Servidor (i proxy) web Nginx

nginx (http://nginx.org , http://nginx.com) consists of modules which are controlled by directives specified in the configuration file (by default, it is **/etc/nginx/nginx.conf**). Directives are divided into simple directives and block directives:

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

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

If you know Apache, maybe this list will help you:

main context --> General configuration (apache2.conf)
http context --> General configuration (apache2.conf)
server context --> VirtualHost configuration
    listen directive --> Listen directive
    server_name directive --> ServerName directive
    root directive --> DocumentRoot directive (can be inside *http* or *location* context too)
    error_log directive --> ErrorLog directive (can be inside *http* or *location* context too)
    access_log directive --> AccessLog directive (can be inside *http* or *location* context too)
    index directive --> DirectoryIndex directive (can be inside *http* or *location* context too)
    autoindex on --> AutoIndex directive (can be inside *http* or *location* context too)
location context --> Location section
    alias directive --> Alias directive
    return 301 [http://www.example.com/moved/here] --> Redirect directive
    error_page 404 /404.html --> ErrorDocument directive (can be inside *server* context too)
    rewrite directive --> Rewrite directive (can be inside *server* context too)
    {deny|all} all --> Require all {denied|granted} (can be inside *server* context too)
    {deny|all} x.x.x.x/y --> Require ip x.x.x.x/y {denied|granted} (can be inside *server* context too)

NOTA: 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.

NOTA: More on: https://www.nginx.com/resources/admin-guide/nginx-web-server/ , https://www.nginx.com/resources/admin-guide/serving-static-content/ , https://linode.com/docs/web-servers/nginx/how-to-configure-nginx/ , https://www.digitalocean.com/community/tutorials/understanding-the-nginx-configuration-file-structure-and-configuration-contexts

How to begin

sudo nano /etc/nginx/sites-available/example.com

```
server {
   listen *:80;
   #listen [::]:80;
   server_name localhost;
   root /var/www/html;
   index index.html index.htm;
   location / {
      # First attempt to serve request as file, then as directory, then fall back to displaying a 404 (by default it is like this)
      try_files $uri $uri/ =404;
   }
}
```

You now have your server blocks created, we need to enable them. We can do this by creating symbolic links from these files to the sites-enabled directory, which Nginx reads from during startup. We can create these links by typing:

sudo ln -s /etc/nginx/sites-available/example.com /etc/nginx/sites-enabled/

If you want to disable default server block, you can simply remove its symbolic link. It will still be available for reference in the sites-available directory, but it won't be read by Nginx on startup:

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

Serving Static Content

Generally, the configuration file may include several server blocks distinguished by ports on which they listen to and by server names. 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.

In general, the location / { ... } section instructs Nginx to forward all requests from / and down in the virtual directory structure of the website, to the folder specified in root directive. It match any part of an HTTP request that comes after the host segment but nginx always fulfills request using the most specific match. So, for example, if we have a location directive pointing to "/planet/blog" and another location directive pointing to "/planet", which request will be fulfilled, http://example.com/planet/blog/ or http://example.com/planet/blog/about/ ? They are fulfilled by the location "/planet/blog/" setting because it is more specific, even though location /planet/ also matches this request.

When a location directive is followed by an equal sign (=), nginx performs an exact match. That is, it doesn't down in the directory structure.

When a location directive is followed by a tilde (~), nginx performs a regular expression match. These matches are always case-sensitive. If you want matches to be case-insensitive, use a tilde with an asterisk (~*) Examples:

```
location ~ IndexPage\.php$ { }
location ~ ^/BlogPlanet(/|/index\.php)$ { }
location ~* \.(pl|cgi|perl|prl)$ { }
```

As an example, 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 and write this:

```
server {
     #This is the shortest prefix, so only if all other location blocks fail to provide a match, this block will be used.
     location / {
        root /data/www;
     }
     #It will be a match for requests starting with /images/ (location / also matches such requests, but has shorter prefix)
     #Trailing dash is important: without it, you risk having weird-looking URLs (e.g., a working /fooen in addition to /foo/en).
     location /images/ {
        root /data;
     }
}
```

In response to the http://localhost/images/example.png request nginx will send the "/data/images/example.png" file. Requests with URIs not starting with /images/ will be mapped onto the "/data/www" directory. For example, in response to the http://localhost/some/example.html request nginx will send the "/data/www/some/example.html" file.

Another example:

```
server {
   root /var/www/main;
   location / {
       try_files $uri $uri.html $uri/ /fallback/index.html;
   }
   location /fallback {
       root /var/www/another;
   }
}
```

In the above example, if a request is made for /blahblah, the first location will initially get the request. It will try to find a file called blahblah in /var/www/main directory. If it cannot find one, it will follow up by searching for a file called blahblah.html. It will then try to see if there is a directory called blahblah/ within the /var/www/main directory. Failing all of these attempts, it will redirect to /fallback/index.html. This will trigger another location search that will be caught by the second location block. This will serve the file /var/www/another/fallback/index.html.

On the other hand, aliases are an interesting tool, too, because define a replacement for the specified location. For example, with the following configuration, on request of “/i/top.gif”, the file /data/w3/images/top.gif will be sent:

```
location /i/ {
        alias /data/w3/images/;
}
```

NOTA: When location matches the last part of the directive’s value, it is better, however,to use the root directive instead of aliases. For example, instead of writing this...:

```
location /images/ {
        alias /data/w3/images/;
}
```

...it would be better writing this one:

```
location /images/ {
      root /data/w3;
}
```

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 like this:

```
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 the server 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" directive with the protocol, name and port of the proxied server specified in the parameter (in our case, it is http://localhost:8080). Moreover, 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 parameter is a regular expression matching all URIs ending with .gif, .jpg, or .png -remember that a regular expression should be preceded with ~ - ; the corresponding requests will be mapped to the /data/images directory):

```
server {
        location / {
             #Note the trailing dash! (it's important to correspond the /foo/ on the front-end location directive with "/" on the backend)
             proxy_pass http://localhost:8080/;
        }
        location ~ \.(gif|jpg|png)$ {
             root /data/images;
        }
}
```

So, in summary, 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.

NOTA: When nginx selects a location block to serve a request it first checks 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.
NOTA: Más en https://www.nginx.com/resources/admin-guide/reverse-proxy/ (select output interface, manipulate request headers...)

Setting Up a HTTP Load Balancer

Load balancing across multiple application instances is a commonly used technique for optimizing resource utilization, maximizing throughput, reducing latency, and ensuring fault-tolerant configurations. The following load balancing mechanisms (or methods) are supported in nginx:

round-robin	— requests to the application servers are distributed in a round-robin fashion,
least-connected	— next request is assigned to the server with the least number of active connections,
ip-hash	— a hash-function is used to determine what server should be selected for the next request (based on the client's IP address).

The simplest configuration for load balancing with nginx may look like the following:

```
http {
        upstream myapp1 {
            server srv1.example.com:8123;
            server srv2.example.com;
            server srv3.example.com;
        }
        server {
            listen 80;
            location / {
                proxy_pass http://myapp1/;
            }
        }
}
```

In the example above, there are 3 instances of the same application running on srv1-srv3. To pass requests to the server group, the name of the group is specified in the proxy_pass directive. When the load balancing method is not specifically configured, it defaults to round-robin. All requests are proxied to the server group myapp1, and nginx applies HTTP load balancing to distribute the requests. To configure load balancing for HTTPS instead of HTTP, just use "https" as the protocol.

Another load balancing discipline, as we already said, is "least-connected". Least-connected allows controlling the load on application instances more fairly in a situation when some of the requests take longer to complete. With the least-connected load balancing, nginx will try not to overload a busy application server with excessive requests, distributing the new requests to a less busy server instead. Least-connected load balancing in nginx is activated when the least_conn directive is used as part of the server group configuration:

```
upstream myapp1 {
        least_conn;
        server srv1.example.com;
        server srv2.example.com;
        server srv3.example.com;
}
```

Please note that with round-robin or least-connected load balancing, each subsequent client's request can be potentially distributed to a different server. There is no guarantee that the same client will be always directed to the same server. If there is the need to tie a client to a particular application server (in other words, make the client's session "sticky" or "persistent" in terms of always trying to select a particular server) the "ip-hash" load balancing mechanism can be used. With ip-hash, the client's IP address is used as a hashing key to determine what server in a server group should be selected for the client's requests. This method ensures that the requests from the same client will always be directed to the same server except when this server is unavailable.To configure ip-hash load balancing, just add the ip_hash directive to the server (upstream) group configuration:

```
upstream myapp1 {
        ip_hash;
        server srv1.example.com;
        server srv2.example.com;
        server srv3.example.com;
}
```

It is also possible to influence nginx load balancing algorithms even further by using server weights. In the examples above, the server weights are not configured which means that all specified servers are treated as equally qualified for a particular load balancing method. With the round-robin in particular it also means a more or less equal distribution of requests across the servers provided there are enough requests, and when the requests are processed in a uniform manner and completed fast enough. When the weight parameter is specified for a server, the weight is accounted as part of the load balancing decision.

```
upstream myapp1 {
            server srv1.example.com weight=3;
            server srv2.example.com;
            server srv3.example.com;
}
```

With this configuration, every 5 new requests will be distributed across the application instances as the following: 3 requests will be directed to srv1, one request will go to srv2, and another one to srv3. Besides "weight" parameter, other ones can be placed in the server directive inside upstream block; they influence the backend selection like this:

max_fails=n: This defines the number of timed-out connections that should occur (in the time frame specified with the fail_timeout parameter below) before Nginx considers the server inoperative.
fail_timeout=n: If Nginx fails to communicate with the backend server max_fails times over fail_timeout seconds, the server is considered inoperative.
down: This server is no longer used. This only applies when the ip_hash directive is enabled.
backup: If a backend server is marked as backup, Nginx will not make use of the server until all other servers (servers not marked as backup) are down or inoperative.

NOTA: Más en http://nginx.org/en/docs/http/load_balancing.html y https://www.nginx.com/resources/admin-guide/load-balancer

SSL

```
server {
            listen 80;
            listen 443 ssl;
            server_name www.cocacola.com;
            ssl_certificate       /etc/ssl/certs/cert.crt;
            ssl_certificate_key   /etc/ssl/private/clauPriv.pem;
            #ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
            #ssl_ciphers HIGH:!aNULL:!MD5;
            root /ruta/carpeta;
            error_log /ruta/arxiu.log;
            access_log /ruta/arxiu.log formatLog;
            location / {
                  index index.html index.htm;
            }
}
```

NOTA: Más en http://nginx.org/en/docs/http/configuring_https_servers.html y https://www.nginx.com/resources/admin-guide/nginx-ssl-termination/

Caching (only in Nginx Plus)

When caching is enabled, Nginx Plus saves responses in a disk cache and uses them to respond to clients without having to proxy requests for the same content every time. To enable caching, include the proxy_cache_path directive in the top-level http context. The mandatory first parameter is the local filesystem path for cached content, and the mandatory keys_zone parameter defines the name and size of the shared memory zone that is used to store metadata about cached items. Then include the proxy_cache directive in the context (protocol type, virtual server, or location) for which you want to cache server responses, specifying the zone name defined by the keys_zone parameter to the proxy_cache_path directive (in this case, "one"):

```
http {
    ...
    proxy_cache_path /data/nginx/cache keys_zone=one:10m;
    server {
        proxy_cache one;
        location / {
            proxy_pass http://localhost:8000/;
        }
    }
}
```

NOTA: Más en: https://www.nginx.com/resources/admin-guide/content-caching/

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 directive instead of the proxy_pass directive, and 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.