Module directives

At each of the three levels, directives can be inserted in order to affect the behavior of the web server. The following is the list of all directives that are introduced by the main HTTP module, grouped thematically. For each directive, an indication regarding the context is given. Some cannot be used at certain levels. For instance, it would make no sense to insert a `server_name` directive at the `http` block level, since `server_name` is a directive directly affecting a virtual host—it should only be inserted in the `server` block. For this purpose, the table indicates the possible levels where each directive is allowed—the `http` block, the `server` block, the `location` block, and additionally the `if` block, later introduced by the **Rewrite module**.

Note

Note that this documentation is valid as of the stable version 1.8.0. Future updates may alter the syntax of some directives or provide new features that are not discussed here.

Socket and host configuration

This set of directives allows you to configure your virtual hosts. In practice, this materializes by creating `server` blocks that you identify either by a hostname or by an IP address and port combination. In addition, some directives let you fine-tune your network settings by configuring the TCP socket options.

listen

Context: `server`

Specifies the IP address and/or the port to be used by the listening socket that serves the website. Sites are generally served on port `80` (the default value) via HTTP, or `443` via HTTPS.

Syntax: `listen [address][:port] [additional options];`

Additional options:

- `default_server` : Specifies this `server` block to be used as the default website for any request received at the specified IP address and port
- `ssl` : Specifies that the website should be served over SSL
- `spdy` : Enables support for the SPDY protocol if the SPDY module is present
- `proxy_protocol` : Enables the PROXY protocol for all the connections accepted on this port
- Other options are related to the `bind` and `listen` system calls: `backlog=num` , `rcvbuf=size` , `sndbuf=size` , `accept_filter=filter` , `deferred` , `setfib=number` , and `bind`

Examples:

Copy

```
listen 192.168.1.1:80;
listen 127.0.0.1;
listen 80 default;
listen [:::a8c9:1234]:80; # IPv6 addresses must be put between square brackets
listen 443 ssl;
```

This directive also allows Unix sockets:

Copy

```
listen unix:/tmp/nginx.sock;
```

server_name

Context: `server`

This assigns one or more hostnames to the `server` block. When Nginx receives an HTTP request, it matches the `Host` header of the request against all the `server` blocks. The first `server` block to match this hostname is selected.

Plan B: if no `server` block matches the desired host, Nginx selects the first `server` block that matches the parameters of the `listen` directive (such as `listen *:80` would be a catch-all for all the requests received on port `80`), giving priority to the first block that has the `default` option enabled on the `listen` directive.

Note that this directive accepts wildcards as well as regular expressions (in which case, the hostname should start with the `~` character).

Syntax: `server_name hostname1 [hostname2…];`

Examples:

Copy

```
server_name www.website.com;
server_name www.website.com website.com;
server_name *.website.com;
server_name .website.com; # combines both *.website.com and website.com
server_name *.website.*;
server_name ~^(www)\.example\.com$; # $1 = www
```

Note that you may use an empty string as the directive value in order to catch all the requests that do not come with a `Host` header, but only after at least one regular name (or " `_` " for a dummy hostname):

Copy

```
server_name website.com "";
server_name _ "";
```

server_name_in_redirect

Context: `http` , `server` , `location`

This directive applies to the case of internal redirects (for more information about internal redirects, check the **Rewrite Module** section given further on in this chapter). If set to `on` , Nginx uses the first hostname specified in the `server_name` directive. If set to `off` , Nginx uses the value of the `Host` header from the HTTP request.

Syntax: `on` or `off`

Default value: `off`

server_names_hash_max_size

Context: `http`

Nginx uses hash tables for various data collections in order to speed up the processing of requests. This directive defines the maximum size of the server names hash table. The default value fits with most configurations. If this needs to be changed, Nginx automatically tells you so on startup, or when you reload its configuration.

Syntax: Numeric value

Default value: `512`

server_names_hash_bucket_size

Context: `http`

Sets the bucket size for server names hash tables. You should only change this value if Nginx tells you to.

Syntax: Numeric value

Default value: `32` (or `64`, or `128`, depending on your processor cache specifications)

port_in_redirect

Context: `http`, `server`, `location`

In case of a redirect, this directive defines whether or not Nginx should append the port number to the redirection URL.

Syntax: `on` or `off`

Default value: `on`

tcp_nodelay

Context: `http`, `server`, `location`

Enables or disables the `TCP_NODELAY` socket option for keep-alive connections only. Quoting the Linux documentation on sockets programming:

> **"TCP_NODELAY is for a specific purpose; to disable the Nagle buffering algorithm. It should only be set for applications that send frequent small bursts of information without getting an immediate response, where timely delivery of data is required (the canonical example is mouse movements)."**

Syntax: `on` or `off`

Default value: `on`

tcp_nopush

Context: `http`, `server`, `location`

Enables or disables the `TCP_NOPUSH` (FreeBSD) or `TCP_CORK` (Linux) socket option. Note that this option applies only if the `sendfile` directive is enabled. If `tcp_nopush` is set to on, Nginx attempts to transmit all the HTTP response headers in a single TCP packet.

Syntax: `on` or `off`

Default value: `off`

sendfile

Context: `http`, `server`, `location`

If this directive is enabled, Nginx uses the `sendfile` kernel call to handle file transmission. If disabled, Nginx handles the file transfer by itself. Depending on the physical location of the file being transmitted (such as NFS), this option may affect the server performance.

Syntax: `on` or `off`

Default value: `off`

sendfile_max_chunk

Context: `http`, `server`

This directive defines the maximum size of data to be used for each call to `sendfile` (read the previous section).

Syntax: Numeric value (size)

Default value: `0`

send_lowat

Context: `http` , `server`

This is an option that allows you to make use of the `SO_SNDLOWAT` flag for TCP sockets under FreeBSD only. This value defines the minimum number of bytes in the buffer for output operations.

Syntax: Numeric value (size)

Default value: `0`

reset_timedout_connection

Context: `http` , `server` , `location`

When a client connection times out, its associated information may remain in memory depending on its state. Enabling this directive will erase all memory associated with the connection after it times out.

Syntax: `on` or `off`

Default value: `off`

Paths and documents

This section describes the directives that configure the documents that should be served for each website, such as the document root, the site index, error pages, and so on.

root

Context: `http` , `server` , `location` , `if` . Variables are accepted.

This directive defines the document root containing the files that you wish to serve to your visitors.

Syntax: Directory path

Default value: `html`

Copy

```
root /home/website.com/public_html;
```

alias

Context: `location` . Variables are accepted.

`alias` is a directive that you place in a `location` block only. It assigns a different path for Nginx to retrieve documents for a specific request. As an example, consider the following configuration:

Copy

```
http {
   server {
       server_name localhost;
       root /var/www/website.com/html;
       location /admin/ {
        alias /var/www/locked/;
      }
   }
}
```

When a request for `http://localhost/` is received, files are served from the `/var/www/website.com/html/` folder. However, if Nginx receives a request for `http://localhost/admin/` , the path used to retrieve the files is `var/www/locked/` . Moreover, the value of the document root directive (`root`) is not altered. This procedure is invisible in the eyes of dynamic scripts.

Syntax: Directory (do not forget the trailing `/`) or file path

error_page

Context: `http` , `server` , `location` , `if` . Variables are accepted.

This allows you to affect URIs to the HTTP response code and optionally, to substitute the code with another.

Syntax: `error_page code1 [code2…] [=replacement code] [=@block | URI]`

Examples :

Copy

```
error_page 404 /not_found.html;
error_page 500 501 502 503 504 /server_error.html;
error_page 403 http://website.com/;
error_page 404 @notfound; # jump to a named location block
error_page 404 =200 /index.html; # in case of 404 error, redirect to index.html with a 200 OK response code
```

if_modified_since

Context: `http` , `server` , `location`

This defines the way Nginx handles the `If-Modified-Since` HTTP header. This header is mostly used by search engine spiders (such as Google web crawling bots). The robot indicates the date and time of the last pass. If the requested file has not been modified since that time, the server simply returns a `304 Not Modified` response code with no body.

This directive accepts the following three values:

- `off` : Ignores the `If-Modified-Since` header.
- `exact` : Returns `304 Not Modified` if the date and time specified in the HTTP header are an exact match with the actual requested file modification date. If the file modification date is anterior or ulterior, the file is served normally (`200 OK` response).
- `before` : Returns `304 Not Modified` if the date and time specified in the HTTP header is anterior or equal to the requested file modification date.

Syntax: `if_modified_since off | exact | before`

Default value: `exact`

index

Context: `http` , `server` , `location` . Variables are accepted.

This defines the default page that Nginx will serve if no filename is specified in the request (in other words, the index page). You may specify multiple filenames, and the first file to be found will be served. If none of the specified files are found, and if the `autoindex` directive is enabled (check the `HTTP Autoindex` module), Nginx will attempt to generate an automatic index of the files. Otherwise, it will return a `403 Forbidden` error page. Optionally, you may insert an absolute filename (such as `/page.html` , based from the document root directory) but only as the last argument of the directive.

Syntax: `index file1 [file2…] [absolute_file];`

Default value: `index.html`

Copy

```
index index.php index.html index.htm;
index index.php index2.php /catchall.php;
```

recursive_error_pages

Context: `http` , `server` , `location`

Sometimes, an error page itself served by the `error_page` directive may trigger an error; in this case, the `error_page` directive is used again (recursively). This directive enables or disables recursive error pages.

Syntax: `on` or `off`

Default value: `off`

try_files

Context: `server` , `location` . Variables are accepted.

This attempts to serve the specified files (arguments 1 to N-1). If none of these files exist, it jumps to the respective named `location` block (last argument) or serves the specified URI.

Syntax: Multiple file paths followed by a named `location` block or a URI

Example:

Copy

```
location / {
    try_files $uri $uri.html $uri.php $uri.xml @proxy;
}
# the following is a "named location block"
location @proxy {
    proxy_pass 127.0.0.1:8080;
}
```

In this example, Nginx tries to serve files normally. If the requested URI does not correspond to any existing file, Nginx appends `.html` to the URI and tries to serve the file again. If it fails again, it tries with `.php` , and then with `.xml` . Eventually, if all of these possibilities fail, another `location` block (`@proxy`) handles the request.

Note

You may also specify `$uri/` in the list of values in order to test for the existence of a directory with that name.

Client requests

This section documents the way that Nginx handles client requests. Among other things, you are allowed to configure the keep-alive mechanism behavior and, possibly, logging the client requests into files.

keepalive_requests

Context: `http` , `server` , `location`

This specifies the maximum number of requests served over a single keep-alive connection.

Syntax: Numeric value

Default value: `100`

keepalive_timeout

Context: `http` , `server` , `location`

This directive defines the number of seconds the server will wait before closing a keep-alive connection. The second (optional) parameter is transmitted as the value of the `Keep-Alive: timeout=` HTTP response header. The intended effect is to let the client browser close the connection itself after this period has elapsed. Note that some browsers ignore this setting. Internet Explorer, for instance, automatically closes the connection after around `60` seconds.

Syntax: `keepalive_timeout time1 [time2];`

Default value: `75`

Copy

```
keepalive_timeout 75;
keepalive_timeout 75 60;
```

keepalive_disable

Context: `http` , `server` , `location`

This option allows you to disable the `keepalive` functionality for the browser families of your choice.

Syntax: `keepalive_disable browser1 browser2;`

Default value: `msie6`

send_timeout

Context: `http` , `server` , `location`

This specifies the amount of time after which Nginx closes an inactive connection. A connection becomes inactive the moment a client stops transmitting data.

Syntax: Time value (in seconds)

Default value: `60`

client_body_in_file_only

Context: `http` , `server` , `location`

If this directive is enabled, the body of the incoming HTTP requests will be stored into actual files on the disk. The **client body** corresponds to the client HTTP request raw data, minus the headers (in other words, the content transmitted in POST requests). Files are stored as plain text documents.

This directive accepts three values:

- `off` : Does not store the request body in a file
- `clean` : Stores the request body in a file, and removes the file after a request is processed
- `on` : Stores the request body in a file, but does not remove the file after the request is processed (not recommended unless for debugging purposes)

Syntax: `client_body_in_file_only on | clean | off`

Default value: `off`

client_body_in_single_buffer

Context: `http` , `server` , `location`

This directive defines whether or not Nginx should store the request body in a single buffer in memory.

Syntax: `on` or `off`

Default value: `off`

client_body_buffer_size

Context: `http` , `server` , `location`

The `client_body_buffer_size` directive specifies the size of the buffer holding the body of client requests. If this size is exceeded, the body (or at least a part of it) will be written to the disk. Note that if the `client_body_in_file_only` directive is enabled, request bodies are always stored to a file on the disk, regardless of their size (whether they fit in the buffer or not).

Syntax: Size value

Default value: `8k` or `16k` (2 memory pages) depending on your computer architecture

client_body_temp_path

Context: `http` , `server` , `location`

This option allows you to define the path of the directory that will store the client request body files. An additional option lets you separate those files into a folder hierarchy of up to three levels.

Syntax: `client_body_temp_path path [level1] [level2] [level3]`

Default value: `client_body_temp`

```
client_body_temp_path /tmp/nginx_rbf;
client_body_temp_path temp 2; # Nginx will create 2-digit folders to hold request body files
client_body_temp_path temp 1 2 4; # Nginx will create 3 levels of folders (first level: 1 digit, second level: 2 digits, third level: 4
```

client_body_timeout

Context: `http` , `server` , `location`

This directive defines the inactivity timeout while reading a client request body. A connection becomes inactive the moment the client stops transmitting data. If the delay is reached, Nginx returns a `408 Request timeout` HTTP error.

Syntax: Time value (in seconds)

Default value: `60`

client_header_buffer_size

Context: `http` , `server` , `location`

This directive allows you to define the size of the buffer that Nginx allocates to request headers. Usually, `1k` is enough. However, in some cases, the headers contain large chunks of cookie data or the request URI is lengthy. If that is the case, then Nginx allocates one or more larger buffers (the size of larger buffers is defined by the `large_client_header_buffers` directive).

Syntax: Size value

Default value: `1k`

client_header_timeout

Context: `http` , `server` , `location`

This defines the inactivity timeout while reading a client request header. A connection becomes inactive the moment the client stops transmitting data. If the delay is reached, Nginx returns a `408 Request timeout` HTTP error.

Syntax: Time value (in seconds)

Default value: `60`

client_max_body_size

Context: `http` , `server` , `location`

This is the maximum size of a client request body. If this size is exceeded, Nginx returns a `413 Request entity too large` HTTP error. This setting is particularly important if you are going to allow users to upload files to your server over HTTP.

Syntax: Size value

Default value: `1m`

large_client_header_buffers

Context: `http` , `server` , `location`

This defines the amount and size of the larger buffers to be used for storing client requests in case the default buffer (`client_header_buffer_size`) is insufficient. Each line of the header must fit in the size of a single buffer. If the request URI line is greater than the size of a single buffer, Nginx returns the `414 Request URI too large` error. If another header line exceeds the size of a single buffer, Nginx returns a `400 Bad request` error.

Syntax: `large_client_header_buffers amount size`

Default value: 4*8 kilobytes

lingering_time

Context: `http` , `server` , `location`

This directive applies to the client requests with a request body. As soon as the amount of uploaded data exceeds `max_client_body_size` , Nginx immediately sends a `413 Request entity too large` HTTP error response. However, most browsers continue uploading data regardless of that notification. This directive defines the amount of time Nginx should wait for after sending this error response before it closes the connection.

Syntax: Numeric value (time)

Default value: 30 seconds

lingering_timeout

Context: `http` , `server` , `location`

This directive defines the amount of time that Nginx should wait between two read operations before it closes the client connection.

Syntax: Numeric value (time)

Default value: 5 seconds

lingering_close

Context: `http` , `server` , `location`

The `lingering_close` directive controls the way Nginx closes client connections. Set this to `off` to immediately close connections after all the request data has been received. The default value (`on`) allows Nginx to wait and process additional data if necessary. If set to `always` , Nginx will always wait to close the connection. The amount of waiting time is defined by the `lingering_timeout` directive.

Syntax: `on` , `off` , or `always`

Default value: `on`

ignore_invalid_headers

Context: `http` , `server`

If this directive is disabled, Nginx returns a `400 Bad Request` HTTP error in case the request headers are malformed.

Syntax: `on` or `off`

Default value: `on`

chunked_transfer_encoding

Context: `http` , `server` , `location`

This directive enables or disables chunked transfer encoding for HTTP 1.1 requests.

Syntax: `on` or `off`

Default value: `on`

max_ranges

Context: `http` , `server` , `location`

This directive defines the number of byte ranges that Nginx will accept to serve when a client requests partial content from a file. If you do not specify a value, there is no limit. If you set this to `0` , the byte range functionality is disabled.

Syntax: Size value

MIME types

Nginx offers two particular directives that help you configure the MIME types: `types` and `default_type` , which defines the default MIME types for documents. This will affect the **Content-Type** HTTP header sent within responses. Read on.

types

Context: `http` , `server` , `location`

This directive allows you to establish correlations between the MIME types and file extensions. It's actually a block accepting a particular syntax:

Copy

```
types {
  mimetype1  extension1;
  mimetype2  extension2 [extension3…];
  […]
}
```

When Nginx serves a file, it checks the file extension in order to determine the MIME type. The MIME type is then sent as the value of the `Content-Type HTTP` header in the response. This header may affect the way in which browsers handle files. For example, if the MIME type of the file you are requesting is `application/pdf` , your browser may, for instance, attempt to render the file using a plugin associated to that MIME type instead of merely downloading it.

Nginx includes a basic set of MIME types as a standalone file (`mime.types`) to be included with the `include` directive:

Copy

```
include mime.types;
```

This file already covers the most important file extensions, so you will probably not need to edit it. If the extension of the served file is not found within the listed types, the default type is used, as defined by the `default_type` directive (read below).

Note that you may override the list of types by re-declaring the `types` block. A useful example would be to force all the files in a folder to be downloaded instead of being displayed:

Copy

```
http {
   include mime.types;
   […]
   location /downloads/ {
       # removes all MIME types
       types { }
       default_type application/octet-stream;
    }
   […]
}
```

Note that some browsers ignore the MIME types and may still display files if their filename ends with a known extension such as `.html` or `.txt` .

Note

To control the way files are handled by your visitors' browsers in a more certain and definitive manner, you should make use of the `Content-Disposition` HTTP header via the `add_header` directive—detailed in the HTTP Headers module (Chapter 4, **Module Configuration**).

The default values, if the `mime.types` file is not included, are:

Copy

```
types {
  text/html html;
  image/gif gif;
  image/jpeg jpg;
}
```

default_type

Context: `http` , `server` , `location`

This defines the default MIME type. When Nginx serves a file, the file extension is matched against the known types declared within the `types` block in order to return the proper MIME type as the value of the `Content-Type` HTTP response header. If the extension doesn't match any of the known MIME types, the value of the `default_type` directive is used.

Syntax: MIME type

Default value: `text/plain`

types_hash_max_size

Context: `http` , `server` , `location`

This defines the maximum size of an entry in the MIME types hash tables.

Syntax: Numeric value.

Default value: `4k` or `8k` (1 line of CPU cache)

types_hash_bucket_size

Context: `http` , `server` , `location`

This directive sets the bucket size for the MIME types hash tables. You should only change this value if Nginx tells you to.

Syntax: Numeric value.

Default value: `64`

Limits and restrictions

This set of directives allow you to add restrictions to be applied when a client attempts to access a particular location or document on your server. Note that you will find additional directives for restricting access in the next chapter.

limit_except

Context: `location`

This directive allows you to prevent the use of all the HTTP methods except the ones that you explicitly allow. Within a `location` block, you may want to restrict the use of some HTTP methods, for instance by forbidding clients from sending POST requests. You need to define two elements—first, the methods that are not forbidden (the allowed methods; all others will be forbidden), and second, the audience that is affected by the restriction:

Copy

```
location /admin/ {
    limit_except GET {
      allow 192.168.1.0/24;
      deny all;
    }
}
```

The preceding example applies a restriction to the `/admin/` location—all visitors are only allowed to use the GET method. Visitors that have a local IP address, as specified with the `allow` directive (detailed in the HTTP Access module), are not affected by this restriction. If a visitor uses a forbidden method, Nginx will return a `403 Forbidden` HTTP error. Note that the GET method implies the HEAD method (if you allow

GET, both GET and HEAD are allowed).

The syntax for this is as follows:

Copy

```
limit_except METHOD1 [METHOD2…] {
  allow | deny | auth_basic | auth_basic_user_file | proxy_pass | perl;
}
```

The directives that you are allowed to insert within the block are documented in their respective module section in Chapter 4, **Module Configuration**.

limit_rate

Context: `http` , `server` , `location` , `if`

This directive allows you to limit the transfer rate of individual client connections. The rate is expressed in bytes per second:

Copy

```
limit_rate 500k;
```

This will limit the connection transfer rates to 500 kilobytes per second. If a client opens two connections, the client will be allowed 2 * 500 kilobytes.

Syntax: Size value

Default value: No limit

limit_rate_after

Context: `http` , `server` , `location` , `if`

This defines the amount of data transferred before the `limit_rate` directive takes effect.

Copy

```
limit_rate 10m;
```

Nginx sends the first `10` megabytes at the maximum speed. Past this size, the transfer rate is limited by the value specified with the `limit_rate` directive (see preceding section). Similarly to the `limit_rate` directive, this setting applies only to a single connection.

Syntax: Size value

Default: None

satisfy

Context: `location`

The `satisfy` directive defines whether clients require all access conditions to be valid (`satisfy all`) or at least one (`satisfy any`).

Copy

```
location /admin/ {
    allow 192.168.1.0/24;
    deny all;
    auth_basic "Authentication required";
    auth_basic_user_file conf/htpasswd;
}
```

In the preceding example, there are two conditions for clients to be able to access the resource:

- Through the `allow` and `deny` directives (HTTP Access module), we only allow clients that have a local IP address; all other clients are denied access.
- Through the `auth_basic` and `auth_basic_user_file` directives (HTTP Auth Basic module), we only allow clients that provide a valid username and password.

With `satisfy all`, the client must satisfy both the conditions in order to gain access to the resource. With `satisfy any`, if the client satisfies either condition, they are granted access.

Syntax: `satisfy any | all`

Default value: `all`

internal

Context: `location`

This directive specifies that the `location` block is internal. In other words, the specified resource cannot be accessed by external requests.

Copy

```
server {
    […]
    server_name .website.com;
    location /admin/ {
        internal;
    }
}
```

With the preceding configuration, clients will not be able to browse `http://website.com/admin/`. Such requests will be met with `404 Not Found` errors. The only way to access the resource is via internal redirects (check the **Rewrite module** section for more information on internal redirects).

File processing and caching

It's important for your websites to be built upon solid foundations. File access and caching is a critical aspect of web serving. In this instance, Nginx lets you perform precise tweaking with the use of the following directives.

disable_symlinks

This directive allows you to control the way Nginx handles symbolic links when they are to be served. By default (the directive value is `off`), symbolic links are allowed and Nginx follows them. You may decide to disable the following symbolic links under different conditions by specifying one of these values:

- `on`: If any part of the requested URI is a symbolic link, access to it is denied, and Nginx returns a `403 HTTP` error page.
- `if_not_owner`: Similar to the preceding value, but access is denied only if the link and the object it points to have different owners.
- The optional parameter `from=` allows you to specify a part of the URL that will not be checked for symbolic links. For example, `disable_symlinks on from=$document_root` will tell Nginx to normally follow the symbolic links in the URI up to the `$document_root` folder. If a symbolic link is found in the URI parts after that, access to the requested file will be denied.

directio

Context: `http`, `server`, `location`

If this directive is enabled, files with a size greater than the specified value will be read with the Direct I/O system mechanism. This allows Nginx to read data from the storage device and place it directly in memory with no intermediary caching process involved.

Syntax: Size value, or off

Default value: `off`

directio_alignment

Context: `http`, `server`, `location`

This directive sets the byte alignment when using `directio`. Set this value to `4k` if you use XFS under Linux.

Syntax: Size value

Default value: 512

open_file_cache

Context: `http` , `server` , `location`

This directive allows you to enable the cache, which stores information about open files. It does not actually store the file contents but only information such as:

- File descriptors (file size, modification time, and so on).
- The existence of files and directories.
- File errors, such as Permission denied, File not found, and so on. Note that this can be disabled with the `open_file_cache_errors` directive.

This directive accepts two arguments:

- `max=X` , where `X` is the number of entries that the cache can store. If this number is reached, older entries will be deleted in order to leave room for newer entries.
- `inactive=Y` , where `Y` is the number of seconds that a cache entry should be stored. By default, Nginx will wait for 60 seconds before clearing a cache entry. If the cache entry is accessed, the timer is reset. If the cache entry is accessed more often than the value defined by `open_file_cache_min_uses` , the cache entry will not be cleared (until Nginx runs out of space and decides to clear out the older entries).

Syntax: `open_file_cache max=X [inactive=Y] | off`

Default value: `off`

Example:

Copy

```
open_file_cache max=5000 inactive=180;
```

open_file_cache_errors

Context: `http` , `server` , `location`

This directive enables or disables the caching of file errors with the `open_file_cache` directive (read the preceding directive).

Syntax: `on` or `off`

Default value: `off`

open_file_cache_min_uses

Context: `http` , `server` , `location`

By default, entries in the `open_file_cache` are cleared after a period of inactivity (60 seconds, by default). However, if there is any activity, you can prevent Nginx from removing the cache entry. This directive defines the number of times an entry must be accessed in order to be eligible for protection.

Copy

```
open_file_cache_min_uses 3;
```

If the cache entry is accessed more than three times, it becomes permanently active and is not removed until Nginx decides to clear out the older entries to clear some space.

Syntax: Numeric value

Default value: `1`

open_file_cache_valid

Context: `http` , `server` , `location`

The open file cache mechanism is important, but the cached information quickly becomes obsolete, especially in the case of a fast-moving filesystem. From that perspective, information needs to be re-verified after a short period of time. This directive specifies the number of seconds that Nginx will wait before revalidating a cache entry.

Syntax: Time value (in seconds)

Default value: `60`

read_ahead

Context: `http` , `server` , `location`

The read_ahead directive defines the number of bytes to be pre-read from the files. Under Linux-based operating systems, setting this directive to a value above `0` will enable reading ahead, but the actual value that you specify has no effect. Set this to `0` to disable pre-reading.

Syntax: Size value

Default value: `0`

Other directives

The following directives relate to various aspects of the web server—logging, URI composition, DNS, and so on.

log_not_found

Context: `http` , `server` , `location`

This directive enables or disables the logging of `404 Not Found` HTTP errors. If your logs get filled with 404 errors due to missing `favicon.ico` or `robots.txt` files, you might want to turn this off.

Syntax: `on` or `off`

Default value: `on`

log_subrequest

Context: `http` , `server` , `location`

This directive enables or disables the logging of sub-requests triggered by internal redirects (see **The Rewrite module** section) or SSI requests (see the **Server Side Includes** module section).

Syntax: `on` or `off`

Default value: `off`

merge_slashes

Context: `http` , `server` , `location`

Enabling this directive will have the effect of merging multiple consecutive slashes in a URI. It turns out to be particularly useful in situations resembling the following:

Copy

```
server {
    […]
    server_name website.com;
    location /documents/ {
        type { }
        default_type text/plain;
    }
}
```

By default, if the client attempts to access `http://website.com//documents/` (note the `//` in the middle of the URI), Nginx will return a `404 Not found` HTTP error. If you enable this directive, the two slashes will be merged into one, and the location pattern will be matched.

Syntax: `on` or `off`

Default value: `off`

msie_padding

Context: `http` , `server` , `location`

This directive functions with the Microsoft Internet Explorer (MSIE) and Google Chrome browser families. In the case of error pages (with error code 400 or higher), if the length of the response body is less than 512 bytes, these browsers will display their own error page, sometimes at the expense of a more informative page provided by the server. If you enable this option, the body of responses with a status code of 400 or higher will be padded to 512 bytes.

Syntax: `on` or `off`

Default value: `off`

msie_refresh

Context: `http` , `server` , `location`

This is another MSIE-specific directive that takes effect in the case of HTTP response codes `301 Moved permanently` and `302 Moved temporarily` . When enabled, Nginx sends a response body containing a refresh meta tag (`<meta http-equiv="Refresh"…>`) to the clients running an MSIE browser in order to redirect the browser to the new location of the requested resource.

Syntax: `on` or `off`

Default value: `off`

resolver

Context: `http` , `server` , `location`

This directive specifies the name servers that should be employed by Nginx to resolve hostnames to IP addresses and vice versa. DNS query results are cached for some time, either by respecting the TTL provided by the DNS server, or by specifying a time value to the valid argument.

Syntax: one or more IPv4 or IPv6 addresses, `valid=Time value, ipv6=on|off`

Default value: None (system default)

Copy

```
resolver 127.0.0.1; # use local DNS
resolver 8.8.8.8 8.8.4.4 valid=1h; # use Google DNS and cache results for 1 hour
```

resolver_timeout

Context: `http` , `server` , `location`

Timeout for a hostname resolution query.

Syntax: Time value (in seconds)

Default value: 30

server_tokens

Context: `http` , `server` , `location`

This directive allows you to define whether or not Nginx should inform clients of the running version number. There are two situations where Nginx indicates its version number:

> - In the server header of HTTP responses (such as `nginx/1.8.0`). If you set `server_tokens` to `off` , the server header will only indicate `Nginx` .
> - On error pages, Nginx indicates the version number in the footer. If you set `server_tokens` to `off` , the footer of error pages will only indicate `Nginx` .

If you are running an older version of Nginx and do not plan to update it, it might be a good idea to hide your version number for security reasons.

Syntax: `on` or `off`

Default value: `on`

underscores_in_headers

Context: `http` , `server`

This directive allows or disallows underscores in custom HTTP header names. If this directive is set to `on` , the following example header is considered valid by `Nginx: test_header: value` .

Syntax: `on` or `off`

Default value: `off`

variables_hash_max_size

Context: `http`

This directive defines the maximum size of the variables hash tables. If your server configuration uses a total of more than 1024 variables, you will have to increase this value.

Syntax: Numeric value

Default value: `1024`

variables_hash_bucket_size

Context: `http`

This directive allows you to set the bucket size for the variables hash tables.

Syntax: Numeric value

Default value: `64` (or `32` or `128` , depending on your processor cache specifications)

post_action

Context: `http` , `server` , `location` , `if`

The `post_action` directive defines a post-completion action, a URI that will be called by Nginx after the request has been completed.

Syntax: URI or named `location` block.

Example:

Copy

```
location /payment/ {
    post_action /scripts/done.php;
}
```