Nginx was originally created as a web server to solve the [C10k problem](https://en.wikipedia.org/wiki/C10k_problem). And, as a web server, it can serve your data with blazing speed. But nginx is so much more than just a web server. You can use it as a reverse proxy, making for easy integration with slower upstream servers (like Unicorn or Puma). You can distribute your traffic properly (load balancing), stream media, resize your images on the fly, cache content, and much more.

The basic nginx architecture consists of a master process and its workers. The master is supposed to read the configuration file and maintain worker processes, while workers do the actual processing of requests.

## Base commands

To start nginx, simply type:

[sudo] nginx

While your nginx instance is running, you can manage it by sending signals:

[sudo] nginx -s signal

Available signals:

* stop: fast shutdown
* quit: graceful shutdown (wait for workers to finish their processes)
* reload: reload the configuration file
* reopen: reopen the log files

## Directive and Context

By default, the nginx configuration file can be found in:

* /etc/nginx/nginx.conf,
* /usr/local/etc/nginx/nginx.conf, or
* /usr/local/nginx/conf/nginx.conf

This file consists of:

* directive: the option that consists of name and parameters; it should end with a semicolon

gzip on;

* context: the section where you can declare directives (similar to scope in programming languages)

worker\_processes 2; # directive in global context

http { # http context

gzip on; # directive in http context

server { # server context

listen 80; # directive in server context

}

}

## Directive types

You have to pay attention when using the same directive in multiple contexts, as the inheritance model differs for different directives. There are 3 types of directives, each with its own inheritance model.

### Normal

Has one value per context. Also, it can be defined only once in the context. Subcontexts can override the parent directive, but this override will be valid only in a given subcontext.

gzip on;

gzip off; # illegal to have 2 normal directives in same context

server {

location /downloads {

gzip off;

}

location /assets {

# gzip is in here

}

}

### Array

Adding multiple directives in the same context will add to the values instead of overwriting them altogether. Defining a directive in a subcontext will override ALL parent values in the given subcontext.

error\_log /var/log/nginx/error.log;

error\_log /var/log/nginx/error\_notive.log notice;

error\_log /var/log/nginx/error\_debug.log debug;

server {

location /downloads {

# this will override all the parent directives

error\_log /var/log/nginx/error\_downloads.log;

}

}

### Action directive

Actions are directives that change things. Their inheritance behaviour will depend on the module.

For example, in the case of the rewrite directive, every matching directive will be executed:

server {

rewrite ^ /foobar;

location /foobar {

rewrite ^ /foo;

rewrite ^ /bar;

}

}

If the user tries to fetch /sample:

* a server rewrite is executed, rewriting from /sample, to /foobar
* the location /foobar is matched
* the first location rewrite is executed, rewriting from /foobar, to /foo
* the second location rewrite is executed, rewriting from /foo, to /bar

This is a different behaviour than what the return directive provides:

server {

location / {

return 200;

return 404;

}

}

In the case above, the 200 status is returned immediately.

## Processing requests

Inside nginx, you can specify multiple virtual servers, each described by a server { } context.

server {

listen \*:80 default\_server;

server\_name netguru.co;

return 200 "Hello from netguru.co";

}

server {

listen \*:80;

server\_name foo.co;

return 200 "Hello from foo.co";

}

server {

listen \*:81;

server\_name bar.co;

return 200 "Hello from bar.co";

}

This will give nginx some insight on how to handle incoming requests. Nginx will first check the listen directive to test which virtual server is listening on the given IP combination. Then, the value from server\_name directive is tested against the Host header, which stores the domain name of the server.

Nginx will choose the virtual server in the following order:

1. Server listing on IP, with a matching server\_name directive;
2. Server listing on IP, with the default\_server flag;
3. Server listing on IP, first one defined;
4. If there are no matches, refuse the connection.

In the example above, this will mean:

Request to foo.co:80 => "Hello from foo.co"

Request to www.foo.co:80 => "Hello from netguru.co"

Request to bar.co:80 => "Hello from netguru.co"

Request to bar.co:81 => "Hello from bar.co"

Request to foo.co:81 => "Hello from bar.co"

### The server\_name directive

The server\_name directive accepts multiple values. It also handles wildcard matching and regular expressions.

server\_name netguru.co www.netguru.co; # exact match

server\_name \*.netguru.co; # wildcard matching

server\_name netguru.\*; # wildcard matching

server\_name ~^[0-9]\*\.netguru\.co$; # regexp matching

When there is ambiguity, nginx uses the following order:

1. Exact name;
2. Longest wildcard name starting with an asterisk, e.g. “\*.example.org”;
3. Longest wildcard name ending with an asterisk, e.g. “mail.\*”;
4. First matching regular expression (in the order of appearance in the configuration file).

Nginx will store 3 hash tables: exact names, wildcards starting with an asterisk, and wildcards ending with an asterisk. If the result is not in any of the tables, the regular expressions will be tested sequentially.

It is worth keeping in mind that

server\_name .netguru.co;

is an abbreviation of:

server\_name netguru.co www.netguru.co \*.netguru.co;

With one difference: .netguru.co is stored in the second table, which means that it is a bit slower than an explicit declaration.

### listen directive

In most cases, you’ll find that the listen directive accepts IP values.

listen 127.0.0.1:80;

listen 127.0.0.1; # by default port :80 is used

listen \*:81;

listen 81; # by default all ips are used

listen [::]:80; # IPv6 addresses

listen [::1]; # IPv6 addresses

However, it is also possible to specify UNIX-domain sockets:

listen unix:/var/run/nginx.sock;

You can even use hostnames:

listen localhost:80;

listen netguru.co:80;

This should be used with caution, as the hostname may not be resolved upon nginx's launch, causing nginx to be unable to bind on a given TCP socket.

Finally, if the directive is not present, \*:80, is used.

## Minimal configuration

With all that knowledge, we should be able to create and understand the minimal configuration needed to run nginx.

# /etc/nginx/nginx.conf

events {} # event context needs to be defined to consider config valid

http {

server {

listen 80;

server\_name netguru.co www.netguru.co \*.netguru.co;

return 200 "Hello";

}

}

## root, location, and try\_files directives

### root directive

The root directive sets the root directory for requests, allowing nginx to map the incoming request onto the file system.

server {

listen 80;

server\_name netguru.co;

root /var/www/netguru.co;

}

Which allows nginx to return server content according to the request:

netguru.co:80/index.html # returns /var/www/netguru.com/index.html

netguru.co:80/foo/index.html # returns /var/www/netguru.com/foo/index.html

### location directive

The location directive sets the configuration depending on the requested URI.

location [modifier] path

location /foo {

# ...

}

When no modifier is specified, the path is treated as prefix, after which anything can follow. The above example will match:

/foo

/fooo

/foo123

/foo/bar/index.html

...

Also, multiple location directives can be used in a given context:

server {

listen 80;

server\_name netguru.co;

root /var/www/netguru.co;

location / {

return 200 "root";

}

location /foo {

return 200 "foo";

}

}

netguru.co:80 / # => "root"

netguru.co:80 /foo # => "foo"

netguru.co:80 /foo123 # => "foo"

netguru.co:80 /bar # => "root"

Nginx also provides a few modifiers which can be used in conjunction with location. These modifiers impact which location block will be used, as each modifier has assigned precedence.

= - Exact match

^~ - Preferential match

~ && ~\* - Regex match

no modifier - Prefix match

Nginx will first check for any exact matches. If it doesn't find any, it'll look for preferential ones. If this match also fails, regex matches will be tested in the order of their appearance. If everything else fails, the last prefix match will be used.

location /match {

return 200 'Prefix match: matches everything that starting with /match';

}

location ~\* /match[0-9] {

return 200 'Case insensitive regex match';

}

location ~ /MATCH[0-9] {

return 200 'Case sensitive regex match';

}

location ^~ /match0 {

return 200 'Preferential match';

}

location = /match {

return 200 'Exact match';

}

/match # => 'Exact match'

/match0 # => 'Preferential match'

/match1 # => 'Case insensitive regex match'

/MATCH1 # => 'Case sensitive regex match'

/match-abc # => 'Prefix match: matches everything that starting with /match'

### try\_files directive

This directive will try different paths, returning whichever is found.

try\_files $uri index.html =404;

So for /foo.html , it will try to return files in the following order:

1. $uri ( /foo.html );
2. index.html;
3. If none is found: 404.

What’s interesting, if we define try\_files in a server context, and then define a location that matches all requests, our try\_files will not be executed. This will happen because try\_files in a server context defines its own pseudo-location, which is the least specific location possible. Therefore, defining location / will be more specific than our pseudo-location.

server {

try\_files $uri /index.html =404;

location / {

}

}

Thus, you should avoid try\_files in a server context:

server {

location / {

try\_files $uri /index.html =404;

}

}

## Concluding remarks

Below, we'll discuss the most common contexts that you're likely to come across when working with Nginx.

## The Core Contexts

The first group of contexts that we will discuss are the core contexts that Nginx utilizes in order to create a hierarchical tree and separate the concerns of discrete configuration blocks. These are the contexts that comprise the major structure of an Nginx configuration.

### The Main Context

The most general context is the "main" or "global" context. It is the only context that is not contained within the typical context blocks that look like this:

# The main context is here, outside any other contexts

. . .

context {

. . .

}

Any directive that exist entirely outside of these blocks is said to inhabit the "main" context. Keep in mind that if your Nginx configuration is set up in a modular fashion, some files will contain instructions that appear to exist outside of a bracketed context, but which will be included within such a context when the configuration is stitched together.

The main context represents the broadest environment for Nginx configuration. It is used to configure details that affect the entire application on a basic level. While the directives in this section affect the lower contexts, many of these aren't inherited because they cannot be overridden in lower levels.

Some common details that are configured in the main context are the user and group to run the worker processes as, the number of workers, and the file to save the main process's PID. You can even define things like worker CPU affinity and the "niceness" of worker processes. The default error file for the entire application can be set at this level (this can be overridden in more specific contexts).

### The Events Context

The "events" context is contained within the "main" context. It is used to set global options that affect how Nginx handles connections at a general level. There can only be a single events context defined within the Nginx configuration.

This context will look like this in the configuration file, outside of any other bracketed contexts:

# main context

events {

# events context

. . .

}

Nginx uses an event-based connection processing model, so the directives defined within this context determine how worker processes should handle connections. Mainly, directives found here are used to either select the connection processing technique to use, or to modify the way these methods are implemented.

Usually, the connection processing method is automatically selected based on the most efficient choice that the platform has available. For Linux systems, the epoll method is usually the best choice.

Other items that can be configured are the number of connections each worker can handle, whether a worker will only take a single connection at a time or take all pending connections after being notified about a pending connection, and whether workers will take turns responding to events.

### The HTTP Context

When configuring Nginx as a web server or reverse proxy, the "http" context will hold the majority of the configuration. This context will contain all of the directives and other contexts necessary to define how the program will handle HTTP or HTTPS connections.

The http context is a sibling of the events context, so they should be listed side-by-side, rather than nested. They both are children of the main context:

# main context

events {

# events context

. . .

}

http {

# http context

. . .

}

While lower contexts get more specific about how to handle requests, directives at this level control the defaults for every virtual server defined within. A large number of directives are configurable at this context and below, depending on how you would like the inheritance to function.

Some of the directives that you are likely to encounter control the default locations for access and error logs (access\_log and error\_log), configure asynchronous I/O for file operations (aio, sendfile, and directio), and configure the server's statuses when errors occur (error\_page). Other directives configure compression (gzip and gzip\_disable), fine-tune the TCP keep alive settings (keepalive\_disable, keepalive\_requests, and keepalive\_timeout), and the rules that Nginx will follow to try to optimize packets and system calls (sendfile, tcp\_nodelay, and tcp\_nopush). Additional directives configure an application-level document root and index files (root and index) and set up the various hash tables that are used to store different types of data (\*\_hash\_bucket\_size and \*\_hash\_max\_size for server\_names, types, and variables).

### The Server Context

The "server" context is declared within the "http" context. This is our first example of nested, bracketed contexts. It is also the first context that allows for multiple declarations.

The general format for server context may look something like this. Remember that these reside within the http context:

# main context

http {

# http context

server {

# first server context

}

server {

# second server context

}

}

The reason for allowing multiple declarations of the server context is that each instance defines a specific virtual server to handle client requests. You can have as many server blocks as you need, each of which can handle a specific subset of connections.

Due to the possibility and likelihood of multiple server blocks, this context type is also the first that Nginx must use a selection algorithm to make decisions. Each client request will be handled according to the configuration defined in a single server context, so Nginx must decide which server context is most appropriate based on details of the request. The directives which decide if a server block will be used to answer a request are:

* listen: The ip address / port combination that this server block is designed to respond to. If a request is made by a client that matches these values, this block will potentially be selected to handle the connection.
* server\_name: This directive is the other component used to select a server block for processing. If there are multiple server blocks with listen directives of the same specificity that can handle the request, Nginx will parse the "Host" header of the request and match it against this directive.

The directives in this context can override many of the directives that may be defined in the http context, including logging, the document root, compression, etc. In addition to the directives that are taken from the http context, we also can configure files to try to respond to requests (try\_files), issue redirects and rewrites (return and rewrite), and set arbitrary variables (set).

### The Location Context

The next context that you will deal with regularly is the location context. Location contexts share many relational qualities with server contexts. For example, multiple location contexts can be defined, each location is used to handle a certain type of client request, and each location is selected by virtue of matching the location definition against the client request through a selection algorithm.

While the directives that determine whether to select a server block are defined within the server context, the component that decides on a location's ability to handle a request is located in the location definition (the line that opens the location block).

The general syntax looks like this:

location match\_modifier location\_match {

. . .

}

Location blocks live within server contexts and, unlike server blocks, can be nested inside one another. This can be useful for creating a more general location context to catch a certain subset of traffic, and then further processing it based on more specific criteria with additional contexts inside:

# main context

server {

# server context

location /match/criteria {

# first location context

}

location /other/criteria {

# second location context

location nested\_match {

# first nested location

}

location other\_nested {

# second nested location

}

}

}

While server contexts are selected based on the requested IP address/port combination and the host name in the "Host" header, location blocks further divide up the request handling within a server block by looking at the request URI. The request URI is the portion of the request that comes after the domain name or IP address/port combination.

So, if a client requests http://www.example.com/blog on port 80, the http, www.example.com, and port 80 would all be used to determine which server block to select. After a server is selected, the /blog portion (the request URI), would be evaluated against the defined locations to determine which further context should be used to respond to the request.

Many of the directives you are likely to see in a location context are also available at the parent levels. New directives at this level allow you to reach locations outside of the document root (alias), mark the location as only internally accessible (internal), and proxy to other servers or locations (using http, fastcgi, scgi, and uwsgi proxying).

## Other Contexts

While the above examples represent the essential contexts that you will encounter with Nginx, other contexts exist as well. The contexts below were separated out either because they depend on more optional modules, they are used only in certain circumstances, or they are used for functionality that most people will not be using.

We will not be discussing each of the available contexts though. The following contexts will not be discussed in any depth:

* split\_clients: This context is configured to split the clients that the server receives into categories by labeling them with variables based on a percentage. These can then be used to do A/B testing by providing different content to different hosts.
* perl / perl\_set: These contexts configures Perl handlers for the location they appear in. This will only be used for processing with Perl.
* map: This context is used to set the value of a variable depending on the value of another variable. It provides a mapping of one variable's values to determine what the second variable should be set to.
* geo: Like the above context, this context is used to specify a mapping. However, this mapping is specifically used to categorize client IP addresses. It sets the value of a variable depending on the connecting IP address.
* types: This context is again used for mapping. This context is used to map MIME types to the file extensions that should be associated with them. This is usually provided with Nginx through a file that is sourced into the main nginx.conf config file.
* charset\_map: This is another example of a mapping context. This context is used to map a conversion table from one character set to another. In the context header, both sets are listed and in the body, the mapping takes place.

The contexts below are not as common as the ones we have discussed so far, but are still very useful to know about.

### The Upstream Context

The upstream context is used to define and configure "upstream" servers. Basically, this context defines a named pool of servers that Nginx can then proxy requests to. This context will likely be used when you are configuring proxies of various types.

The upstream context should be placed within the http context, outside of any specific server contexts. The general form looks something like this:

# main context

http {

# http context

upstream upstream\_name {

# upstream context

server proxy\_server1;

server proxy\_server2;

. . .

}

server {

# server context

}

}

The upstream context can then be referenced by name within server or location blocks to pass requests of a certain type to the pool of servers that have been defined. The upstream will then use an algorithm (round-robin by default) to determine which specific server to hand the request to. This context gives our Nginx the ability to do some load balancing when proxying requests.

### The Mail Context

Although Nginx is most often used as a web or reverse proxy server, it can also function as a high performance mail proxy server. The context that is used for directives of this type is called, appropriately, "mail". The mail context is defined within the "main" or "global" context (outside of the http context).

The main function of the mail context is to provide an area for configuring a mail proxying solution on the server. Nginx has the ability to redirect authentication requests to an external authentication server. It can then provide access to POP3 and IMAP mail servers for serving the actual mail data. The mail context can also be configured to connect to an SMTP Relayhost if desired.

In general, a mail context will look something like this:

# main context

events {

# events context

}

mail {

# mail context

}

### The If Context

The "if" context can be established to provide conditional processing of directives defined within. Like an if statement in conventional programming, the if directive in Nginx will execute the instructions contained if a given test returns "true".

The if context in Nginx is provided by the rewrite module and this is the primary intended use of this context. Since Nginx will test conditions of a request with many other purpose-made directives, if should not be used for most forms of conditional execution. This is such an important note that the Nginx community has created a page called [if is evil](https://www.nginx.com/resources/wiki/start/topics/depth/ifisevil/).

The problem is basically that the Nginx processing order can very often lead to unexpected results that seem to subvert the meaning of an if block. The only directives that are considered reliably safe to use inside of these contexts are the return and rewrite directives (the ones this context was created for). Another thing to keep in mind when using an if context is that it renders a try\_files directive in the same context useless.

Most often, an if will be used to determine whether a rewrite or return is needed. These will most often exist in location blocks, so the common form will look something like this:

# main context

http {

# http context

server {

# server context

location location\_match {

# location context

if (test\_condition) {

# if context

}

}

}

}

### The Limit\_except Context

The limit\_except context is used to restrict the use of certain HTTP methods within a location context. For example, if only certain clients should have access to POST content, but everyone should have the ability to read content, you can use a limit\_except block to define this requirement.

The above example would look something like this:

. . .

# server or location context

location /restricted-write {

# location context

limit\_except GET HEAD {

# limit\_except context

allow 192.168.1.1/24;

deny all;

}

}

This will apply the directives inside the context (meant to restrict access) when encountering any HTTP methods except those listed in the context header. The result of the above example is that any client can use the GET and HEAD verbs, but only clients coming from the 192.168.1.1/24 subnet are allowed to use other methods.

## General Rules to Follow Regarding Contexts

Now that you have an idea of the common contexts that you are likely to encounter when exploring Nginx configurations, we can discuss some best practices to use when dealing with Nginx contexts.

### Apply Directives in the Highest Context Available

Many directives are valid in more than one context. For instance, there are quite a few directives that can be placed in the http, server, or location context. This gives us flexibility in setting these directives.

However, as a general rule, it is usually best to declare directives in the highest context to which they are applicable, and overriding them in lower contexts as necessary. This is possible because of the inheritance model that Nginx implements. There are many reasons to use this strategy.

First of all, declaring at a high level allows you to avoid unnecessary repetition between sibling contexts. For instance, in the example below, each of the locations is declaring the same document root:

http {

server {

location / {

root /var/www/html;

. . .

}

location /another {

root /var/www/html;

. . .

}

}

}

You could move the root out to the server block, or even to the http block, like this:

http {

root /var/www/html;

server {

location / {

. . .

}

location /another {

. . .

}

}

}

Most of the time, the server level will be most appropriate, but declaring at the higher level has its advantages. This not only allows you to set the directive in fewer places, it also allows you to cascade the default value down to all of the child elements, preventing situations where you run into an error by forgetting a directive at a lower level. This can be a major issue with long configurations. Declaring at higher levels provides you with a sane default.

### Use Multiple Sibling Contexts Instead of If Logic for Processing

When you want to handle requests differently depending on some information that can be found in the client's request, often users jump to the "if" context to try to conditionalize processing. There are a few issues with this that we touched on briefly earlier.

The first is that the "if" directive often return results that do not align with the administrator's expectations. Although the processing will always lead to the same result given the same input, the way that Nginx interprets the environment can be vastly different than can be assumed without heavy testing.

The second reason for this is that there are already optimized, purpose-made directives that are used for many of these purposes. Nginx already engages in a well-documented selection algorithm for things like selecting server blocks and location blocks. So if it is possible, it is best to try to move your different configurations into their own blocks so that this algorithm can handle the selection process logic.

For instance, instead of relying on rewrites to get a user supplied request into the format that you would like to work with, you should try to set up two blocks for the request, one of which represents the desired method, and the other that catches messy requests and redirects (and possibly rewrites) them to your correct block.

The result is usually easier to read and also has the added benefit of being more performant. Correct requests undergo no additional processing and, in many cases, incorrect requests can get by with a redirect rather than a rewrite, which should execute with lower overhead.