**Nginx Reverse Proxy Setup for Apache web server**

Nginx also pronounced “Engine-X” is a free, open-source HTTP Web server and one of the best alternative to Apache http server.

It is a high-performance edge web server with the lowest memory footprint possible and optimizes CPU usage while delivering maximum performance even on a very cheap server hardware.

Nginx also provides a combination of

Nginx web servers

Reverse proxy and

Load balancing solution

**What Is a Reverse Proxy?**

A reverse proxy is an intermediary proxy service which takes a client request, passes it on to one or more servers, and subsequently delivers the server’s response to the client. A common reverse proxy configuring is to put **Nginx in front of an Apache web server**. Using this method will allow both web servers to work together enabling each to do what they do best. Learn more about the differences between [Nginx vs Apache](https://www.keycdn.com/support/nginx-vs-apache/).

This post will outline the benefits of using an Nginx reverse proxy as well as how to configure one.

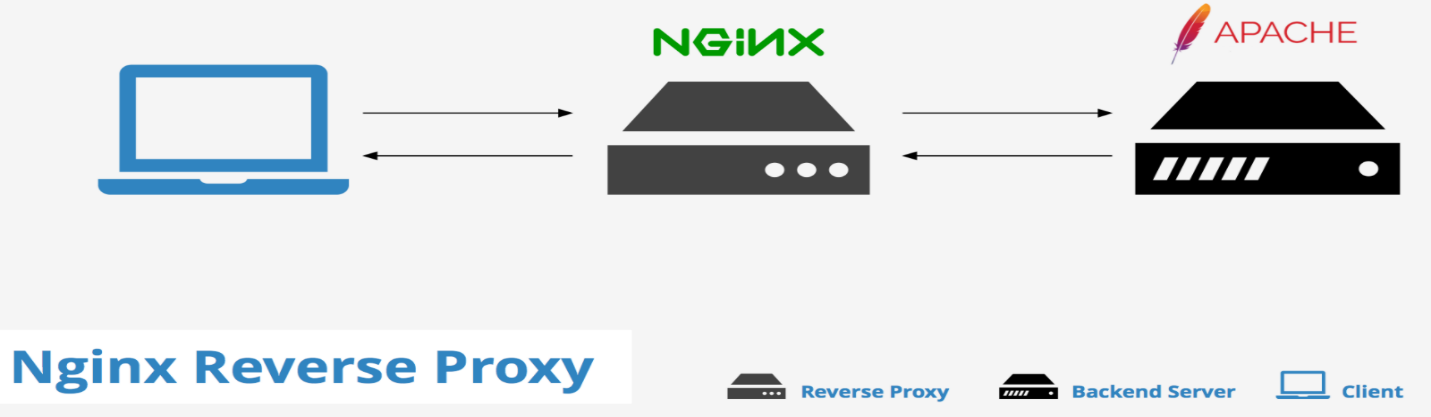
## **Benefits of an Nginx Reverse Proxy**

There are a few benefits to setting up an Nginx reverse proxy. Although not required in all cases, it can be beneficial depending upon your particular scenario / setup. The following outlines a few benefits implementing a reverse proxy.

* **Load Balancing** – A reverse proxy can perform load balancing which helps distribute client requests evenly across backend servers. This process greatly helps in avoiding the scenario where a particular server becomes overloaded due to a sudden spike in requests. Load balancing also improves redundancy as if one server goes down, the reverse proxy will simply reroute requests to a different server. Read our complete article to learn more about [load balancing](https://www.keycdn.com/support/load-balancing/).
* **Increased Security** – A reverse proxy also acts as a line of defense for your backend servers. Configuring a reverse proxy ensures that the identity of your backend servers remains unknown. This can greatly help in protecting your servers from attacks such as [DDoS](https://www.keycdn.com/support/ddos-attack/) for example.
* **Better Performance** – Nginx has been known to perform better in delivering static content over Apache. Therefore with an Nginx reverse proxy, all client requests can be handled by Nginx while all requests for dynamic content can be passed on to the backend Apache server. This helps improve performance by **optimizing the delivery of assets based on their type**. Additionally, reverse proxies can also be used to serve cached content and perform SSL encryption to take a load off the web server(s).
* **Easy Logging and Auditing** – Since there is only one single point of access when a reverse proxy is implemented, this makes logging and auditing much simpler. Using this method, you can easily monitor what goes in and out through the reverse proxy.

## **What is Nginx Reverse Proxy ?**

When [Nginx](https://www.nginx.com/resources/admin-guide/reverse-proxy/) reverse proxy received request, it sends a request to the specified proxied server. In this case the specified proxied server is Apache web server. When Nginx reverse proxy fetches the response from Apache web server, It will sends it back to the client. In other words, Nginx reverse proxy serve as front-end server for Apache web service.



**Nginx Reverse Proxy Server : 192.168.1.54  
Apache Web server : 192.168.1.55**

1. Prepared yum repository for nginx : (OR) sudo yum -y update install epel-release

# vi /etc/yum.repos.d/nginx.repo

[nginx]

name=nginx repo

baseurl=http://nginx.org/packages/centos/$releasever/$basearch/

gpgcheck=0

enabled=1

2. Perform yum install for nginx :

# yum install nginx -y

3. Backup original nginx config file :

# cp /etc/nginx/nginx.conf /etc/nginx/nginx.conf.bak

4. Modify default nginx config file :

Disable the Nginx as a webserver in configuration file by commenting the following lines.

# server {

# listen 80 default\_server;

# listen [::]:80 default\_server;

# server\_name \_;

# root /usr/share/nginx/html;

# Load configuration files for the default server block.

# include /etc/nginx/default.d/\*.conf;

# location / {

# }

# error\_page 404 /404.html;

# location = /40x.html {

# }

# error\_page 500 502 503 504 /50x.html;

# location = /50x.html {

# }

# }

Then add the following configuration to enable Nginx as a reverse proxy.

server {

listen 80;

server\_name node2.example.com;

access\_log off;

error\_log off;

location / {

proxy\_pass http://192.168.5.238/;

proxy\_redirect off;

proxy\_set\_header Host $host;

proxy\_set\_header X-Real-IP $remote\_addr;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

proxy\_max\_temp\_file\_size 0;

client\_max\_body\_size 10m;

client\_body\_buffer\_size 128k;

proxy\_connect\_timeout 90;

proxy\_send\_timeout 90;

proxy\_read\_timeout 90;

proxy\_buffer\_size 4k;

proxy\_buffers 4 32k;

proxy\_busy\_buffers\_size 64k;

proxy\_temp\_file\_write\_size 64k;

}

}

**NOTE:** Mention the IP address and Hostname for the backend webserver

# vi /etc/nginx/nginx.conf

user nginx;

worker\_processes 1;

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

pid /var/run/nginx.pid;

events {

worker\_connections 1024;

}

http {

include /etc/nginx/mime.types;

default\_type application/octet-stream;

log\_format main '$remote\_addr - $remote\_user [$time\_local] "$request" '

'$status $body\_bytes\_sent "$http\_referer" '

'"$http\_user\_agent" "$http\_x\_forwarded\_for"';

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

charset utf-8;

keepalive\_timeout 65;

server\_tokens off;

sendfile on;

tcp\_nopush on;

tcp\_nodelay off;

server {

listen 80;

server\_name \_;

root /usr/share/nginx/html;

index index.html index.htm; }

include conf.d/\*.conf;

}

5. Add the host configuration file under /etc/nginx/conf.d/. Nginx config file will load all \*.conf files under conf.d folder :

As example, the website domain is ehowstuff.local. So virtual server ehowstuff.local will be created and named as ehowstuff.local.conf :

# vi /etc/nginx/conf.d/ehowstuff.local.conf

server {

listen 80;

server\_name ehowstuff.local www.ehowstuff.local;

access\_log off;

error\_log off;

location / {

proxy\_pass http://192.168.1.55:80/;

proxy\_set\_header X-Real-IP $remote\_addr;

proxy\_set\_header Host $host;

proxy\_redirect off;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

proxy\_connect\_timeout 90;

proxy\_send\_timeout 90;

proxy\_read\_timeout 90;

client\_max\_body\_size 10m;

client\_body\_buffer\_size 128k;

proxy\_buffer\_size 4k;

proxy\_buffers 4 32k;

proxy\_busy\_buffers\_size 64k;

}

}

On above configuration file, all the traffic to ehowstuff.local port 80 will be redirected to the Apache web server that hosted at 192.168.1.55. Nginx reverse proxy serve as front-end server for Apache web service.

6. Verify the configuration file :

# /usr/sbin/nginx -t

nginx: the configuration file /etc/nginx/nginx.conf syntax is ok

nginx: configuration file /etc/nginx/nginx.conf test is successful

7. Start nginx server :

# service nginx start

sudo systemctl enable nginx

8. Configure nginx start at boot :

# chkconfig nginx on

just a reminder that these commands leave the nginx server http/https ports open to the internet.

sudo firewall-cmd --permanent --zone=public --add-service=http

sudo firewall-cmd --permanent --zone=public --add-service=https

Run this command for your changes to take affect. That’s it!

sudo firewall-cmd --reload

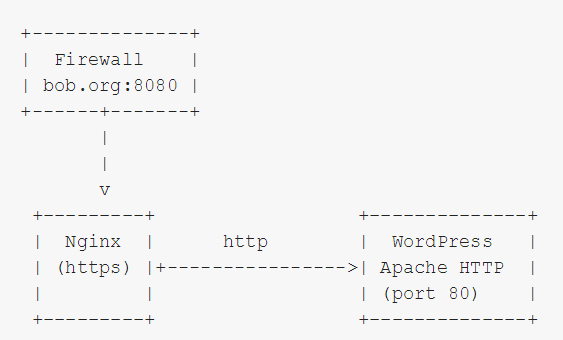
|  |
| --- |
| apt-get install nginx  Once Nginx has been installed, the next step is to disable the default virtual host.  unlink /etc/nginx/sites-enabled/default  Then, we need to create a file within the /etc/nginx/sites-available directory that contains the reverse proxy information. We can name this reverse-proxy.conf for example.  server {  listen 80;  location / {  proxy\_pass http://192.x.x.2;  }  }  The above snippet is quite minimalistic and straightforward. The important part here is the proxy\_pass directive which is essentially telling any requests coming through the Nginx reverse proxy to be passed along to the Apache remote socket 192.x.x.2:80. There are many other proxy configurations you can define within your Nginx configuration settings. To learn more about the directives available, check out the [Nginx proxy module documentation](http://nginx.org/en/docs/http/ngx_http_proxy_module.html).  Once you’ve added the appropriate directives to your .conf file, activate it by linking to /sites-enabled/ using the following command.  ln -s /etc/nginx/sites-available/reverse-proxy.conf /etc/nginx/sites-enabled/reverse-proxy.conf  Lastly, run an Nginx configuration test and restart Nginx.  service nginx configtest  service nginx restart |

[**Nginx https reverse proxy to WordPress with Apache, http and different port**](https://cmanios.wordpress.com/2014/04/12/nginx-https-reverse-proxy-to-wordpress-with-apache-http-and-different-port/)

Today I had to hide a [WordPress 3.8.1](http://wordpress.org/latest.zip) blog behind an Nginx reverse proxy configured to use only **https**. Nginx was behind an external firewall which forwarded <https://bob.org:8080/blog> to Nginx using https (port 4443). The difficulty was that whe WordPress blog was installed in an Apache HTTP server in port 80 and it worked using **http**!. Lets say that my site was [https://bob.org](https://bob.org/) and I had to put the blog in <https://bob.org:8080/blog>. Thus my list of burdens was the following:

1. Configure Nginx to use only https and redirect http to https.
2. Reverse proxy in Nginx from **bob.org:8080/blog** to **192.168.1.10/wordpress** in Apache.
3. Force WordPress links to use port 8080

The following is the diagram summarised my server configuration:



So you will say: “Are you completely insane ??!!”  
  
I was asked to do it! That is my answer!

Finally after one hour of searching I used a combination of two support articles ([post1](http://wordpress.org/support/topic/wordpress-behind-reverse-proxy-1), [post2](http://wordpress.org/support/topic/compatibility-with-wordpress-behind-a-reverse-proxy)), some PHP knowledge and experience and a little of imagination, I created a solution:

1. Configure Nginx to reverse proxy all requests to /blog

This is a fragment of my /etc/nginx/sites-available/default-ssl Nginx configuration file:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51 | upstream blog-webservers {      server 192.168.1.10:80;  }    # redirect http to https  # <http://serverfault.com/a/171238>  server {      listen 80;      rewrite ^(.\*) [https://](NULL)$host$1 permanent;  }    server {      listen \*:443;        ssl on;      ssl\_certificate     /etc/nginx/crypto/server.crt;      ssl\_certificate\_key /etc/nginx/crypto/server.key;        ssl\_client\_certificate /etc/nginx/crypto/ca.crt;      ssl\_verify\_client optional;      ssl\_verify\_depth 10;        # pem key asking for password problem      # <http://pandemoniumillusion.wordpress.com/2008/04/21/nginx-ssl-passphrase-at-startup/>        server\_name bob.org;      access\_log  /var/log/nginx/ssl.access.log;      error\_log   /var/log/nginx/ssl.error.log;      error\_page  404 /404.html;          # reverse proxy to blog web servers      location /blog {          proxy\_pass <http://blog-webservers/wordpress/;>          proxy\_redirect [https://server\_name](https://server_name/) <http://blog-webservers/wordpress;>            proxy\_read\_timeout       3500;          proxy\_connect\_timeout    3250;            proxy\_set\_header   X-Real-IP          $remote\_addr;          proxy\_set\_header   Host               $host;          proxy\_set\_header   X-Forwarded-For    $proxy\_add\_x\_forwarded\_for;          proxy\_set\_header   X-Forwarded-Proto  https;              proxy\_set\_header   SSL\_PROTOCOL $ssl\_protocol;          proxy\_set\_header   SSL\_CLIENT\_CERT $ssl\_client\_cert;          proxy\_set\_header   SSL\_CLIENT\_VERIFY $ssl\_client\_verify;          proxy\_set\_header   SSL\_SERVER\_S\_DN $ssl\_client\_s\_dn;      }  } |

1. Configure WordPress wp-config.php  
   I added the following lines on top of wp-config.php file:

|  |  |
| --- | --- |
| **1**  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | // If WordPress is behind reverse proxy  // which proxies https to http  if ( (!empty( $\_SERVER['HTTP\_X\_FORWARDED\_HOST'])) ||       (!empty( $\_SERVER['HTTP\_X\_FORWARDED\_FOR'])) ) {        // <http://wordpress.org/support/topic/wordpress-behind-reverse-proxy-1>      $\_SERVER['HTTP\_HOST'] = $\_SERVER['HTTP\_X\_FORWARDED\_HOST'];        define('WP\_HOME', '<https://bob.org:8080/blog>');      define('WP\_SITEURL', '<https://bob.org:8080/blog>');        // rewrite blog word with wordpress      $\_SERVER['REQUEST\_URI'] = str\_replace("wordpress", "blog",      $\_SERVER['REQUEST\_URI']);        // <http://wordpress.org/support/topic/compatibility-with-wordpress-behind-a-reverse-proxy>      $\_SERVER['HTTPS'] = 'on';  } |

If you want to debug the former PHP code fragment you can print $\_SERVER variables before and after configuration changing command inside the if block:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | echo '<br>Before:';  echo '<br>$\_SERVER[\'HTTP\_HOST\'] : ' . $\_SERVER['HTTP\_HOST'];  echo '<br>$\_SERVER[\'HTTP\_X\_FORWARDED\_HOST\']: ' . $\_SERVER['HTTP\_X\_FORWARDED\_HOST'];  echo '<br>$\_SERVER[\'REQUEST\_URI\']: ' . $\_SERVER['REQUEST\_URI'];  echo '<br>$\_SERVER[\'HTTP\_X\_FORWARDED\_SERVER\']: ' . $\_SERVER['HTTP\_X\_FORWARDED\_SERVER'];  echo '<br>$\_SERVER[\'HTTP\_X\_FORWARDED\_FOR\']: ' . $\_SERVER['HTTP\_X\_FORWARDED\_FOR'];  echo '<br>$\_SERVER[\'HTTP\_X\_FORWARDED\_FOR\']: ' . $\_SERVER['HTTP\_X\_FORWARDED\_FOR'];  echo '<br>$\_SERVER[\'HTTPS\']: ' . $\_SERVER['HTTPS'];  echo '<br>$\_SERVER[\'REMOTE\_ADDR\']: ' . $\_SERVER['REMOTE\_ADDR'];  echo '<br>$\_SERVER[\'SERVER\_NAME\']: ' . $\_SERVER['SERVER\_NAME'];  echo '<br>$\_SERVER[\'SERVER\_PROTOCOL\']: ' . $\_SERVER['SERVER\_PROTOCOL'];  echo '<br>WP\_HOME: ' . WP\_HOME;  echo '<br>WP\_SITEURL : ' . WP\_SITEURL; |

I am exhausted, just writting it down! Be patient!

**LOAD BALANCER**

Load balancing a great method to reduce latency, improve resource utilization and ultimately increase fault-tolerance. If you already use nginx as a reverse proxy, you can easily update your configuration to enable load balancing functionality for your application.

This guide shows you how to configure nginx to forward requests to a group of servers instead of passing requests to a single one.

1. How to Use Nginx as a Load Balancer
2. [Nginx Load Balancing — Advanced Configuration](https://futurestud.io/blog/nginx-load-balancing-advanced-configuration/)

## Preparation

If you didn’t configure nginx for load balancing before, use a Vagrant box to get started. We used the following **Vagrantfile** to bootstrap an initial Ubuntu box. During Vagrant’s provision phase, we install nginx and Node.js.

Vagrant.configure(2) do |config|

config.vm.box = "ubuntu/trusty64"

config.vm.network "forwarded\_port", guest: 80, host: 8080

config.vm.network "private\_network", ip: "192.168.33.10"

config.vm.provision "shell", inline: <<-SHELL

sudo apt-get update

sudo add-apt-repository ppa:nginx/stable

sudo apt-get install -y software-properties-common

sudo apt-get update

sudo apt-get install -y nginx

curl -sL https://deb.nodesource.com/setup | sudo bash -

sudo apt-get install -y nodejs

SHELL

end

Besides the initial provisioning, we forward port **80** of the box to **8080** on your host machine. By default, nginx binds to port 80 to listen for every HTTP traffic and that’s why we forward to your host machine. Besides that, you can just use a private network to bind your Vagrant box to a given IP address. If you use the **private\_network** option, you can use this IP directly within your browser to test the functionality.

## Start Multiple Node Servers

Before heading towards the nginx configuration, we need to boot our cluster to handle incoming requests and response respectively. For our cluster setup, we use basic Node.js servers which start on different ports. You can find the code to start a single Node.js server on the startpage of [Node.js](https://nodejs.org/).

Create a new file and copy/paste the following content into your new file. We choose to name the file **servers.js**.

**servers.js**

var http = require('http');

function startServer(host, port) {

http.createServer(function (req, res) {

res.writeHead(200, {'Content-Type': 'text/plain'});

res.end('Nice to meet you at ' + host + ':' + port);

}).listen(port, host);

console.log('Warming up your server at ' + host + ':' + port);

}

startServer('localhost', 3000);

startServer('localhost', 3001);

startServer('localhost', 3002);

We wrapped the code to kick off the single Node.js server into a method and call this method 3 times. The method expects two parameters: **host** and **port**. Since we use the same host, we need to run the servers on different ports.

Your console output should indicate the start of the 3 servers:

$ node servers.js

Warming up your server at localhost:3000

Warming up your server at localhost:3001

Warming up your server at localhost:3002

## Configure nginx

The default installation of nginx passes traffic to a predefined HTML file directly delivered with the installation process. Since we want nginx to load balance requests between multiple servers, we need to update the default configuration. nginx offers multiple configuration options and we’re explaining them in more detail within the next blog post. This article shows you the basic nginx configuration to enable load balancing. By default, requests are routed round-robin within the group of servers.

Now let’s update nginx’s configuration. Therefore, change into the following directory: **/etc/nginx/sites-available/**

Open the **default** nginx config as root within your editor of choice (we use **nano**). You need to open nginx’s config as root, because you won’t have write access without the root privileges.

sudo nano default

Delete the existing content and copy/paste the following into the **default** file. We’re going to describe the nginx configuration in more detail below the code block.

upstream node\_cluster {

server 127.0.0.1:3000; # Node.js instance 1

server 127.0.0.1:3001; # Node.js instance 2

server 127.0.0.1:3002; # Node.js instance 3

}

server {

listen 80;

server\_name yourdomain.com www.yourdomain.com;

location / {

proxy\_set\_header X-Real-IP $remote\_addr;

proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for;

proxy\_set\_header Host $http\_host;

proxy\_set\_header X-NginX-Proxy true;

proxy\_pass http://node\_cluster/;

proxy\_redirect off;

}

}

The **upstream** block with name **node\_cluster** defines the servers (including ports because we’re on localhost and need to start them on different ports) which should be available to handle requests. We’ll need this upstream block to proxy any requests internally to the Node.js mini-cluster.

If you’re familiar with nginx, you’re recognizing the **server** block and parts of its configuration. While listening on port **80**, we route incoming traffic to the defined **location**. Within the **location** block, we need to set HTTP headers and pass the actual request to the previously defined **upstream** block, **node\_cluster**.

Close and save the **default** file, restart nginx to publish the configuration and let’s head towards the visual testing in the browser!

## Test Load Balancing

If you didn’t start or in the meantime stopped the Node.js servers, make sure they are running. On your host machine, open your favorite browser and put the private network IP of your Vagrant box into the url bar, press enter and you’re hopefully seeing a response like **Nice to meet you at localhost:3000**. Refresh the page multiple times to see how nginx passes the requests in round-robin style to the servers.

nginx load balancing test

## Outlook

You’ve learned how to set up nginx load balancing by passing requests to a predefined **upstream** block which includes your cluster to handle the traffic. nginx provides many more configuration options and we’re going to touch them within the next blog post. There, we’ll show you how to define various load balancing mechanisms, how to assign weights to individual servers and how to handle failover situations.

# How to Configure nginx as a Load Balancer

For very small websites, a single server running a web application is usually enough. However, this doesn’t work well for larger websites. In such cases, we make use of “load balancing”, a technique that distributes incoming requests to a number of other servers, each running a copy of the web application. This is one of the techniques large websites use to stay up and running even with large volumes of traffic.

Although most people use nginx as a HTTP server, it also has built-in load balancing features. In this article, we’re going to look at how to set it up on your system.

## **nginx.conf**

**http** {

**upstream** myproject {

**server** 127.0.0.1:8000 weight=3;

**server** 127.0.0.1:8001;

**server** 127.0.0.1:8002;

**server** 127.0.0.1:8003;

}

**server** {

**listen** 80;

**server\_name** www.domain.com;

**location** / {

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

}

}

}

### **NGINX load balancing configuration**

You need x2 modules which are built into the NGINX core: [Proxy](http://wiki.nginx.org/HttpProxyModule), which forwards requests to another location, and [Upstream](http://wiki.nginx.org/HttpUpstreamModule), which defines the other location(s). They should be available by default.

Within your nginx.conf file you need to specify 2 blocks. The first of these is upstream which defines the nodes within the load balanced cluster:

upstream web\_rack { server 10.0.0.1:80; server 10.0.0.2:80; server 10.0.0.3:80; }

Here you have 3 nodes with a web server listening on port 80. The group has been called web\_rack. This is the destination for the proxy and this upstream module deals with distributing that proxied request across the defined nodes. There are [different options](http://wiki.nginx.org/HttpUpstreamModule#Directives) for how the distribution works including defining nodes with higher priority and what happens if nodes are down.

Next you tell the “vhost” about this upstream rack:

server { listen 80; server\_name www.example.com; location / { proxy\_pass http://web\_rack; } }

This creates an equivalent of an Apache vhost listening on www.example.com port 80 and all requests are proxied to the web\_rack, which then distributes them to the 3 nodes we have configured.

The full nginx.conf file would look like this:

http { upstream web\_rack { server 10.0.0.1:80; server 10.0.0.2:80; server 10.0.0.3:80; } server { listen 80; server\_name www.example.com; location / { proxy\_pass http://web\_rack; } } }

#### **More advanced options**

The docs for each module contain more examples of the options you can include but some of the ones we make use of are:

#### **NGINX load balancer log formatting**

It can be useful for debugging to dump a load of info into the request logs. We use:

log\_format upstreamlog '[$time\_local] $remote\_addr - $remote\_user - $server\_name to: $upstream\_addr: $request upstream\_response\_time $upstream\_response\_time msec $msec request\_time $request\_time';

so we can see where things are coming from and where they’re going, plus how long the response took.

This goes in the http config block and you can [find other variables in the docs](http://wiki.nginx.org/HttpUpstreamModule#Variables).

#### **Conditional forwarding based on the HTTP Method**

We distribute GET and POST requests in certain situations e.g. our graphing is very write heavy so we have dedicated, separate processes dealing with POSTs and GETs to avoid contention. Nginx splits these up:

location / { if ($request\_method = POST) { proxy\_pass http://post\_rack; break; } proxy\_pass http://get\_rack; }

This goes in the server config block.

#### **Proxy headers**

Again useful for debugging we add headers into the proxied request so we can see where things are going and where they have come from, plus some timestamps for monitoring:

proxy\_set\_header Host $host; proxy\_set\_header X-Real-IP $remote\_addr; proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for; proxy\_set\_header X-Queue-Start "t=${msec}000";

This goes in the http config block.

#### **Deploying NGINX with Puppet**

All the examples above are hand written but we actually make use of Puppet to configure everything, using our own fork to add some non-supported features like the conditional HTTP method routing, custom logging and SSL improvements (not mentioned above since they’re not load balancer specific but things like better cipher choices and [HSTS](https://en.wikipedia.org/wiki/HTTP_Strict_Transport_Security) headers).

Read more about our usage of [deploying NGINX with Puppet](http://blog.serverdensity.com/deploying-nginx-with-puppet/) or check out our tutorial content on [monitoring NGINX](https://blog.serverdensity.com/monitor-nginx/).

## **Load balancing basics**

Load balancing is a method to distribute workloads across multiple computing resources, such as computers, network links or disks. It aims to improve use of resources, maximize [throughput](https://en.wikipedia.org/wiki/Throughput), improve response times, and ensure fault-tolerance. In this article, we will talk specifically about the types of load balancing supported by nginx. However, other applications (such as database servers) can also make use of load balancing.

A typical HTTP load balancing setup looks like the one shown in the diagram below. There are many users who send in requests, and a load balancer distributes them to many servers (called backends). Each of these backends run a copy of the same application. When these servers send out a response, the load balancer send proxies them out to the clients.

Load balancers are usually categorized as hardware-based and software-based.

**Hardware-based load balancers** are dedicated equipment, and they carry [application-specific integrated circuits](https://en.wikipedia.org/wiki/Application-specific_integrated_circuit) (ASICs) that routes traffic very efficiently.

On the other hand, **software-based load balancers** such as nginx or HAproxy perform the load balancing in software.

In addition, load balancing may be performed at various network layers. As the name suggests, **layer 4 load balancers** balance traffic by inspecting the requests and responses at the [transport layer](https://en.wikipedia.org/wiki/Transport_layer). In addition, there are **layer 7 load balancers**, which inspect traffic at the [application layer](https://en.wikipedia.org/wiki/Application_layer)to figure things out. The latter is slightly more powerful, as it has a high level view of things. For instance, a layer 7 balancer that’s aware of HTTP can inspect URLs, and dispatch requests to different sets of backends depending upon the URL.

nginx can perform both layer 4 load balancing for TCP and UDP, as well as layer 7 HTTP load balancing. In the next few sections, we’re going to see how to configure nginx for this purpose.

## A note about the configuration file

On most Linux distributions, the nginx configuration file is in /etc/nginx/nginx.conf. However, on Debian/Ubuntu, this file is split into two different ones: /etc/nginx/nginx.conf for the main configuration, and /etc/nginx/sites-enabled/default for the individual websites that you host. If you are using Debian/Ubuntu, you should add the upstream blocks into the former, and the location block in the latter.

## Load balancing web applications

Before you continue any further, you need to set up a few backend servers where our web application will be running. These servers will be running a HTTP server on each of these servers. For our example, we will assume that we’ve already set them up, and these backend servers are available at 192.168.0.1, 192.168.0.2 and 192.168.0.3. Of course, these need not be internal IPs — they can be external IPs or hostnames as well.

Now, there are two things you need to configure nginx as a load balancer. First, in the http section, you need to specify a set of backend servers with an upstream block. Then, you need to proxy these requests to the servers. In order to do so, open the configuration file and add the following lines:

http {

upstream backend1 {

server 192.168.0.1;

server 192.168.0.2;

server 192.168.0.3;

}

server {

listen 80;

location / {

proxy\_pass [http://backend1](http://backend1/);

}

}

}

This tells nginx to pass over HTTP requests for any URL to the set of backend servers we’ve named backend1. If you want to pass over specific URLs only, you could do something like:

location /shop {

proxy\_pass [http://backend1](http://backend1/);

}

This will match all URLs that begin with “/shop” and hand them over to the backend servers. You can also extend this concept to pass over different URLs to different backends. In this example, we’ve added another backend for the servers that host a blog. Then we’ve configured nginx to pass over URLs that begin with “/blog” to the new backend.

upstream backend1 {

# list of servers

}

upstream backend2 {

# list of servers

}

location /shop {

proxy\_pass [http://backend1](http://backend1/);

}

location /blog {

proxy\_pass [http://backend2](http://backend2/);

}

nginx also supports a couple proxying to a couple of other protocols with directives such as FastCGI and uWSGI with the [fastcgi\_pass](https://nginx.org/en/docs/http/ngx_http_fastcgi_module.html" \l "fastcgi_pass) and [uwsgi\_pass](https://nginx.org/en/docs/http/ngx_http_uwsgi_module.html" \l "uwsgi_pass) directives. Say, for example, there is a PHP-FPM daemon running on every backend server on port 9000, and you want to proxy the requests to them. This is how the configuration would look like:

http {

upstream backend1 {

server 192.168.0.1:9000;

server 192.168.0.2:9000;

server 192.168.0.3:9000;

}

server {

listen 80;

location / {

fastcgi\_pass backend1;

}

}

}

## Load balancing strategies

nginx has a couple of strategies to pick a server to send requests to. By default, it uses a round robin algorithm to decide the server to which a request should be sent. However, other strategies are available too, which you can enable manually. The least\_conn strategy chooses a server that is handling the least number of connections. On the other hand, the ip\_hash strategy chooses servers based on the result of running a [hash function](https://en.wikipedia.org/wiki/Hash_function) on the IP. This means that requests from the same IP end up going to the same server.

To use a strategy other than the default, simply mention its name in the upstream block:

upstream backend {

ip\_hash;

server 192.168.0.1;

server 192.168.0.2;

server 192.168.0.3;

}

There’s also a a generic hash directive, which can be used on any HTTP value. In this example, we’ve distributed requests based on the URL:

hash $request\_uri;

You can further change the default strategy by assigning weights to a server. By default, all servers have a weight of 1, which means that nginx does not “prefer” a server over another, apart from applying the rules of the strategy used. An example configuration with weights looks like the following:

upstream backend {

server 192.168.0.1 weight=4;

server 192.168.0.2 weight=2;

server 192.168.0.3;

}

As you can tell, we’re using the round robin strategy in this example. With the weights we’ve shown in the example above, every 4 out of 7 requests would go to the first server, 2 of them to the second, and 1 to the third. (As we’ve mentioned previously, the default weight is 1, so we’ve left it out in the above example.)

## **Other advanced configuration**

When nginx proxies requests to these servers, it also passively performs a health check. If a server fails to send out a response, it is automatically marked as such, and nginx avoids sending requests to it for some time. The fail\_timeout value controls the time till which nginx avoids sending requests to the server. The max\_fails value sets the number of requests that should fail, after which nginx marks the server as such.

In the example below, we’ve set max\_fails to 3, and the fail\_timeout to 20 seconds for the first server:

upstream backend {

server 192.168.0.1 max\_fails=3 fail\_timeout=20s;

server 192.168.0.2;

}

In addition, you can mark a particular server to be treated differently. If you’re performing maintenance on one of the backend servers, you can mark it as down. The requests that would be processed by this server are now sent over to the next server in the group. You can also mark a particular server as a backup server. Requests will be passed to it when all other servers are unavailable. An example of these two directives is shown below:

upstream backend {

server 192.168.0.1;

server 192.168.0.2;

server 192.168.0.3 down;

server 192.168.0.4 backup;

}

## Some common problems

Depending upon the kinds of applications that you have deployed, a load balancer may cause certain problems. Often, you will get 404 or other errors from the web application. This is usually because the “Host” header contains the address of the backend server instead of its original value. To fix this, you can manually set it in the location block:

location / {

proxy\_set\_header Host $host;

proxy\_pass [http://backend](http://backend/);

}

Many complex web applications store state locally, and they can fail to work if a load balancer distributes requests of a session across different servers. In order to resolve this, the load balancer must keep track of the original backend server that responded to the initial request. This concept is known as “session persistence”. Unfortunately, this feature is only available in the commercial version of nginx. However, as a workaround, you can try the ip\_hash directive, which will forward requests from the same IP to the same backend server. IPs usually don’t change very frequently, so this may work well enough for your use case.

## Transport layer load balancing

As we’ve mentioned previously, nginx can also perform load balancing at transport layer. The syntax is slightly different than the one we’ve seen previously. Specifically, both the upstream and server sections are contained inside a stream block.

To use this feature, you have to compile nginx with the --with-stream flag. On Debian and Ubuntu, the version of nginx in the repositories is already compiled with this flag, so you can directly use it.

Suppose, you have a number of DNS servers, and you want to use nginx as a load balancer. You simply need to put this in the configuration file:

stream {

upstream dns\_backends {

server 8.8.8.8:53;

server 8.8.4.4:53;

}

server {

listen 53 udp;

proxy\_pass dns\_backends;

proxy\_responses 1;

}

}

Here, we’ve defined a number of DNS backends, and then configured nginx to listen to incoming UDP packets on port 53. The proxy\_pass directive sends them over to the backend servers. Also, by default, nginx expects that the backend may send out one or more responses. Because there’s going to be a single reply against a single request, we’ve set proxy\_responses to 1.

Load balancing with TCP is quite similar.

stream {

upstream tcp\_backend {

server [srv1.example.com](http://srv1.example.com/):3306;

server [srv2.example.com](http://srv2.example.com/):3306;

}

server {

listen 3306;

proxy\_pass tcp\_backend;

}

}

The load balancing strategies and other configuration options that we’ve described previously also apply here.

## **Conclusion**

nginx provides powerful load balancing capabilities with a wide variety of options. If you want to learn more, be sure to check out the [official documentation](https://nginx.org/en/docs/http/ngx_http_upstream_module.html).

# Load Balancing with Nginx and Docker

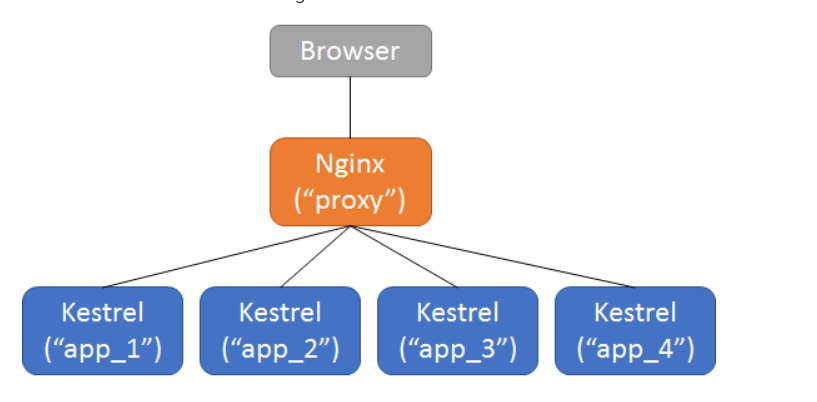
[**Aaron Alexander**](https://www.sep.com/sep-blog/author/ajalexander/)February 28, 2017[0](https://www.sep.com/sep-blog/2017/02/28/load-balancing-with-nginx-and-docker/#comments)

The [previous post](http://www.sep.com/sep-blog/2017/02/27/nginx-reverse-proxy-to-asp-net-core-separate-docker-containers/) showed how to use Nginx as a reverse proxy to an ASP.NET Core application running in a separate Docker container. This time, I’ll show how to use a similar configuration to spin up multiple application containers and use Nginx as a load balancer to spread traffic over them.

## **Desired architecture**

The architecture we’re looking for is to have four application servers running in separate Docker containers. In front of those application servers, there will be a single Nginx server. That Nginx server will reverse proxy to the application servers and will load balance using a round-robin methodology.

The desired state looks something like this:



## **Example Application**

This example uses the same application and directory structure as the previous example.

## **Docker Compose Configuration**

The configuration file for Docker compose remains exactly the same as in the previous example.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | version: '2'    services:    app:      build:        context:  ./app        dockerfile: Dockerfile      expose:        - "5000"      proxy:      build:        context:  ./nginx        dockerfile: Dockerfile      ports:        - "80:80"      links:        - app |

So, while we will eventually end up with four running instances of the app service, it only needs to be defined within the docker-compose.yml file once.

## **Nginx Configuration**

The first thing we’ll need to update is the Nginx configuration. Instead of a single upstream server, we now need to define four of them.

When updated, the nginx.conf file should look like the following:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | worker\_processes 4;    events { worker\_connections 1024; }    http {      sendfile on;        upstream app\_servers {          server example\_app\_1:5000;          server example\_app\_2:5000;          server example\_app\_3:5000;          server example\_app\_4:5000;      }        server {          listen 80;            location / {              proxy\_pass         http://app\_servers;              proxy\_redirect     off;              proxy\_set\_header   Host $host;              proxy\_set\_header   X-Real-IP $remote\_addr;              proxy\_set\_header   X-Forwarded-For $proxy\_add\_x\_forwarded\_for;              proxy\_set\_header   X-Forwarded-Host $server\_name;          }      }  } |

With this configuration, the reverse proxy defined with proxy-pass will use each of the defined upstream application servers. The requests will be passed between them in a round-robin fashion.

## **Starting the services**

The first thing we need to do is build the collection of services:

docker-compose build

Before bringing up the services, we need to tell Docker to bring multiple instances of the app service online. The Nginx configuration used four instances of the application, so we need to set the number of containers for that service to four as well.

docker-compose scale app=4

You should see each service starting

Starting example\_app\_1 ... done

Starting example\_app\_2 ... done

Starting example\_app\_3 ... done

Starting example\_app\_4 ... done

Now you can bring up all of the services

docker-compose up

## **Testing the load balancer**

Let’s navigate to the site at http://localhost:80. We can look at the output from the services to see that the requests are being split across multiple application service instances.

proxy\_1 | 172.20.0.1 - - [24/Feb/2017:18:59:39 +0000] "GET / HTTP/1.1" 200 2490 "-" "Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/56.0.2924.87 Safari/537.36"

app\_3 | info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]

app\_3 | Request starting HTTP/1.0 GET http://localhost/js/site.min.js?v=47DEQpj8HBSa-\_TImW-5JCeuQeRkm5NMpJWZG3hSuFU

app\_2 | info: Microsoft.AspNetCore.Hosting.Internal.WebHost[1]

app\_2 | Request starting HTTP/1.0 GET http://localhost/css/site.min.css?v=78TaBTSGdek5nF1RDwBLOnz-PHnokB0X5pwQZ6rE9ZA

app\_3 | info: Microsoft.AspNetCore.StaticFiles.StaticFileMiddleware[6]

app\_3 | The file /js/site.min.js was not modified

app\_3 | info: Microsoft.AspNetCore.Hosting.Internal.WebHost[2]

app\_3 | Request finished in 97.6534ms 304 application/javascript

app\_2 | info: Microsoft.AspNetCore.StaticFiles.StaticFileMiddleware[2]

app\_2 | Sending file. Request path: '/css/site.min.css'. Physical path: '/app/wwwroot/css/site.min.css'

proxy\_1 | 172.20.0.1 - - [24/Feb/2017:18:59:39 +0000] "GET /js/site.min.js?v=47DEQpj8HBSa-\_TImW-5JCeuQeRkm5NMpJWZG3hSuFU HTTP/1.1" 304 0 "http://localhost/" "Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/56.0.2924.87 Safari/537.36"

proxy\_1 | 172.20.0.1 - - [24/Feb/2017:18:59:39 +0000] "GET /css/site.min.css?v=78TaBTSGdek5nF1RDwBLOnz-PHnokB0X5pwQZ6rE9ZA HTTP/1.1" 200 251 "http://localhost/" "Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/56.0.2924.87 Safari/537.36"

In that output, you can see that the app\_1, app\_2, and app\_3 instances were all to responding to requests.

## **Further scaling**

For the purpose of this blog series, I think I intend to stop at this scaling capability level.

If you find you need to scale beyond what you can easily accomplish with Nginx, there are a couple good options to look into.

[Docker Swarm](https://www.docker.com/products/docker-swarm) provides a facade projecting multiple clustered Docker engines as a single engine. The tooling and infrastructure will feel similar to the other Docker tooling you will have used.

A more extensive option would be to use [Kubernetes](https://kubernetes.io/). That system is designed for automated deployment and scaling. It does provide an good toolset for massive scaling. But, it involves quite a bit of complexity in creating and managing your clustered services.

Either option could serve you well. Before picking up either option, I’d first make sure the scaling needs of the application really warrant the added complexity.