

Nginx:

NGINX is open source software for web serving, reverse proxying, caching, load balancing, media streaming, and more. It started out as a web server designed for maximum performance and stability. In addition to its HTTP server capabilities, NGINX can also function as a proxy server for email (IMAP, POP3, and SMTP) and a reverse proxy and load balancer for HTTP, TCP, and UDP servers.

Apache Vs Nginx

- > Apache is by default configured at **PREFORK** mode, which means response is to handle one request at a time.
- > Nginx deals with requests asynchronously, which means a single nginx process can serve multiple requests concurrently.
- > Because of this asynchronous design nginx can't embed server side programming languages into its own process. Meaning all requests for dynamic content have to deal with a complete spread process like PHP-FPM then reverse proxy back to client via Nginx.
- > When there is mix content like static(images,etc) & dynamic(PHP scripts) Apache involves the php as well to serve the static content but in Nginx it won't involve php for processing the static content that's why Nginx is faster than apache.
- > Nginx has high concurrency as it serves requests asynchronously.
- > Configuration : Nginx configuration also takes different approach for content

Nginx : URI location

```
Location /images{  
}
```

Apache : File system location

```
<Directory "www/sites/images"> </Directory>
```

Refer below URL what is C10K problem i.e 10K concurrent connection problem

https://en.wikipedia.org/wiki/C10k_problem

Installing Nginx Via package manager:

- > apt-get install nginx

For installing from releases:

<https://fedoraproject.org/wiki/EPEL>

Building Nginx from source & adding modules:

<http://nginx.org/en/download.html>

<http://nginx.org/en/docs/configure.html>

-> wget <http://nginx.org/download/nginx-1.19.0.tar.gz>

-> switch to folder nginx-1.19.0 then configure it with **./configure**

-> Nginx path to start & stop nginx servers : **--sbin-path=/usr/local/bin/nginx**

-> Nginx configuration path : **--conf-path=/usr/local/etc/nginx/nginx.conf**

<https://thoughtbot.com/blog/the-magic-behind-configure-make-make-install>

./configure is responsible for getting ready to build the software on your specific system. It makes sure all of the dependencies for the rest of the build and install process are available, and finds out whatever it needs to know to use those dependencies.

make - utility for building and maintaining groups of programs.

Once **configure** has done its job, we can invoke **make** to build the software. This runs a series of tasks defined in a **Makefile** to build the finished program from its source code.

make install: Now that the software is built and ready to run, the files can be copied to their final destinations. The make install command will copy the built program, and its libraries and documentation, to the correct locations.

Steps for manual configuration of nginx from start:

- 1) **sudo ./configure --sbin-path=/usr/local/bin/nginx
--conf-path=/usr/local/etc/nginx/nginx.conf
--error-log-path=/var/log/nginx/error.log
--http-log-path=/var/log/nginx/access.log --with-pcre
--pid-path=/var/run/nginx.pid**
- 2) **sudo make**
- 3) **sudo make install**
- 4) **ls -l /usr/local/etc/nginx/ ⇒ To check nginx executable file present**
- 5) **nginx -v ⇒ output : nginx version: nginx/1.19.0**
- 6) Start the nginx with **nginx** command and check in the browser by default nginx runs on port 8080.

Adding an Nginx Service:

-> systemd : systemd is only supported by linux machines. If you have then add the systemd service to nginx for start and stop. Example is shown in the below link

<https://www.nginx.com/resources/wiki/start/topics/examples/systemd/>

Note: just change the path according to the PID path & nginx path which we have set while configuration above. And Configure the systemctl to automatically start the nginx after reboot machines.

Understanding configuration terms:

- > **server_name localhost** ⇒ Name and value are directives
- > **http {}** ⇒ **Context**, Section which holds the directives. So basically it is **http context** like wise there is **server context** , **location context**.

Creating Virtual Host:

- > To create virtual host or host configuration we need to define it inside the http context.
- > **include mime.types;** is for including all extensions like html, css, images, etc which is defined inside a mime.types file. mime .types file directory is **MAC: /usr/local/etc/nginx/** , **Ubuntu : etc/nginx/**
- > **listen** is for the port on which your application will listen.
- > **server_name** as name suggests its name of your application. So either put an application name or Ip address which is pointing to your server.
- > **root** holds the path of your application from where it needs to serve the request or process the request. i.e your code directory
- > Below code is written inside **nginx.conf** file

```
events {  
}  
  
http {  
    include mime.types;  
  
    server {  
        listen 8080;  
        server_name localhost;  
  
        root /Users/Apple/learning/Nginx/demo;  
    }  
}
```

For deep learning on server & location block refer below url:

<https://www.digitalocean.com/community/tutorials/understanding-nginx-server-and-location-block-selection-algorithms#:~:text=A%20location%20block%20lives%20within,i%20an%20extremely%20flexible%20model.>

Location Block:

- > A location block lives within a server block and is used to define how Nginx should handle requests for different resources and URIs for the parent server. The URI space can be subdivided in whatever way the administrator likes using these blocks. It is an extremely flexible model.
- > Location block is at server level means there will be different location blocks for different servers which are defined inside the server block .
- > location block defined outside block will throw the following error
nginx: [emerg] "location" directive is not allowed here in /usr/local/etc/nginx/nginx.conf:27

-> Location syntax:

```
location uri {  
    task which needs to perform.  
}
```

-> I have defined **/process url** so any requests that come to that url will give the response.



```
events {  
}
```





```
http {  
    include mime.types;  
  
    server {  
        listen 8080;  
        server_name localhost;  
  
        root /Users/Apple/learning/Nginx/demo;  
        location /process {  
            return 200 "text serve apart from parent uri";  
        }  
    }  
}
```

 **localhost:8080/process**

text serve apart from parent uri

-> below are defined for uri match:

- | | |
|------------------------------|---|
| 1. Exact Match |  uri |
| 2. Preferential Prefix Match |  uri |
| 3. REGEX Match |  uri |
| 4. Prefix Match | uri |

- (none): If no modifiers are present, the location is interpreted as a *prefix* match. This means that the location given will be matched against the beginning of the request URI to determine a match.
- : If an equal sign is used, this block will be considered a match if the request URI exactly matches the location given.
- : If a tilde modifier is present, this location will be interpreted as a case-sensitive regular expression match.
- : If a tilde and asterisk modifier is used, the location block will be interpreted as a case-insensitive regular expression match.
- : If a caret and tilde modifier is present, and if this block is selected as the best non-regular expression match, regular expression matching will not take place.

Variables:

-> As we can see that nginx syntax closely resembles a programming language only where you can define variables, you can write if statements as well.

-> You can create your own variable like this with prefix **\$** e.g. **\$var** or **\$abc** or you can use the nginx defined variables.

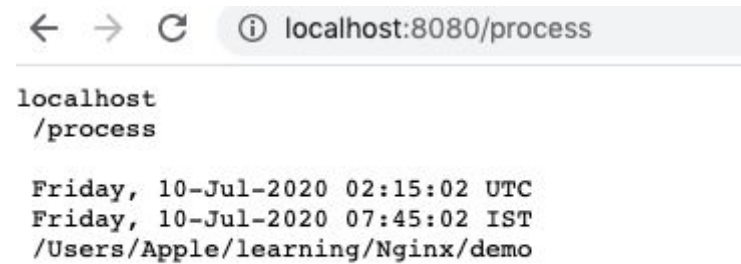
For system defined variable :

<http://nginx.org/en/docs/varindex.html>

```
location /process {  
    return 200 "$host\n $uri \n $args \n $date_gmt \n $date_local \n  
$document_root";  
}
```

Output of above code:

args returned empty because we didn't pass any arguments in uri.



```
localhost
/process

Friday, 10-Jul-2020 02:15:02 UTC
Friday, 10-Jul-2020 07:45:02 IST
/Users/Apple/learning/Nginx/demo
```

-> As you can see above we have used system defined variables like we can use our own variables as well to check or process some events with some condition or so on.

NOTE: Using conditional operator (if block) inside location is quite risky as it can behave in unexpected or quite different ways then you expect to work.

For detail please refer <https://www.nginx.com/resources/wiki/start/topics/depth/ifisevil/>

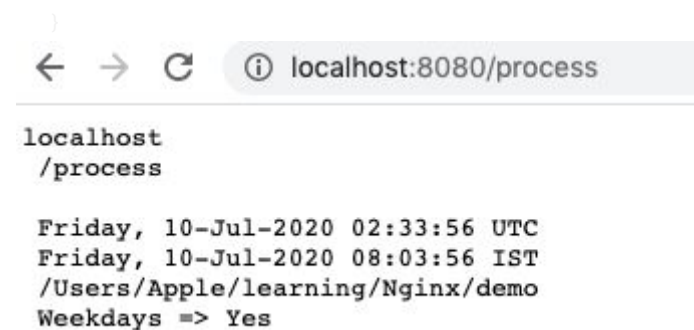
```
server {
    listen 8080;
    server_name localhost;

    root /Users/Apple/learning/Nginx/demo;

    set $weekdays "NO";
    if ( $date_local ~ 'Friday|friday' ) {
        set $weekdays "Yes";
    }

    location /process {
        return 200 "$host\n $uri \n $args \n $date_gmt \n $date_local \n $document_root
\n Weekdays => $weekdays";
    }
}
```

Output:



```
localhost
/process

Friday, 10-Jul-2020 02:33:56 UTC
Friday, 10-Jul-2020 08:03:56 IST
/Users/Apple/learning/Nginx/demo
Weekdays => Yes
```

Redirect & Rewrite:

NOTE: Difference between redirect & rewrite is, redirect change the uri to redirected path where else rewrite doesn't change the uri but internally it redirects to that page.

Redirect

-> Redirect status code is 300 series i.e 307.

-> So to handle redirect requests while returning we need to pass the status code as 307.

-> For example if we want to access image then the path is

<http://localhost:8080/thumb.png>

But we want it to be served on /logo as well. Currently if someone enters /logo then it will throw 404 errors.

To redirect we can do it like this and it will redirect to thumb.png location

```
location /logo {  
    return 307 /thumb.png;  
}
```

Rewrite:

-> Rewrite doesn't change the uri but internally it redirects to that page.

-> When the value is rewritten it also gets reevaluated by nginx as completely new requests.

-> Reevaluation also makes the rules more powerful but it also takes additional system resources to process them.

```
rewrite ^/users/\w+ /redirectedurl;  
location /redirectedurl {  
    return 200 "Request served from redirectedurl";  
}
```

Output :



-> As you can see that browser url is still pointing to uses/swapnil but the request is served from /redirectedurl. So the request is rewritten internally without the knowledge of the client or users.

-> So consider a scenario where you want a specific user you need to serve the request from a different location.

```
rewrite ^/users/(\w+) /redirectedurl/$1;
```

```
location /redirectedurl {  
    return 200 "Request served from redirectedurl";  
}
```

```
location = /redirectedurl/swapnil {  
    return 200 "Request served specially for swapnil from this location";  
}
```

Output : As you can see for swapnil request served from different location & request for abc is served from different location.



-> **(\w+)** captures the username & serves as 1st capture group in **/redirectedurl/\$1 as \$1**.

-> so if there is a second capture group like this **/users/(\w+)/something** then we could have access to that value like this **/redirectedurl/\$1 \$2**.

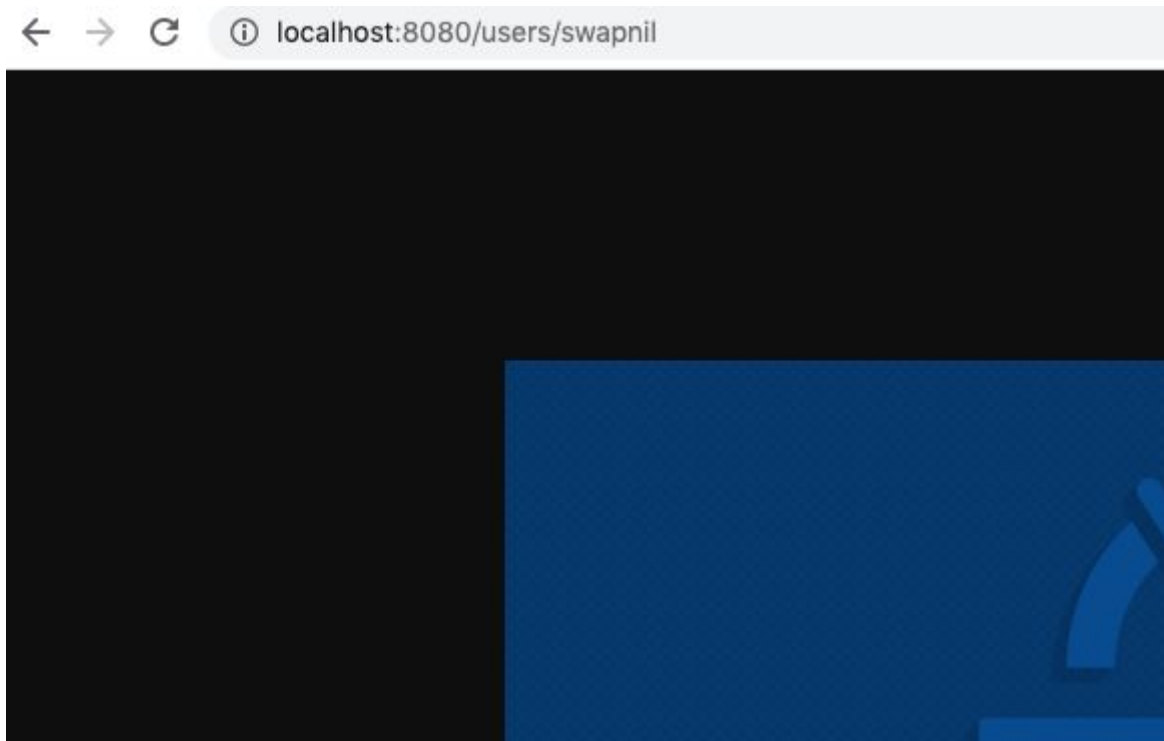
-> So it gives priority to **/redirectedurl/swapnil** in case of swapnil user served from a different location.

-> We can further rewrite the above url to serve some other request like this.

-> So i have rewritten the **/redirectedurl/swapnil to /thumb.png** so now it will not serve the request from a special location block but it will display the image.

```
rewrite ^/users/(\w+) /redirectedurl/$1;  
rewrite ^/redirectedurl/swapnil /thumb.png;  
location /redirectedurl {  
    return 200 "Request served from redirectedurl";  
}
```

```
location = /redirectedurl/swapnil {  
    return 200 "Request served specially for swapnil from this location";  
}
```

-> To stop from further rewritten & ignore any rewritten url further we will use the **last** keyword at the end of url rewrite.

```
rewrite ^/users/(\w+) /redirectedurl/$1 last;  
rewrite ^/redirectedurl/swapnil /thumb.png;  
location /redirectedurl {  
    return 200 "Request served from redirectedurl";  
}
```

```
location = /redirectedurl/swapnil {  
    return 200 "Request served specially for swapnil from this location";  
}
```

Output:



try_files

try_files path1 path2 final;

-> In try_files it will first search for the 1st path that is present in application then regardless of what uri path you have specified it will serve the request for path1 uri & if path1 doesn't exist then it will serve the request from path2 & so on.

-> if none of the path exists then it will go to the final path & for final path nginx does internal rewrite & reevaluation for processing request.

-> if you specified path1 ,path2 ,final path & location is set for path2 or some other location then the specified path will be able to serve the request.

-> path1 is always given priority in the list.

try_files /index.html /process ;

```
    set $weekdays "NO";
    if ( $date_local ~ 'Friday|friday' ) {
        set $weekdays "Yes";
    }

    location /process {
        return 200 "$host\n $uri \n $args \n $date_gmt \n $date_local \n $document_root
\n Weekdays => $weekdays";
    }

    location /logo{
        return 307 /thumb.png;
    }

    #rewrite ^/users/(\w+) /redirectedurl/$1 last;
    #rewrite ^/redirectedurl/swapnil /thumb.png;
    location /redirectedurl {
        return 200 "Request served from redirectedurl";
    }

    location = /redirectedurl/swapnil {
        return 200 "Request served specially for swapnil from this location";
    }
```

-> So over here index.html is given priority and by default request is served from this location only regardless of what uri you enter in the browser except location uri.

-> In the below output you can see I have entered <http://localhost:8080/thumb.png> but page showing is index.html.

-> but if i enter the **/redirectedurl, /redirectedurl/swapnil ,/process** it will serve the request from location blocks.

← → ↻ ⓘ localhost:8080/thumb.png



NGINX Fundamentals

Learn how to install & configure an Nginx web server from scratch.

-> In the above nginx configuration if you can see this location block over here when you enter the /logo it serves index.html page only.

```
location /logo{  
    return 307 /thumb.png;  
}
```

-> The /logo is redirected to **thumb.png** url and there is no location block for thumb.png so the request is served from 1st path i.e index.html.

-> In my nginx configuration file i have changed the index.html to index1.html now if path1 doesn't exist it will serve the request from path2.

```
try_files /index1.html /process;
```

-> I have entered thumb.png in url but data showing is of process location block.

← → ↻ ⓘ localhost:8080/thumb.png

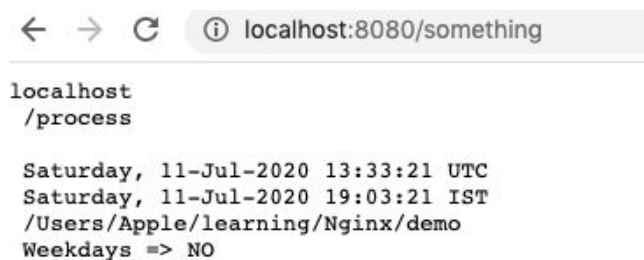
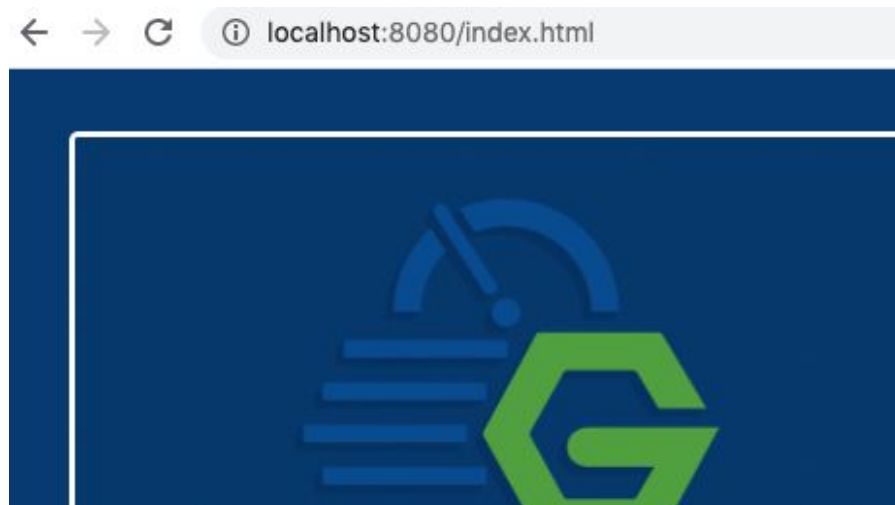
localhost
/process

Saturday, 11-Jul-2020 13:22:39 UTC
Saturday, 11-Jul-2020 18:52:39 IST
/Users/Apple/learning/Nginx/demo
Weekdays => NO

-> Now i have specified **\$uri** at first path which means serve the request from project first and if some other url entered outside project or not specified in location block the request will serve from path 2,3 or final path.

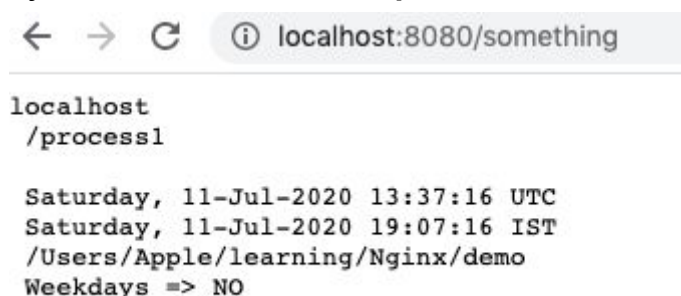
```
try_files $uri /index1.html /process ;
```

-> In the below screenshot as you can see request is served for index.html but in 2nd screenshot you can see i have entered /something and request is served from final path that is process location block because path1 is uri & its not the path of project, path2 index1.html doesn't exists so the last is final block and it has served the request.



-> So there might be a question what if the final path also doesn't exist then what?
-> I have previously specified that before processing the final url it does internal rewrite & reevaluation for processing requests.
-> I have changed the final path **process** to **process1** for which there is no location block but still request is served.
-> So to handle 404 requests we can use the final path & design a good web page for 404 pages.

try_files \$uri /index1.html /process1;



Named Location:

-> There is something called a **named** location in this revaluation is not done but a call is definitely made.

-> So we need to define it with @ & same thing in the location block as well.

try_files \$uri /index1.html /process @friendly_location_404 ;

```
location @friendly_location_404{
    return 404 "Request not found";
}
```

Logging :

-> Logging is a very important aspect of a web server whether it's an error or some malicious things happening on the server.

-> Nginx provides 2 logs type :

Access Log & Error Log

-> Error log as name suggests any error comes while serving the request then it is logged in error.log like 404 or internal server error.

-> Access log for all requests made to the server.

-> Logging is also enabled by default so in most cases leaving the log configuration as it will work but understanding how to change it is important.

-> So for some conditions like js or css error you might need to off the log or creating resource specific log can both help resource usage and help to improve.

-> So common mistake we made is 404 is logged in only error.log but not the requested uri is logged in access.log but with actual error with details is logged in error.log

```
==> access.log <==
127.0.0.1 - - [11/Jul/2020:20:34:56 +0530] "GET /missing HTTP/1.1" 404 555 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7; rv:68.0) Gecko/20100101 Firefox/68.0"

==> error.log <==
2020/07/11 20:34:56 [error] 89743#0: *71 open() "/Users/Apple/learning/Nginx/demo/missing" failed (2: No such file or directory) while serving request, client: 127.0.0.1, server: localhost, request: "GET /missing HTTP/1.1", host: "localhost:8080"
```

-> A properly handled 404 request won't produce any error like above we have handled.

-> If there is some error while configuring the nginx server apart from syntax error that too is logged in error.log.

I have tried something like this

root /Users/Apple/learning/Nginx/demo /path/demosite;

Error is logged in error.log and it is displayed in cmd line as well.

```
2020/07/11 20:44:22 [emerg] 90079#0: invalid number of arguments in "root" directive in /usr/local/etc/nginx/nginx.conf:24
```

-> For some url you might need to create custom log file

```
location /process {
    access_log /var/log/nginx/custom_access.log;
    return 200 "$host\n $uri \n $args \n $date_gmt \n $date_local \n $document_root
\n Weekdays => $weekdays";
}
```

-> As soon as you reload the server on the following path a new file is created.

```
-rw-r--r-- 1 root wheel 88341 Jul 11 20:41 access.log
-rw-r--r-- 1 root wheel 0 Jul 11 20:48 custom_access.log
-rw-r--r-- 1 root wheel 8792 Jul 11 20:48 error.log
```

-> So whenever we hit above url request is logged in custom_access.log file.

```
MacFreakss-MacBook-2:nginx Apple$ tail -f custom_access.log
127.0.0.1 - - [11/Jul/2020:20:53:17 +0530] "GET /process HTTP/1.1" 200 148 "-" "MacFreakss-MacBook-2:nginx Apple$
```

-> To disable the access log for certain location we can simply so it

access_log off

-> You can set the error log level as well according to your preferences.

For more deep learning following below urls:

http://nginx.org/en/docs/http/nginx_http_log_module.html

http://nginx.org/en/docs/nginx_core_module.html#error_log

<https://docs.nginx.com/nginx/admin-guide/monitoring/logging/>

#Inheritance & directives:

Directives:

1) Array directives: Array directive like the logging directive can be applied multiple times within the same context to specify multiple values. For example you could specify 3 different access logs by using the access_log directive with 3 different values.

2) Standard directives

3) Action directives

```
events {}

#####
# (1) Array Directive
#####
# Can be specified multiple times without overriding a previous setting
# Gets inherited by all child contexts
# Child context can override inheritance by re-declaring directive
access_log /var/log/nginx/access.log;
access_log /var/log/nginx/custom.log.gz custom_format;
```

```

http {

    # Include statement - non directive
    include mime.types;

    server {
        listen 80;
        server_name sitel.com;

        # Inherits access_log from parent context (1)
    }
    server {
        listen 80;
        server_name site2.com;

        #####
        # (2) Standard Directive
        #####
        # Can only be declared once. A second declaration overrides the first
        # Gets inherited by all child contexts
        # Child context can override inheritance by re-declaring directive
        root /sites/site2;

        # Completely overrides inheritance from (1)
        access_log off;
        location /images {

            # Uses root directive inherited from (2)
            try_files $uri /stock.png;
        }

        location /secret {
            #####
            # (3) Action Directive
            #####
            # Invokes an action such as a rewrite or redirect
            # Inheritance does not apply as the request is either stopped
            (redirect/response) or re-evaluated (rewrite)
            return 403 "You do not have permission to view this.";
        }
    }
}

```

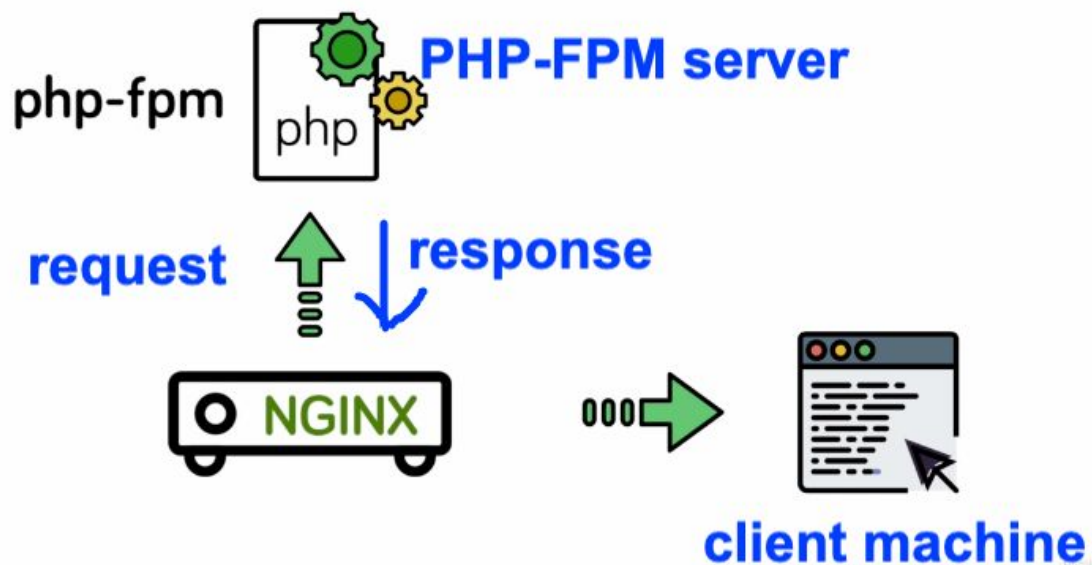
PHP Processing, php & nodejs configured :

-> Upto now we have configured the nginx to serve static content and leave the rendering of content based on its content type.

-> Nginx can't embed server side programming languages into its own process. Meaning all requests for dynamic content have to deal with a complete spread process like PHP-FPM.

-> So to handle PHP requests we need to create stand-alone php server i.e PHP-FPM to process the request & return the response to nginx server.

-> Nginx is a reverse proxy by design, and passing dynamic requests to a backend like PHP-fpm and serving the processed content back to the user is doing exactly that. Being a reverse proxy.



```
location / {  
    try_files $uri $uri/ =404 ;  
}  
location ~\.php$ {  
    include fastcgi.conf;  
    fastcgi_pass 127.0.0.1:9000; or fastcgi_pass  
unix:/run/php/php7.1-fpm.sock; (either add the php-fpm listen ip which is  
inside php-fpm config file or php-fpm.sock file path both will work)  
}
```

-> Configuring the nginx with node

```
listen 3001;  
server_name localhost;  
  
root /Users/Apple/learning/node/node_basics;
```



```
location / {
    proxy_pass http://localhost:3000/;
}
```

-> **Porxy_pass** : As you know nginx usually runs on its own express.js server for serving the request then what's the use of nginx server? Why do we need it?

-> Nginx has the ability to scale out your infrastructure. Nginx is built to handle many concurrent connections at the same time.

-> So proxy_pass basically used to divert the request to another server.

-> Proxying in Nginx is accomplished by manipulating a request aimed at the Nginx server and passing it to other servers for the actual processing. The result of the request is passed back to Nginx, which then relays the information to the client. The other servers in this instance can be remote machines, local servers, or even other virtual servers defined within Nginx. The servers that Nginx proxies requests to are known as *upstream servers*.

#Worker Processes:

-> Worker processes are used to process the request from the client.

-> As you can see in the below image there are 2 process running for nginx

nginx: master process nginx which is an actual nginx service which responds to **nginx: worker process** which is responsible for listening to the client request & returns the response.

-> The default number of processes is 1 which is shown below, you can change the worker process to any number you want from nginx.conf file but it's not that much useful .

-> Nginx's asynchronous design means a single worker can fully utilise a single CPU core. No more and no less. This directive can simply be set to 'auto' to follow this principle.

worker_processes 1

```
MacFreakss-MacBook-2:nginx-1.19.0 Apple$ ps aux | grep "nginx"
Apple      82822  0.0  0.0  4296336   580 s014  R+   10:52PM   0:00.00 grep nginx
nobody     82755  0.0  0.0  4289344    84  ??   S    10:50PM   0:00.00 nginx: worker process
root       82754  0.0  0.0  4279660    656  ??   Ss   10:50PM   0:00.00 nginx: master process nginx
Apple      81531  0.0  0.0  4267924    668 s000  S+   10:15PM   0:00.01 tail -fn 10 /var/log/nginx/access
```

-> Just to demonstrate I have set the **worker_processes 3** and you can see below its showing 3 worker processes are running.

```
MacFreakss-MacBook-2:nginx-1.19.0 Apple$ ps aux | grep "nginx"
root       82754  0.0  0.0  4289156   1400  ??   Ss   10:50PM   0:00.01 nginx: master process nginx
Apple      81531  0.0  0.0  4267924    668 s000  S+   10:15PM   0:00.01 tail -fn 10 /var/log/nginx/
Apple      84003  0.0  0.0  4277256    816 s014  S+   11:35PM   0:00.00 grep nginx
nobody     84001  0.0  0.0  4289344    880  ??   S    11:35PM   0:00.00 nginx: worker process
nobody     84000  0.0  0.0  4297536    864  ??   S    11:35PM   0:00.00 nginx: worker process
nobody     83999  0.0  0.0  4289344    872  ??   S    11:35PM   0:00.00 nginx: worker process
MacFreakss-MacBook-2:nginx-1.19.0 Apple$
```

-> As nginx processes handles the request asynchronously meaning they will handle the request as fast as hardware capabilities and creating a 2nd or 3rd server will not increase the hardware capabilities.

-> So a single worker process will run at its 100% capabilities on one cpu and adding 2nd means both processes will run at 50% of its capabilities and so on in that cpu.

-> You can check the how many cpu are there in your system

Linux : nproc or lscpu

Mac : sysctl -n hw.physicalcpu

-> There is option to set the worker process to auto which means nginx will spin the worker automatically according to cpu

worker_processes auto;

Worker_connections: this sets the number of connections each worker can accept against the cpu. And this number you can't define at its own as there is some limit that the server can accept the connections.

ulimit -n will show the max number of connections a server can accept.

-> worker_connection is defined inside the events block.

-> one more thing now we also have the maximum number of concurrent connections our server accepts.

-> which means number of **worker_processes * worker_connections = max connections.**

-> Another directives is **pid** which we defined while configuring the nginx that we can change it without rebuilding the nginx again using **pid new path.**

Buffering & Timeout

NOTE: Buffering & timeout value should be set as default but we should know it if sometime we need to tweak it.

Buffers

-> Another incredibly important tweak we can make is to the buffer size. If the buffer sizes are too low, then Nginx will have to write to a temporary file causing the disk to read and write constantly. There are a few directives we'll need to understand before making any decisions.

-> **client_body_buffer_size:** This handles the client buffer size, meaning any POST actions sent to Nginx. POST actions are typically form submissions.

-> **client_header_buffer_size:** Similar to the previous directive, only instead it handles the client header size. For all intents and purposes, 1K is usually a decent size for this directive.

-> **client_max_body_size:** The maximum allowed size for a client request. If the maximum size is exceeded, then Nginx will spit out a 413 error or *Request Entity Too Large*.

-> **large_client_header_buffers:** The maximum number and size of buffers for large client headers.

```
client_body_buffer_size 10K;  
client_header_buffer_size 1k;  
client_max_body_size 8m;  
large_client_header_buffers 2 1k;
```

Timeouts

-> Timeouts can also drastically improve performance.

-> The **client_body_timeout** and **client_header_timeout** directives are responsible for the time a server will wait for a client body or client header to be sent after request. If neither a body or header is sent, the server will issue a 408 error or *Request time out*.

-> The `keepalive_timeout` assigns the timeout for keep-alive connections with the client. Simply put, Nginx will close connections with the client after this period of time.

-> Finally, the `send_timeout` is established not on the entire transfer of answer, but only between two operations of reading; if after this time client will take nothing, then Nginx is shutting down the connection.

```
client_body_timeout 12;
client_header_timeout 12;
keepalive_timeout 15;
send_timeout 10;
```

-> `tcp_nopush` is used to tell the server that while serving the static file don't forward it to memory instead directly send it to the client.

```
sendfile on;
tcp_nopush on;
```

<https://www.digitalocean.com/community/tutorials/how-to-optimize-nginx-configuration>

Adding Dynamic Modules:

-> Adding a dynamic module is something like you want to update the existing nginx with more functionalities and all.

-> So we will rebuild the existing nginx module, if you want you can create another nginx with an updated version.

-> To get the existing configure modules run the command

nginx -V

```
MacFreaks-MacBook-2:nginx-1.19.0 Apple$ nginx -V
nginx version: nginx/1.19.0
built by clang 10.0.1 (clang-1001.0.46.4)
configure arguments: --sbin-path=/usr/local/bin/nginx --conf-path=/usr/local/etc/nginx/nginx.conf --error-log-path=/var/log/nginx/error.log --http-log-path=/var/log/nginx/access.log --with-pcre --pid-path=/var/run/nginx.pid
```

./configure --help

-> Will list all the modules & flags available that you can configure with nginx. This list consists of all modules static or inbuilt modules & dynamic modules.

-> to list only dynamic modules use grep command with help command shown below.

./configure --help | grep dynamic

-> While configuring the dynamic modules if the dependent library is not present then you need to install the dependent library first then start again.

-> i have added the image filter module & we have specified the module path as well where **ngx_http_image_filter_module.so** is created.

--with-http_image_filter_module=dynamic --modules-path=/usr/local/etc/nginx/modules

-> Once we are done with the configuration you can check the module is integrated properly using the same command **nginx -V** in configure image filter module will also be there.

-> And now to use the image module we need to first load that module in **nginx.conf** file.

load_module <path_to_module>.

load_module /usr/local/etc/nginx/modules/ngx_http_image_filter_module.so;

-> So we can configure nginx without downtime.

Headers & Expires:

- > Expire headers are useful especially while serving the static requests.
- > So there is an image which will not change for a specific amount of time so by setting up the expire headers nginx tells the browser that cache this image for a specified amount of time and not make server requests & serve directly from the browser.
- > Like that we can set the expire headers for css file as but for a short amount of time as css might get updated but we can do that as well.
- > So I have configured the nginx expire headers for js ,css, jpg, jpeg, png.

```
location ~* \.(css|jpg|jpeg|png|js) {  
    add_header Cache-Control public; (telling the receiving client that this response  
can be cached in anyway)  
    add_header Pragma public;(Older version of this cache-control header)  
    add_header Vary Accept-Encoding;(vary meaning the content of this response vary  
)  
    expires 1M; (expire is the time you can set 30m(minute), 60m(minute),  
1h(hour),1M(month)  
}
```

Output :

As we have the set the header expires after 1 month the same you can check the Expires Value.

```
[MacFreakss-MacBook-2:nginx-1.19.0 Apple$ curl -I http://localhost:3001/helper.js  
HTTP/1.1 200 OK  
Server: nginx/1.19.0  
Date: Wed, 15 Jul 2020 05:15:15 GMT  
Content-Type: application/javascript  
Content-Length: 984  
Last-Modified: Sat, 04 Jul 2020 16:00:40 GMT  
Connection: keep-alive  
ETag: "5f00a7a8-3d8"  
Expires: Fri, 14 Aug 2020 05:15:15 GMT  
Cache-Control: max-age=2592000  
Cache-Control: public  
Pragma: public  
Vary: Accept-Encoding  
Accept-Ranges: bytes
```

Compressed Responses with gzip:

- > With gzip compression we can send the static content by compressing the content and making it in lesser byte as compared to original size.
- > As most of the modern browsers have the ability to decompress the compressed static content. So while making a request in the header browser will send the Accept-Encoding:gzip so that we can send the response in gzip format.

```
gzip on;  
gzip_comp_level 4;  
gzip_types text/css;  
gzip_types text/javascript;
```

curl -I -H "Accept-Encoding: gzip" <http://localhost:3001/helper.js>

FASTCGI Caching/Proxy Caching

Note: FASTCGI caching is used in php and Proxy caching is used for Nodejs.

-> As commonly you know, the Caching technique is used to reduce the server load or cache some data so that processing time will save and we can directly serve the request to the client.

-> Nginx cache mechanism also works the same so we can cache some data in micro cache so that nginx doesn't make contact with PHP-FPM server or node server and serve the request immediately.

PHP cache configuration:

Inside http context

```
# Configure microcache (fastcgi)

http {
    fastcgi_cache_path /tmp/nginx_cache levels=1:2 keys_zone=ZONE_1:100m inactive=60m;
    fastcgi_cache_key "$scheme$request_method$host$request_uri";
    add_header X-Cache $upstream_cache_status;
}

server {
    # Cache by default
    set $no_cache 0;

    # Check for cache bypass
    if ($arg_skipcache = 1) {
        set $no_cache 1;
    }

    location ~\.php$ {
        # Pass php requests to the php-fpm service (fastcgi)
        include fastcgi.conf;
        fastcgi_pass unix:/run/php/php7.1-fpm.sock;

        # Enable cache
        fastcgi_cache ZONE_1;
        fastcgi_cache_valid 200 60m;
        fastcgi_cache_bypass $no_cache;
        fastcgi_no_cache $no_cache;
    }
}
```

Nodejs cache configuration :

As node js used the proxy mechanism whereas php FASTCGI mechanism.

Inside http context

```
http {
    proxy_cache_path /usr/local/etc/nginx/cache levels=1:2 keys_zone=backcache:8m
    max_size=50m;
    proxy_cache_key "$request_method$host$request_uri";
    add_header X-Proxy-Cache $upstream_cache_status;

    server {
        set $no_cache 0;
        if ($arg_skippingcache = 1) {
            set $no_cache 1;
        }
        location / {
            proxy_pass "http://localhost:3000/";
            proxy_cache backcache;
            proxy_cache_valid 200 302 10m;
            proxy_cache_valid 404 1m;
            proxy_cache_bypass $no_cache;
            proxy_no_cache $no_cache;
        }
    }
}
```

proxy_cache_path/fastcgi_cache_path : directive, we have defined a directory on the filesystem where we would like to store our cache.

Levels: parameter specifies how the cache will be organized. Nginx will create a cache key by hashing the value of a key (configured below). The levels we selected above dictate that a single character directory (this will be the last character of the hashed value) with a two character subdirectory (taken from the next two characters from the end of the hashed value) will be created. You usually won't have to be concerned with the specifics of this, but it helps Nginx quickly find the relevant values shown in the screenshot below.

```
/tmp/nginx_cache
├── 2
│   ├── 13
│   │   └── 30ea56f1e9fd583e5b5f4afcd3194132
│   ├── 19
│   │   └── c9c7336be416ea846f4a1fa208200192
│   ├── 5c
│   │   └── 1255078577a61d2f1a0a707f14dd55c2
│   └── 62
│       └── d3d5bfee66483e71a40ff3e77e207622
```

Keys_zone: parameter defines the name for this cache zone, which we have called backcache/ZONE_1. This is also where we define how much metadata to store. In this case, we are storing 8 MB of keys. For each megabyte, Nginx can store around 8000 entries. The max_size parameter sets the maximum size of the actual cached data.

proxy_cache_key/fastcgi_cache_key: This is used to set the key that will be used to store cached values. This same key is used to check whether a request can be served from the cache. We are setting this to a combination of the scheme (http or https), the HTTP request method, as well as the requested host and URI.

```
$scheme$request_method$host$request_uri
```

https://GETdomain.com/blog/article

64CA8686873CA2055EE55F7BAA61961D

X-Proxy-Cache \$upstream_cache_status: We also added an extra header called X-Proxy-Cache. We set this header to the value of the \$upstream_cache_status variable. Basically, this sets a header that allows us to see if the request resulted in a cache hit, a cache miss, or if the cache was explicitly bypassed. This is especially valuable for debugging, but is also useful information for the client.

Proxy_cache: We need to set the keys_zone value here.

Proxy_cache_valid: The proxy_cache_valid directive can be specified multiple times. It allows us to configure how long to store values depending on the status code. In our example, we store successes and redirects for 10 minutes, and expire the cache for 404 responses every minute.

Proxy_cache_bypass: proxy_cache_bypass values tell the nginx whether to bypass cache or not. As in our example if value is 1 then we need to bypass the cache else not.

Proxy_no_cache: proxy_no_cache specifies whether to store the cache or not.

Refer below url for more detail on understanding the above parameters:

<https://www.digitalocean.com/community/tutorials/understanding-nginx-http-proxying-load-balancing-buffering-and-caching>

Apache Benchmark:

-> I have used the apache benchmark for checking the cache response.
Below screenshot are from apache benchmark.

NOTE: For mac apache benchmark is preinstalled.

ab -n 1000 -c 100 http://localhost:3001/getdata

I have run the apache benchmark for 1000 connections at 100 concurrent requests.

Benchmark before cache: Request per second: 135.52

```
Benchmarking localhost (be patient)
Completed 100 requests
Completed 200 requests
Completed 300 requests
Completed 400 requests
Completed 500 requests
Completed 600 requests
Completed 700 requests
Completed 800 requests
Completed 900 requests
Completed 1000 requests
Finished 1000 requests

Server Software:      nginx/1.19.0
Server Hostname:      localhost
Server Port:          3001

Document Path:        /getdata
Document Length:       537 bytes

Concurrency Level:     100
Time taken for tests:   7.379 seconds
Complete requests:      1000
Failed requests:         0
Total transferred:      768000 bytes
HTML transferred:       537000 bytes
Requests per second:    135.52 [#/sec] (mean)
Time per request:       737.925 [ms] (mean)
Time per request:       7.379 [ms] (mean, across all concurrent requests)
Transfer rate:          101.64 [Kbytes/sec] received
```

Benchmark after cache: Request per second 6547.89

```
Completed 900 requests
Completed 1000 requests
Finished 1000 requests

Server Software:      nginx/1.19.0
Server Hostname:      localhost
Server Port:          3001

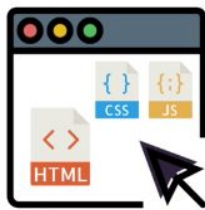
Document Path:        /getdata
Document Length:       537 bytes

Concurrency Level:     100
Time taken for tests:   0.153 seconds
Complete requests:      1000
Failed requests:         0
Total transferred:      768000 bytes
HTML transferred:       537000 bytes
Requests per second:    6547.89 [#/sec] (mean)
Time per request:       15.272 [ms] (mean)
Time per request:       0.153 [ms] (mean, across all concurrent requests)
Transfer rate:          4910.92 [Kbytes/sec] received
```


HTTP2 & SSL CERTIFICATE

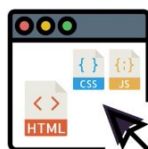
- > HTTP2 is binary protocol whereas HTTP1 is textual protocol.
- > **Binary protocol**: Binary data(0110101) is a far more compact way of transferring data & it greatly reduces the chance of error while transferring data.
- > **Compressed Header** : compresses response header which again reduces the transfer time.
- > **Persistent connections**: HTTP2 uses persistence connection.
- > **Multiplex Streaming** : And those persistence connections are used in multiplex streaming meaning that multiple aspects such as **html, css, js** can be combined in one binary data and transmit it in a single connection. As HTTP1 requires dedicated connection for each resource or aspects.
- > To communicate both server and client need to send the header e.g TLS handshake and then connection opens.
- > As shown below HTTP1 uses 3 connections to transfer the data.
- > So the number of files increases the number of connections will increase and browsers too have its limit to open connections.

HTTP 1.1



Connections: 3

- > Whereas in HTTP2 it uses a single connection to transfer all data which is multiplexed streaming.



HTTP 2
Connections: 1

NOTE: To use HTTP2 we need to enable the SSL.

NOTE: Before loading http2 module & enabling ssl certificate check the http1.1 is loaded just to compare after the http2 module, so check in browsers console check in network.

-> To enable the http2 module we need to load the HTTP2 module from configure as we load the modules.

-> As we know how to load the module previously it is explained.

nginx -V will the previous ./configure modules add **--with-http_v2_module** init and rest all previous steps make & make install.

--with-http_v2_module

-> After we load the module http2 we will generate the self sign certificate i.e test certificate.

-> Create the dir where you want to store the ssl certificate.

MAC:

Openssl installed path. For me it is here

-> `cd /usr/local/Cellar/openssl/1.0.2t/bin`

-> **Next run the openssl command which will open the openssl shell.**

-> **Run below command.**

`req -x509 -days 10 -nodes -newkey rsa:2048 -keyout /usr/local/etc/nginx/ssl/self.key -out /usr/local/etc/nginx/ssl/self.crt`

Ubuntu :

Directly run the below command

`openssl req -x509 -days 10 -nodes -newkey rsa:2048 -keyout /usr/local/etc/nginx/ssl/self.key -out /usr/local/etc/nginx/ssl/self.crt`

To understand about above command visit below link

<https://www.digitalocean.com/community/tutorials/how-to-create-a-self-signed-ssl-certificate-for-nginx-in-ubuntu-16-04>

-> On the execution of above command a self sign certificate is generated in the mentioned folder.

```
MacFreakss-MacBook-2:bin Apple$ cd /usr/local/etc/nginx/ssl/
MacFreakss-MacBook-2:ssl Apple$ ls -l
total 16
-rw-r--r--  1 Apple  admin  1424 Jul 19 21:20 self.crt
-rw-r--r--  1 Apple  admin  1704 Jul 19 21:20 self.key
MacFreakss-MacBook-2:ssl Apple$ pwd
/usr/local/etc/nginx/ssl
MacFreakss-MacBook-2:ssl Apple$
```

-> Load ssl certificate & enable the http2 module.

server {

listen 443 ssl http2;(port 443 for https, ssl to use ssl certificate, http2 module)

server_name localhost; (server_name either domain name or ip)

root /Users/Apple/learning/node/node_basics;

ssl_certificate /usr/local/etc/nginx/self.crt; (this is self generated ssl certificate)
ssl_certificate_key /usr/local/etc/nginx/self.key; (this is self generated ssl key)

-> Reload or restart the nginx and you can see the connection is secure and in the browser console's network section you can see http2 is used.

-> So if in some cases the browser doesn't support the http2 and need not to worry as it will automatically fallback to http1.

-> As we have not set anything on port 80 i.e if someone requests without https then the website will not open. To do so we have 2 methods:

-> Listen on port 80 which is not a feasible solution.

-> Permanent Redirect to https. It's Feasible solution.

-> Below we have enabled some of the flags

Ssl_protocols : Over here we have disabled the ssl handshake mechanism and enable the TLS handshake mechanism which is more secure.

```
# Disable SSL
ssl_protocols TLSv1 TLSv1.1 TLSv1.2;

# Optimise cipher suits
ssl_prefer_server_ciphers on;
ssl_ciphers ECDH+AESGCM:ECDH+AES256:ECDH+AES128:DH+3DES:!ADH:!AECDH:!MD5;

# Enable DH Params
ssl_dhparam /etc/nginx/ssl/dhparam.pem;

# Enable HSTS
add_header Strict-Transport-Security "max-age=31536000" always;

# SSL sessions
ssl_session_cache shared:SSL:40m;
ssl_session_timeout 4h;
ssl_session_tickets on;
```

#Rate Limiting :

-> It allows you to limit the amount of HTTP requests a user can make in a given period of time.

-> A request can be as simple as a `GET` request for the homepage of a website or a `POST` request on a login form.

-> Rate limiting can be used for security purposes, for example to slow down brute-force password-guessing attacks.

-> To test Rate limiting we are using **SIEGE**.

-> Siege is mostly like Apache benchmark but it's used for load testing.

-> Siege is an http load testing and benchmarking utility. It was designed to let web developers measure their code under duress, to see how it will stand up to load on the internet.

-> Install Siege on your system & run the below command

Link to install siege on mac

<https://jasonmccreary.me/articles/installing-siege-mac-os-x-lion/>

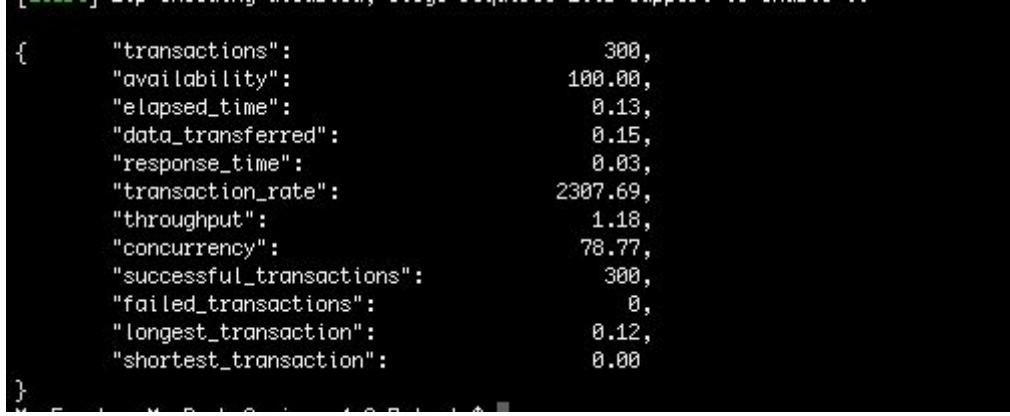
For ubuntu you can directly do it from the command line.

siege -v -r 3 -c 100 http://localhost:3001/getdata

-r : repetitions

-c : concurrent connection

-v : Verbose output. With this option selected, siege will print transaction information to the screen.



```
{
  "transactions": 300,
  "availability": 100.00,
  "elapsed_time": 0.13,
  "data_transferred": 0.15,
  "response_time": 0.03,
  "transaction_rate": 2307.69,
  "throughput": 1.18,
  "concurrency": 78.77,
  "successful_transactions": 300,
  "failed_transactions": 0,
  "longest_transaction": 0.12,
  "shortest_transaction": 0.00
}
```

Defining Ratelimit

-> **limit_req_zone \$request_uri zone=myzone:10m rate=1r/s;**

Key – Defines the request characteristic against which the limit is applied.

\$request_uri or server_name or \$binary_remote_addr

Zone – Defines the shared memory zone used to store the state of each IP address and how often it has accessed a request-limited URL.

Rate – Sets the maximum request rate. In our example we set 1 request per second.

-> In the location block we used the above zone

limit_req zone=myzone;

```

#define Rate limiting
limit_req_zone $request_uri zone=myzone:10m rate=1r/s;

server {
    listen 3001;
    server_name localhost;

    root /Users/Apple/learning/node/node_basics;

    #ssl_certificate /usr/local/etc/nginx/self.crt;
    #ssl_certificate_key /usr/local/etc/nginx/self.key;

    #index index.php index.html;
    set $no_cache 0;
    if ($arg_skippingcache = 1) {
        set $no_cache 1;
    }
    location / {
        proxy_pass "http://localhost:3000/";
        proxy_cache backcache;
        proxy_cache_valid 200 302 10m;
        proxy_cache_valid 404 1m;
        proxy_cache_bypass $no_cache;
        proxy_no_cache $no_cache;
        #add_header X-Proxy-Cache $upstream_cache_status;
        limit_req zone=myzone burst=10;
    }
}

```

Output: first output is without burst so after the 1st request reset all failed as all requests fell under 1sec. In the second burst is set which means the buffer all requests are successful but it takes time. Burst basically slowing down the connections.

```

[MacFreakss-MacBook-2:siege-4.0.7 Apple$ siege -v -r 3 -c 5 http://localhost:3001/getdata
[alert] Zip encoding disabled; siege requires zlib support to enable it

{
  "transactions":          1,
  "availability":         6.67,
  "elapsed_time":         0.06,
  "data_transferred":     0.00,
  "response_time":        0.10,
  "transaction_rate":    16.67,
  "throughput":           0.05,
  "concurrency":          1.67,
  "successful_transactions": 1,
  "failed_transactions":   14,
  "longest_transaction":  0.06,
  "shortest_transaction": 0.00
}
[MacFreakss-MacBook-2:siege-4.0.7 Apple$ siege -v -r 3 -c 5 http://localhost:3001/getdata
[alert] Zip encoding disabled; siege requires zlib support to enable it

{
  "transactions":          15,
  "availability":        100.00,
  "elapsed_time":        14.00,
  "data_transferred":    0.01,
  "response_time":       4.00,
  "transaction_rate":    1.07,
  "throughput":          0.00,
  "concurrency":         4.29,
  "successful_transactions": 15,
  "failed_transactions":   0,
  "longest_transaction":  5.02,
  "shortest_transaction": 0.07
}
[MacFreakss-MacBook-2:siege-4.0.7 Apple$

```

For more information on rate limiting follow below URL as rate limiting has end less parameters: <https://www.nginx.com/blog/rate-limiting-nginx/>

#Basic Auth

-> Basic Auth is used where application is for internal use only and no outsider is allowed to access.

-> To do so we need to generate the password. So here i have used the apache utils feature of htpasswd.

htpasswd -c /usr/local/etc/nginx/.htpasswd user1

User1 is the user & .htpasswd is hidden file in that directory

```
[MacFreakss-MacBook-2:siege-4.0.7 Apple$ htpasswd -c /usr/local/etc/nginx/.htpasswd user1
New password:
Re-type new password:
Adding password for user user1
[MacFreakss-MacBook-2:siege-4.0.7 Apple$ cd /usr/local/etc/nginx/
[MacFreakss-MacBook-2:nginx Apple$ cat .htpasswd
user1:$apr1$9CwPXyIG$YoVPxWw3sX2kHEMLrP.WD.
[MacFreakss-MacBook-2:nginx Apple$ pwd
```

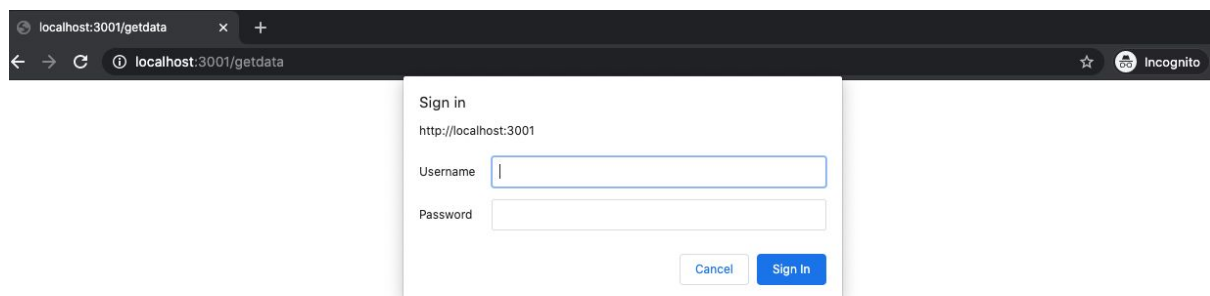
-> Inside location block i have written this code

auth_basic "Secure Area";

auth_basic_user_file /usr/local/etc/nginx/.htpasswd; => Path to password file

```
location / {
    proxy_pass "http://localhost:3000/";
    proxy_cache backcache;
    proxy_cache_valid 200 302 10m;
    proxy_cache_valid 404 1m;
    proxy_cache_bypass $no_cache;
    proxy_no_cache $no_cache;
    #add_header X-Proxy-Cache $upstream_cache_status;
    #limit_req zone=myzone burst=10;
    auth_basic "Secure Area";
    auth_basic_user_file /usr/local/etc/nginx/.htpasswd;
}
```

-> Now when you hit the url you will be prompted for entering the username & password.



#Hardening Nginx

-> So when you do the curl request with header it gives the nginx version as well.
-> And whenever nginx releases some security patch we need to update it. This can be found on nginx website change log.
-> So the attacker can only attack on the previous version of nginx and from the curl request attacker can get to know the version of nginx, so we need to hide that.
-> As you can see below the nginx version is visible on making the curl request.
curl -lk http://localhost:3001/getdata

```
HTTP/1.1 200 OK
Server: nginx/1.19.0
Date: Wed, 12 Aug 2020 05:54:51 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 537
Connection: keep-alive
X-Powered-By: Express
ETag: W/"219-sKN2ay2bnkamclXuMBcpjYmjEwQ"
X-Proxy-Cache: MISS
```

-> To hide the nginx version we need to off the server token in http context.

server_tokens off;

This is an important step in hardening your install as malicious users won't know which version of Nginx you are running and thus won't be able to target your version of Nginx with 'known' vulnerabilities.

Reload the nginx and hit again. Now the nginx version is hidden.

```
curl -lk http://localhost:3001/getdata
HTTP/1.1 200 OK
Server: nginx
Date: Wed, 12 Aug 2020 07:07:19 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 537
Connection: keep-alive
X-Powered-By: Express
ETag: W/"219-sKN2ay2bnkamclXuMBcpjYmjEwQ"
X-Proxy-Cache: MISS
```

-> Some might take the website url and load it in xframe on some other side. We can stop that too by setting the header

add_header X-Frame-Options "SAMEORIGIN";

-> And it wont load it on any other server whose origin is not matched.

#Reverse Proxy :

-> Proxying is typically used to distribute the load among several servers, seamlessly show content from different websites, or pass requests for processing to application servers over protocols other than HTTP.

-> Reverse proxy is basically acting as an intermediate between client and other server.

-> Like we for PHP we have PHP-FPM or for nodejs we have express server running we just pass the request to that server, server processes the request returns the response to nginx and nginx forwards the request to client.

```
Location / {  
    proxy_pass "http://localhost:3000/";  
}
```

For more details visit below url:

<https://docs.nginx.com/nginx/admin-guide/web-server/reverse-proxy/>

#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.

-> **round-robin**: Nginx load balancing uses the round-robin requests to the application servers are distributed in a round-robin fashion.

-> Inside Upstream context we need to define all the servers of the same application and call that same upstream in the proxy pass as shown below.

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

-> **Sticky session** : there are scenarios like user login where user session should stick to one session for that request we can achieve that using the sticky session or persistent session.

To enable sticky session just ip_hash inside upstream block

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



```
server srv3.example.com;  
}
```

-> **Least connection** : Another load balancing discipline 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.

To enable Least connection just least_conn inside upstream block.

-> 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.

For more details follow below URL :

http://nginx.org/en/docs/http/load_balancing.html