how to set up nginx to serve out static content

how to configure nginx as a proxy server, and

how to connect it with a FastCGI application.

reverse proxy for web application

load balancing

caching

**Load Balancing:** If we have high traffic website with requests more than 1000 req/sec. Response will be slower.

So, we will use nginx as front-end HTTP server, that intercepts all these incoming requests and distribute requests to different servers.

For 1000 req/sec., we balance like 300 req for sserver1, 300 req. for server 2,…

**Caching:**

web apps most of the times we respond with CSS file and JSS file and images. Most of the times these static files don't change very often.

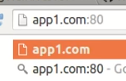
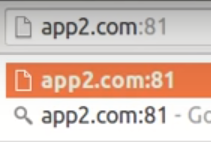
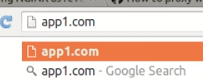
So we can use nginx caching feature for that that will improve the response time and user will have a better experience so rather than asking for these static resources from actual application server web server we can use nginx caching and we can use nginx caching to respond back to these static resources directly to the client.

so we don't have to ask our application server to respond back to these static resources that will also improve the response time and the page will load faster.

**Reverse proxy**

**S**uppose you are having multiple applications on your server and you can only run one application at a port, so if you are already running one application on port 80 you cannot run another application on port 80 you have to use a different port for that.

So now what problem arises is suppose if you have an application app 1.com that is running on port 80 so user can simply request for this application without even specifying the boat so they can say app dot 1.com app 1.com. so they will get the page

** **

but if you are running another application on the same server you have to run on different POrts say you are running on different ports say app2.com on the port 81 now to access this application clients have to specify the port number because this is not the default port.

Browsers for HTTP request use the default port as 80 .So to make this request happen you have to specify the port 81.

To deal with this problem we don't want to specify actually Port 81 we just want to give the client a simple URL app to client. where they can access the application so to how to do that???

we can put nginx on the server which will intercept the request that is coming and if you'll see okay this request is for app2.com and it will like distribute or rather than distribute I should say it will transfer that request to that application that is running on port 81 on the server so this is what reverse proxy means.

->directories are sites-available and sites-enabled

sites-enabled :whatever configuration you will put into sites enabled this configuration will apply nginx reload or nginx restart

sites-available: if you put any configuration file in sites available directory it will not be taken into effect by nginx when you do nginx restart or in the next reload or in the next start.so that's an important thing.

Now, we will be creating a file in sites-available directory, then we will create a symbolic link for that file in sites-enable directory which then can be reloaded by nginx when we reload.

1. Delete default file in site-enabled dir, which is created during installation

rm -rf sites-enabled/default

**/etc/nginx/nginx.conf**

Simple directives and context directives

**Simple directives:**  are Ex: user, worker, pid, and ones inside context-directives

**Context directives**: blocks Ex: events, http

1. **user www-data;**

User: User and group that nginx and processes going to run-as ----Here www-data.

We want our services to run with least privileges..

So Nginx runs as root, And its spawns worker\_process as normal user www-data

root@c75192caee4d:/# ps -aef | grep -in nginx

root 64 0 0 06:14 ? 00:00:00 nginx: master process /usr/sbin/nginx

www-data 90 64 0 06:24 ? 00:00:00 nginx: worker process

www-data 91 64 0 06:24 ? 00:00:00 nginx: worker process

www-data 92 64 0 06:24 ? 00:00:00 nginx: worker process

www-data 93 64 0 06:24 ? 00:00:00 nginx: worker process

1. **worker\_process 4;**

This is number of worker\_process we want to create when nginx starts.

The number we want o put depends on number of CPU cores you have

Ex: for 2 core cpu-2 , for Quad core- 4 etc

Put auto for nginx to figure-out itself--- **worker\_process auto;**

root@c75192caee4d:/# grep processor /proc/cpuinfo | wc -l

1

1. pid /run/nginx.pid;

It will store processed of nginx here

root@c75192caee4d:/# cat /run/nginx.pid

64

**EVENTS (Context directive)**

events {

worker\_connections 768;

# multi\_accept on;-----not sure what is is

}

1. **Worker\_connections(768) :**

This sets max. number of simultaneous connections that can be open by single worker process.

Total number of connections= **Worker\_connections(768) \* worker\_process(4)**

**Note: By default, linux comes with a certain number of open files. Whenever we try to open connection, it’s a new file.**

**If we have more connections---files—may get into problem**

**HTTP (Context directive)**

1. **Sendfile on; ----good feature**

If we are serving very large files …This is crucial.

Drastically speeds up large file transfers.

If we turn this on, we also need to turn tcp\_nopush

Optimizes data sent

1. **Tcp\_nopush on;**
2. **Tcp\_nodelay on;**

Forces socket to send data

1. **Keepalive\_timeout 65; ----good feature**

Set how long client connection open on server.

Can reduce cpu, memory usages, handshakes etc..

1. **Logging paths: -----access logs , error logs**

##

# Logging Settings

##

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

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

1. Gzip settings -🡪 for compression

##

##

# Gzip Settings

##

gzip on;

gzip\_disable "msie6";

# gzip\_vary on;

# gzip\_proxied any;

# gzip\_comp\_level 6;

# gzip\_buffers 16 8k;

# gzip\_http\_version 1.1;

# gzip\_types text/plain text/css application/json application/javascript text/xml application/xml application/xml+rss text/javascript;

Which files are compressed, and Sizes etc…

1. Virtual host settings

**##**

**# Virtual Host Configs**

**##**

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

**include /etc/nginx/sites-enabled/\*;-----------------We can work with any off these directories**

These are diff. paths... This is how we can bring our sites online

Static Sites Configuration:

2 directories: sites-available and sites-enabled

sites-enabled :whatever configuration you will put into sites enabled this configuration will apply nginx reload or nginx restart

sites-available: if you put any configuration file in sites available directory it will not be taken into effect by nginx when you do nginx restart or in the next reload or in the next start….so that's an important thing.

Will work with sites-available, and create symlink to sites-enabled.

Now, we will be creating a file in sites-available directory, and then we will create a symbolic link for that file in sites-enable directory which then can be reloaded by nginx when we reload.

**Cd /etc/nginx/sites-available**

**Inside server context directive, we have simple directive**

# Default server configuration

#

server {

listen 80 default\_server; -----Tells nginx which port to listen on, Default http port:80

---- Default server/per nginx configuration.

When request comes in, and host doesn’t match any servername, then it will be default to server block here

listen [::]:80 default\_server; ----Does the same as above for ipv6

# SSL configuration

#

# listen 443 ssl default\_server;

# listen [::]:443 ssl default\_server;

#

# Note: You should disable gzip for SSL traffic.

# See: https://bugs.debian.org/773332

#

# Read up on ssl\_ciphers to ensure a secure configuration.

# See: https://bugs.debian.org/765782

#

# Self signed certs generated by the ssl-cert package

# Don't use them in a production server!

#

# include snippets/snakeoil.conf;

root /var/www/html; ----🡪 Tells nginx where to look for things, when requests comes in

# Add index.php to the list if you are using PHP

index index.html index.htm index.nginx-debian.html; ----Look for these files

server\_name \_localhost; -----------If request doesn’t match any servername, it will be default one.

I can change to my domainname.com

location / { ---Once nginx decides, which server to process, it will test url headers against these parameter directives.

Here we are looking for only / (forward slash)

# First attempt to serve request as file, then

# as directory, then fall back to displaying a 404.

try\_files $uri $uri/ =404;

----Tries the literal paths asked for in relation to our root directive

}

First it will look for URI, file in document root Ex: localhost/hello.html or whatever defined in root /var/www/html;

If not found, it will look for directive defined $uri $uri/.

If still doesn’t find ---Displays 404 error

Demo:



In this case, if I goto localhost/images/main.jpeg..Instead of looking in /usr/share/nginx/index.html…will look in /uploads/images/main.jpeg and retune if it exists or not.

**Note: So, this shows more specific location requests will get checked first.**

**Create another site on same server**

1. mkdir /usr/share/nginx/test.com
2. vim /usr/share/nginx/test.com/index.html --🡪 this is test file
3. **cd /etc/nginx/sites-available**

vim test.com

root@c75192caee4d:/etc/nginx/sites-available# cat test.com

server{

listen 8000; ----------🡪 Change port if we want

root /usr/share/nginx/test.com;

index index.html index.html;

server\_name test.com;

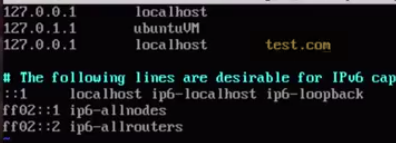
location / {

try\_files $uri $uri/ =404;

}

}

1. Change hosts file /etc/hosts



1. Create symlink

ln -s /etc/nginx/sites-available/test.com /etc/nginx/sites-enabled/

1. Restart nginx

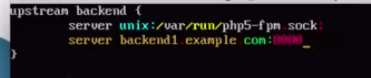
Serving Dynamic PHP sites:

1. Upstream backend: Tell nginx that we want to use Unix socket to pass PHP requests (php-fpm), so that it can process it …Return back to nginx , and then return back to user.

We can use nginx to process php applications directly can be done….>we have to use php-fpm.

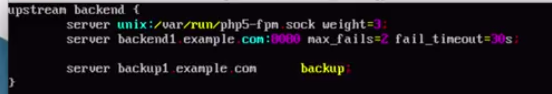
We don’t have to separate this (server unix:/var/….) in upstream backend. We can put it in server directive, inside location directive.

But this is simpler, when we have multiple different backend servers.., if you want to pass to multiple different ones.

Ex: 

We can use as load balancer, and have even distribution.

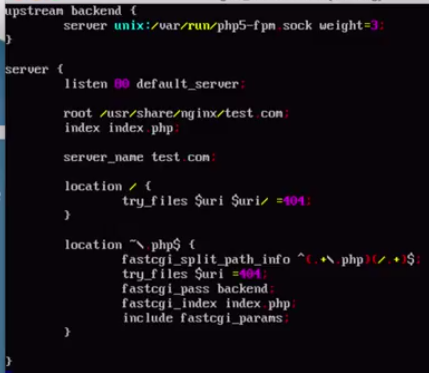
Other options:



weight=3 : for every one request to 2nd server(backend1), 3 requests goes to fhp-fpm.sock

max\_fails=2 fail\_timeout=30s-🡪 So if you have 2 fail attempts, when passing request to backend1…And those 2 in 30 sec Then nginx is not going to pass any more requests to backend1 server for 30 sec…fail\_timeout is iteration time.

We can also have backup …Only this works is when other 2 fails..



Location /-🡪because not everything will be there in php

Location \php$-->If matches php extension, will be using this directive.

Fastcgi\_split\_path🡪 resgular expression used to separate split filename, and path info.

Try\_files 🡪 if file not found, that matches uri ---404 error

Fastcgi\_pass backend-🡪 this is where we push upstream, by name backend here

Fastcgi\_index 🡪 look for index.php file

Include fastcgi\_params🡪 this is to include files for fastcgi in nginx directory.

nginx has one master process and several worker processes. The main purpose of the master process is to read and evaluate configuration, and maintain worker processes. Worker processes do actual processing of requests. nginx employs event-based model and OS-dependent mechanisms to efficiently distribute requests among worker processes. The number of worker processes is defined in the configuration file and may be fixed for a given configuration or automatically adjusted to the number of available CPU cores (see [worker\_processes](https://nginx.org/en/docs/ngx_core_module.html#worker_processes)).

**Configuration File’s Structure**

nginx consists of modules which are controlled by directives specified in the configuration file. Directives are divided into simple directives and block directives.

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

A block directive has the same structure as a simple directive, but instead of the semicolon it ends with a set of additional instructions surrounded by braces ({ and }). If a block directive can have other directives inside braces, it is called a context (examples: [events](https://nginx.org/en/docs/ngx_core_module.html#events), [http](https://nginx.org/en/docs/http/ngx_http_core_module.html" \l "http),[server](https://nginx.org/en/docs/http/ngx_http_core_module.html" \l "server), and [location](https://nginx.org/en/docs/http/ngx_http_core_module.html#location)).

Directives placed in the configuration file outside of any contexts are considered to be in the [main](https://nginx.org/en/docs/ngx_core_module.html) context. The events and http directives reside in the main context, server in http, and location in server.

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

Directives placed in the configuration file outside of any contexts are considered to be in the [main](https://nginx.org/en/docs/ngx_core_module.html) context. The events and http directives reside in the main context, server in http, and location in server.

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

**Serving Static Content**

An important web server task is serving out files (such as images or static HTML pages). You will implement an example where, depending on the request, files will be served from different local directories: /data/www(which may contain HTML files) and /data/images (containing images).

This will require editing of the configuration file and setting up of a [server](https://nginx.org/en/docs/http/ngx_http_core_module.html#server) block inside the [http](https://nginx.org/en/docs/http/ngx_http_core_module.html#http) block with two [location](https://nginx.org/en/docs/http/ngx_http_core_module.html#location) blocks.

First, create the /data/www directory and put an index.html file with any text content into it and create the /data/images directory and place some images in it.

Next, open the configuration file. The default configuration file already includes several examples of the serverblock, mostly commented out. For now comment out all such blocks and start a new server block:

http {

server {

}

}

Generally, the configuration file may include several server blocks [distinguished](https://nginx.org/en/docs/http/request_processing.html) by ports on which they [listen](https://nginx.org/en/docs/http/ngx_http_core_module.html" \l "listen)to and by [server names](https://nginx.org/en/docs/http/server_names.html). Once nginx decides which server processes a request, it tests the URI specified in the request’s header against the parameters of the location directives defined **inside the server block.**

**Add the following location block to the server block:**

location / {

root /data/www;

}

This location block specifies the “/” prefix compared with the URI from the request. For matching requests, the URI will be added to the path specified in the [root](https://nginx.org/en/docs/http/ngx_http_core_module.html#root) directive, that is, to /data/www, to form the path to the requested file on the local file system. If there are several matching location blocks nginx selects the one with the longest prefix. The location block above provides the shortest prefix, of length one, and so only if all other location blocks fail to provide a match, this block will be used.

**Next, add the second location block:**

location /images/ {

root /data;

}

It will be a match for requests starting with /images/ (location / also matches such requests, but has shorter prefix).

The resulting configuration of the server block should look like this:

server {

location / {

root /data/www;

}

location /images/ {

root /data;

}

}

This is already a working configuration of a server that listens on the standard port 80 and is accessible on the local machine at http://localhost/. In response to requests with URIs starting with /images/, the server will send files from the /data/images directory. For example, in response to the http://localhost/images/example.pngrequest nginx will send the /data/images/example.png file. If such file does not exist, nginx will send a response indicating the 404 error. Requests with URIs not starting with /images/ will be mapped onto the /data/wwwdirectory. For example, in response to the http://localhost/some/example.html request nginx will send the /data/www/some/example.html file.

To apply the new configuration, start nginx if it is not yet started or send the reload signal to the nginx’s master process, by executing:

nginx -s reload

In case something does not work as expected, you may try to find out the reason in **access.log anderror.log files in the directory /usr/local/nginx/logs or /var/log/nginx.**

#### Setting Up a Simple Proxy Server

One of the frequent uses of nginx is setting it up as a proxy server, which means a server that receives requests, passes them to the proxied servers, retrieves responses from them, and sends them to the clients.

We will configure a basic proxy server, which serves requests of images with files from the local directory and sends all other requests to a proxied server. In this example, both servers will be defined on a single nginx instance.

First, define the proxied server by adding one more server block to the nginx’s configuration file with the following contents:

server {

listen 8080;

root /data/up1;

location / {

}

}

This will be a simple server that listens on the port 8080 (previously, the listen directive has not been specified since the standard port 80 was used) and maps all requests to the /data/up1 directory on the local file system. Create this directory and put the index.html file into it. Note that the root directive is placed in theserver context. Such root directive is used when the location block selected for serving a request does not include own root directive.

Next, use the server configuration from the previous section and modify it to make it a proxy server configuration. In the first location block, put the [proxy\_pass](https://nginx.org/en/docs/http/ngx_http_proxy_module.html#proxy_pass) directive with the protocol, name and port of the proxied server specified in the parameter (in our case, it is http://localhost:8080):

server {

location / {

proxy\_pass http://localhost:8080;

}

location /images/ {

root /data;

}

}

We will modify the second location block, which currently maps requests with the /images/ prefix to the files under the /data/images directory, to make it match the requests of images with typical file extensions. The modified location block looks like this:

location ~ \.(gif|jpg|png)$ {

root /data/images;

}

The parameter is a regular expression matching all URIs ending with .gif, .jpg, or .png. A regular expression should be preceded with ~. The corresponding requests will be mapped to the /data/images directory.

When nginx selects a location block to serve a request it first checks [location](https://nginx.org/en/docs/http/ngx_http_core_module.html#location) directives that specify prefixes, remembering location with the longest prefix, and then checks regular expressions. If there is a match with a regular expression, nginx picks this location or, otherwise, it picks the one remembered earlier.

The resulting configuration of a proxy server will look like this:

server {

location / {

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

}

location ~ \.(gif|jpg|png)$ {

root /data/images;

}

}

This server will filter requests ending with .gif, .jpg, or .png and map them to the /data/images directory (by adding URI to the root directive’s parameter) and pass all other requests to the proxied server configured above.

To apply new configuration, send the reload signal to nginx as described in the previous sections.

There are many [more](https://nginx.org/en/docs/http/ngx_http_proxy_module.html) directives that may be used to further configure a proxy connection.

#### Setting Up FastCGI Proxying

nginx can be used to route requests to FastCGI servers which run applications built with various frameworks and programming languages such as PHP.

The most basic nginx configuration to work with a FastCGI server includes using the [fastcgi\_pass](https://nginx.org/en/docs/http/ngx_http_fastcgi_module.html" \l "fastcgi_pass) directive instead of the proxy\_pass directive, and [fastcgi\_param](https://nginx.org/en/docs/http/ngx_http_fastcgi_module.html" \l "fastcgi_param) directives to set parameters passed to a FastCGI server. Suppose the FastCGI server is accessible on localhost:9000. Taking the proxy configuration from the previous section as a basis, replace the proxy\_pass directive with the fastcgi\_pass directive and change the parameter to localhost:9000. In PHP, the SCRIPT\_FILENAME parameter is used for determining the script name, and the QUERY\_STRING parameter is used to pass request parameters. The resulting configuration would be:

server {

location / {

fastcgi\_pass localhost:9000;

fastcgi\_param SCRIPT\_FILENAME $document\_root$fastcgi\_script\_name;

fastcgi\_param QUERY\_STRING $query\_string;

}

location ~ \.(gif|jpg|png)$ {

root /data/images;

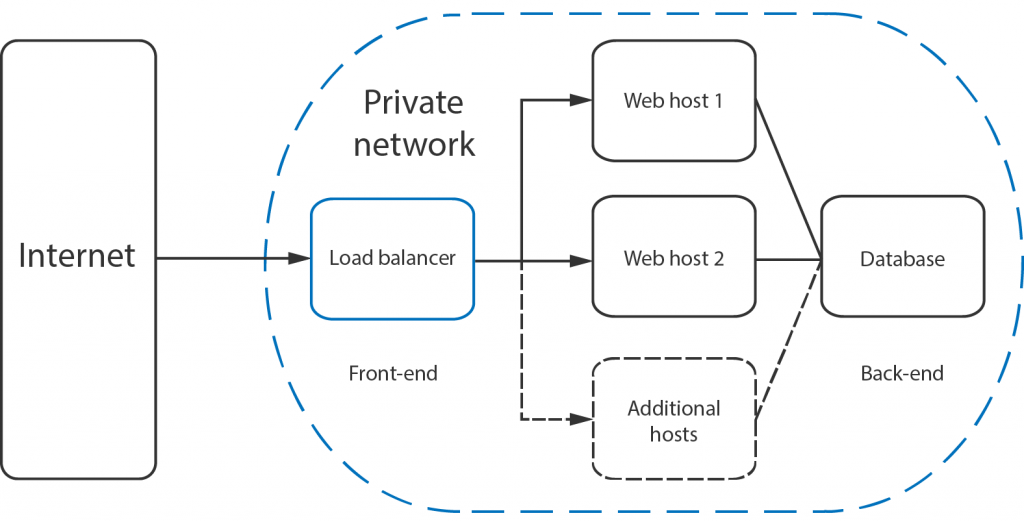
}

}

This will set up a server that will route all requests except for requests for static images to the proxied server operating on localhost:9000 through the FastCGI protocol.

How to Configure Load Balancing with Nginx

Load balancing is an excellent way to scale out your application and increase it’s performance and redundancy. Nginx, which is a popular web server software, can be configured as a simple yet powerful load balancer to improve your servers resource availability and efficiency. In a load balancing configuration nginx acts as single entrance point to a distributed web application working on multiple separate servers.



This guide describes how to set up load balancing with nginx for your cloud servers. As a prerequisite you’ll need to have at least two hosts with a web server software installed and configured to see the benefit of the load balancer. If you already have one web host set up, you can use the [Server Cloning](https://www.upcloud.com/support/server-cloning/) feature available at your [UpCloud Control Panel](https://my.upcloud.com/" \t "_blank).

Installing nginx

The first thing to do is to set up a new host that will serve as your load balancer. Go ahead and deploy a new instance at your UpCloud Control Panel if you haven’t already. Currently nginx packages are available on the latest versions of CentOS, Debian and Ubuntu, so pick which ever of these you prefer.

After setting up the server the way you like, adding users, running updates, and so on, install the latest stable nginx using one of the following methods.

# Enter the following to the file

[nginx]

name=nginx repo

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

gpgcheck=0

enabled=1

sudo yum update

sudo yum install nginx

Once installed change directory into the nginx main configuration folder.

cd /etc/nginx/

Now depending on your OS the web server configuration files will be in one of two places.

Ubuntu and Debian follow a rule for storing virtual host files in /etc/nginx/sites-available/, which are enabled through symbolic links to /etc/nginx/sites-enabled/. You can use the command below to enable any new virtual host files.

sudo ln -s /etc/nginx/sites-available/<vhost> /etc/nginx/sites-enabled/<vhost>

CentOS users can find their host configuration files under /etc/nginx/conf.d/ in which any *.conf* -type virtual host file gets loaded.

Check that you can find at least the default configuration and then restart nginx.

sudo service nginx restart

Test that the server replies to HTTP requests by opening the load balancer server’s public IP address in your web browser. When you see the default welcoming page for nginx the installation was successful.



If you are having trouble loading the page, check that a firewall is not blocking your connection. For example on CentOS 7 the default firewall rules do not allow HTTP traffic, enable it with the commands below.

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

sudo firewall-cmd --reload

Then try reloading your browser.

Configuring nginx as a load balancer

With nginx installed and tested you can start configuring it for load balancing. In essence all you need to do is setup nginx with instructions for which type of connections to listen to and where to redirect them. To accomplish this, create a new configuration file using which ever text editor you prefer, for example with *nano*:

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

In the *load-balancer.conf* you’ll need to define the following two segments, *upstream* and *server*, see the examples below.

# Define which servers to include in the load balancing scheme.

# It's best to use the servers' private IPs for better performance and security.

# You can find the private IPs at your [UpCloud Control Panel](https://my.upcloud.com/network) Network section.

upstream backend {

server 10.1.0.101;

server 10.1.0.102;

server 10.1.0.103;

}

# This server accepts all traffic to port 80 and passes it to the upstream.

# Notice that the upstream name and the proxy\_pass need to match.

server {

listen 80;

location / {

proxy\_pass http://backend;

}

}

Then save the file and exit the editor.

Next you’ll need to disable the default server configuration you earlier tested was working after the installation. Again depending on your OS this part differs slightly.

On Debian and Ubuntu systems you’ll need to remove the *default* symbolic link from the *sites-enabled* folder.

sudo rm /etc/nginx/sites-enabled/default

CentOS hosts don’t use the same linking, instead simply rename the *default.conf* in the *conf.d/* directory to something that doesn’t end with *.conf*, for example:

sudo mv /etc/nginx/conf.d/default.conf /etc/nginx/conf.d/default.conf.disabled

Then use the following to restart nginx.

sudo service nginx restart

Check that nginx starts successfully. If the restart fails, take a look at the  */etc/nginx/conf.d/load-balancer.conf* you just created to make sure there are no mistypes or missing semicolons.

You should now be passed to one of your back-end servers when entering the load balancer’s public IP address in your web browser.

Load balancing methods

Load balancing with nginx uses 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 an other straight forward 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 server 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;

}

For 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://www.upcloud.com/support/install-lets-encrypt-nginx/).

Using encryption with a load balancer is easier than you might think. All you need to do is add an other 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 as failed and will try to avoid directing 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.

Conclusions

If you wish to improve your web application performance and availability, setting up a load balancer is definitely something to consider. Load balancing with nginx is powerful yet relatively simple to setup and together with an easy encryption solution, such as Let’s Encrypt client, it makes for a great front-end to your web farm. Check out [the documentation for upstream](http://nginx.org/en/docs/http/ngx_http_upstream_module.html) over at nginx.org to learn more.

While using multiple hosts protects your web service with redundancy, the load balancer itself can still leave a single point of failure. You can further improve high availability by setting up a floating IP between multiple load balancers. You can find out more at our article for [Floating IPs on UpCloud](https://www.upcloud.com/support/floating-ips-on-upcloud/).



a request was initiated by your browser leaving your local network traversing the global network or the Internet as it is called till it reaches one of the computers which are assigned with serving Google's homepage that computer the one that was serving the Google's homepage in return initiates a response containing the contents of the requested page this response takes a similar route through the Internet till it reaches your local network and finally your computer's browser he came here that was tasket with serving you this web page is called a web server.

visit the Google web page file typing www.google.com on in your browser and when you receive the response this request in response in order for them to be successful in order for them to display the page the way you saw it on your screen on your browser or on your mobile device if you are using a cell phone or a tablet this is called the HTTP protocol.

HTTP stands for hypertext Transfer Protocol a protocol and Ikey jargon refers to a set of rules that are agreed upon between two or more parties for communication.

when you requested www.google.com you initiated an HTTP GET request what's that you have gold in HTTP language the scope and get request you are asking the server that you want to get a page that resides on the file system somewhere on that server when the web server receives this request it knows that it should reply with a response containing the HTML CSS JavaScript images and any other components that happen to be on the page.

your browser is responsible for interpreting this data and displaying it in a proper manner by default the HTTP protocol is served on port 80 of the web server and if you are not aware of ports a port is just a window from which the server accepts connections just think of it as a window in your house if you want air and Sun energy house you will have to open a window in your room or where you are sitting but imagine that you have several windows in your house and imagine that you have different types of things that you want to receive through those windows perhaps some people want to talk to you through the windows they are standing outside your home and you want to talk to you to these windows so you will need to know which person can talk to you through which windows so you numbers your windows let's say from 1 to 10 if you have 10 windows and then you say ok I want to talk to Joe I'm going to talk to him on window number 5 I want to talk to Mary I can talk to her on window number 9 and so on and so forth in compare networking this is done through something called port support it's just a window on the server on the computer it agrees or it specifies that this port will be used to receive that sort of connection that sort of communication in HTTP world this window or this port by default is port 80 and this port may be easily changed and many web servers actually do this for reasons of security and for other reasons why for example if there is already another window or another port that is already open for some other application and it's already listening for the accepting connections on port 80 this port can be changed to another port and it will act perfect and as efficient as sport a teetotum so the number here is just a number it's useless is just a default number that is agreed upon by the HTTP developers around the world and web developers that this sport will be used for HTTP connections.

