2021

NGINX

A GUIDELINE

TAHIRA BISHWAS ANNY

JANUARY 13, 2021

Contents

Introduction

NGINX is a web server which Initially worked as HTTP web serving. However, today, it also serves as a reverse proxy, HTTP load balancer and email proxy for IMAP, POP3 and SMTP.

Igor Sysoev, creator of the Software, initiated his research in 2002 as an attempt to solve the C10K problem. C10K is a challenge of managing ten thousand connections at the same time. To resolve the problem, NGINX offers an **event-driven** and **asynchronous** architecture, which made NGINX one of the most reliable servers for speed and it's highly scalable as well, means its service grows along with its client's traffic.

As NGINX has the superb ability to handle a lot of connections and speed, many high traffic websites like Instagram, Netflix, Adobe, Cloudflare, Uber, Slack and many more use NGINX.

Overview

Before learning more about NGINX, let's just take a general look how it's actually works. But before that let's have a quick check in how web server works.

When someone makes a request, the browser contacts the server of that website. Then, the server looks for the requested item for the page and sends it back to the browser. This is how traditional web servers works, create a single thread for every single request.

On the other hand, NGINX has **one master process** and **several worker processes**. The main purpose of the master process is to read and evaluate configuration and maintain worker processes. Then worker processes are the one who actually process a request.

NGINX manage similar threads under one **worker process** and each worker process contains smaller units called **worker connections**. This whole unit is then responsible for handling request threads. Primarily, Worker connections deliver the requests to a worker process, then worker will send it to a master process. Finally, master process provides the result of those requests. Actually, the number of worker process is defined in the configuration file or it can be altered for a given configuration or automatically adjusted to the number of CPU cores available.

One point about NGINX is, it's performance on Windows is not as great as on other platforms. Detail explanation can be found here.

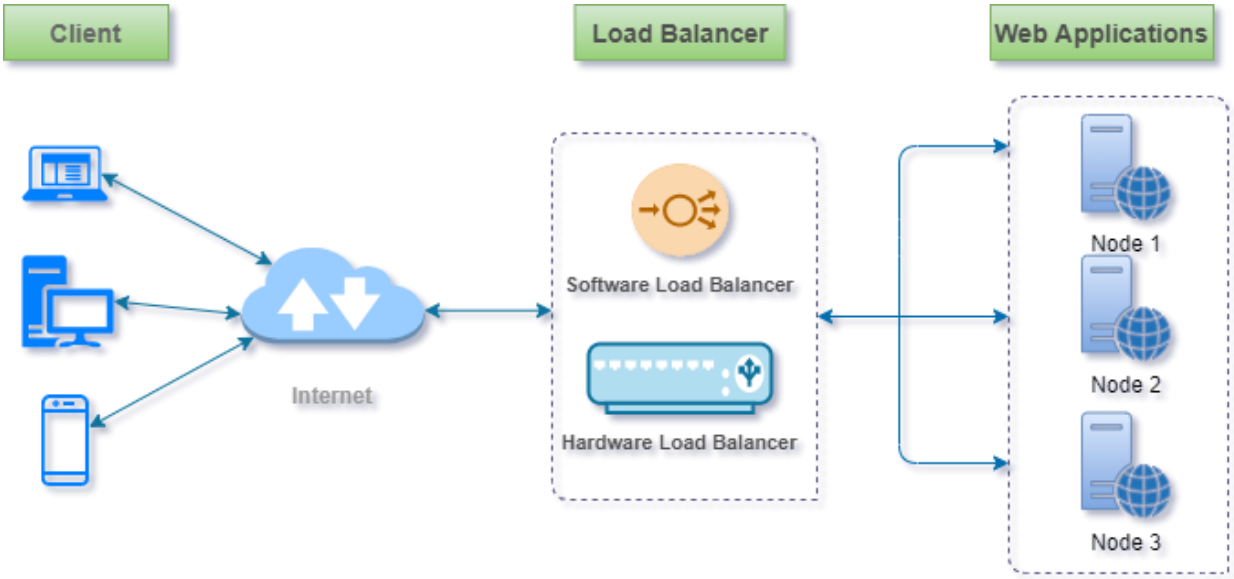
General Proxy Information

Proxy servers and load balancers are components in a client-server computing architecture. Both act as intermediaries in the communication between the clients and servers, performing functions that improve efficiency. They can be implemented as dedicated, purpose-built devices, but increasingly in modern web architectures they are software applications that run on commodity hardware.

We often got tricked by the definition of the basic Proxy server and load balancer. Actually, both types of application reside between clients and servers, accepting requests from the former and delivering responses from the latter. Obviously, there are some serious confusion going on, let's just take a look on both types and tease them apart.

Load Balancing

When a site needs multiple servers because the volume of the requests is way too much for a single server to handle efficiently, there comes load balancers to rescue. Deploying multiple servers, commonly same content in all the server, makes the website more reliable, prevent overload on any server and also most importantly eliminates a single point failure.

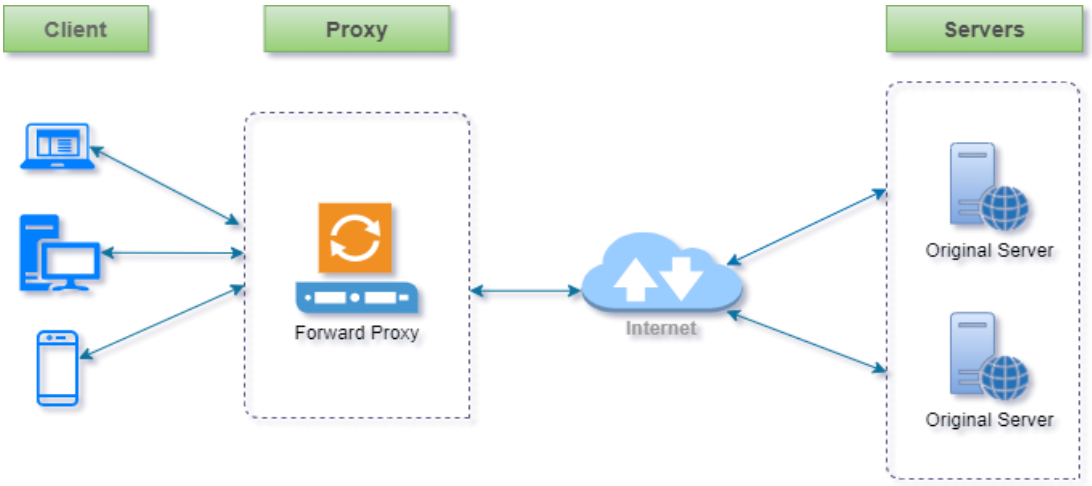


Proxy Servers

"Reverse proxies shouldn't be confused with forward proxies, which are used not for load balancing but for passing requests to the internet from private networks through a firewall and can act as cache servers to reduce outward traffic."

- **Forward Proxy**

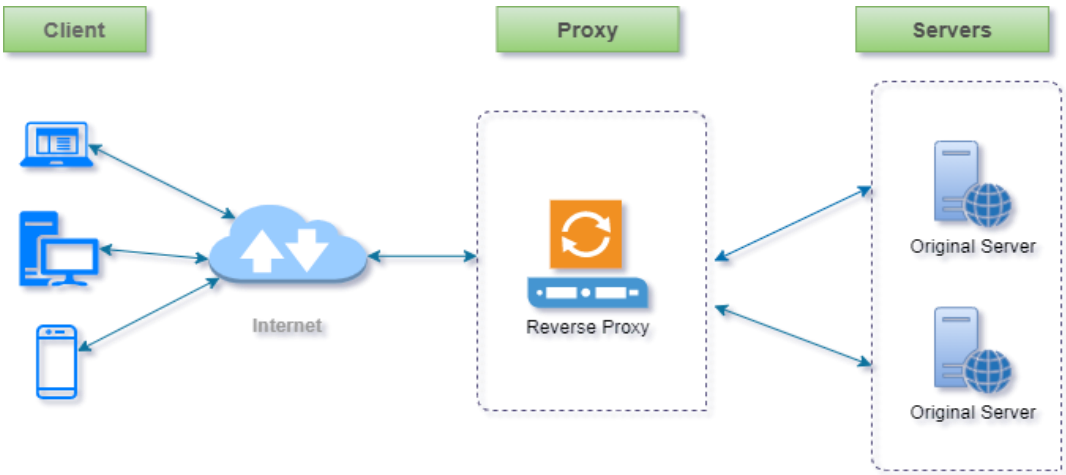
  A forward proxy, often called as proxy or proxy server or web proxy, is a server that serves as a middleman between clients and the services of the internet.
  A forward proxy can be useful to get around restrictions or blocked users, as they let the user connect to the proxy rather than directly to the sites they are visiting.



- **Reverse Proxy**

  Whereas deploying a load balancer makes sense only when you have multiple servers, it often makes sense to deploy a reverse proxy even with just one web server or application server.

  Reverse proxies have built-in security services, including authentication and abstraction of web servers. A reverse proxy stores a copy of response locally before responding to the client, when the client makes the same request again, the reverse proxy can provide the response from the cache instead of forwarding the request to the backend server, which decrease the response time of the client as well as reduces the load on the backend server.



Version Used

NGINX: nginx-1.18.0 (Download From here)

N.B: Normally nginx will recommend you to download the latest Mainline version, but I installed the recent stable version.

Installation Process

STEP 1: Download the file from here.

N. B: Nginx recommends using the "mainline version." However, you will not find any issues if you download its most recent stable version for Windows.

STEP 2: Then moved the zip file to C drive to access it easily, unzip it and then start it using command prompt.

N.B: Firewall will block it, when you will try to run it. For some reason, in my PC firewall notification panel was not showing. So, I run it through command prompt and it was working fine.

Commands:

- `cd C:\`
- `cd nginx-1.18.0`
- `nginx-1.18.0>start nginx`

STEP 3: Open a new Tab and then go to localhost and you should see the "Welcome to Nginx" default page. If you see this page, then we can be sure that Nginx has been installed properly.

Starting, Stopping, and Reloading Configuration

Once nginx is started, it can be controlled by invoking the executable with the -s parameter. Use the following syntax:

```
nginx -s "commands"
```

Commands

nginx/Windows runs as a standard console application (not a service), and it can be managed using the following commands:

nginx -s stop	fast shutdown
nginx -s quit	graceful shutdown
nginx -s reload	changing configuration, starting new worker processes with a new configuration, graceful shutdown of old worker processes
nginx -s reopen	re-opening log files

This command should be executed under the same user that started nginx.

Configuring

Nginx and its modules work are determined in the configuration file. By default, the configuration file is named `nginx.conf` and placed in the directory `/usr/local/nginx/conf`, `/etc/nginx`, or `/usr/local/etc/nginx`. The location will vary depending on the installation procedure of NGINX. In my case it was in **nginx.conf (C:\nginx-1.18.0\conf)** directory.

Modules: NGINX is actually a collection of modules. Even very basic functions like HTTP, or serving static files inside HTTP – these basic things are modules.

Let's take a detail look in conf file.

Open the core NGINX configuration file in a text editor. Preferable- Visual Studio Code.

The very first thing that you will notice that the configurations are organized in a tree-like structure surrounded by curly braces { and }. These locations surrounded by braces is known as Context for placing configuration directive. Moreover, the configuration directives along with their parameters end with a semicolon.

In Nginx parlance, the areas that these brackets define are called "contexts" because they contain configuration details that are separated according to their area of concern. Basically, these divisions provide an organizational structure along with some conditional logic to decide whether to apply the configurations within. (examples: events, http, server, and location).

Because contexts can be layered within one another, Nginx provides a level of directive inheritance. As a general rule, if a directive is valid in multiple nested scopes, a declaration in a broader context will be passed on to any child contexts as default values. The children contexts can override these values at will. It is worth noting that an override to any array-type directives will *replace* the previous value, not append to it.

Directives can only be used in the contexts that they were designed for. Nginx will error out on reading a configuration file with directives that are declared in the wrong context. The Nginx documentation contains information about which contexts each directive is valid in, so it is a great reference if you are unsure.

Directives placed in the configuration file outside of any contexts are considered to be in the main context. The `events` and `http` directives reside in the `main` context, `server` in `http`, and `location` in `server`.

The rest of a line after the # sign is considered a comment.

If you don't want to read more then you can directly jump to The Upstream Context after knowing a bit about main context.

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 a Nginx configuration.

NGINX Config File – Main context

The main context is used to set the settings for NGINX globally and is the only context that is not surrounded by curly braces. Generally, the main context is placed at the beginning of the core NGINX configuration file. The directives for the main context cannot be inherited in any other context and therefore cannot be overridden.

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:

```
# Main context is outside of any other contexts
.....
context {
  .....
}
```

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).

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
      .....
}
```

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
      .....
}
```

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
events {
    # Events 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.

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. Please have a look here if you want to know more context and wish to make with own module or some other purpose.

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:

```
# http context

upstream backend_hosts {
    server host1.example.com;
    server host2.example.com;
    server host3.example.com;
}

server {
    listen 80;
    server_name example.com;

    location /proxy-me {
        proxy_pass http://backend_hosts;
    }
}
```

In the above example, we've set up an upstream context called `backend hosts`. Once defined, this name will be available for use within proxy passes as if it were a regular domain name. As you can see, within our server block we pass any request made to `example.com/proxy-me/...` to the pool we defined above. Within that pool, a host is selected by applying a configurable algorithm. By default, this is just a simple round-robin selection process (each request will be routed to a different host in turn).

Changing the Upstream Balancing Algorithm

You can modify the balancing algorithm used by the upstream pool by including directives or flags within the upstream context:

- **(round robin)**: The default load balancing algorithm that is used if no other balancing directives are present. Each server defined in the upstream context is passed requests sequentially in turn.
- **least_conn**: Specifies that new connections should always be given to the backend that has the least number of active connections. This can be especially useful in situations where connections to the backend may persist for some time.
- **ip_hash**: This balancing algorithm distributes requests to different servers based on the client's IP address. The first three octets are used as a key to decide on the server to handle the request. The result is that clients tend to be served by the same server each time, which can assist in session consistency.
- **hash**: This balancing algorithm is mainly used with memcached proxying. The servers are divided based on the value of an arbitrarily provided hash key. This can be text, variables, or a combination. This is the only balancing method that requires the user to provide data, which is the key that should be used for the hash.

When changing the balancing algorithm, the block may look something like this:

```
# http context

upstream backend_hosts {

    least_conn;

    server host1.example.com;
    server host2.example.com;
    server host3.example.com;
}

. . .
```

In the above example, the server will be selected based on which one has the least connections. The ip_hash directive could be set in the same way to get a certain amount of session "stickiness".

Round-robin and least connections balancing schemes are fair and have their uses. However, they cannot provide session persistence.

nGinx Worker Process:

nGinx has one master process and several worker processes. The main purpose of the master process is to read and evaluate configuration, and maintain worker processes. Worker processes do actual processing of requests. nginx employs event-based model and OS-dependent mechanisms to efficiently distribute requests among worker processes. The number of worker processes is defined in the configuration file and may be fixed for a given configuration or automatically adjusted to the number of available CPU cores.

N.B: This is all I read, What I wrote above, you might find sites containing the exact same information. So please don't claim any copy right issue on me.

References:

Beginner Guide

How to configure load balancing using Nginx

Understanding Nginx HTTP Proxying, Load Balancing, Buffering, and Caching