# ****[In short how an SSL works between web servers and browsers](https://linuxtechme.wordpress.com/2015/01/16/in-short-how-an-ssl-works-between-web-servers-and-browsers/)****

1. A browser requests a secure page (usually **https://**)
2. The web server sends its public key with its certificate.
3. The browser checks that the certificate was issued by a trusted party (usually a trusted root CA), that the certificate is still valid and that the certificate is related to the site contacted.
4. The browser then uses the public key, to encrypt a random symmetric encryption key and sends it to the server with the encrypted URL required as well as other encrypted http data.
5. The web server decrypts the symmetric encryption key using its private key and uses the symmetric key to decrypt the URL and http data.
6. The web server sends back the requested html document and http data encrypted with the symmetric key.
7. The browser decrypts the http data and html document using the symmetric key and displays the information.

**What is Difference between event, worker and prefork**

Apache (HTPD) is  very popular and widely deployed web server arround the world. A-Patchy server comes with multiple modules. The term MPM is used for multiprocessing module. We can check for default mpm by running this command “ httpd -l ”

Apache 2 is available with following 3 MPM modules.

**PREFORK**  
**WORKER**

**EVENT**

(mpm\_winnt This Multi-Processing Module is optimized for Windows NT.)  
(mpm\_netware Multi-Processing Module implementing an exclusively threaded web server optimized for Novell NetWare)

**Prefork MPM**

A prefork mpm handles http requests just like older Apache 1.3. As the name suggests it will pre-fork necessary child process while starting Apache. It is suitable for all those websites which don’t want threading for compatibility. i.e for non-thread-safe libraries . It is also known as the best MPM for isolating each incoming http request.

**How it works**

A single control (master) process is responsible for launching multiple child processes which serves incoming http requests. Apache always tries to maintain several spare (not-in-use) server processes, which stand ready to serve incoming requests. In this way, clients do not need to wait for a new child processes to be forked before their requests can be served.  
We can adjust this spare process through the Apache configuration. Default settings are usually enough for small amount of traffic. One can always tune those Directives / Values as per their requirements.

Pre-Fork is the default module given by Apache.

#prefork MPM  
#StartServers: number of server processes to start  
#MinSpareServers: minimum number of server processes which are kept spare  
#MaxSpareServers: maximum number of server processes which are kept spare  
#ServerLimit: maximum value for MaxClients for the lifetime of the server  
#MaxClients: maximum number of server processes allowed to start  
#MaxRequestsPerChild: maximum number of requests a server process serves

<IfModule prefork.c>  
StartServers       8  
MinSpareServers    5  
MaxSpareServers   20  
ServerLimit      256  
MaxClients       256  
MaxRequestsPerChild  4000  
</IfModule>

**Worker MPM**

A worker mpm is an Multi-Processing Module (MPM) which implements a hybrid multi-process multi-threaded server. By using threads to serve requests, it is able to serve a large number of requests with fewer system resources than a process-based server.

The most important directives used to control this MPM are ThreadsPerChild, which controls the number of threads deployed by each child process and MaxClients, which controls the maximum total number of threads that may be launched.

Strength : Memory usage and performance wise its better than prefork  
Weakness : worker will not work properly with languages like php

**How it works**

A single control process (the parent) is responsible for launching child processes. Each child process creates a fixed number of server threads as specified in the ThreadsPerChild directive, as well as a listener thread which listens for connections and passes them to a server thread for processing when they arrive.

Apache always tries to maintain a group of spare or idle server threads, which stand ready to serve incoming requests. In this way, clients do not need to wait for a new threads or processes to be created before their requests can be served. The number of processes that will initially launched is set by the StartServers directive. During operation, Apache assesses the total number of idle threads in all processes, and forks or kills processes to keep this number within the boundaries specified by MinSpareThreads and MaxSpareThreads. Since this process is very self-regulating, it is rarely necessary to modify these directives from their default values. The maximum number of clients that may be served simultaneously (i.e., the maximum total number of threads in all processes) is determined by the MaxClients directive. The maximum number of active child processes is determined by the MaxClients directive divided by the ThreadsPerChild directive

#worker MPM  
#StartServers: initial number of server processes to start  
#MaxClients: maximum number of simultaneous client connections  
#MinSpareThreads: minimum number of worker threads which are kept spare  
#MaxSpareThreads: maximum number of worker threads which are kept spare  
#ThreadsPerChild: constant number of worker threads in each server process  
#MaxRequestsPerChild: maximum number of requests a server process serves

<IfModule worker.c>  
StartServers         4  
MaxClients         300  
MinSpareThreads     25  
MaxSpareThreads     75  
ThreadsPerChild     25  
MaxRequestsPerChild  0  
</IfModule>

**Event MPM**

The event Multi-Processing Module (MPM) is designed to allow more requests to be served simultaneously by passing off some processing work to supporting threads, freeing up the main threads to work on new requests. Event has been released with stable in Apache 2.4. The Event MPM works the exact same way as the Worker MPM when it comes to processes and threads. The big difference is that an Event MPM will dedicate a thread to a request, not the whole HTTP connection.

**How it works**

This MPM tries to fix the ‘keep alive problem’ in HTTP. After a client completes the first request, the client can keep the connection open, and send further requests using the same socket. This can save significant overhead in creating TCP connections. However, Apache HTTP Server traditionally keeps an entire child process/thread waiting for data from the client, which brings its own disadvantages. To solve this problem, this MPM uses a dedicated thread to handle both the Listening sockets, all sockets that are in a Keep Alive state, and sockets where the handler and protocol filters have done their work and the only remaining thing to do is send the data to the client. The status page of mod\_status shows how many connections are in the mentioned states.

This is useful in a situation where you like the idea of threading, but have an application that uses rather long KeepAlive timeouts. With the Worker MPM, the thread would be bound to the connection, and stayed tied up regardless if a request was being processed or not.

With the Event MPM, the connection the thread is only used for requests and frees backup immediately after the request is fulfilled, regardless of the actual HTTP connection, which is handled by the parent process.  Since the thread frees up immediately after the request is fulfilled,  it can be used for other requests.

<IfModule event.c>  
MinSpareThreads 64  
MaxSpareThreads 128  
ThreadsPerChild 64  
ThreadLimit 64  
MaxRequestsPerChild 20000  
ListenBacklog 4096  
</IfModule>

**How to configure multiple apache virtual hosts on multiple ports**

**I have a server with one IP address. I want to run several virtual hosts in an Apache 2.4. And I want all virtual hosts to be accessible on port 80,8000 and 443 (ssl).**

The Listen directive tells the server to accept incoming requests only on the specified ports or address-and-port combinations. If only a port number is specified in the Listen directive, the server listens to the given port on all interfaces. If an IP address is given as well as a port, the server will listen on the given port and interface. Multiple Listen directives may be used to specify a number of addresses and ports to listen on. The server will respond to requests from any of the listed addresses and ports.  
**Listen \*:80  
Listen \*:8080**

**NameVirtualHost \*:80  
NameVirtualHost \*:8080**

**VirtualHost \*:80  
ServerName**[**a.foo.com**](http://a.foo.com/) **DocumentRoot /www/a  
/VirtualHost**

**VirtualHost \*:8080  
ServerName**[**b.foo.com**](http://b.foo.com/) **DocumentRoot /www/ab  
VirtualHost**  
=-=-=-

**How to Install an apache module without recompiling (Easyapache)**

**Login the server**

**cd /home/cpeasyapache/src/httpd-2.x.x/modules/mappers/**

Make sure the module is in uncompiled format (mod\_module.c).

From command prompt run:

**/usr/local/apache/bin/apxs -c mod\_module.c**  
Example : /usr/local/apache/bin/apxs -c mod\_imagemap.c

This will create the DSO in **/home/cpeasyapache/src/httpd-2.x.x/modules/mappers/.libs/** folder.

Copy the mod\_module.so file to **/usr/local/apache/modules/ file**

**Load the module and enable it in Apache configuration file.**

Check whether module is installed or not by using the below command :

**/usr/local/apache/bin/apachectl -t -D DUMP\_MODULES**

**How to Redirect HTTP traffic to another IP using iptables**

I want to redirect all traffic coming to the old server’s http port(during the TTL change period) to the webserver running in new server with a different ip .

Service : Apache(port 80 and 443)  
Interface name in old server(this is a vps) : venet0  
Destination ip : 198.89.54.263

SOLUTION:

Use iptables nat to redirect http and https traffic to another server ip and port. Execute the following commands in the source server.

**iptables -t nat -A PREROUTING -i venet0 -p tcp –dport 80 -m conntrack –ctstate NEW -j DNAT –to 98.59.254.163:80  
iptables -t nat -A PREROUTING -i venet0 -p tcp –dport 443 -m conntrack –ctstate NEW -j DNAT –to 98.59.254.163:443  
iptables -t nat -A PREROUTING -m conntrack –ctstate ESTABLISHED,RELATED -j ACCEPT  
iptables -A POSTROUTING -t nat -j MASQUERADE**

**Name or service not known: mod\_unique\_id: unable to find IPv4 address**

[Thu Jan 02 05:29:41 2014] [alert] (EAI 2)Name or service not known: mod\_unique\_id: unable to find IPv4 address of “server hostname”  
Configuration Failed

I have checked the Apache configuration syntax file and it is fine.

**root@ [/]# httpd -t  
Syntax OK**

While checking the file **“/etc/hosts”** the server hostname was incorrect. After correcting the Apache service was working fine.

(98)Address already in use: make\_sock: could not bind to address 0.0.0.0:80 no listening sockets available, shutting down Unable to open logs

Check the Apache process

root@ [~]# **ps aux | grep httpd**  
root 6846 0.0 0.0 4956 700 pts/0 S+ 04:22 0:00 grep httpd  
root 29439 0.0 0.7 145736 125784 ? S 03:06 0:00 /usr/local/apache/bin/httpd -k start -DSSL

Check whether port 80 is listening on the server.

root@ [~]# **netstat -lnp | grep :80**  
tcp 0 0 0.0.0.0:80 0.0.0.0:\* LISTEN 29439/httpd

The below command will display which service is running on port 80. Please use lsof command before and after the process start. You can see the changes.

**lsof -i tcp:80  
lsof -i | grep httpd**  
Kill the process and start the service

root@ [~]# **kill -9 29439**  
root@ [~]# **ps aux | grep httpd**  
root 7258 0.0 0.0 4956 704 pts/0 S+ 04:22 0:00 grep httpd

root@ [~]# **/etc/init.d/httpd start**  
root@ [~]# **ps aux | grep httpd**  
root 7312 17.2 0.7 151624 127688 ? Ss 04:23 0:00 /usr/local/apache/bin/httpd -k start -DSSL  
root 7335 0.0 0.7 143376 122460 ? S 04:23 0:00 /usr/local/apache/bin/httpd -k start -DSSL  
nobody 7336 6.0 0.7 153152 127272 ? S 04:23 0:00 /usr/local/apache/bin/httpd -k start -DSSL  
nobody 7337 2.0 0.7 152180 126460 ? S 04:23 0:00 /usr/local/apache/bin/httpd -k start -DSSL  
root 7361 0.0 0.0 4960 700 pts/0 S+ 04:23 0:00 grep httpd

# mod\_fcgid: can’t apply process slot for /usr/local/cpanel/cgi-sys/php5

Thu Aug 22 13:05:06 2013] [warn] [client 200.150.249.63] mod\_fcgid: can’t apply process slot for /usr/local/cpanel/cgi-sys/php5,

This issue commonly occur due to lower value of MaxRequestsPerProcess.

Solution :-

#vim /usr/local/apache/conf/php.conf

**MaxRequestsPerProcess 1000  
DefaultMaxClassProcessCount 120  
IdleTimeout 60  
MaxProcessCount 2000  
IPCCommTimeout 40  
IPCConnectTimeout 10  
MaxRequestLen 10240000**

[**Apache error “Error retrieving pid file logs/httpd.pid”**](https://linuxtechme.wordpress.com/2013/01/24/apache-error-error-retrieving-pid-file-logshttpd-pid/)

**Unable to start Apache service on cPanel server.**

If you are getting the following error while trying to restart the Apache service.

=====================>  
-bash-3.2# /etc/init.d/httpd start  
(20014)Internal error: Error retrieving pid file logs/httpd.pid  
Remove it before continuing if it is corrupted.  
=====================>

**Solution :-**

-bash-3.2# mv /usr/local/apache/logs/httpd.pid /usr/local/apache/logs/httpd.pid.bk  
-bash-3.2# /etc/init.d/httpd restart  
httpd not running, trying to start

Be sure that the apache service is running :

# **/etc/init.d/httpd status**

# [****Quick install for SourceGuardian on Debian Server****](https://linuxtechme.wordpress.com/2012/11/07/quick-install-for-sourceguardian-on-debian-server/)

Untar the tarball

Copy the loader file ixed.5.3.lin to the /usr/lib/php5/20090626/

**cp ixed.5.3\* to /usr/lib/php5/20090626/**

Create a Source Guardian configuration file:

**vim /etc/php5/conf.d/sourceguardian.ini**

Add the following two lines to sourceguardian.ini:

**[sourceguardian]  
zend\_extension=/usr/lib/php5/20090626/ixed.5.3.lin**

**Restart Apache:**

# **php -v**  
PHP 5.3.3-7+squeeze14 with Suhosin-Patch (cli) (built: Aug 6 2012 14:18:06)  
Copyright (c) 1997-2009 The PHP Group  
Zend Engine v2.3.0, Copyright (c) 1998-2010 Zend Technologies  
with SourceGuardian v9.0, Copyright (c) 2000-2012, by Inovica Ltd.  
with Suhosin v0.9.32.1, Copyright (c) 2007-2010, by SektionEins GmbH

**Why I Find Nginx Practically Better Than Apache**

Apache is a free, open-source HTTP server for Unix-like operating systems and Windows. It was designed to be a secure, efficient and extensible server that provides HTTP services in sync with the prevailing HTTP standards.

Ever since it’s launch, Apache has been the most popular web server on the Internet since 1996. It is the de facto standard for Web servers in the Linux and open source ecosystem. New Linux users normally find it easier to set up and use.

Nginx (pronounced ‘Engine-x’) is a free, open-source, high-performance HTTP server, reverse proxy, and an IMAP/POP3 proxy server. Just like Apache, it also runs on Unix-like operating systems and Windows.

Well known for it’s high performance, stability, simple configuration, and low resource consumption, it has over the years become so popular and its usage on the Internet is heading for greater heights. It is now the web server of choice among experienced system administrators or web masters of top sites.

Some of the busy sites powered by:

Apache are: PayPal, BBC.com, BBC.co.uk, SSLLABS.com, Apple.com plus lots more.

Nginx are: Netflix, Udemy.com, Hulu, Pinterest, CloudFlare, WordPress.com, GitHub, SoundCloud and many others.

There are numerous resources already published on the web concerning the comparison between Apache and Nginx (i really mean ‘Apache Vs Nginx’ articles), many of which clearly explain into detail, their top features and operations under various scenarios including performance measures in lab benchmarks. Therefore that will not be addressed here.

I will simply share my experience and thoughts about the whole debate, having tried out Apache and Nginx, both in production environments based on requirements for hosting modern web applications, in the next section.

Reasons Why I Find Nginx Practically Better Than Apache

Following are reasons why I prefer Nginx web server over Apache for modern web content delivery:

1. Nginx is Lightweight

Nginx is one of light weight web servers out there. It has small footprints on a system compared to Apache which implements a vast scope of functionality necessary to run an application.

Because Nginx puts together a handful of core features, it relies on dedicated third‑party upstream web servers such as an Apache backend, FastCGI, Memcached, SCGI, and uWSGI servers or application server, i.e language specific servers such as Node.js, Tomcat, etc.

Therefore its memory usage is far better suited for limited resource deployments, than Apache.

2. Nginx is Designed for High Concurrency

As opposed to Apache’s threaded- or process-oriented architecture (process‑per‑connection or thread‑per‑connection model), Nginx uses a scalable, event-driven (asynchronous) architecture. It employs a liable process model that is tailored to the available hardware resources.

It has a master process (which performs the privileged operations such as reading configuration and binding to ports) and which creates several worker and helper processes.

The worker processes can each handle thousands of HTTP connections simultaneously, read and write content to disk, and communicate with upstream servers. The helper processes (cache manager and cache loader) can manage on‑disk content caching operations.

This makes its operations scalable, and resulting into high performance. This design approach further makes it fast, favorable for modern applications. In addition, third-party modules can be used to extend the native functionalities in Nginx.

3. Nginx is Easy to Configure

Nginx has a simple configuration file structure, making it super easy to configure. It consists of modules which are controlled by directives specified in the configuration file. In addition, directives are divided into block directives and simple directives.

A block directive is defined by braces ({ and }). If a block directive can have other directives inside braces, it is called a context such as events, http, server, and location.

http {

server {

}

}

A simple directive consists of the name and parameters separated by spaces and ends with a semicolon (;).

http {

server {

location / {

## this is simple directive called root

root /var/www/hmtl/example.com/;

}

}

}

You can include custom configuration files using the include directive, for example.

http {

server {

}

## examples of including additional config files

include /path/to/config/file/\*.conf;

include /path/to/config/file/ssl.conf;

}

A practical example for me was how I managed to easily configure Nginx to run multiple websites with different PHP versions, which was a little of a challenge with Apache.

**4. Nginx is an Excellent Frontend Proxy**

One of the common uses of Nginx is setting it up as a proxy server, in this case it receives HTTP requests from clients and passes them to proxied or upstream servers that were mentioned above, over different protocols. You can also modify client request headers that are sent to the proxied server, and configure buffering of responses coming from the proxied servers.

Then it receives responses from the proxied servers and passes them to clients. It is mush easier to configure as a proxy server compared to Apache since the required modules are in most cases enabled by default.

**5. Nginx is Remarkable for Serving Static Content**

Static content or files are typically files stored on disk on the server computer, for example CSS files , JavaScripts files or images. Let’s consider a scenario where you using Nginx as a frontend for Nodejs (the application server).

Although Nodejs server (specifically Node frameworks) have built in features for static file handling, they don’t need to do some intensive processing to deliver non-dynamic content, therefore it is practically beneficial to configure the web server to serve static content directly to clients.

Nginx can perform a much better job of handling static files from a specific directory, and can prevent requests for static assets from choking upstream server processes. This significantly improves the overall performance of backend servers.

**6. Nginx is an Efficient Load Balancer**

To realize high performance and uptime for modern web applications may call for running multiple application instances on single or distributed HTTP servers. This may in turn necessitate for setting up load balancing to distribute load between your HTTP servers.

Today, load balancing has become a widely used approach for optimizing operating system resource utilization, maximizing flexibility, cutting down latency, increasing throughput, achieving redundancy, and establishing fault-tolerant configurations – across multiple application instances.

Nginx uses the following load balancing methods:

round-robin (default method) – requests to the upstream servers are distributed in a round-robin fashion (in order of the list of servers in the upstream pool).

least-connected – here the next request is proxied to the server with the least number of active connections.

ip-hash – here a hash-function is used to determine what server should be selected for the next request (based on the client’s IP address).

Generic hash – under this method, the system administrator specifies a hash (or key) with the given text, variables of the request or runtime, or their combination. For example, the key may be a source IP and port, or URI. Nginx then distributes the load amongst the upstream servers by generating a hash for the current request and placing it against the upstream servers.

Least time (Nginx Plus) – assigns the next request to the upstream server with the least number of current connections but favors the servers with the lowest average response times.

7. Nginx is Highly Scalable

Furthermore, Nginx is highly scalable and modern web applications especially enterprise applications demand for technology that provides high performance and scalability.

One company benefiting from Nginx’s amazing scalability features is CloudFlare, it has managed to scale its web applications to handle more than 15 billion monthly page views with a relatively modest infrastructure, according to Matthew Prince, co-founder and CEO of CloudFare.

# how is nginx different from apache

Many people in the industry might be aware of the thing called “speed” for which nginx is famous for. There are some other important difference between apache and nginx's working model, we will be discussing, that differences in detail.

Lets discuss two working models used by the apache web server. We will get to nginx later. Most people who are associated with apache might be knowing about these two models, through which apache servers its requests. These models are mentioned below.

1.Apache MPM Prefork

2.Apache MPM Worker.

Note: there are many different MPM modules available, for different platforms and functionalities but we will be discussing only the above two here.

Lets have a look at the main difference between MPM Prefork and MPM Worker. MPM stands forMulti Processing Module.

## MPM Prefork:

Most of the functionality in apache comes from modules, even this MPM Prefork comes as a module, and can be enabled or disabled. This prefork model of apache is non-threaded and is a good model, as it makes each and every connection isolated from each other.

So if one connection is is having some issues, the other one is not all effected. By default if no MPM module is specified then apache uses this MPM Prefork as its MPM module. But this model is very resource intensive.

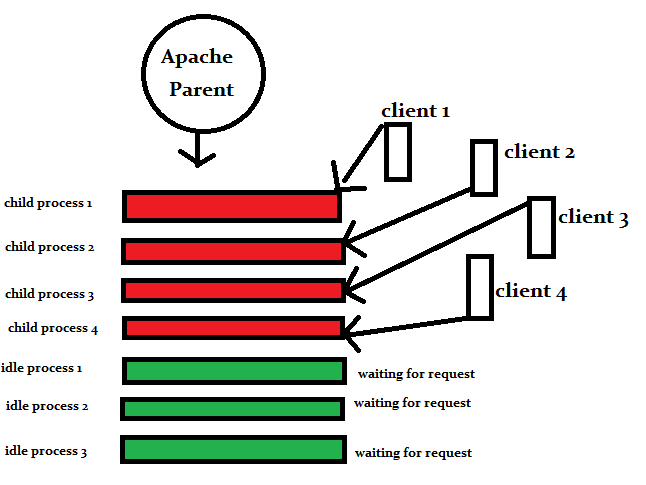
## ****Why is prefork model reource intensive?****

Because in this model a single parent process sits and creates many child processes which wait for requests and serve as the requests arrive. Which means each and every request is served by a seperate process. In other words, we can say its “process per request”. And apache maintains, several number of idle process before the requests arrive. Due to these idle processes waiting for requests, they can serve fast when requests arrive.

If you are interested in learning and understanding more about processes in linux. Then go through the below post.

**Read:** [Monitoring and administering processes in Linux](http://slashroot.in/linux-processes-administration-and-monitoring-tutorial)

But each and every process will utilize system resources like RAM, and CPU. And equal amount of RAM is utilized for each and every process.



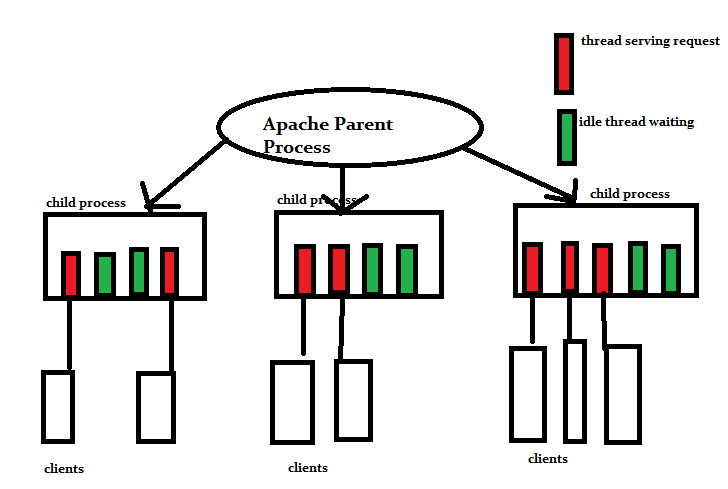
If you have got a lot number of requests at one time, then you will have lot number of child processes spawned by apache, and which will result in heavy resource utilization, as each process will utilize a certain amount of system memory and cpu resources.

## MPM Worker:

This model of apache can serve a large number of requests with less system resources than the prefork model because here a limited number of process will serve, many number of requests.

This is multi threaded architecture of apache. This model uses thread rather than process to serve requests. Now what is thread??

In Operating System's thread is a small instance of a process which does some job and exits. Thread is sometimes called a process inside a process.



In this model also there is one single parent process, which spawns some child processes. But there is no “process per requests”, but instead “thread per requests”. So the child process will have a certain number of threads inside it. Each child process will have certain “server threads” and certain “idle threads”. Idle threads are waiting for new requests, so there is no time wasted in creating threads when the requests arrive.

There is a directive inside apache config file /etc/httpd/conf/httpd.conf called “StartServers” which says how many child process will be there when apache starts.

Child process handles requests with the help of a fixed number of threads inside them which is specified by the argument “ThreadsPerChild” in the config file.

Note: there are some php module issues reported while working with apache MPM worker model.

Now lets discuss nginx.

## ****Nginx:****

## ****Nginx was made, to solve the c10k problem in apache.****

**C10k:** its a name given to the issue of optimizing the web server software to handle large number of requsts at one time. In the range of 10000 requests at a time, hence the name

Nginx is known for its speed in serving static pages, much faster than apache and keeping the machine resources very low.

**Fundamentally both apache and nginx differs a lot.**

Apache works in a multi process/multi threaded architecture, While nginx is an event driven single threaded architecture.(i will come back to event driven later). The main difference this even driven architecture makes is that, a very small number of nginx worker process can serve a very very large number of requests.

Sometimes nginx is also deployed as a front end server, serving static content requests faster to the clients, and apache in behind.

Each worker process handles requests with the help of the event driven model. Nginx does this with the help of a special functionality in linux kernel called as epoll and select poll. Apache when even run by its threaded model utilizes considerably much more system resource than nginx.

### Why does nginx run more efficiently than apache?

In apache when a request is being served, either a thread or a process is created which serves the request. Now if one requst needs some data from the database,and files from disk, etc the process waits for that.

So some processes in apache just sits and wait for certain task to complete(eating system resources).

Suppose a client with a slow internet connection connects to a web server running apache, the apache server retrieves the data from the disk, to serve the client. Now even after serving the client that process will wait until a confirmation is received from that cliet(which will waste that much process resource)

Nginx avoids the idea of child processes. All requests are handled by a single thread. And this single thread will handle everything, with the help of something called as event loop. So the thread pops up whenever a new connection, or some thing is required(not wasting resources.).

**Step 1**: Gets Request

**Step 2**: Request Triggers events inside the process

**Step 3:** Process manages all these events and returns the output(and simultaniously handles other events for other requests)

Nginx also supports major functionalists which apache supports like the following.

* SSL/TLS
* Virtual Hosts
* Reverse Proxy
* Load Balence
* Compression
* URL rewrite

# Nginx Vs Apache: Nginx Basic Architecture and Scalability

The need for serving large number of concurrent requests is raising every day. The prediction of C10K problem (i.e 10,000 concurrent clients) started the research on web server architecture which could solve this problem. As a result Nginx architecture was developed.

This article explains on a very high-level how Nginx works to solve the scalability problem, along with high level differences between Nginx and Apache webserver.

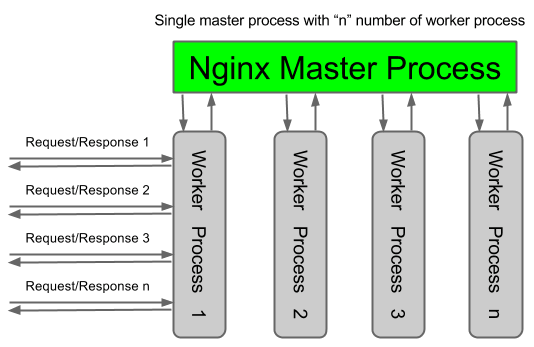
### Increasing Web Server Scalability

The web server scalability problem can be solved either by increasing the hardware capabilities (i.e memory, CPU, etc ) or by improving the web server architecture. The goal here is to improve the web server architecture to optimize the hardware resources, which will eventually lead to a cost effective architecture.

For solving this problem, In 2002 Igor Sysoev started developing a web server written in C, which had an efficient thread management.

Nginx was quite different than the traditional way in which web servers implement thread-based models for serving the request. In the traditional thread-based models, for each client there is one thread which is completely separate and is dedicated to serve that thread. This might cause I/O blocking problems when process is waiting to get completed to release the resources (memory, CPU) in hold. Also, creating separate processes consumes more resources.

In Nginx, the solution to solve the above problem is to use Event-driven, asynchronous, non-blocking and single threaded architecture as shown in the diagram below.



### How Nginx Works

By Event-driven it means that notifications or signals are used to mark the initiation or completion of a process. Thus, the resources can be used by other process until a process initiation event is triggered and resource can be allocated and released dynamically. This leads to the optimized use of memory and CPU.

By Asynchronous it means that the threads can be executed concurrently with out blocking each other. It enhances the sharing of resources without being dedicated and blocked.

By Single threaded it means that, multiple clients can be handled by a single worker process as the resources are not blocked.

Nginx do not create a new process or thread for a new request. Here the worker process accepts the requests and process thousands of it with the implementation of highly efficient event loops. As shown in the above diagram, Nginx can be configured to have n number of worker process with a single master process over them.

Thus nginx is able to do the same work with less memory as it is utilized in a very optimized way.

If you are new to Nginx, get started by [installing Nginx from source](https://www.thegeekstuff.com/2011/07/install-nginx-from-source/).

### Nginx Vs Apache

* As discussed above Nginx is based on event-driven architecture. Apache is based on process-driven architecture. It is interesting to note that Apache in its earliest release was not having multitasking architecture. Later Apache MPM (multi-processing module) was added to achieve this.
* Nginx doesn’t create a new process for a new request. Apache creates a new process for each request.
* In Nginx, memory consumption is very low for serving static pages. But, Apache’s nature of creating new process for each request increases the memory consumption.
* Several benchmarking results indicates that when compared to Apache, Nginx is extremely fast for serving static pages.
* Nginx development started only in 2002. But Apache initial release was in 1995.
* In complex configurations situation, when compared to Nginx, Apache can be configured easily as it comes with lot of configuration features to cover wide range of requirements.
* When compared to Nginx, Apache has excellent documentation.
* In general, Nginx have less components to add more features. But Apache has tons of features and provides lot more functionality than Nginx.
* Nginx do not support Operating Systems like OpenVMS and IBMi. But Apache supports much wider range of Operating Systems.
* Since Nginx comes only with core features that are required for a web server, it is lightweight when compared to Apache.
* The performance and scalability of Nginx is not completely dependent on hardware resources, whereas the performance and scalability of the Apache is dependent on underlying hardware resources like memory and CPU.

[https://linuxtechlab.com/?s=Nginx**(LinuxTechLab)**](https://linuxtechlab.com/?s=Nginx(LinuxTechLab))

# Simple way to configure Nginx Reverse Proxy

A reverse proxy is a server that takes the requests (http/https) & then transfers or distributes them to backend server. Backend server can be an application server like Tomcat, wildfly or Jenkins etc or it can even be another web server like Apache.

But why do we even need a reverse proxy in front of app or web server at all, we need it cause,

1- It hides point of origin, thus making our backend server more secure & less suseptable to attacks,

2- Since reverse proxy is first point of contact for all requests, it can help encrypt/decrypt the request. This takes the load off from backend server,

3- It can also be used for caching of content, which again reduces the load from other servers,

4- it can also act as a load-balancer.

We have already discussed how we can [**configure Apache Web Server as reverse proxy,**](https://linuxtechlab.com/apache-as-reverse-proxy-centos-rhel/) now let’s talk about how we configure a Nginx reverse proxy.

We will need a backend server, it can be any app server or even a webserver. But remember, if you are using a web server that is also on the same server as nginx reverse proxy, make sure that the other web server is not using same tcp port as nginx reverse proxy i.e. 80 & 443.

For the purpose of this tutorial, I will using a tomcat server hosted at a different server on IP 192.168.1.110 , working at port 8080 (refer to our tutorial here for detailed Apache Tomcat installation). As mentioned above, you can opt for different application server or web server.

**Installation**

**Ubuntu**

Nginx is available with default Ubuntu Repositories. So simple install it using the following command,

**$ sudo apt-get update && sudo apt-get install nginx**

**CentOS/RHEL**

We need to add some repos for installing nginx on CentOS & we have created a detailed [**ARTICLE HERE**](https://linuxtechlab.com/installing-nginx-server-configuring-virtual-hosts/) for nginx installation on CentOS/RHEL.

Now start the services & enable it for boot,

**# systemctl start nginx**

**# systemctl enable nginx**

At this point, we can open the web-browser & enter the server IP of nginx, to see a default webpage & make sure the nginx is working with no issues.

**Configuration**

Now that nginx is installed & working we will move ahead with the Nginx reverse proxy configuration part. But first we will remove the default configuration for the nginx, it can be done with the following command,

**# rm /etc/nginx/conf.d/default.conf**

Alternatively, we can also remove the content inside the above mentioned file & make the configuration for Nginx reverse proxy there, but I prefer to use separate file for each site configured. So let’s create a new conf file for our nginx reverse proxy,

**# vi /etc/nginx/conf.d/test-proxy.conf**

& make the following entries to the file,

**server {**

**listen 80;**

**listen [::]:80;**

**server\_name test-reverse-proxy.com;**

**location / {**

**proxy\_pass ;**

**}**

**}**

Now save file & exit. Here in the configuration, we are telling the about the server\_name & than under ‘location’ section, we are providing the backend server i.e. our Apache tomcat server. Now to implement the changes made, we will restart the nginx service but before that we must check if the configuration made are correct or not,

**# ngnix -t**

or we can also provide the complete path for configuration file,

**# nginx -t -c /etc/nginx/conf.d/test-proxy.conf**

Once the check returns with zero errors, we can restart the nginx service,

**# systemctl restart nginx**

**Note :-** Also make sure that your backend server is working properly before moving onto next step.

**Testing**

Now the next & final step is to check if the nginx reverse proxy is working fine or not. So open a web browser & enter the nginx server address/URL. Now when the page finishes loading, we should be seeing the apache tomcat page & not the default nginx page,which we saw earlier.

# Simple guide to configure Nginx reverse proxy with SSL

A reverse proxy is a server that takes the requests made through web i.e. http & https, then sends them to backend server (or servers). A Backend server can be a single or group of application server like Tomcat, wildfly or Jenkins etc or it can even be another web server like Apache etc.

**– A backend server:** For purpose of this tutorial we are using an tomcat server running on localhost at port 8080. If want to learn how to setup a apache tomcat server.

**– SSL cert :** We would also need an SSL certificate to configure on the server. We can use let’s encrypt certificate, you can get one using the procedure [**mentioned HERE**](https://linuxtechlab.com/complete-guide-to-configure-ssl-on-nginx-with-lets-encrypt-ubuntu-centos-rhel/). But for this tutorial, we will using a self signed certificates, which can be created by running the following command from terminal,

**$ openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout /etc/nginx/certs/cert.key -out /etc/nginx/certs/cert.crt**

Next step on configuring nginx reverse proxy with ssl will be nginx installation,

### **Install Nginx**

### **Ubuntu**

Nginx is available with default Ubuntu Repositories. So simple install it using the following command,

**$ sudo apt-get update && sudo apt-get install nginx**

### **CentOS/RHEL**

We need to add some repos for installing nginx on CentOS & we have created a detailed [**ARTICLE HERE**](https://linuxtechlab.com/installing-nginx-server-configuring-virtual-hosts/) for nginx installation on CentOS/RHEL.

Now start the services & enable it for boot,

**# systemctl start nginx**

**# systemctl enable nginx**

Now to check the nginx installation, we can open web browser & enter the system ip as url to get a default nginx webpage, which confirms that nginx is working fine.

### ****Configuring Nginx reverse proxy with SSL****

Now we have all the things we need to configure nginx reverse proxy with ssl. We need to make configurations in nginx now, we will using the default nginx configuration file i.e. ‘/etc/nginx/conf.d/default.conf’.

Assuming this is the first time we are making any changes to configuration, open the file & delete or comment all the old file content, then make the following entries into the file,

**# vi /etc/nginx/conf.d/default.conf**

**server {**

**listen 80;**

**return 301 https://$host$request\_uri;**

**}**

**server {**

**listen 443;**

**server\_name linuxtechlab.com;**

**ssl\_certificate /etc/nginx/ssl/cert.crt;**

**ssl\_certificate\_key /etc/nginx/ssl/cert.key;**

**ssl on;**

**ssl\_session\_cache builtin:1000 shared:SSL:10m;**

**ssl\_protocols TLSv1 TLSv1.1 TLSv1.2;**

**ssl\_ciphers HIGH:!aNULL:!eNULL:!EXPORT:!CAMELLIA:!DES:!MD5:!PSK:!RC4;**

**ssl\_prefer\_server\_ciphers on;**

**access\_log /var/log/nginx/access.log;**

**location / {**

**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\_set\_header X-Forwarded-Proto $scheme;**

**proxy\_pass http://localhost:8080;**

**proxy\_read\_timeout 90;**

**proxy\_redirect http://localhost:8080 https://linuxtechlab.com;**

**}**

**}**

Once all the changes have been made, save the file & exit. Now before we restart the nginx service to implement the changes made, we will discuss the configuration that we have made , section by section,

**Section 1**

**server {**

**listen 80;**

**return 301 https://$host$request\_uri;**

**}**

here, we have told that we are to listen to any request made to port 80 & then redirect it to https,

**Section 2**

**listen 443;**

**server\_name linuxtechlab.com;**

**ssl\_certificate /etc/nginx/ssl/cert.crt;**

**ssl\_certificate\_key /etc/nginx/ssl/cert.key;**

**ssl on;**

**ssl\_session\_cache builtin:1000 shared:SSL:10m;**

**ssl\_protocols TLSv1 TLSv1.1 TLSv1.2;**

**ssl\_ciphers HIGH:!aNULL:!eNULL:!EXPORT:!CAMELLIA:!DES:!MD5:!PSK:!RC4;**

**ssl\_prefer\_server\_ciphers on;**

Now these are some of the default nginx ssl options that we are using, which tells what kind of protocol version, SSL ciphers to support by nginx web server,

**Section 3**

**location / {**

**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\_set\_header X-Forwarded-Proto $scheme;**

**proxy\_pass http://localhost:8080;**

**proxy\_read\_timeout 90;**

**proxy\_redirect http://localhost:8080 https://linuxtechlab.com;**

**}**

**}**

Now this section tells about proxy & where the incoming requests are sent once they come in. Now that we have discussed all the configurations, we will check & then restart the nginx service,

To check the nginx , run the following command,

**# nginx -t**

Once we have configuration file as OKAY, we will restart the nginx service,

**# systemctl restart nginx**

That’s it, our nginx reverse proxy with ssl is now ready. Now to test the setup, all you have to do is to open web browser & enter the URL. We should now be redirected to the apache tomcat webpage.

# Complete guide to configure SSL on Nginx with Let’s Encrypt (Ubuntu/Centos/RHEL)

Securing your websites with an SSL certificate is now a must for all website admins, else the web browsers will mark the website as unsafe to visit, causing the loss of website traffic. But SSL certificate is not cheap, but there is a way around to get an SSL certificate for free with only downside that we need to renew SSL cert every 90 days but that process can also be automated.

## Pre-Requisites

**–** We will need a registered Domain address,  
**–** We will need a CentOS/RHEL or Ubuntu server with Ngnix installed. Installation steps are mentioned below,

## Ubuntu

Nginx is available with default Ubuntu Repositories. So simple install it using the following command,

**$ sudo apt-get update && sudo apt-get install nginx**

## CentOS/RHEL

We need to add some repos for installing nginx on CentOS & we have created a detailed [ARTICLE HERE for nginx installation](https://linuxtechlab.com/installing-nginx-server-configuring-virtual-hosts/) on CentOS/RHEL.

Now start the services & enable it for boot,

**# systemctl start nginx**

**# systemctl enable nginx**

Once its installed, we can move to next part i.e. installing let’s encrypt & issuing of SSL certificate for website.

## Let’s Encrypt on Ubuntu

Firstly we need to install Certbot on Ubuntu system, but its not available with default Ubuntu repositories. Install the Ubuntu repos with the following command,

**$ sudo apt-get install software-properties-common**  
**$ sudo add-apt-repository universe**  
**$ sudo add-apt-repository ppa:certbot/certbot**  
**$ sudo apt-get update**

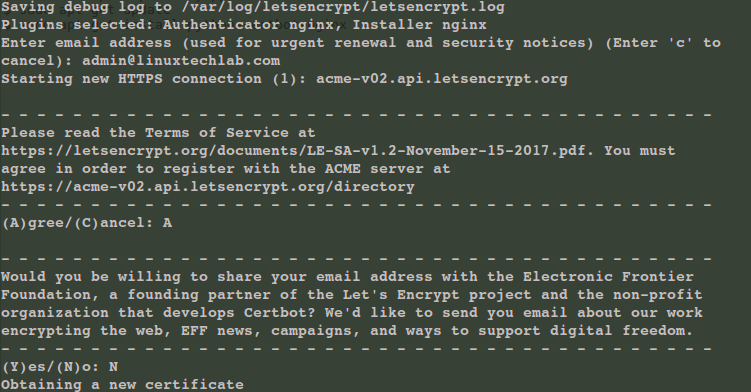
Now to install the Certbot , execute the following command from terminal,

**$ sudo apt-get install python-certbot-nginx**

Now, we will issue a new SSL certificate using certbot command. Use the following command as reference ,

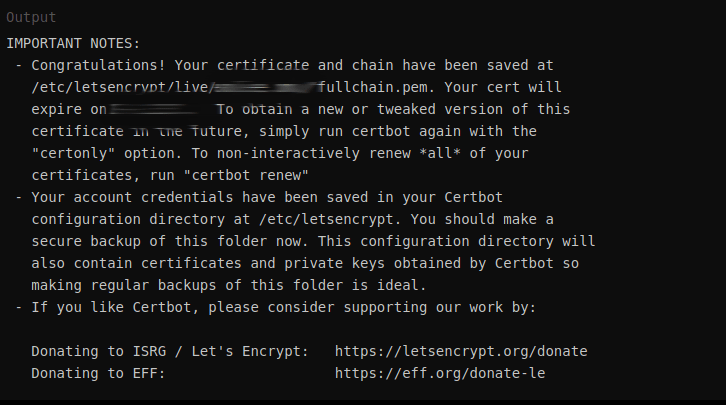
**$ sudo certbot –nginx -d**[**linuxtechlab.com**](https://linuxtechlab.com/)**-d**[**www.linuxtechlab.com**](https://linuxtechlab.com/)

Here linuxtechlab.com is the name of the website for which the certificate will be issues, replace this with the name of your website. If this is the first time you are using Certbot, you will be asked to enter an Email address & also to agree to User Agreement,



Now certbot will then check with let’s encrypt server to verify that you are the web admin of the domain that you are trying to get an SSL for (usually you need to place two files with random text provided by let’s encrypt at location http://domain-name/.well-known/acme-challenge.

Once site ownership has been confirmed, we will be asked to configure redirect settings for Nginx, you can choose 1 (No-Redirect) or 2 (Redirect). If you choose 1, than you will have to configure redirect yourselves in Nginx configuration afterwards, with option 2 , the configuration will be updated & Nginx will be reloaded to implement the new changes made.



Our cert is now installed & new configurations also have been loaded. As mentioned above, we need to renew the cert every 90 days, for that we can create a new cronjob, mentioned at the end of this tutorial.

Now let’s discuss the SSL issue procedure for CentOS & RHEL,

## Let’s Encrypt on CentOS/RHEL

To install Certbot on CentOS, we will need to first install EPEL repositories first on our system. Install EPEL using following command on your system,

**RHEL/CentOS 7**

# rpm -Uvh https://dl.fedoraproject.org/pub/epel/7/x86\_64/Packages/e/epel-release-7-11.noarch.rpm

RHEL/CentOS 6 (64 Bit)

# rpm -Uvh http://download.fedoraproject.org/pub/epel/6/x86\_64/epel-release-6-8.noarch.rpm

**RHEL/CentOS 6 (32 Bit)**

**# rpm -Uvh http://dl.fedoraproject.org/pub/epel/6/i386/epel-release-6-8.noarch.rpm**

Now we to install certbot use the following command,

# yum install certbot-nginx

Once the certbot has been installed, we can then issue the SSL certificate with the same command as mentioned above,

**# certbot –nginx -d linuxtechlab.com -d www.linuxtechlab.com**

Now the process will same as has been mentioned above for Ubuntu. Now after the cert has been installed, we need to make sure that the certificate is renewed before 90 days.

**Automatic Certificate Renewal**

Following cron job will take care of the automatic certificate renewal,

**# crontab -e**

05 01 30 \* \* /usr/bin/certbot renew –quiet

this job will renew certificate every 30 days at 1:05 AM. We can also run the following command to dry-run or test the renewal of certificate,

**# certbot renew –dry-run**

# Easy guide to install NGINX server & configuring Virtual hosts (Server Blocks)

Nginx is a great alternative to apache & has been gaining popularity in recent times. Though Nginx has been known to being a web server with less modules/add-ons & more complex settings than apache but if you are hosting a single website that you want to work at fast speed, than Nginx is right fit for you.

Nginx is an open source has been known to address some of performance related issues that apache has. It uses an event driven design, as opposed to process driven design used by apache which causes apache to slow down under heavy loads as it needs to create new processes which causes increased memory consumption. Worker processes can be configured for Nginx (each worker process is equal to one cpu) & each worker process can handle thousand concurrent connections.

In this tutorial, we will learn to install Nginx server, but before we move onto install Nginx server, we need to perform the following steps.

**1- Remove Apache**

If you are using an old installation on apache server, then remove it as both nginx & apache uses same ports i.e. 80 & 443. To remove apache

|  |
| --- |
| **$ yum remove httpd** |

2- Entry for our virtual host in /etc/hosts

We need to create an entry for out virtual host (newhost.com) in /etc/hosts,

|  |
| --- |
| **$ vi /etc/hosts** **192.168.0.100                     newhost.com** |

3- Epel or Nginx repositories enabled

Since Nginx is available with default RHEL/CentOS repositories, we either need to enable official Nginx repository or Epel repository

Nginx Repository

To enable Nginx repository, create a file named ‘nginx.repo’ in ‘/etc/yum.repos.d’ & add the following repo information

|  |
| --- |
| **$ vi /etc/yum.repos.d/nginx.repo**  **[nginx] name=nginx repo baseurl=http://nginx.org/packages/OS/OSRELEASE/$basearch/ gpgcheck=0 enabled=1** |

Replace “OS” with “rhel” or “centos”, depending on the distribution used, and “OSRELEASE” with “5”, “6”, or “7”, for 5.x, 6.x, or 7.x versions, respectively.

**Epel Repository**

**RHEL/CentOS 7**

|  |
| --- |
| **$ rpm -Uvh https://dl.fedoraproject.org/pub/epel/7/x86\_64/Packages/e/epel-release-7-11.noarch.rpm** |

**RHEL/CentOS 6 (64 Bit)**

|  |
| --- |
| **$ rpm -Uvh http://download.fedoraproject.org/pub/epel/6/x86\_64/epel-release-6-8.noarch.rpm** |

**RHEL/CentOS 6 (32 Bit)**

|  |
| --- |
| **$ rpm -Uvh http://dl.fedoraproject.org/pub/epel/6/i386/epel-release-6-8.noarch.rpm** |

Install Nginx server

Once the Nginx/Epel repository has been enabled, installation can be easily done with yum. To install nginx server, run

|  |
| --- |
| **$ yum install nginx** |

Now start the nginx service & also enable it after reboot,

|  |
| --- |
| **$ systemctl start nginx** **$ systemctl enable nginx** |

We can also test the installation by opening web browser & entering IP address/FQDN of our server

|  |
| --- |
| **http://IPAddress or FQDN/** |

& a test page will open.

**Configuring Virtual hosts (aka Server Blocks)**

Like apache, document root directory for nginx is also ‘/var/www/html’ & it works fine if hosting a single wesbite (virtual host) but when deploying multiple websites, it is advisable to create our custom document directory. So, we will first create a document directory for our website ‘newhost.com’

|  |
| --- |
| **$ mkdir -p /var/www/newhost/html** |

**& provide necessary permissions i.e. 755 to all directories created in /var/www**

|  |
| --- |
| **$ chmod 755 /var/www** |

Next, create an index.html file to greet website users,

|  |
| --- |
| **$ cd /var/www/newhost/html $ echo “Welcome to newhost.com” > index.html** |

Last thing now to configure newhost.com is to create an entry for the website/virtual host in configuration file i.e. ‘/etc/nginx/conf.d/virtual.conf’ . Open the file & create an entry for our host,

|  |
| --- |
| **$  vi /etc/nginx/conf.d/virtual.conf**  server { listen       80; #    listen       \*:80; server\_name  newhost.com;  location / { root   /var/www/newhost.com/html/; index  index.html index.htm; } } |

Save file & exit it. Lastly restart nginx services to implement the changes made to configuration file.

|  |
| --- |
| **$ systemctl restart nginx** |

Now open web browser & enter ‘newhost.com’ in the address bar. This will open our website newhost.com.

Note – If we need to create more than one virtual host, repeat the same process. Create a document root directory, place website files in it, create an entry for it in virtual.conf & restart services.

# How to use Apache reverse proxy as Load Balancer

Reverse proxy is a kind of a proxy that takes http or https request & transfers/distributes them to one or more backend servers. Reverse proxy is useful in many ways, like

**–** It can hide the origin serve, thus making it more secure & immune to attacks,

**–** It can act as a load balancer,

**–** Reverse proxy can also be used to encrypting/decrypting webserver traffic, thus taking some load off from the backend servers.

**–** It can also be used for caching static as well as dynamic contents, which also reduces load off the web servers.

We have covered up the redirection part in our earlier tutorial, in this tutorial we are going the discuss how we can use apache reverse proxy as load balancer.

In this tutorial, we will be using three instances of Apache tomcat server & will than use apache reverse proxy as load balancer to distribute & redirect the requests to these three tomcat servers.

**Pre-requisites**

**–** Install Apache on the server meant to be used as reverse proxy with the following command,

**$ sudo yum install httpd**

**For detailed installation of Apache webserver, refer to our article**[‘Step by Step guide to configure APACHE server ‘](https://linuxtechlab.com/beginner-guide-configure-apache/)

–**On the three backend servers, install Apache tomcat. Read our detailed tutorial on**[how to install Apache tomcat](https://linuxtechlab.com/complete-guide-apache-tomcat-installation-linux/)**to setup the backend servers.**

**Apache modules required**

1- mod\_proxy**– it is the main module responsible for redirecting the connections,**

2- mod\_proxy\_http**– add the support for proxying HTTP connections,**

3- mod\_proxy\_balancer**and**mod\_lbmethod\_byrequests**– Both these modules are required if you are planning to use reverse proxy as load balance as well.**

**Check if the following modules are installed & working with the following command,**

**$ httpd -M**

This command will generate the list of modules that are currently working . If these modules are not among the list, than we need to enable them by making the following entry in httpd.conf,

**$ sudo vim /etc/httpd/conf/httpd.conf**

**LoadModule proxy\_module modules/mod\_proxy.so**

**LoadModule proxy\_http\_module modules/mod\_proxy\_h**

**LoadModule lbmethod\_byrequests\_module modules/mod\_lbmethod\_byrequests.so**

**LoadModule proxy\_balancer\_module modules/mod\_proxy\_balancer.so**

Now save the file & exit, than restart the apache service to implement the changes made,

**$ sudo systemctl restart httpd**

**Configuring Backend servers**

Once we have apache tomcat installed, we don’t need to make any changes to the app server. We will just need the URL for the application hosted on all the tomcat servers, i.e.

**http://192.168.1.110:8080/test/**

**http://192.168.1.120:8080/test/**

**http://192.168.1.130:8080/test/**

Note:- For testing purposes, we can host three different pages on all the tomcat instances so as to identify which webpage is coming from which server.

**Configuring the reverse proxy**

To configure the apache reverse proxy as load balancer, we need to add some configurations to main apache configuration file ,’ httpd.conf ‘,

**$ sudo vim /etc/httpd/conf/httpd.conf**

**<VirtualHost \*:80>**

**<Proxy balancer://cluster>**

**BalancerMember http://192.168.1.110:8080/test**

**BalancerMember http://192.168.1.120:8080/test**

**BalancerMember http://192.168.1.130:8080/test**

**</Proxy>**

**ProxyPreserveHost On**

**ProxyPass / balancer://cluster/**

**ProxyPassReverse / balancer://cluster/**

**</VirtualHost>**

here ‘<Proxy balancer://cluster>’ is the part where we mention the all the tomcat instances & ‘ProxyPass’ is the part which handles the redirection. After making changes to file, save it & restart the apache service to implement the changes.

**$ sudo systemctl restart httpd**

**Checking the load balancing**

To check the reverse proxy & load balancing, open the web browser & enter the following URL,

**http://192.168.1.100**

Where, 192.168.1.100 is the IP address of the reverse proxy server. Now open the URL from two other machines or browsers, all the three opened webpages should be coming from the different application servers. We can make sure by checking the server/access logs of tomcat servers or we can also host 3 different pages on the application servers as mentioned above.

# How to use Apache as Reverse Proxy on CentOS & RHEL

Reverse proxy is a kind of proxy server that takes http or https request & transfers/distributes them to one or more backend servers. Reverse proxy is useful in many ways, like

– It can hide the origin serve, thus making it more secure & immune to attacks,

– It can act as a load balancer,

– Reverse proxy can also be used to encrypting/decrypting webserver traffic, thus taking some load off from the backend servers.

– It can also be used for caching static as well as dynamic contents, which also reduces load off the web servers.

In this tutorial, we are going to discuss how we can use Apache as reverse proxy server on CentOS/RHEL machines. So let’s start with the per-requisites needed for creating apache as reverse proxy,

**Pre-requisites**

– We will be using both apache as reverse proxy as well as backend server, though we can also use some other application or webserver like wildfly or nginx as backend servers . But for the purpose of this tutorial, we will be using apache server only.

So we need to have Apache server installed on both the servers. Install apache with the following command,

**$ sudo yum install httpd**

**Modules needed for using Apache as reverse proxy**

After the apache has been installed on the machine, we need to make sure that following modules are installed & activated on the apache machine, that will be used as reverse proxy,

**1- mod\_proxy –** it is the main module responsible for redirecting the connections,

**2- mod\_proxy\_http –** add the support for proxying HTTP connections,

Check if the following modules are installed & working with the following command,

**$ httpd -M**

This command will generate the list of modules that are currently working . If these modules are not among the list, than we need to enable them by making the following entry in httpd.conf,

**$ sudo vim /etc/httpd/conf/httpd.conf**

**LoadModule proxy\_module modules/mod\_proxy.so**

**LoadModule proxy\_http\_module modules/mod\_proxy\_http.so**

Now save the file & exit, than restart the apache service to implement the changes made,

**$ sudo systemctl restart httpd**

**Configuring Backend test server**

We have also installed apache on backend server & will now add a simple html page for testing purposes,

**$ sudo vim /var/www/html/index.html**

**<html>**

**<head>**

**<title>Test page for Backend server</title>**

**</head>**

**<body>**

**This is a simple test page hosted on backend server.**

**</body>**

**</html>**

Save the file & exit. Now restart the apache service to implement the changes made. Next test the page from a browser on local or remote system with the following URL,

**http://192.168.1.50**

where, 192.168.1.50 is the IP address of the backend server.

### Configuring simple reverse proxy

After the backend server is ready, next thing to do is to make our front end i.e. reverse proxy ready. To do so, we need to make the following entry in apache configuration file i.e. httpd.conf,

**$ sudo vim /etc/httpd/conf/httpd.conf**

**<VirtualHost \*:80>**

**ProxyPreserveHost On**

**ProxyPass / http://192.168.1.50/**

**ProxyPassReverse / http://192.168.1.50/**

**</VirtualHost>**

here, we are telling with ‘ProxyPass’ parameter that whatever request s received at ‘/’ , redirect it to ‘http://192.168.1.50/’. Now restart the apache services to implement the changes,

**$ sudo systemctl restart httpd**

**Note:-**We can also add port numbers here, like for example we are using this reverse proxy with tomcat as backend server, we can also this frontend server as reverse proxy for apache tomcat with the following entries in httpd.conf,

**<VirtualHost \*:80>**

**ProxyPreserveHost On**

**ProxyPass / http://192.168.1.50:8080/test/**

**ProxyPassReverse / http://192.168.1.50:8080/test/**

**</VirtualHost>**

### Testing the reverse proxy

To test the reverse proxy, open the following URL from a web browser,

**http://192.168.1.100/**

here 192.168.1.100 is the IP address of the reverse proxy server. As soon as the URL loads up, we can than see the page that was hosted on backend server. This shows that our reverse proxy is correctly configured & working.

# Step by Step guide to configure APACHE server

Apache is also has a major advantage, that it can support multiple website hosting on a single server. There are actually two types of hosting :-

**1- IP address based hosting**– For IP based hosting, we need to have a different IP for every website that we are hosting. These IPs are then attached to a single or multiple NICs.

**2- Name based Virtual hosting**– Name based hosting is used to host multiple virtual websites using a single IP address.

In this tutorial, we are going to learn to configure Apache server & host both IP address based websites & name based websites but before we do that, we are going to install apache on our server.

**APACHE Installation**

There are three different ways in which we can install apache on our RHEL/CentOS machines  
**1-** **Using RPM** – We need rpm package for it. Once you obtain the rpm package for apache, we can install it using the following command

$ rpm –ivh httpd-2.4.x-1.i686.rpm

**2- Using YUM**– Yum is the easiest way to install apache on your system. We don’t need to download any package when using yum, just use the following

$ yum install httpd

**3- Using source package**– This process is a bit complex but provide us with various options to modify our apache installation as we are using source packages to compile & install the apache package. Firstly we need to download source package (Download from here), we will then extract it, compile it & then install it,

$ tar–xvf httpd-2.4.6.tar.gz

$ cd httpd-2.2.26

$./configure –prefix=PREFIX (here you will modify the apache installation)

$ make

$ make install

This way we will install apache from source, alternatively you can also build a rpm package from source package

$ rpmbuild -tb httpd-2.4.6.tar.bz

& then install it using the rpm command, as we did above.  
Now we have our apache server ready & will now configure configure to host websites.

**Configure Apache server Name Based hosting**

We will first configure 2 websites (test1.com & test2.com) using the name based hosting. Firstly open the apache configuration file which is **/etc/httpd/conf/httpd.conf** & add the following lines to the bottom of file,

$ vi /etc/httpd/conf/httpd.conf

**<VirtualHost 192.168.0.120:80>**  
**ServerAdmin webmaster@test1.com**  
**DocumentRoot /var/www/html/test1.com**  
**ServerName www.test1.com**  
**ErrorLog logs/www.test1.com-error\_log**  
**CustomLog logs/www.test1.com-access\_log common**  
**</VirtualHost>**

**<VirtualHost \*:80>**  
**ServerAdmin webmaster@test2.com**  
**DocumentRoot /var/www/html/test2.com**  
**ServerName www.test2.com**  
**ErrorLog logs/www.test2.com-error\_log**  
**CustomLog logs/www.test2.com-access\_log common**  
**</VirtualHost>**  
Next, search for the “*NameVirtualHost*” in the same file & uncomment it by removing ‘*#’* from the starting of the line & add the IP of the server, so it should look like

NameVirtualHost 192.168.0.120:80

Now save the file & exit.

We will now create ‘index.html’ file for both the websites but we will firstly create a directory for both the websites,

**$ mkdir /var/www/html/test1.com**

**$ mkdir /var/www/html/test2.com**

Now add some content to index.html to identify both the sites,

**$ vi /var/www/html/test1.com/index.html**

**<html>**

**<head>**

**<title>This is test1.com</title>**

**</head>**

**& do the same for test2.com**

**$ vi /var/www/html/test2.com/index.html**

**<html>**

**<head>**

**<title>This is test2.com</title>**

**</head>**

You can also type ‘this is test1.com or test2.com ‘ without using html tags. We now have our apache server ready with name based hosting. We will now restart our server to implement changes & will then check our sites

**$ systemctl restart httpd.service or**

**$ service httpd restart**

Now, open your browser & enter the website url

**www.test1.com**

**www.test2.com**

You should now see two different webpages with their unique contents.

**IP based hosting**

We need 2 different IPs assigned to our single NIC or 2 different NICs with different IPs to achieve IP based hosting. We are going to use a single NIC card with different IPs assigned to it. if you don’t know how to do it, read our tutorial here.

After we have configured our IP addresses (192.168.0.120 , 192.168.0.125), we will start our configuration by editing ‘/etc/httpd/conf/httpd.conf’

**$ vi /etc/httpd/conf/httpd.conf**

& search for ‘Listen 80’ . Next uncomment it & add IP address, so it should look like

Listen 192.168.0.120:80

& add the following server configurations to the bottom of the same file i.e. httpd.conf

**VirtualHost 192.168.0.120:80>**

**ServerAdmin webmaster@test1.com**

**DocumentRoot /var/www/html/test1**

**ServerName www.test1.com**

**ErrorLog logs/www.test1.com-error\_log**

**TransferLog logs/www.test1.com-access\_log**

**</VirtualHost>**

**<VirtualHost 192.168.0.125:80>**

**ServerAdmin webmaster@test2.com**

**DocumentRoot /var/www/html/test2**

**ServerName www.test2.com**

**ErrorLog logs/www.test2.com-error\_log**

**TransferLog logs/www.test2.com-access\_log**

**</VirtualHost>**

Create index.html for both websites in their respective folders & write content to it , as we did above & lastly restart your apache service for changes to take effect.

Open your browser & access your websites, either using name or IP address.

# Redirect http to https: Apache Server

SSL secured webstites or HTTPS has now become a must to have on website, especially those handling sensitive client information. Having a HTTPS enabled website means that a intruder can’t intrude to communication between users & website.

HTTPS not only secures communication but is now a requirement for many new features like http2, which requires you to have https enabled on your server. Having a HTTPS enabled website also improves your Google SEO (Search Engine Optimization) ranking.

We will be discussing three different methods to redirect all the web traffic i.e. redirect http to https. But before we do that let’s see the files which we will use to make the redirection of http to https,

**/etc/httpd/conf/httpd.conf (RHEL/CentOS)-** If you have access to this & have defined the Apache hosts entries in httpd.conf, than use this file to define the redirection options. There can be another user defined location, for ex /etc/httpd/conf/virtual-hosts.conf. But there will be entry for the same in httpd.conf, like

**include /virtual-hosts.conf**

If you are using a custom location for defining the virtual hosts, than that’s where you will make the redirection entry.

**/etc/apache2/sites-available/test.com (Ubuntu)-** This file is same as custom defined virtual hosts configuration file. If using Ubuntu, than make redirection entry to this file. Here test.com is the name of the website.

**.htaccess-** This file is used when handling a number of servers or when you don’t have access to main configuration file like in case of wordpress installation but you can also use it on any case as well. This file is usually located in the root of the website.

All these files can be used to redirect traffic, so choose the file that’s suitable to your needs.

**Method 1 : Using Redirect**

This method is pretty simple, make the following entry in virtual host config file (httpd.conf for CentOS/RHEL or test.com for Ubuntu),

**<VirtualHost \*:80>**

**ServerName test.com**

**<Location />**

**Redirect permanent / https://test.com/**

**</Location>**

**</VirtualHost>**

**<VirtualHost \*:443>**

**ServerName example.com**

**DocumentRoot /var/www/html/test.com**

**SSLEngine On**

**SSLCertificateFile /etc/apache2/ssl/test.com.crt**

**SSLCertificateKeyFile /etc/apache2/ssl/test.com.key**

**SSLCertificateChainFile /etc/apache2/ssl/test.com.ca-bundle**

**</VirtualHost>**

If using .htaccess file, make the following entry to the file,

**Redirect permanent / https://test.com/**

Save the file & web server traffic redirect http to https would start working.

**Method 2 : Using mod\_rewrite**

This is my personal favorite method. Here we will use mod\_rewrite module to redirect the traffic to https. To use this method, make sure that we have mod\_rewrite module enabled. To enable mod\_rewrite module, open file ‘httpd.conf’ & make the following entry

**For CentOS**

**$ vi /etc/httpd/conf/httpd.conf**

**LoadModule rewrite\_module modules/mod\_rewrite.so**

**For Ubuntu**

**$ sudo a2enmod rewrite**

Restart the Apache service to implement the changes. Next make the following entry in the virtual host file ( httpd.conf for CentOS/RHEL or test.com for Ubuntu )

**<VirtualHost \*:80>**

**ServerName test.com**

**RewriteEngine On**

**RewriteCond %{HTTPS} !on**

**RewriteRule (.\*) https://%{HTTP\_HOST}%{REQUEST\_URI}**

**</VirtualHost>**

**<VirtualHost \*:443>**

**ServerName example.com**

**DocumentRoot /var/www/html/test.com**

**SSLEngine On**

**SSLCertificateFile /etc/apache2/ssl/test.com.crt**

**SSLCertificateKeyFile /etc/apache2/ssl/test.com.key**

**SSLCertificateChainFile /etc/apache2/ssl/test.com.ca-bundle**

**</VirtualHost>**

For **.htaccess** file make the following entry to the file,

**RewriteEngine On**

**RewriteCond %{HTTPS} !on**

**RewriteRule (.\*) https://%{HTTP\_HOST}%{REQUEST\_URI}**

**Note:-** Make sure that you are not duplicating ‘RewriteEngine On’ & that the ‘RewriteCond’ , ‘RewriteRule’ immediately follow ‘RewriteEngine’.

**Method 3 : Using PHP**

Using this method, we can create a function to redirect traffic & call this function in the page when we need to redirect the traffic or on top of all the scripts. Use the following PHP code,

**< ?php**

**function redirectTohttps() {**

**if($\_SERVER[‘HTTPS’]!=”on”) {**

**$redirect= “https://”.$\_SERVER[‘HTTP\_HOST’].$\_SERVER[‘REQUEST\_URI’];**

**header(“Location:$redirect”); } }**

**?>**

Before you use this code, make sure that SSL has been properly configured.

# Installing mod\_pagespeed for Apache webserver

Pagespeed or mod\_pagespeed is an open source module used in apache & is used to speed up the websites. It applies various layers of filters to all the associated assets like pages, style-sheets, images etc, which than helps in optimizing the assets & this leads to boost in speed of the website. The best part of using pagespeed is that, no changes modifications/changes are required to the current existing content. So changes are done at the server side & users accessing the website are presented with the modified content.

**Pre-requisites**

We need to have Apache web-server installed on our machines. To install apache on CentOS/RHEL, run

**$ yum install httpd**

Also, we need the package ‘at’ for CentOS like os, so install that as well if not already installed,

**$ yum install at**

**For Ubuntu, run**

**$ sudo apt-get install apache2**

We now have our web server ready, you can put some files or a complete website. Once you are ready, we can proceed with the page speed setup,

**Installing Mod\_pageSpeed on CentOS/RHEL/Fedora**

Before installing mod\_pagespeed module, we will download it’s package as per your system architecture,

For 64 Bit OS,  use the following command

**$ wget https://dl-ssl.google.com/dl/linux/direct/mod-pagespeed-stable\_current\_x86\_64.rpm**

For 32 but OS, use

**$ wget https://dl-ssl.google.com/dl/linux/direct/mod-pagespeed-stable\_current\_i386.rpm**

Now install the downloaded package with the command, (64 bit in my case)

**$ rpm -ivh mod-pagespeed-stable\_current\_x86\_64.rpm**

That’s it, PageSpeed module has been installed & it will be activated on its own. We just need to restart the webserver to implement the changes.

**$ service httpd restart**

**Installing mod\_pagespeed on Ubuntu**

Installating mod\_pagespeed module on Ubuntu is similar to above mentioned procedure for CentOS, we will download the packages,

For 64 Bit,  
**$ wget https://dl-ssl.google.com/dl/linux/direct/mod-pagespeed-stable\_current\_amd64.deb**

For 32 Bit,  
**$ wget https://dl-ssl.google.com/dl/linux/direct/mod-pagespeed-stable\_current\_i386.deb**

& install the downloaded package,

**$ dpkg -i mod-pagespeed-stable\_current\_amd64.deb**

**$ sudo apt-get -f install**

Now restart the webserver to implment the changes,

$ sudo service apache2 restart

Configuration

After installing mod\_pagespeed module, we will now configure it our servers. Main configuration file is named ‘pagespeed.conf’ & its location is ‘/etc/httpd/conf.d/pagespeed.conf’ for CentOS/RHEL & ‘/etc/apache2/mods-available’ for Ubuntu.

Open the file & firstly enable the module (its enabled by default),

**$ vi pagespeed.conf**  
**ModPagespeed on**

To turn-off the module, use ‘off’ in place on ‘on’ . You can then also enable core filters to allow pagespeed module to rewrite the files. To enable corefilters create the following entry at the bottom of pagespeed.conf,

**ModPagespeedRewriteLevel CoreFilters**

**ModPagespeedAllow “http://\*test-domain.com/\*.html”**

**ModPagespeedAllow “http://\*test-domain.com/\*.css”**

**ModPagespeedAllow “http://\*test-domain.com/\*.js”**

**ModPagespeedAllow “http://\*test-domain.com/\*.png”**

PageSpeed will now rewrite the files with extensions .html, .css, .js, & .png. You can also disallow filters using “ModPagespeedDisallow” parameter. After making the changes to the file, save file & exit. Then restart your webserver to implement the changes.

**Install Varnish Cache for Apache Web Server to Speed up Website**

Varnish or Varnish cache is an open source http proxy/accelerator that is put in front of webserver, for the purpose of speeding up the webserver (Apache & nginx both), it’s especially meant for webservers that are serving content heavy dynamic websites.

Varnish works by redirecting user requests to static, thus reducing the calls to dynamic pages & thereby reducing load. It visits server once to cache the page & then all the future calls to the page are served by varnish cache.

It makes website really fast. Depending upon the architecture being used, it can make website 3 to 10 times faster. In this tutorial, we are going to discuss  how we can install Varnish Cache for Apache Webserver on RHEL/CentOS to speed up your website .

**Pre-requisites**

– We need to have an apache webserver installed

– We also need to have EPEL repositories installed on our system

To install Varnish package, we need EPEL repository installed on our systems. Install EPEL repositories on the system using the below mentioned command,

**RHEL/CentOS 7**

**$ sudo rpm -Uvh https://dl.fedoraproject.org/pub/epel/7/x86\_64/Packages/e/epel-release-7-11.noarch.rpm**

**RHEL/CentOS 6 (64 Bit)**

**$ sudo rpm -Uvh http://download.fedoraproject.org/pub/epel/6/x86\_64/epel-release-6-8.noarch.rpm**

RHEL/CentOS 6 (32 Bit)

$ sudo rpm -Uvh http://dl.fedoraproject.org/pub/epel/6/i386/epel-release-6-8.noarch.rpm

Installation

We are now ready to install varnish on our machine & with EPEL repositories installed & enabled, it can be done through yum. To install varnish, open your machine terminal & execute the following command,

$ sudo yum install varnish

After installation is complete, we will start the service & enable it at boot,

$ sudo systemctl start varnish

Configuration

We will now configure both Varnish cache & apache server to work with each other. As mentioned above varnish server is placed before apache webserver, so firstly we will configure varnish server to listen to port 80. To do this open the varnish configuration file ‘/etc/varnish/varnish.params’

**$ sudo vim /etc/varnish/varnish.params**

& change the port from 6081 to port 80,

**VARNISH\_LISTEN\_PORT=80**

After making changes, save file & exit. Next, we will set the IP address & port on which the varnish server will look for the server content, to do this we need to edit ‘/etc/varnish/default.vc1’

**Open ‘default.vc1 & make the following entries,**

**$ sudo vim /etc/varnish/default.vc1**

**backend default {**

**.host = “127.0.0.1”;**

**.port = “8080”;**

**}**

We can change the IP address to 127.0.0.1 to IP address of the webserver, if varnish cache & webserver are on separate machines. Also we can make change to port, as per our requirement. Next, restart the varnish service to implement the changes made,

**$ sudo systemctl restart varnish**

Now we will make changes to apache webserver, so that it listens to the port that we have mentioned in on varnish cache in ‘default.vc1’ file. Open ‘httpd.conf’

**$ sudo vim /etc/httpd/httpd.conf**

& change the port number,

**Listen 8080**

Save file, exit & restart the apache service to implement the changes.

**$ sudo systemctl restart httpd**

Our setup is now complete. To test that things are workin as they are supposed to, execute the following command,

**$ curl -I**[**http://localhost**](http://localhost/)

It should return with output with varnish version, something like

**HTTP/1.1 403 Forbidden**

**Date: Mon, 18 Sep 2017 01:28:41 GMT**

**Server: Apache/2.4.27 (CentOS) PHP/5.4.16**

**Last-Modified: Tue, 14 Mar 2017 11:27:18 GMT**

**ETag: “1896-6152e1i536981”**

**Accept-Ranges: bytes**

**Content-Length: 4897**

**Content-Type: text/html; charset=UTF-8**

**X-Varnish: 13**

**Age: 0**

**Via: 1.1 varnish-v4**

**Connection: keep-alive**

This shows varnish is working as it’s supposed to & serving the httpd pages.

# Easy way to integrate Apache with modsecurity on Ubuntu

ModSecurity WAF (Web Application Firewall ) is an open source software firewall that is used to safeguard applications from attacks & unauthorised access. ModSecurity WAF can be used as a module with already installed Apache web server or also with Nginx server or IIS. It continuously checks the incoming traffic & performs real time analysis .

The best feature that makes ModSecurity wonderful is that it supports flexible rule engine using which we can safeguard our applications from a number of threats like XSS attacks, SQL injection attacks, Local file include, remote file include attacks etc.

**Pre-requisites**

Before we can intergrate Apache with modsecurity, we must have LAMP server installed on our systems. To install LAMP on Ubuntu system, please refer to the [detailed article HERE](https://linuxtechlab.com/install-lamp-stack-on-ubuntu/). Once we have installed it, we can now proceed to use apache with modsecurity.

Also we will need git installed on our system, install git with the following command,

**$ sudo apt-get install git**

**Installing ModSecurity**

To install ModSecurity, open the terminal & execute the following command,

**$ sudo apt-get install libapache2-modsecurity**

Once the installation has been completed, we can also check whether the module has been loaded or not using the following command,

**$ apachectl -M | grep security**

& we should get the following output,

security2\_module (shared)

**Configuring ModSecurity**

Next step will be configure modsecurity, so that we can use apache with modsecurity module. When we install modsecurity, a sample configuration file for apache is also installed & we can use this file to configure WAF. First create a copy of the file,

**$ cp /etc/modsecurity/modsecurity.conf-recommended /etc/modsecurity/modsecurity.conf**

Now edit the file & make the following changes to the file,

**$ sudo nano /etc/modsecurity/modsecurity.conf**

Change the following in the file,

SecRuleEngine On                              # it should be “SecRuleEngine DetectionOnly ” by default

Once done, save the file & exit. Next, restart the apache service to implement the changes,

**$ sudo systemctl restart apache2**

**Configuring rules for ModSecurity**

Until now, we have integrated Apache with modsecurity. Now we will install CoreRule Set aka CRS to be able to fully utilize modsecurity. CRS is basically a collection of rules which tells how a web server should behave under certain conditions.

When ModSecurity is installed, a default set of CRS is also installed under /usr/share/modsecurity-crs. We should remove this folder & install CRS using the above method.

**$ rm -rf /usr/share/modsecurity-crs**

Next download the latest CRS with the following command,

**$ git clone https://github.com/SpiderLabs/owasp-modsecurity-crs.git /usr/share/modsecurity-crs**

**$ cd /usr/share/modsecurity-crs**

Now to enable the rules in apache configuration, so open the following file,

**$ sudo nano /etc/apache2/mods-enabled/security2.conf**

& add the following lines in the files (or modify if already there)

**<IfModule security2\_module>**

**SecDataDir /var/cache/modsecurity**

**IncludeOptional /etc/modsecurity/\*.conf**

**IncludeOptional “/usr/share/modsecurity-crs/\*.conf**

**IncludeOptional “/usr/share/modsecurity-crs/rules/\*.conf**

**</IfModule>**

Save the file & exit, than restart the apache service to implement the changes,

**$ sudo systemctl restart apache2**

That’s it, we have successfully integrated apache with modsecurity & we can now check the modsecurity logs to make sure that everything is fine,

**$ tail -f /var/log/apache2/modsec\_audit.log**

# Block Web traffic in Apache server using .htaccess

.htaccess file is a very important & useful file used to alter the configuration of Apache web server. .htaccess files can be used to change the configuration of the Apache Web Server to enable/disable additional functionality and features.

There are two ways by which we can block web traffic in apache server from a country, either we can use the IP addresses to block web traffic or we can use the country codes to block traffic. We will now discuss the both ways,

**Block web traffic in Apache based on IP address**

Using this way, we can also block traffic from any source & not only from a particular country. For this method to work, open the .htaccess file (usually located in the website document directory)

**$ sudo vim /var/www/html/linuxtechlab/.htaccess**

& make the following entries into the file,

**Order allow,deny**

**Allow from all**

**Deny from 150.20.0.0/12**

**Deny from 191.16.0.0/16**

**Deny from 12.16.1.0/8**

Here the IP address mentioned are from the country for which you need to block the web traffic. But this method has a downside when using it to block traffic from a country as we need to have all the IP addresses for that particular country. But we can generate the list of IP address used by a country

**Block web traffic in Apache based on Country code**

This method is much more easy method to block the web traffic originating from a particular country as we only need the country code & not the list of IP addresses to block the traffic.

We can get the country code list from the [following website](http://www.asiteaboutnothing.net/c_country-codes.html). Once you the country make the following entry to the .htaccess file,

**$ sudo vim /var/www/html/linuxtechlab/.htaccess**

**SetEnvIf CF-IPCountry DE BuzzOff=1**

**Order allow,deny**

**Allow from all**

**Deny from env=BuzzOff**

Here, we have used the country code ‘DE’ which is the country code for Germany. Similarly we can block the traffic from more than once country,

**SetEnvIf CF-IPCountry DE BuzzOff=1**

**SetEnvIf CF-IPCountry CN BuzzOff=1**

**SetEnvIf CF-IPCountry IN BuzzOff=1**

**SetEnvIf CF-IPCountry FR BuzzOff=1**

**SetEnvIf CF-IPCountry GR BuzzOff=1**

**Order allow,deny**

**Allow from all**

**Deny from env=BuzzOff**

# To create SSL Certificate for Apache Server

# SSL certificate

**SSL** or **Secure Socket Layer** is web protocol, which is used to encrypt our web-server traffic. Data transfer during website access is in plain text which can easily be hacked. This might not seem like a threat when we are reading tutorials on [Linuxtechlab.com](https://linuxtechlab.com/) but it will be a major issue when you are shopping at amazon & someone hacks your credit card info. So in order to prevent any eavesdropping/hacking , we secure our websites with a SSL certificate. If our website has a SSL cert & someone hacks the data, it will be of no use to him since it will be encrypted.

If you have a website that is accessed globally, we need to [**get an SSL certificate from a Global certificate authorities (CAs)**](https://www.cheapsslshop.com/) as self-signed certificates, which we will be creating, are not identified by web-browsers. Self-signed certificates are signed by the same person as the person creating & are good for internal & testing purposes.

Now let’s create SSL Certificate (self-signed certificate),

# Pre-requisite

Firstly, we will need a working apache web-server with a website of at least a single page hosted,

Secondly, we will need the ‘mod\_ssl’ & ‘openssl’ installed on our web-server. We can install them by using YUM

**$ yum install mod\_ssl openssl**

**Create SSL Certificate**

We will now generate the certificate using the following steps, but first let’s create a folder where we will be doing our certificate generation

**$ mkdir /etc/httpd/ssl**

**$ cd /etc/httpd/ssl**

Now, we can create a self-signed key and certificate pair with OpenSSL in a single command by typing

**$ openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout /etc/httpd/ssl/apache.key -out /etc/httpd/ssl/apache.crt**

here, openssl is the command for creating and managing ssl,

**req –x509** is public key infrastructure for ssl,

-nodes, means we don’t need a passphrase,

**-days 365** is the validity of the certificate,

-newkey rsa:2048 means cert will 2048 bit long,

-keyout, means where to place Private key,

-out means where to place our certificate.

Once you run this command you will asked to provide some information , provide the information & your certificate will be created.

ssl

Adding Certificate to Web Server

After generating the certificate, we need to add in to our apache server. Open ‘/etc/httpd/conf.d/ssl.conf’ & we will make some changes to it

$ vi/etc/httpd/conf.d/ssl.conf

Now search for the line with ‘VirtualHost \_default\_:443’ & change the server name to one you used as common name on your ssl certificate (test1.com), so it look like

**<VirtualHost \_default\_:443>**

**. . .**

**DocumentRoot “/var/www/html”**

**ServerNamewww.test.com:443**

Next we will add the path to our certificate & Private Key,

**SSLEngine on**

**SSLCertificateFile /etc/httpd/ssl/apache.crt**

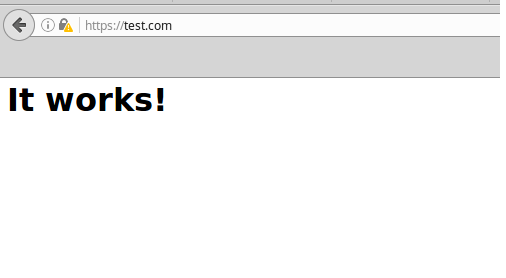
**SSLCertificateKeyFile /etc/httpd/ssl/apache.key**

After making these changes, save & exit the file. Restart your apache service

**$ systemctl restart httpd**

& now open your web browser & try accessing your website using https,

<https://www.test.com>



# 12 Important things to do for securing APACHE Web Server

Apache is used for web hosting for millions of websites (I am not exaggerating)  & is target of millions of hacks taking place on daily basis. So being sysadmins, we must know how we can secure our apache servers as securing our precious data is of utmost importance in today’s world.

Listed below are some points that we can use to secure our web-servers;-

### Disable unnecessary modules

This is one of the most common cause for web attacks. By default, apache enables lots of modules but we usually don’t need them. We must disable all these unnecessary modules to make our server less susceptible to threats.

Some of the modules that are not usually needed are **mod\_imap, mod\_include, mod\_info, mod\_userdir, mod\_autoindex** etc. If you are not sure which  modules are of use to you, refer to [Apace Module Documentation](http://httpd.apache.org/docs/2.4/mod/) & remove them during installation if you are using source files or if you have a working server, you  can run the following command

**$ grep LoadModule /etc/httpd/conf/httpd.conf**

& just put**‘#’** (comment it) in front of the unnecessary modules. Restart apache service to implement changes.

### Disable directory listing

In the absence of index file, apache lists all the files & directory which is again a serious security threat as it can enable access to places we don’t want anyone to enter. So we must disable directory listing, it can be done by making the following entry in **‘/etc/httpd/conf/httpd.conf’**file,

**<Directory /var/www/html>**

**Options -Indexes**

**</Directory>**

### Hide apache identity i.e. version & OS identity

By default, apache shows its version, OS & php versions. This makes an attacker task much less easier since he has the version & he can devise an attack plan based on vulnerabilities of these version. To disable this, we need to make changes to ‘ServerSignature’ & ‘ ServerTokens’parameters in ‘httpd.conf’,

**ServerSignature Off**

**ServerTokens Prod**

Enable mod\_security & mod\_evasive

Both these modules, mod\_security & mod\_evasive, are very good modules when it comes to securing our apache servers. **mod\_security** works as a firewall for our web applications and allows us to monitor traffic on a real time basis. It also helps us to protect our websites or web server from brute force attacks.

**mod\_evasive** enables it to handle the HTTP brute force and Dos or DDos attack. It takes one request to process and processes it. It prevents DDOS attacks from doing as much damage.

We can install *mod\_security* by using yum

**$ yum install mod\_security**

& *mod\_evasive* can be installed from source.

### Limit large requests

By default, apache puts no restriction on request size, which an attacker can use to put your website under DOS (Denial of service ) attack. So we must limit the size for our website directory. We can set the value in bytes from 0 (unlimited) to 2147483647(2GB).

An example of doing so is mentioned below

**<Directory “/var/www/test1.com/upload”>**

**LimitRequestBody 204800**

**</Directory>**

Here, we restricted users to upload files of size more than 2 Mb to ‘*/var/www/test1.com/upload*’

Run apache as separate user & group

Apache usually runs with users ‘nobody’ or ‘daemon’ but it’s a good practise to run with its own user,

**$ groupadd apache**

**$ useradd -d /var/www/ -g apache -s /bin/nologin apache**

& edit ‘httpd.conf’ to reflect new user & group. Open file & search for ‘User’ & ‘Group’ & change them

**$ vi /etc/httpd/conf/httpd.conf**

**User apache**

**Group apache**

### Block unwanted services

Certain services such as CGI execution and symbolic links are usually not required. So we must disable them. Open ‘httpd.conf’ & add the following lines

**<Directory /var/www/test1.com>**

**Options -ExecCGI -FollowSymLinks -Includes**

**</Directory>**

### Restrict access to root directory

We must also secure our root directory. Open ‘httpd.conf’ & add the following lines

**<Directory />**

**Options None**

**Order deny,allow**

**Deny from all**

**</Directory>**

### Don’t allow access to .htaccess

.htaccess files is used to modify the behaviour of our site, using it we can  customize error pages, password protect our site, deny access based on IP, change index.html page, redirect to another page etc.

.htaccess file can be used to overwrite the default apache directives. So we should not allow users to access .htaccess & override directive. We do this by adding following lines in our ‘htpd.conf’ file

**<Directory />**

**Options None**

**AllowOverride None**

**Order allow,deny**

**Allow from all**

**</Directory>**

### Enable logging

Apache logging provides detailed information about client requests made on our web server, so logging must be enabled as it will help in investigating an issue. Logging in apache is achieved by mod\_log\_config module.

To enable website-wise logging, we must provide ‘ErrorLog’ & ‘CustomLog’ directive for the site while creating an entry in ‘httpd.conf’.

**<VirtualHost \*:80>**

**DocumentRoot /var/www/html/test1.com/**

**ServerName www.test1.com**

**ServerAlias test1.com**

**ErrorLog /var/log/httpd/test1.com\_error\_log**

**CustomLog /var/log/httpd/test1.com\_access\_log combined**

**</VirtualHost>**

### Update apache on regular basis

Apache continuously works on resolving any bugs or security vulnerabilities & keep updating  apache to address these issue, so we must keep our apache updated to latest version to make our server more secure. You can update your apache using yum

**$ yum update httpd**

### Secure apache with SSL certificates

Securing web-server with an SSL certificate is necessary when we are dealing with sensitive information on our website like account information etc. SSL certificate encrypts the data & even if data is hacked, it will be of no use to hacker as it will be encrypted. You can refer to our tutorial for [**creating a Self signed SSL certificate**](https://linuxtechlab.com/create-ssl-certificate-apache-server/).

# Complete guide for Apache TOMCAT installation on Linux

Apache Tomcat, also known as Tomcat server is an open source java servlet container & provides a pure JAVA http webserver environment to run java code. It is very famous & widely used to run java based websites around the world. Apache Tomcat software powers numerous large-scale, mission-critical web applications across a diverse range of industries and organisations .

Apache tomcat is availble for various operating systems like Linux, Unix, Windows etc. Apache Tomcat has recently announced the End Of Life for Tomcat 8.0.x versions, though this will not affect the 8.5.x versions.

**Pre-requisites**

– To install Tomcat 8.5, we will require JAVA 7 or later version. So firstly we need to install and setup JAVA on our system. You can install JAVA with the following commands,

**CentOS/RHEL**  
**$ sudo yum install java-1.8.0**

**Ubuntu/Debian**

**$ sudo apt-get install openjdk-8\***

**Apache Tomcat installation**

For Apache Tomcat installation, we will download the archived package from the official website with the following command,

$ wget [http://redrockdigimark.com/apachemirror/tomcat/tomcat-8/v8.5.31/bin/apache-tomcat-8.5.31.tar.gz](http://redrockdigimark.com/apachemirror/tomcat/tomcat-8/v8.5.37/bin/apache-tomcat-8.5.37.tar.gz)

Once the file has finished downloading, extract the package in /opt directory (we can also use some other directory like /data etc). Run the following command in terminal to extract the file,

**$ tar -xvzf pache-tomcat-8.5.23.tar.gz -C /opt**

**$ mv /opt/apache-tomcat-8.5.23/ /opt/tomcat**

Apache tomcat is now ready to be started but before we do that we need to assign credentials to access ‘Manager’ & ‘GUI’ page of tomcat, as by default no user name and pasword is setup. To asisgn the credentials, we will use the **‘/opt/tomcat/conf/tomcat-users.xml ‘** file,

**$ vim /opt/tomcat/conf/tomcat-users.xml**

& make the following entries to the file,

**<role rolename=”manager-gui” />**

**<user username=”manager” password=”Password@123″ roles=”manager-gui” />**

**<role rolename=”admin-gui” />**

**<user username=”admin” password=”Password@123″ roles=”admin-gui” />**

Make sure that you make these entries before the tag ‘tomcat-users’, i.e. make above entries before the below mentioned lines,

**<tomcat-users xmlns=”http://tomcat.apache.org/xml”**

**xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”**

**xsi:schemaLocation=”http://tomcat.apache.org/xml tomcat-users.xsd”**

**version=”1.0″>**

Once done, save the file & exit.

We need to make another change, though this is completely optional & completely depends on how you will access the manager page.We can only access the manager page from the browser on localhost but it we to access it on some other remote machines, we need to modify the ‘context.html’ for manager to comment the lines which disables the remote login,

**$ vi /opt/tomcat/webapps/manager/META-INF/context.xml**

**<Context antiResourceLocking=”false” privileged=”true” >**

**<!– <Valve className=”org.apache.catalina.valves.RemoteAddrValve”**

**allow=”127\.\d+\.\d+\.\d+|::1|0:0:0:0:0:0:0:1″ /> –>**

**</Context>**

Same is to be for host-manager’s context file as well,

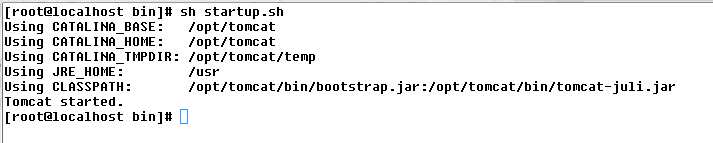
**$ vi /opt/tomcat/webapps/host-manager/META-INF/context.xml**

**<Context antiResourceLocking=”false” privileged=”true” >**  
**<!– <Valve className=”org.apache.catalina.valves.RemoteAddrValve”  
allow=”127\.\d+\.\d+\.\d+|::1|0:0:0:0:0:0:0:1″ /> –>**  
**</Context>**

We are now ready to start the tomcat server, goto /opt/tomcat/bin folder & run the script named ‘startup.sh’,

**$ cd /opt/tomcat/bin**

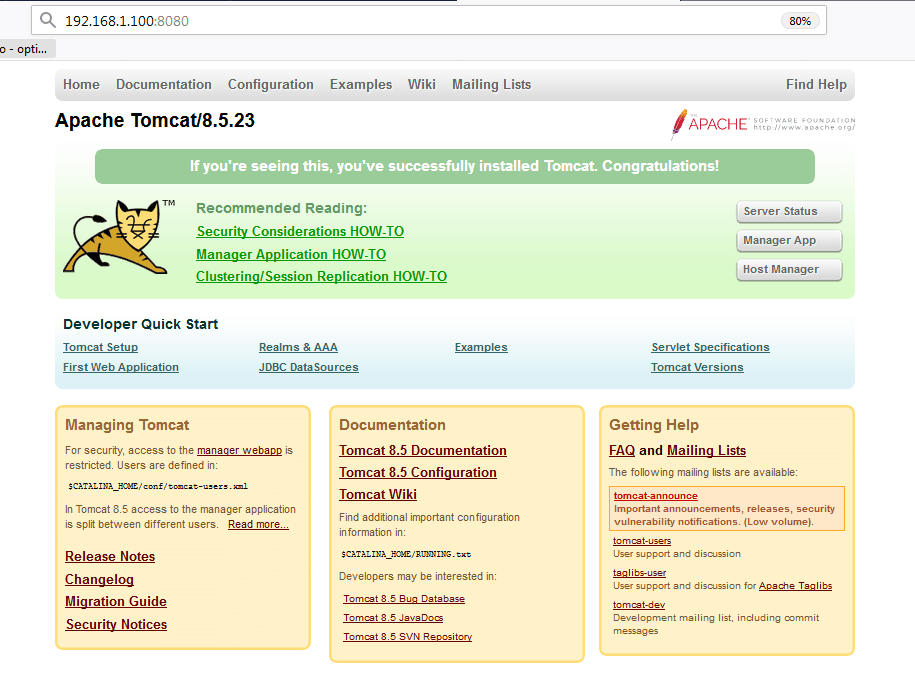
**$ sh startup.sh**



Apache tomcat will start with the following output,

Now open a web browser of your choice & use the following URL to access the Apache tomcat UI,

**http://192.168.1.100:8080**



here, 192.168.1.100 is the IP address to apache tomcat server. We will now be directed to following page,

From this page only, we can access ‘Manager’ & ‘Host Manager’ pages by clicking on the respective tabs. Upon clicking any of the tabs, we would be asked to authenticate, in order to access the page. We need to enter the credentials which we have earlier setup to access the page,

# nginx: See Active connections / Connections Per Seconds

nginx server has a module called HttpStubStatusModule. This module provides the ability to get some status from nginx. You will get the following information:

1. Number of all open connections.
2. Stats about accepted connections.
3. Connections per second and so on.

**Configuration**

Edit nginx.conf file:  
# vi nginx.conf  
Add or append the following in context location:

|  |
| --- |
| location /nginx\_status {  # Turn on stats  stub\_status on;  access\_log off;  # only allow access from 192.168.1.5 #  allow 192.168.1.5;  deny all;  } |

Save and close the file. Reload nginx server:  
# service nginx reload  
OR  
# nginx -s reload

**Test it**

Open a web-browser and type the following url:  
http://your-domain-name-here/nginx\_status  
OR  
<http://ip.address.here/nginx_status>

**Nginx Performance Tuning**

Nginx is one of the best web servers which unleash the high performance even with minimal resources.

Usually, all web servers are performing mainly based on the server configuration.

Not all the times are same for a website.

When the traffic to the website increases and goes beyond the server capacity, then your server will be down.

There are some certain ways which we can use to optimize the web server so that it will give a higher performance to the existing resources.

Here I am going to show you,

* Worker Processes
* Worker connection
* Buffers
* Timeouts
* Gzip compression
* Static File caching
* Logging

**Prerequisites:**

* Here we are going to use Debian 7 droplet.
* The server should have Nginx installed

**Worker Processes and Worker connection**

There are two main directives that we are going to discuss here.

They are worker processes and worker connection. The worker processes work as the backbone of the server.

This directive let the virtual server know to allocate the workers once it is connected to the IP address and port.

The default worker processes allocation is 1 worker processes per core.

However, you can use more than one worker processes.

To know how much worker processes you need, you have to check your server configuration.

Based on the number of cores you can set the worker\_processes.

If you want to know how many processes your server has, just execute the following command.

**$ grep processor /proc/cpuinfo | wc -l**

If it returns value 2, then it means your server has two cores.

The worker\_connections tells the worker\_processes about how much simultaneous visitors the Nginx server can serve.

The default simultaneous connection is 768.

Since each browser opens two connections with the server, the exact count will be half of the default connection.

So, we have to adjust the worker connections to unleash the full potential.

To check the core limitation use the following command.

**ulimit -n**

If your server has single core, you will see the following output

**1024**

We have to update the configuration.

**sudo nano /etc/nginx/nginx.conf**

**worker\_processes 1;**

**worker\_connections 1024;**

The number of clients can be multiplied by the number cores. For this configuration, the server can serve 1024 clients/second.

**Buffers:**

We have to make some changes in the buffer.

if the buffer size is low, the server has to do read and write at hard disk.

Here are the some of the directives which we have to know before making any changes.

client\_body\_buffer\_size: This directive handles the client buffer size.

client\_header\_buffer\_size: This directive handles the client header size. The 1K is sufficient for the header.

client\_max\_body\_size: If the client request is too large to handle the Nginx will return the 413 error or Request Entity too large.

large\_client\_header\_buffers: this directive contains the maximum buffer size and number of buffers for large client headers.

**client\_body\_buffer\_size 10K;**

**client\_header\_buffer\_size 1k;**

**client\_max\_body\_size 8m;**

**large\_client\_header\_buffers 2 1k;**

**Timeout**:

This directive also improves the performance.

The client\_body\_timeout and client\_header\_timeout directories are making the server to wait for some particular time to receive the client header and client body to be received.

If the time exceeds, the Nginx will respond, "Request Timeout or 408 error".

The keepalive\_timeout: This directive makes the connection to last up to the mentioned time. After that time, the server will disconnect the client.

send\_timeout: If the send timeout expires, the server will disconnect the connection to the client.

**client\_body\_timeout 12;**

**client\_header\_timeout 12;**

**keepalive\_timeout 15;**

**send\_timeout 10;**

**Gzip Compression**

Gzip helps the Nginx server reduce its network transfer amount. You should not set the gzip\_comp\_level too high as it will waste the CPU cycles.

**gzip on;**

**gzip\_comp\_level 2;**

**gzip\_min\_length 1000;**

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

**gzip\_types text/plain application/x-javascript text/xml text/css application/xml;**

Static File Caching:

You can set the expire header for web files which are not changed often and served frequently. This will let the Nginx load the required content only.

**location ~\* .(jpg|jpeg|png|gif|ico|css|js)$ {**

**expires 365d;**

**}**

You can add and remove the files in the above lines which the Nginx server will serve.

**Logging:**

The Nginx will record each visit to your server. If you are using any other tool to watch the traffic, you may want to turn off this future.

For that, you have to edit the access\_log directive.

**access\_log off;**

Now, save and close the file. After that run

**sudo service nginx restart**

Conclusion

If you are a system admin, make sure you are tunning the Nginx for maximum performance.

These settings are not constant. According to the requirement, you have to tweak the settings to get more performance.

### Worker Processes

NGINX can run multiple worker processes, each capable of processing a large number of simultaneous connections. You can control the number of worker processes and how they handle connections with the following directives:

* [**worker\_processes**](https://nginx.org/en/docs/ngx_core_module.html#worker_processes) – The number of NGINX worker processes (the default is 1). In most cases, running one worker process per CPU core works well, and we recommend setting this directive to auto to achieve that. There are times when you may want to increase this number, such as when the worker processes have to do a lot of disk I/O.
* [**worker\_connections**](https://nginx.org/en/docs/ngx_core_module.html#worker_connections) – The maximum number of connections that each worker process can handle simultaneously. The default is 512, but most systems have enough resources to support a larger number. The appropriate setting depends on the size of the server and the nature of the traffic, and can be discovered through testing.

### 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](https://nginx.org/en/docs/http/ngx_http_core_module.html#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](https://nginx.org/en/docs/http/ngx_http_core_module.html#keepalive_timeout) – How long an idle keepalive connection remains open.

The following directive relates to upstream keepalives:

* [keepalive](https://nginx.org/en/docs/http/ngx_http_upstream_module.html#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](https://nginx.org/en/docs/http/ngx_http_proxy_module.html#proxy_http_version) 1.1;

[proxy\_set\_header](https://nginx.org/en/docs/http/ngx_http_proxy_module.html#proxy_set_header) Connection "";

**Nginx Worker Tuning**

On high traffic servers running Nginx, you will want to ensure the proper number of processes are available for the requests. This can be fine tuned in the nginx.conf configuration file via both worker\_processes and worker\_connections.

**# grep worker /etc/nginx/nginx.conf**

**worker\_processes 1;**

**worker\_connections 1024;**

**worker\_processes:**

The number of NGINX worker processes (the default is 1). In most cases, running one worker process per CPU core works well. There are times when you may want to increase this number, such as when the worker processes have to do a lot of disk I/O.

**worker\_connections:**

The maximum number of connections that each worker process can handle simultaneously. The default is 512, but most systems have enough resources to support a larger number. The appropriate setting depends on the size of the server and the nature of the traffic, and can be discovered through testing.

**Maximum Connections = worker\_processes \* worker\_connections.**

Both of these values multiplied result in the maximum number of concurrent connections. Its important to keep in mind that each connection will require a file handler, and two if the connection is proxied, so you will need to verify the open file limit. You will find errors in the Nginx log if you hit the open file limit.

The Nginx configuration option worker\_rlimit\_nofile sets the value for the maximum file descriptors that can be opened by a single worker process. If you don't set the worker\_rlimit\_nofile directive, then the OS settings will determine how many FDs can be used by Nginx. One method of changing the FD limit for Nginx is to change the values in the OS.

## Worker Modifications

The easiest thing to set in your configuration is the right number of workers and connections

### Worker Processes

In /etc/nginx/nginx.conf, set worker\_processes 1; if you have a lower traffic site where nginx, a database, and a web application all run on the same server.

If you have a higher traffic site or a dedicated instance for nginx, set one worker per CPU core: worker\_processes auto;

If you’d like to set this manually, you can utilize grep ^processor /proc/cpuinfo | wc -l to find the number of processes that the server can handle and adjust accordingly.

### Worker Connections

The option worker\_connections sets the maximum number of connections that can be processed at one time by **each**worker process. By default, the worker connection limit is 512, but many systems can handle more.

The appropriate sizing can be discovered through testing, as it is variable based on the type of traffic nginx is handling. The system’s core limitations can also be find through using ulimit:

|  |
| --- |
| **ulimit** –n |

It will output a number as a result: 65536

You can also set use epoll, a scalable I/O event notification mechanism to trigger on events and make sure that I/O is utilized to the best of its ability.

Lastly, you can utilize multi\_accept in order for a worker to accept all new connections at one time.

|  |  |
| --- | --- |
| 1  2  3  4  5 | events {  worker\_connections 4096;  use epoll;  multi\_accept on;  } |

# Maximum open file descriptors per process. Should be > (worker\_connections \* worker\_processes)

worker\_rlimit\_nofile 32768;

## HTTP and TCP Optimizations

### Keep Alive

Keep alive allows for fewer reconnections from the browser. Remember to place these settings inside the **http** {} directive.

* 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.

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

### 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. Remember to place these settings inside the **http** {} directive.

* 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.

|  |  |
| --- | --- |
| 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 2 32k;  postpone\_output 1460; |

### Timeouts

Timeouts can also drastically improve performance. Remember to place these settings inside the http {} directive.

* 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.

|  |  |
| --- | --- |
| 1  2  3  4 | reset\_timedout\_connection on;  client\_header\_timeout 60s;  client\_body\_timeout 15s;  send\_timeout 10s; |

### 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. Remember to place these settings inside the **http** {} directive.

#also see: https://easyengine.io/tutorials/nginx/open-file-cache/

|  |  |
| --- | --- |
| 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; |

You can also cache via a particular location. Caching files for a long time is beneficial, especially if the files have a version control system delivered by the build process or CMS. This should be placed in the virtual host configuration or main configuration (e.g: /etc/nginx/sites-available/default)

|  |  |
| --- | --- |
| 1  2  3 | location ~\* .(woff|eot|ttf|svg|mp4|webm|jpg|jpeg|png|gif|ico|css|js)$ {  expires 365d;  } |

### Gzipping Content

For content that is plain text, nginx can use gzip compression to serve back these assets compressed to the client. Modern web browsers will accept gzip compression and this will shave bytes off of each request that comes in for plain text assets. The list below is a “safe” list of compressible content types; however, you only want to enable the content types that you are utilizing within your web application. Remember to place these settings inside the http {} directive.

gzip on;

gzip\_min\_length 1000;

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;

# [Different Types of Load Balancing through Nginx](https://www.oodlestechnologies.com/blogs/Different-Types-of-Load-Balancing-through-Nginx)

**Intro:** When we use multiple servers in the production environment then load balance between these servers must require. Nginx can handle 10K concurrent connections and proxy pass to the backend server.Nginx provides the different ways to use load balance:

1.Round Robin load balancing

2.Least Connection load balancing

3.Session Persistence load balancing

4.Weighted load balancing

**A)Round Robin Load Balancing:** This default load balancing in provide by Nginx.In this requests are provided by one by one to each server.  
**Example:**

http {  
    upstream backendserver {  
        server http://server1.com;  
        server http://server2.com;  
        server http://server3.com;  
    }

    server {  
        listen 80;

        location / {  
            proxy\_pass http://backendserver;  
        }  
    }  
}

**B)Least Connection load balancing:** In this technique, nginx check the load of the server and send the request to a less busy server.   
**Ex:**  
    upstream backendserver {  
        least\_conn;  
        server http://server1.com;  
        server http://server2.com;  
        server http://server3.com;  
    }

**C)Session Persistence load balancing:**In round-robin, each request equally distributed to each server but in session persistence, the each request of same client bound with the same server except when server unavailable.

**Ex:**  
    upstream backendserver {  
        ip\_hash;  
        server http://server1.com;  
        server http://server2.com;  
        server http://server3.com;  
    }

**D)Weighted load balancing:**In this technique, we provide the weight to the server in round robin manner.In the example, server3 handle the twice request from server2.

**Ex:** upstream backendserver {  
        server http://server1.com weight=3;  
        server http://server2.com weight=2;  
        server http://server3.com weight=1;  
    }

There are three algo used by nginx to distribute the load:-

* Round-Robin.
* Least Connection.
* Hashing

Round Robin distribute the load in sequential manner.First request goes to first server and second goes to second server and so on.

**Least Connection** - in this nginx distribute the load to the backend sever with the least active connection.

**Hash method** - This uses a key to determine how to map the request with one of the upstream servers. Generally, this is set to the client’s IP address, which allows you to map the requests to the same upstream server each time.

Configuring The LB.

Round Robin based load balancing

**create an upstream**

**upstream backend {**

**server 127.0.0.1:8080;**

**server 127.0.0.1:8081;**

**server 127.0.0.1:8082;**

**}**

Define a server block

**server {**

**listen 80;**

**server\_name example.com;**

**access\_log /var/log/nginx/access.log combined;**

**location / {**

**proxy\_pass http://backend;**

**}**

**}**

In this scenario, request to example.com will distribute between the backend server deines in upstream block.

First request to example.com will proxy pass to backend app with port 8080, likewise second request to backend app with port 8081 and third request to backend app with port 8082 and so on.

Also, it’s possible to weight the servers, meaning it will preference upstream servers with a higher weight. If your servers aren’t exactly the same, you can use weighting to preference your higher capacity systems so that they receive more request.

**upstream backend {**

**server 127.0.0.1:8080 weight=2;**

**server 127.0.0.1:8081;**

**server 127.0.0.1:8082;**

**}**

Because we set the first server with a weighted value of 2, it will receive twice as many requests as the others.

Least connected load balancing

The upstream block will almost same as of Round Robin but there will be an extra key word i.e least\_conn;.

**upstream backend {**

**least\_conn;**

**server 127.0.0.1:8080;**

**server 127.0.0.1:8081;**

**server 127.0.0.1:8082;**

**}**

As each new request comes in, NGINX determines which upstream server has the least amount of connections and directs requests to this server.

**Hash-based load balancing**

The upstream block will almost same as of Round Robin but we need to explicitly tell NGINX to use the hash method.

**upstream backend {**

**hash $remote\_addr consistent;**

**server 127.0.0.1:8080;**

**server 127.0.0.1:8081;**

**server 127.0.0.1:8082;**

**}**

For this hash method, we used the client IP ($remote\_addr) as the determining factor to build up the hash map.

The consistent parameter at the end of the hash line implements the Ketama consistent hashing method, which helps to minimize the amount or remapping (and therefore potential disruption or cache loss) if you need to add or remove servers from your upstream block directive. If your upstream servers remain constant, then you can omit this parameter.

For those who have used older versions of NGINX, the ip\_hash method is still available, but with one distinct difference.

upstream backend {

ip\_hash;

server 127.0.0.1:8080;

server 127.0.0.1:8081;

server 127.0.0.1:8082;

}

While this method still works, if you need better consistency for ip\_hash mapping, then using hash $remote\_addr will match the full IP address.

# [Run multiple Sites in single url on same port with NGINX](https://www.oodlestechnologies.com/blogs/Run-multiple-Sites-in-single-url-on-same-port-with-NGINX)

Consider that in your project you have multiple module such as admin panel , buyer front end , seller front end and you need to run these module in single url like localhost:8080/admin for admin ,localhost:8080/ for buyer and localhost:8080/seller for seller.The NGINX configuration that can be used is as follows:

server {

listen 8080;

server\_name localhost;

root /home/ajit/git/univisior;

location / {

alias /home/ajit/git/project-x/buyer/dist/;

index index.html;

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

}

location /admin{

alias /home/ajit/git/project-x/admin/dist/;

index index.html;

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

}

location /seller {

alias /home/ajit/git/project-x/seller/dist/;

index index.html;

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

}

location /api {

proxy\_pass http://localhost:3000/api;

}

}

**Load balancing methods in Nginx**

Load balancing with nginx uses a round-robin algorithm by default if no other method is defined, like in the first example above. With round-robin scheme each server is selected in turns according to the order you set them in the load-balancer.conf file. This balances the number of requests equally for short operations.

Least connections based load balancing is another straightforward method. As the name suggests, this method directs the requests to the server with the least active connections at that time. It works more fairly than round-robin would with applications where requests might sometimes take longer to complete.

To enable least connections balancing method, add the parameter least\_conn to your upstream section as shown in the example below.

**upstream backend {**

**least\_conn;**

**server 10.1.0.101;**

**server 10.1.0.102;**

**server 10.1.0.103;**

**}**

While round-robin and least connections balancing schemes are fair and have their uses, they, however, cannot provide session persistence. If your web application requires that the users are subsequently directed to the same back-end server as during their previous connection, you should use IP hashing method instead. IP hashing uses the visitors IP address as a key to determine which host should be selected to service the request. This allows the visitors to be each time directed to the same server, granted that the server is available and the visitor’s IP address hasn’t changed.

To use this method, add the *ip\_hash* -parameter to your *upstream* segment like in the example underneath.

**upstream backend {**

**ip\_hash;**

**server 10.1.0.101;**

**server 10.1.0.102;**

**server 10.1.0.103;**

**}**

In a server setup where the available resources between different hosts are not equal, it might be desirable to favour some servers over others. Defining server weights allows you to further fine-tune load balancing with nginx. The server with the highest weight in the load balancer is selected the most often.

**upstream backend {**

**server 10.1.0.101 weight=4;**

**server 10.1.0.102 weight=2;**

**server 10.1.0.103;**

**}**

or example in the configuration shown above the first server is selected twice as often as the second, which again gets twice the requests compared to the third.

## Load balancing with HTTPS enabled

Enabling HTTPS for your site is a great way to protect your visitors and their data. If you haven’t yet implemented encryption on your web hosts, we highly recommend taking a look at our guide for [how to install Let’s Encrypt on nginx](https://upcloud.com/community/tutorials/install-lets-encrypt-nginx/).

Using encryption with a load balancer is easier than you might think. All you need to do is add another server section to your load balancer configuration file which listens to HTTPS traffic at port 443 with SSL and set up a proxy\_pass to your upstream segment like with the HTTP in the previous example above.

Open your configuration file again for edit.

**sudo nano /etc/nginx/conf.d/load-balancer.conf**

Then add the following server segment to the end of the file

**server {**

**listen 443 ssl;**

**server\_name domain\_name;**

**ssl\_certificate /etc/letsencrypt/live/domain\_name/cert.pem;**

**ssl\_certificate\_key /etc/letsencrypt/live/domain\_name/privkey.pem;**

**location / {**

**proxy\_pass http://backend;**

**}**

**}**

Then save the file, exit the editor and restart nginx again with

**sudo service nginx restart**

Setting up encryption at your load balancer while using the private network connections to your back-end has some great advantages.

As only your UpCloud servers have access to your private network, it allows you to terminate the SSL at the load balancer and thus only passing forward HTTP connections.

It also greatly simplifies your certificate management as you can obtain and renew the certificates from a single host.

With the HTTPS-enabled you also have the option to enforce encryption to all connections to your load balancer. Simply update your server segment listening to port 80 with a server name and a redirection to your HTTPS port, then remove or comment out the location portion as it’s no longer needed. See the example below.

**server {**

**listen 80;**

**server\_name domain\_name;**

**return 301 https://$server\_name$request\_uri;**

**#location / {**

**# proxy\_pass http://backend;**

**#}**

**}**

**Save the file again after making the changes and then restart nginx.**

**sudo service nginx restart**

Now all connections to your load balancer will be served over encrypted HTTPS connection and requests to the unencrypted HTTP will be redirected to use HTTPS as well. This provides a seamless transition into encryption with nothing required from your visitors.

**Health checks**

In order to know which servers are available nginx’s implementations of reverse proxy includes passive server health checks. If a server fails to respond to a request or replies with an error, nginx will note the server has failed and will try to avoid forwarding connections to that server for a time.

The number of consecutive unsuccessful connection attempts within a certain time period can be defined in the load balancer configuration file by setting a parameter max\_fails to the server lines. By default, when no max\_fails is specified, this value is set to 1. Optionally setting the max\_fails to 0 will disable health checks to that server.

If *max\_fails* is set to a value greater than 1 the subsequent fails must happen within a specific time frame for the fails to count. This time frame is specified by a parameter *fail\_timeout*, which also defines how long the server should be considered failed. By default the *fail\_timeout* is set to 10 seconds.

After a server is marked failed and the time set by *fail\_timeout* has passed, nginx will begin to gracefully probe the server with client requests. If the probes return successful, the server is again marked live and included in the load balancing as normal.

**upstream backend {**

**server 10.1.0.101 weight=5;**

**server 10.1.0.102 max\_fails=3 fail\_timeout=30s;**

**server 10.1.0.103;**

**}**

Using the health checks allows you to adapt your server back-end to the current demand by powering up or down hosts as required. Starting up additional servers during high traffic can easily increase your application performance when new resources become automatically available to your load balancer.