses with the code 200 or 302 are considered valid for 10 minutes, and responses with code 404 are valid for 1 minute. To define the validity time for responses with all status codes, specify any as the first parameter:

proxy\_cache\_valid any 5m;

To define conditions under which NGINX Plus does not send cached responses to clients, include the [proxy\_cache\_bypass](http://nginx.org/en/docs/http/ngx_http_proxy_module.html" \l "proxy_cache_bypass) directive. Each parameter defines a condition and consists of a number of variables. If at least one parameter is not empty and does not equal “0” (zero), NGINX Plus does not look up the response in the cache, but instead forwards the request to the backend server immediately.

proxy\_cache\_bypass $cookie\_nocache $arg\_nocache$arg\_comment;

To define conditions under which NGINX Plus does not cache a response at all, include the [proxy\_no\_cache](http://nginx.org/en/docs/http/ngx_http_proxy_module.html" \l "proxy_no_cache) directive, defining parameters in the same way as for the proxy\_cache\_bypassdirective.

proxy\_no\_cache $http\_pragma $http\_authorization;

## **Byte-Range Caching**

Sometimes, the initial cache fill operation may take some time, especially for large files. When the first request starts downloading a part of a video file, next requests will have to wait for the entire file to be downloaded and put into the cache.

NGINX makes it possible cache such range requests and gradually fill the cache with the [cache slice module](http://nginx.org/en/docs/http/ngx_http_slice_module.html). The file is divided into smaller “slices”. Each range request chooses particular slices that would cover the requested range and, if this range is still not cached, put it into the cache. All other requests to these slices will take the response from the cache.

To enable byte-range caching:

1. Make sure your NGINX is compiled with the [slice](http://nginx.org/en/docs/http/ngx_http_slice_module.html) module.
2. Specify the size of the slice with the [slice](http://nginx.org/en/docs/http/ngx_http_slice_module.html#slice) directive:
3. location / {
4. slice 1m;

}

The slice size should be adjusted reasonably enough to make slice download fast. A too small size may result in excessive memory usage and a large number of opened file descriptors while processing the request, a too large value may result in latency.

1. Include the [$slice\_range](http://nginx.org/en/docs/http/ngx_http_slice_module.html#var_slice_range) variable to the cache key:

proxy\_cache\_key $uri$is\_args$args**$slice\_range**;

1. Enable caching of responses with 206 status code:

proxy\_cache\_valid 200 **206** 1h;

1. Enable passing range requests to the proxied server by passing the [$slice\_range](http://nginx.org/en/docs/http/ngx_http_slice_module.html#var_slice_range) variable in the Range header field:

proxy\_set\_header Range $slice\_range;

Byte-range caching example:

location / {

slice 1m;

proxy\_cache cache;

proxy\_cache\_key $uri$is\_args$args$slice\_range;

proxy\_set\_header Range $slice\_range;

proxy\_cache\_valid 200 206 1h;

proxy\_pass http://localhost:8000;

}

Note that if slice caching is turned on, the initial file should not be changed.

**Compression**

## **Enabling Compression**

To enable compression, include the [gzip](http://nginx.org/en/docs/http/ngx_http_gzip_module.html" \l "gzip) directive with the on parameter.

gzip on;

By default, NGINX compresses responses only with MIME type text/html. To compress responses with other MIME types, include the [gzip\_types](http://nginx.org/en/docs/http/ngx_http_gzip_module.html" \l "gzip_types) directive and list the additional types.

gzip\_types text/plain application/xml;

To specify the minimum length of the response to compress, use the [gzip\_min\_length](http://nginx.org/en/docs/http/ngx_http_gzip_module.html" \l "gzip_min_length)directive. The default is 20 bytes (here adjusted to 1000):

gzip\_min\_length 1000;

By default, NGINX does not compress responses to proxied requests (requests that come from the proxy server). The fact that a request comes from a proxy server is determined by the presence of the Via header field in the request. To configure compression of these responses, use the [gzip\_proxied](http://nginx.org/en/docs/http/ngx_http_gzip_module.html" \l "gzip_proxied) directive. The directive has a number of parameters specifying which kinds of proxied requests NGINX should compress. For example, it is reasonable to compress responses only to requests that will not be cached on the proxy server. For this purpose the gzip\_proxied directive has parameters that instruct NGINX to check the Cache-Controlheader field in a response and compress the response if the value is no-cache, no-store, or private. In addition, you must include the expired parameter to check the value of the Expires header field. These parameters are set in the following example, along with the authparameter, which checks for the presence of the Authorization header field (an authorized response is specific to the end user and is not typically cached):

gzip\_proxied no-cache no-store private expired auth;

As with most other directives, the directives that configure compression can be included in the http context or in a server or location configuration block.

The overall configuration of gzip compression might look like this.

server {

gzip on;

gzip\_types text/plain application/xml;

gzip\_proxied no-cache no-store private expired auth;

gzip\_min\_length 1000;

...

}

## **Enabling Decompression**

Some clients do not support responses with the gzip encoding method. At the same time, it might be desirable to store compressed data, or compress responses on the fly and store them in the cache. To successfully serve both clients that do and do not accept compressed data, NGINX can decompress data on the fly when sending it to the latter type of client.

To enable runtime decompression, use the [gunzip](http://nginx.org/en/docs/http/ngx_http_gunzip_module.html" \l "gunzip) directive.

location /storage/ {

gunzip on;

...

}

The gunzip directive can be specified in the same context as the gzip directive:

server {

gzip on;

gzip\_min\_length 1000;

gunzip on;

...

}

Note that this directive is defined in a separate [module](http://nginx.org/en/docs/http/ngx_http_gunzip_module.html) that might not be included in an open source NGINX build by default.

## **Sending Compressed Files**

To send a compressed version of a file to the client instead of the regular one, set the [gzip\_static](http://nginx.org/en/docs/http/ngx_http_gzip_static_module.html" \l "gzip_static) directive to on within the appropriate context.

location / {

gzip\_static on;

}

In this case, to service a request for **/path/to/file**, NGINX tries to find and send the file **/path/to/file.gz**. If the file doesn’t exist, or the client does not support gzip, NGINX sends the uncompressed version of the file.

Note that the gzip\_static directive does not enable on-the-fly compression. It merely uses a file compressed beforehand by any compression tool. To compress content (and not only static content) at runtime, use the gzip directive.

This directive is defined in a separate [module](http://nginx.org/en/docs/http/ngx_http_gzip_static_module.html) that might not be included in an open source NGINX build by default.

### Turning off Logging Completely

Logging can be turned off completely if you have an alternative logging methodology or if you don’t care about logging any of the requests to the server. Turning off logging can be performed with the following server directives

**/etc/nginx/nginx.conf**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | server {  listen 80;  server\_name example.com;  access\_log off;  error\_log off;  } |

**E**

## **Example Files**

Several tweaks have now been made across three files to improve nginx performance on your system. Full snippets of the files are included below.

### sysctl.conf

**/etc/sysctl.conf**

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | net.core.somaxconn = 65536  net.ipv4.tcp\_max\_tw\_buckets = 1440000  net.ipv4.ip\_local\_port\_range = 1024 65000  net.ipv4.tcp\_fin\_timeout = 15  net.ipv4.tcp\_window\_scaling = 1  net.ipv4.tcp\_max\_syn\_backlog = 3240000 |

### limits.conf

**/etc/security/limits.conf**

|  |  |
| --- | --- |
| 1  2 | soft nofile 4096  hard nofile 4096 |

### nginx.conf

**nginx.conf**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83 | pid /var/run/nginx.pid;  worker\_processes 2;  events {  worker\_connections 65536;  use epoll;  multi\_accept on;  }  http {  keepalive\_timeout 65;  keepalive\_requests 100000;  sendfile on;  tcp\_nopush on;  tcp\_nodelay on;  client\_body\_buffer\_size 128k;  client\_max\_body\_size 10m;  client\_header\_buffer\_size 1k;  large\_client\_header\_buffers 4 4k;  output\_buffers 1 32k;  postpone\_output 1460;  client\_header\_timeout 3m;  client\_body\_timeout 3m;  send\_timeout 3m;  open\_file\_cache max=1000 inactive=20s;  open\_file\_cache\_valid 30s;  open\_file\_cache\_min\_uses 5;  open\_file\_cache\_errors off;  gzip on;  gzip\_min\_length 1000;  gzip\_buffers 4 4k;  gzip\_types text/html application/x-javascript text/css application/javascript text/javascript text/plain text/xml application/json application/vnd.ms-fontobject application/x-font-opentype application/x-font-truetype application/x-font-ttf application/xml font/eot font/opentype font/otf image/svg+xml image/vnd.microsoft.icon;  gzip\_disable "MSIE [1-6]\.";  *# [ debug | info | notice | warn | error | crit | alert | emerg ]*  error\_log /var/log/nginx.error\_log warn;  log\_format main '$remote\_addr - $remote\_user [$time\_local] '  '"$request" $status $bytes\_sent '  '"$http\_referer" "$http\_user\_agent" '  '"$gzip\_ratio"';  log\_format download '$remote\_addr - $remote\_user [$time\_local] '  '"$request" $status $bytes\_sent '  '"$http\_referer" "$http\_user\_agent" '  '"$http\_range" "$sent\_http\_content\_range"';  map $status $loggable {  ~^[23] 0;  default 1;  }  server {  listen 127.0.0.1;  server\_name 127.0.0.1;  root /var/www/html;  access\_log /var/log/nginx.access\_log main;  location / {  proxy\_pass http://127.0.0.1/;  proxy\_redirect off;  proxy\_set\_header Host $host;  proxy\_set\_header X-Real-IP $remote\_addr;  proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;  proxy\_connect\_timeout 90;  proxy\_send\_timeout 90;  proxy\_read\_timeout 90;  proxy\_buffer\_size 4k;  proxy\_buffers 4 32k;  proxy\_busy\_buffers\_size 64k;  proxy\_temp\_file\_write\_size 64k;  proxy\_temp\_path /etc/nginx/proxy\_temp;  }  location ~\* .(woff|eot|ttf|svg|mp4|webm|jpg|jpeg|png|gif|ico|css|js)$ {  expires 365d;  }  }  } |