2025-11-17

Research on Nginx, its features, and documentation.

Table of Contents

1. Install NGINX, Serve Content, Configure Reverse-proxy

Install NGINX, serve static content from a document root using `root` and `index`, and configure NGINX as a reverse proxy that forwards requests to an internal backend service while preserving important client headers.

Prerequisites

- Docker installed
- Docker Compose installed
- Basic terminal access
- Project directory containing `nginx.conf`, `conf.d/`, `html/`, `backend/`

1.1 Docker-Compose Setup

This project uses Docker Compose to run NGINX as the main web server and to serve backend content.

The `docker-compose.yml` mounts:

- `html/` → for static content

2025-11-17

- `conf.d/` → for NGINX configs
- `backend/app/` → backend response for reverse proxy

docker-compose.yml

```yaml
version: "3.9"

services:
  nginx:
    image: nginx:latest
    container_name: nginx-wu
    ports:
      - "8080:80"
      - "8443:443"
    volumes:
      - ./nginx.conf:/etc/nginx/nginx.conf:ro
      - ./conf.d:/etc/nginx/conf.d
      - ./html:/usr/share/nginx/html
      - ./log:/var/log/nginx
    networks:
      - nginx-net
    restart: unless-stopped

  backend:
    image: nginx:alpine
    container_name: nginx-backend
    volumes:
      - ./backend/app:/usr/share/nginx/html
    expose:
      - "80"
    networks:
      - nginx-net

networks:
  nginx-net:
```

1.2 Install & Start NGINX (using Docker Compose)

Run:

```
docker-compose up -d
```

This automatically:

- pulls the NGINX container
- reads your configs
- serves files from local folders

Check that the container is running:

```
docker ps
```

1.3 Serving Static Files (root + index)

2025-11-17

The static files are placed inside:

```
html/
   ├── index.html
   └── images/
```

Your `default.conf` tells NGINX to:

- use `/usr/share/nginx/html` as `root`
- load `index.html` as the landing page

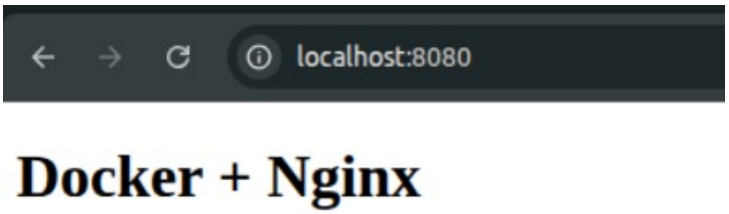
```
server {
    listen       80;
    listen  [::]:80;
    server_name  localhost;

    #access_log  /var/log/nginx/host.access.log  main;

    location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
    }
```

Test in browser:

http://localhost:8080



1.4 Reverse Proxy Setup

The reverse proxy is configured inside `default.conf` (or `reverse-proxy.conf`).

`reverse-proxy.conf`:

2025-11-17

```
server {
    listen 80;
    server_name localhost;

    location /app/ {
        proxy_pass http://backend/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        add_header X-Proxied-By "Nginx-Reverse-Proxy 123";
    }
}
```

This means:

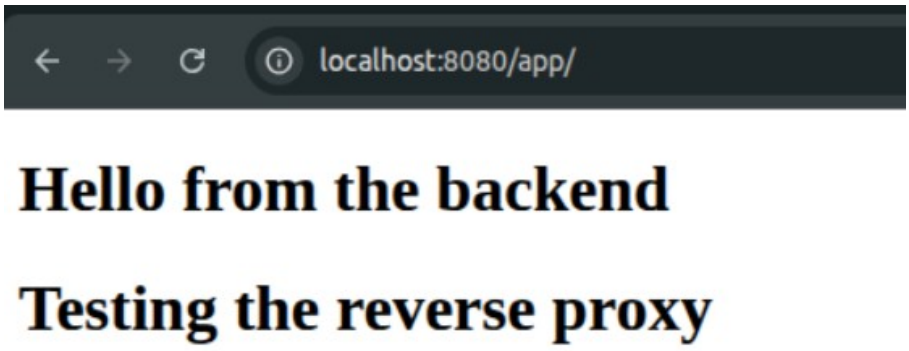
- any request to `/app/`
  → is forwarded to the backend container
- headers are forwarded so backend knows client details

Backend files are inside:

```
backend/app/
   └── index.html
```

Test:

http://localhost:8080/app/



2025-11-17

2. Setup Load Balancer (round robin, least_conn & ip_hash)

For the Load balancing setup, first create 3 backend servers using docker-compose. Update your docker-compose and make the volumes directories.

Clean Up:
Before moving onto to the next part, make sure you cleanup the previous configs to avoid any confusions and conflicts. If you have made changes in the default.conf file, rename it to default.conf.bak (backup file) so it does not conflicts with the load-balancing configs.

2.1 Round Robin

Create the file with the **Round Robin** config:

```
upstream backend_group {
    server backend1:80;
    server backend2:80;
    server backend3:80;
}

server {
    listen 80;
    server_name localhost;

    # Serve static files from the root
    location / {
        root /usr/share/nginx/html;
        index index.html index.htm;
    }

    # Proxy requests to /app/ to the backend group
    location /app/ {
        proxy_pass http://backend_group/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
    }
}
```

Apply changes

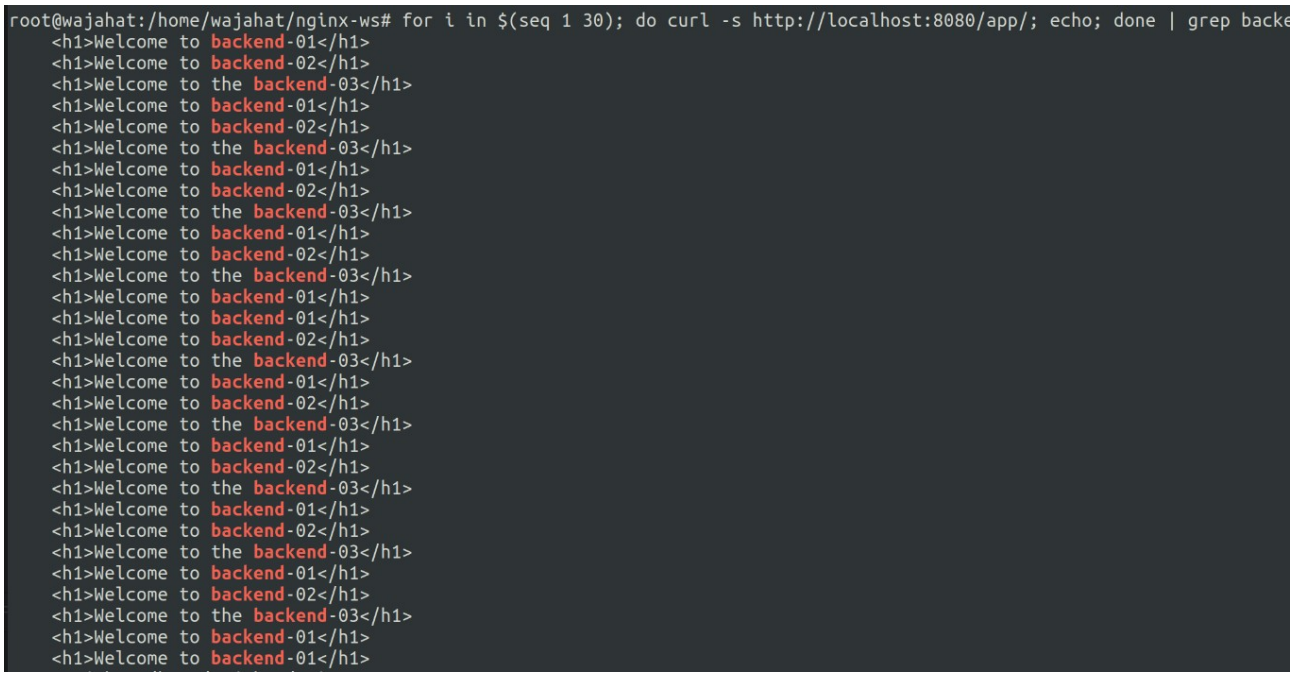
```
docker-compose down && docker-compose up -d
```

Test

for i in $(seq 1 30); do curl -s http://localhost:8080/lb/; echo; done | grep backend

Expected Behavior: Outputs alternate among Backend 1 / Backend 2 / Backend 3.

2025-11-17



2.2 Least-conn

Edit `conf.d/load-balancer.conf` , change the `upstream` block:

```
# LEAST CONNECTIONS
upstream backend_group {
    least_conn;
    server backend1:80;
    server backend2:80;
    server backend3:80;
}
```

- Keep the same `server` block below it as in Round Robin (location `/lb/`).
- NGINX sends the next request to the backend with the fewest active connections.

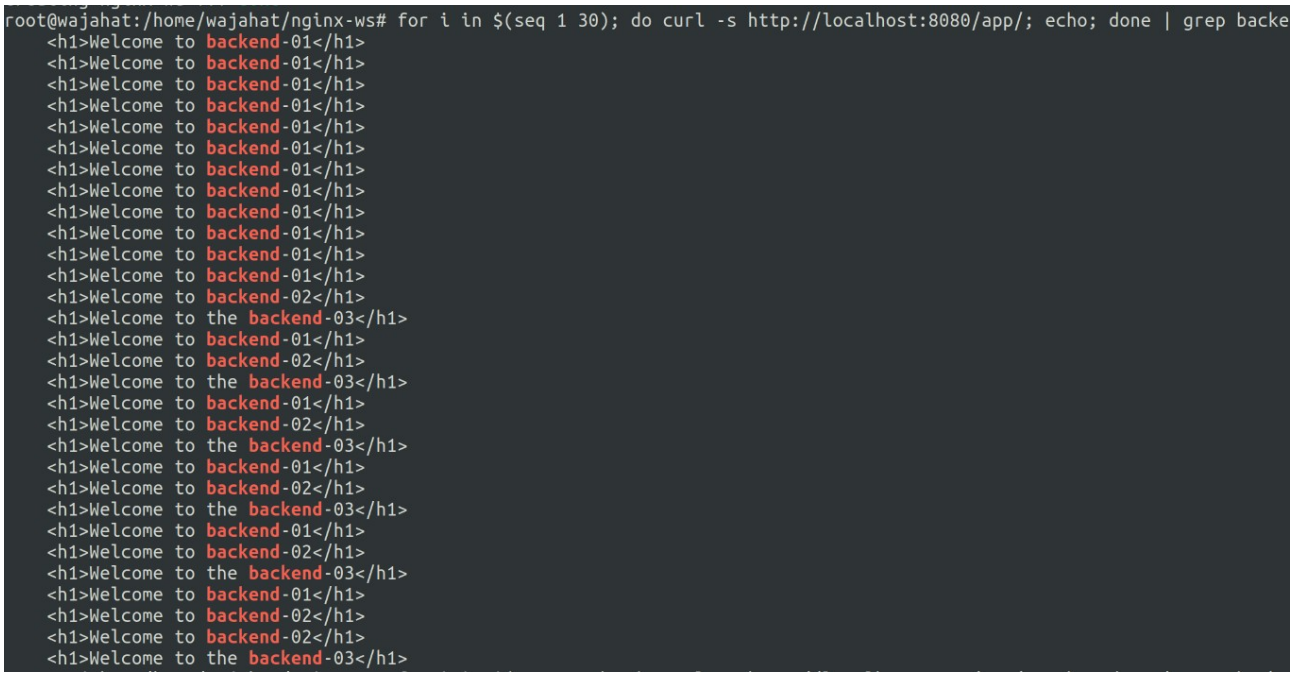
Apply changes

```
docker-compose down && docker-compose up -d
```

Test

for i in $(seq 1 30); do curl -s http://localhost:8080/lb/; echo; done | grep backend

Expected Behavior: If one backend responds slower, NGINX sends more traffic to the faster server, balancing active connections rather than request count.



2.3 ip_hash

Edit `conf.d/load-balancer.conf` — change the `upstream` block:

```
# IP HASH - session sticky by client IP
upstream backend_group {
    ip_hash;
    server backend1:80;
    server backend2:80;
    server backend3:80;
}
```

NGINX chooses the backend based on the client IP hash. Requests from the same client IP consistently go to the same backend.

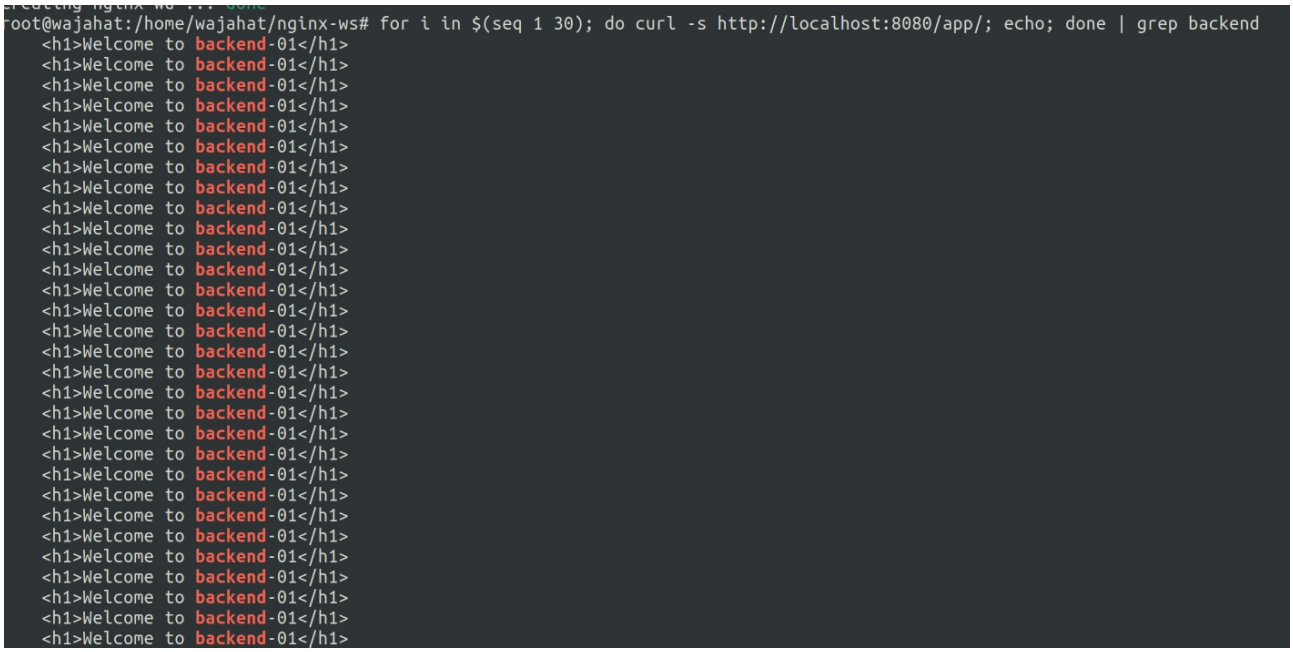
Apply changes

`docker-compose down && docker-compose up -d`

Test

- From the same client machine (same public IP), repeated requests should hit the same backend:

  ```
  for i in $(seq 1 20); do curl -s http://localhost:8080/lb/; echo; done  | grep backend
  ```



Expected Behavior: All requests from your IP consistently hit the same backend server unless NGINX restarts or the upstream configuration changes.

Useful when you need session affinity without backend-side session sharing.

Clean Up:
Before moving onto to the next part, make sure you cleanup the previous configs to avoid any confusions and conflicts. If you have made changes in the load-balancer.conf file, rename it to load-balancer.conf.bak (backup file) so it does not conflicts with the next configs. Also rename the default.conf to default.conf.bak

3. Enable SSL/TLS Using a Self-Signed Certificate

Step 1 : Create a Directory for Certificates

Create a directory on your local machine to store the certificate and private key:

```
mkdir certs
```

This directory will later be mounted into the NGINX container.

Step 2 : Generate a Self-Signed Certificate and Private Key

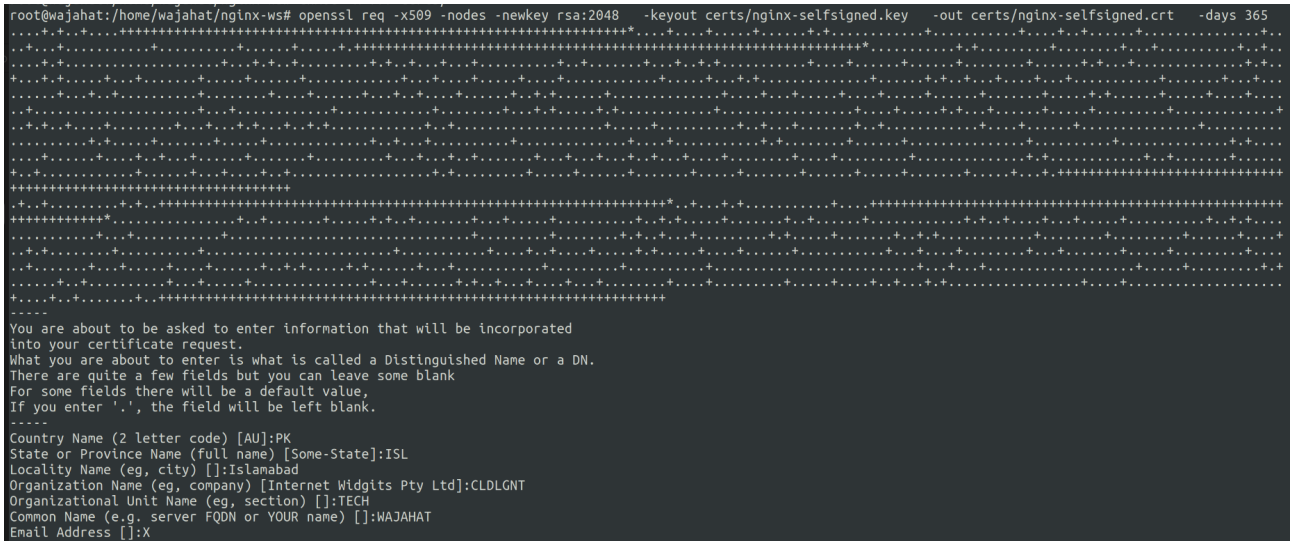
Use OpenSSL to generate a certificate valid for 365 days:

```
openssl req -x509 -nodes -days 365 \
  -newkey rsa:2048 \
  -keyout certs/nginx-selfsigned.key \
  -out certs/nginx-selfsigned.crt \
  -subj "/CN=localhost"
```

This produces two files:

- `nginx-selfsigned.key` — the private key
- `nginx-selfsigned.crt` — the certificate presented during TLS handshake

Both must be readable by the NGINX container.



Step 3 : Mount Certificates into the NGINX Container

Modify the `nginx` service in `docker-compose.yml`:

```
services:
  nginx:
    ...
    volumes:
      - ./certs:/etc/nginx/certs:ro
```

This binds your local `certs/` directory to `/etc/nginx/certs` inside the container.

Step 4 : Create the SSL Server Block

`conf.d/ssl.conf`

and add the following configuration:

2025-11-17

```
# HTTP → HTTPS REDIRECT (Port 80)
server {
    listen 80;
    listen [::]:80;
    server_name localhost;
    # Redirect all HTTP traffic to HTTPS
    return 301 https://$host$request_uri;
}

# HTTPS SERVER BLOCK (Port 443)
server {
    listen 443 ssl;
    listen [::]:443 ssl;
    server_name localhost;

    ssl_certificate     /etc/nginx/certs/nginx-selfsigned.crt;
    ssl_certificate_key /etc/nginx/certs/nginx-selfsigned.key;
    ssl_protocols TLSv1.2 TLSv1.3;
    ssl_prefer_server_ciphers on;
    ssl_ciphers HIGH:!aNULL:!MD5;

    location / {
        root   /usr/share/nginx/html;
        index  index.html index.htm;
    }
    # REVERSE PROXY (HTTPS)
    location /app/ {
        proxy_pass http://backend/;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;

        add_header X-Proxied-By "Nginx-Reverse-Proxy";
    }
    # ERROR PAGES (HTTPS)
    error_page 500 502 503 504 /50x.html;
    location = /50x.html {
        root /usr/share/nginx/html;
    }
}
```

Explanation:

1. HTTP → HTTPS Redirect Block

- Listens on port 80 (plain HTTP).
- Instantly redirects every request to HTTPS using a `301` redirect.
- Ensures users always land on the secure version of the site.
- `$host$request_uri` preserves the original domain + path.

2. HTTPS Server Block

- Listens on port **443**, which is the standard TLS/HTTPS port.
- The `ssl` parameter activates TLS handling for this server block.

3. TLS Certificate & Security Settings

Directive	Purpose
`ssl_certificate`	Public certificate to prove server identity.
`ssl_certificate_key`	Private key to decrypt traffic (keep secret!).
`ssl_protocols TLSv1.2 TLSv1.3`	Enforces modern, secure protocols.

2025-11-17

Directive	Purpose
`ssl_prefer_server_ciphers on`	Uses the server's preferred ciphers (stronger encryption).
`ssl_ciphers HIGH:!aNULL:!MD5`	Disables weak ciphers and authentication methods.

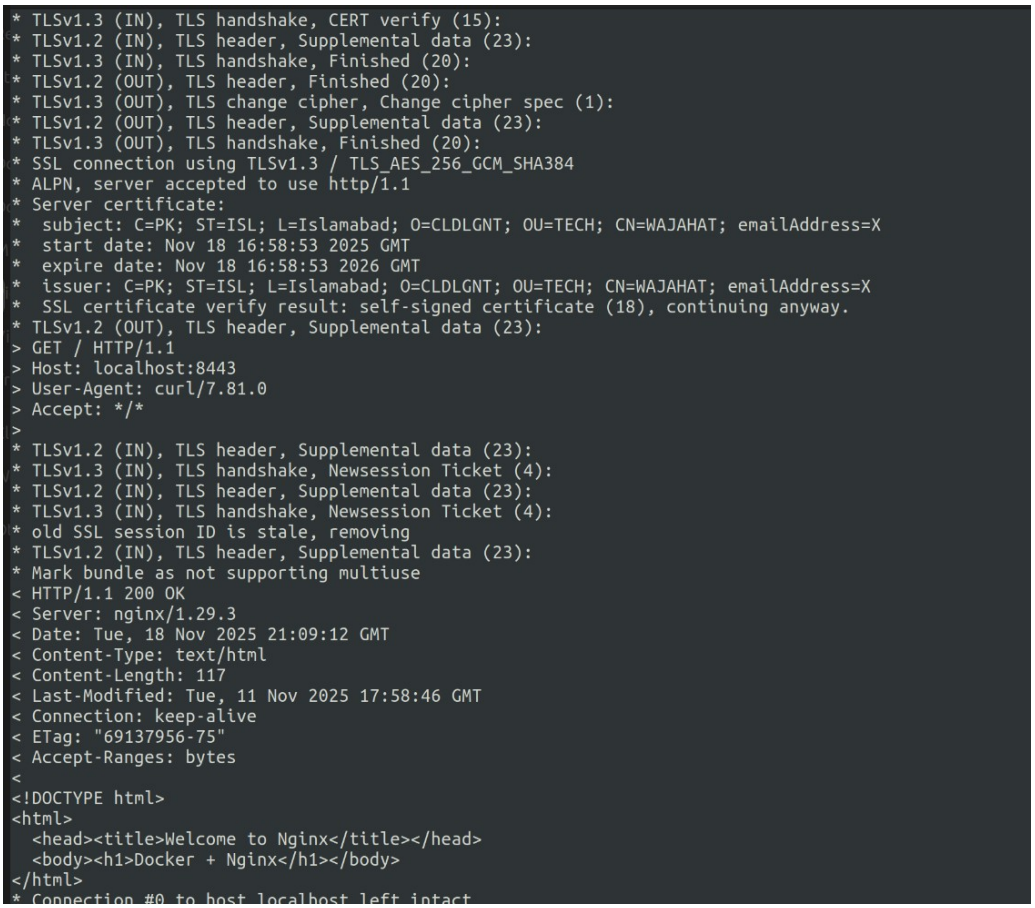
Step 5 : Apply changes

```
docker-compose down && docker-compose up -d
```

Step 6 : Test HTTPS Access

Test using curl (insecure mode because it's self-signed):

```
curl -vk https://localhost
```



```
* TLSv1.3 (IN), TLS handshake, CERT verify (15):
* TLSv1.2 (IN), TLS header, Supplemental data (23):
* TLSv1.3 (IN), TLS handshake, Finished (20):
* TLSv1.2 (OUT), TLS header, Finished (20):
* TLSv1.3 (OUT), TLS change cipher, Change cipher spec (1):
* TLSv1.2 (OUT), TLS header, Supplemental data (23):
* TLSv1.3 (OUT), TLS handshake, Finished (20):
* SSL connection using TLSv1.3 / TLS_AES_256_GCM_SHA384
* ALPN, server accepted to use http/1.1
* Server certificate:
*  subject: C=PK; ST=ISL; L=Islamabad; O=CLDLGNT; OU=TECH; CN=WAJAHAT; emailAddress=X
*  start date: Nov 18 16:58:53 2025 GMT
*  expire date: Nov 18 16:58:53 2026 GMT
*  issuer: C=PK; ST=ISL; L=Islamabad; O=CLDLGNT; OU=TECH; CN=WAJAHAT; emailAddress=X
*  SSL certificate verify result: self-signed certificate (18), continuing anyway.
* TLSv1.2 (OUT), TLS header, Supplemental data (23):
> GET / HTTP/1.1
> Host: localhost:8443
> User-Agent: curl/7.81.0
> Accept: */*
>
* TLSv1.2 (IN), TLS header, Supplemental data (23):
* TLSv1.3 (IN), TLS handshake, Newsession Ticket (4):
* TLSv1.2 (IN), TLS header, Supplemental data (23):
* TLSv1.3 (IN), TLS handshake, Newsession Ticket (4):
* old SSL session ID is stale, removing
* TLSv1.2 (IN), TLS header, Supplemental data (23):
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< Server: nginx/1.29.3
< Date: Tue, 18 Nov 2025 21:09:12 GMT
< Content-Type: text/html
< Content-Length: 117
< Last-Modified: Tue, 11 Nov 2025 17:58:46 GMT
< Connection: keep-alive
< ETag: "69137956-75"
< Accept-Ranges: bytes
<
<!DOCTYPE html>
<html>
  <head><title>Welcome to Nginx</title></head>
  <body><h1>Docker + Nginx</h1></body>
</html>
* Connection #0 to host localhost left intact
```

Or test using OpenSSL client:

```
openssl s_client -connect localhost:443
```

You should see certificate information and a successful TLS handshake.

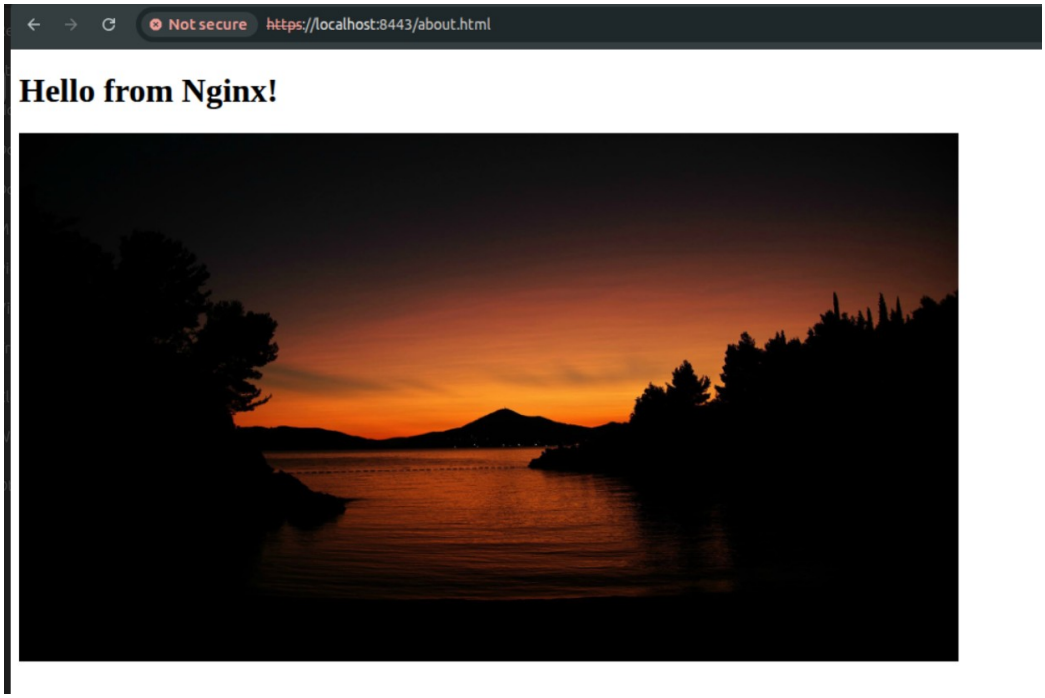
2025-11-17

```
root@wajahat:/home/wajahat/nginx-ws# openssl s_client -connect localhost:8443 -servername localhost
CONNECTED(00000003)
depth=0 C = PK, ST = ISL, L = Islamabad, O = CLDLGNT, OU = TECH, CN = WAJAHAT, emailAddress = X
verify error:num=18:self-signed certificate
verify return:1
depth=0 C = PK, ST = ISL, L = Islamabad, O = CLDLGNT, OU = TECH, CN = WAJAHAT, emailAddress = X
verify return:1
---
Certificate chain
 0 s:C = PK, ST = ISL, L = Islamabad, O = CLDLGNT, OU = TECH, CN = WAJAHAT, emailAddress = X
   i:C = PK, ST = ISL, L = Islamabad, O = CLDLGNT, OU = TECH, CN = WAJAHAT, emailAddress = X
   a:PKEY: rsaEncryption, 2048 (bit); sigalg: RSA-SHA256
   v:NotBefore: Nov 18 16:58:53 2025 GMT; NotAfter: Nov 18 16:58:53 2026 GMT
---
Server certificate
-----BEGIN CERTIFICATE-----
MIIDyTCCArGgAwIBAgIUL1Zte7EP/dDOiUl727yxhEZFdg4wDQYJKoZIhvcNAQEL
BQAwdDELMAkGA1UEBhMCUEsxDDAKBgNVBAgMA0lTTDESMBAGA1UEBwwJSXNsYW1h
YmFkMRAwDgYDVQQKDAdTERMR05UMQ0wCwYDVQQLDARURUNIMRAwDgYDVQQDDAdX
QUpBSEFUMRAwDgYJKoZIhvcNAQkBFgFYMB4XDTI1MTExODE2NTg1M1oXDTI2MTEx
ODE2NTg1M1owdDELMAkGA1UEBhMCUEsxDDAKBgNVBAgMA0lTTDESMBAGA1UEBwwJ
SXNsYW1hYmFkMRAwDgYDVQQKDAdTERMR05UMQ0wCwYDVQQLDARURUNIMRAwDgYD
VQQDDAdXQUpBSEFUMRAwDgYJKoZIhvcNAQkBFgFYMIIBIjANBgkqhkiG9w0BAQEF
AAOCAQ8AMIIBCgKCAQEAzOWdNVqGttijgaXiixpBMKS96+9HNjNGZ1AlAeuTd9tz
OPTC0I+lOqIfPP8f0XAE6Z1/xzoNtguzQqyyQmTaVz4OLgX0Fc5IG/h31AK2pCze
Qkx6Fkl7EDafwSDF9HmEBSe3OyD2Bhi2eBdAUNNdWjigmSLfY5fbdxs7X9OSbGa5
qHtYVCSkhtFeKftzzMwVQiowiVSGhOrh2fwTUNCploZjupyAHGIa4kREFs6hOHSu
8IpHY5h8Ohy/tw8uW8Btd0PbTZpVz8UFx6vS4ZF6+El2dAwAq1cmK0LxmjzFh7FA
wF0l+6zcTkxBK66kfosEgVXzEqEYxSMy55dHRPHt1wIDAQABo1MwUTAdBgNVHQ4E
FgQUN9QrXGCWNvJLaxKTdDUMHpPUWW8wHwYDVR0jBBgwFoAUN9QrXGCWNvJLaxKT
dDUMHpPUWW8wDwYDVR0TAQH/BAUwAwEB/zANBgkqhkiG9w0BAQsFAAOCAQEAfp1B
jH2W9tmEKFpzaAfr/SlRYMoa2WIpoGUB1hav5l3kHdmyenafiSmwkwxFAT0tMTFj
c5rPn8hgHIzP+Zf8fGxPYhO0mXvP9xD127GQTGhwse+8Ota8bvyPya8ldaPjByoA
afstR/2A1QzKjRS2OR8p0DXUEzmDL49KpoQii5K1NE3HXosAx5ZlVOrhbQIJ5C2r
Nn+3jwmTl9qhOOg9JOUl+fS7tbbr8Qzc+n0GvK2kpzVsv/KkAxrCbWTS5Dw7qWQT
ljTmaEf5oECmaehgqIOVVhtV038IGAAkNLGIXxT3ypRNJvA9cYYkrLGKIm5uASJT
Rs2lMgdRUJwYpbsmuQ==
```

Test in a browser:
Visit:

`https://localhost:8443`

(If using port mapping to 8443.)



The browser will warn about a self-signed certificate, which is expected.

2025-11-17

4. Implement Security Headers in NGINX (HSTS, CSP, XSS, Frame, MIME-Types)

Security headers help protect applications from common web vulnerabilities.
They enforce browser-level security rules that reduce attack surface without changing your application code.

Configuration (Insert Into ssl.conf)

Put these directly below your SSL directives, before any `location` blocks.

```
# Enable in HTTPS server block
add_header Strict-Transport-Security "max-age=31536000; includeSubDomains" always;
add_header Content-Security-Policy "default-src 'self';";
add_header X-XSS-Protection "1; mode=block";
add_header X-Frame-Options "DENY";
add_header X-Content-Type-Options "nosniff";
```

Explanation

1. Strict-Transport-Security (HSTS)

- Tells browsers to always use HTTPS, even if the user types `http://`.
- Prevents protocol downgrade attacks.
- `always` ensures header is included even on error pages.

2. Content-Security-Policy (CSP)

- Defines what resources the browser is allowed to load.
- Using `"default-src 'self'"` limits everything (scripts, images, styles) to same origin unless explicitly allowed.
- Strong protection against XSS.

3. X-XSS-Protection

- Enables browser's XSS filter (useful for older browsers).
- `"1; mode=block"` blocks the page if an attack is detected.

4. X-Frame-Options

- `"DENY"` prevents your website from being embedded in any iframe.
- Protects against clickjacking.

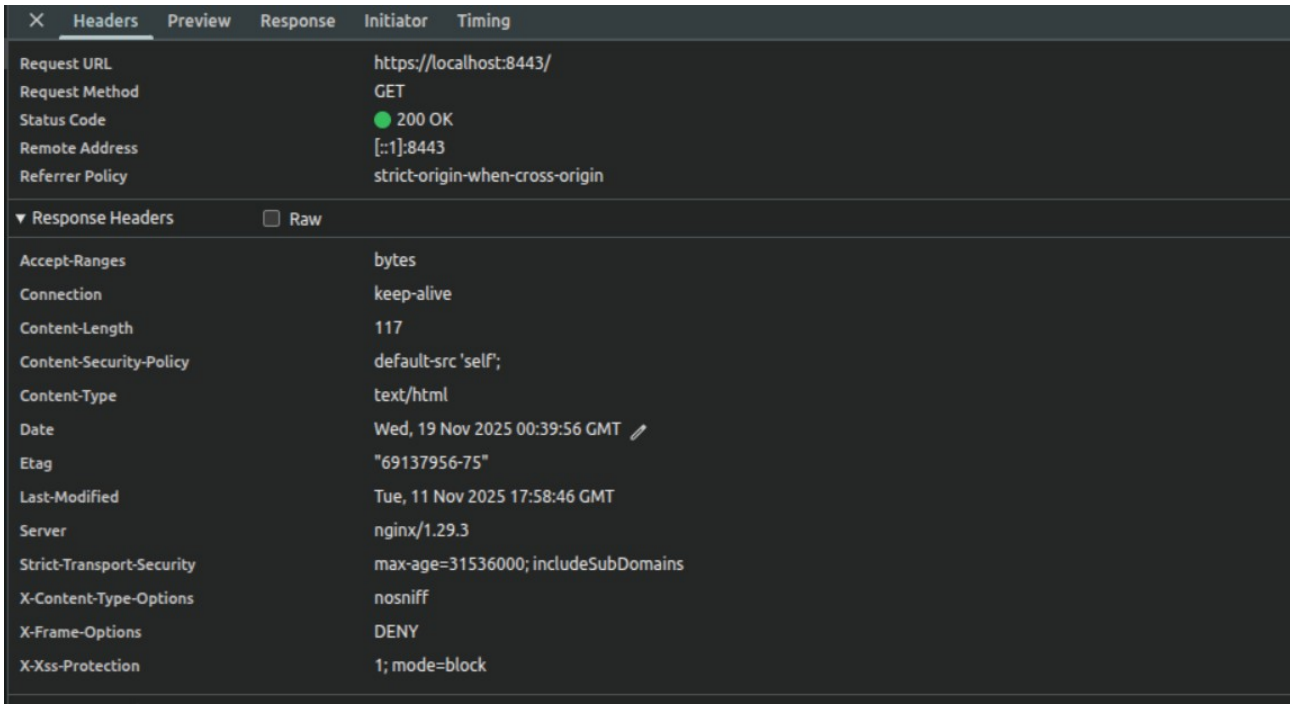
5. X-Content-Type-Options

- `"nosniff"` tells browsers not to guess MIME types.
- Prevents malicious files pretending to be harmless (e.g., `.jpg` that is actually JS).

2025-11-17

Apply changes
```
docker-compose down && docker-compose up -d
```

Test with curl
```
curl -vk https://localhost:8443/
```



4.b Implement Basic Authentication

Basic Authentication restricts access to an NGINX location or entire server block using a username and password. It provides a simple method to prevent unauthorized access to private services, dashboards, or internal sites.

Step 1 : Install htpasswd util
```
sudo apt install apache2-utils
```

Step 2 : Create auth directory
Inside your project:

```
mkdir auth
```

This directory will store the authentication file used by NGINX.

Step 3 : Create user credentials

2025-11-17

Run the following to create a password file and an initial user:

```
htpasswd -c ./auth/.htpasswd <username>
```

- `-c` creates the file.
- The command will prompt for a password.
- Credentials are stored in encrypted form inside `.htpasswd`.

```
root@wajahat:/home/wajahat/nginx-ws/conf.d# htpasswd -c ./auth/.htpasswd wajahat
New password:
Re-type new password:
Adding password for user wajahat
```

Step 4 : Mount auth to docker-compose

In `docker-compose.yml`, mount the directory so NGINX can access it:

```
volumes:
  - ./auth:/etc/nginx/auth:ro
```

This ensures:

- NGINX sees the `.htpasswd` file at runtime.
- File is read-only for safety.

Step 5 : NGINX configuration

In the desired server or location block (for example inside `ssl.conf` or `default.conf`):

```
location /app/ {

    # basic authentication
    auth_basic "Restricted Area!";
    auth_basic_user_file /etc/nginx/auth/.htpasswd;
```

Here the backend app is being password protected.

This configuration instructs NGINX to:

- Prompt users with a browser login dialog.
- Validate credentials using the `.htpasswd` file.

Step 6 : Validate & apply

Check whether NGINX syntax is valid:

2025-11-17

```
docker compose exec nginx nginx -t
```

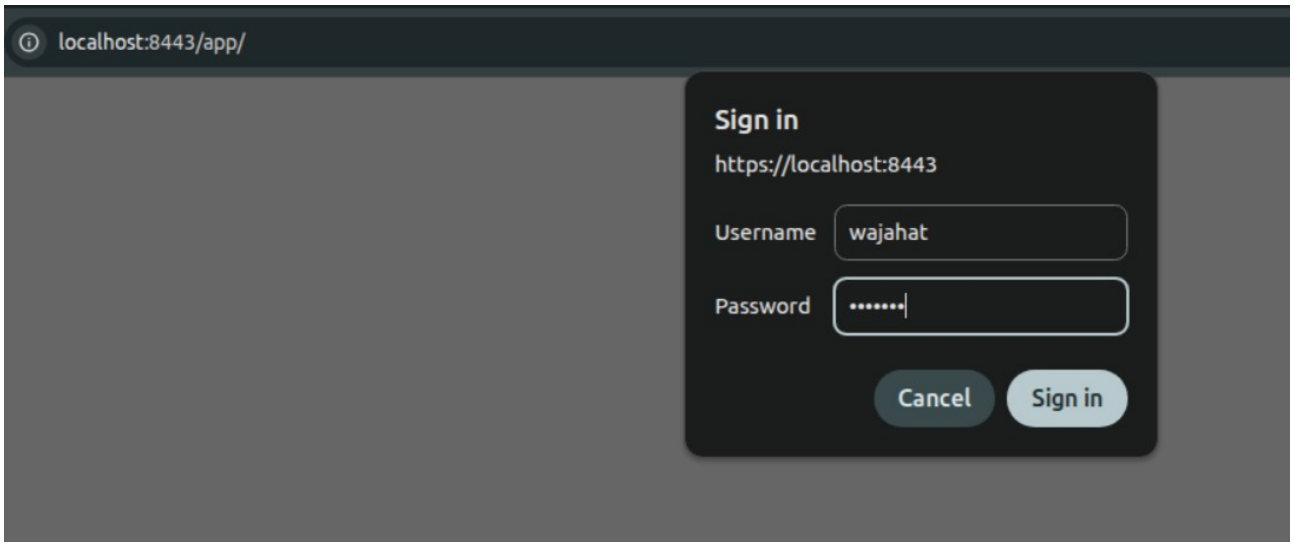
If the output contains:

```
syntax is ok
test is successful
```

Apply the changes
```
docker-compose down && docker-compose up -d
```

Step 7 : Test the setup

Navigate to https://localhost:8443:/app/ it will prompt for username and password



Expected Behavior

- Access requires correct username/password.
- Incorrect credentials return `401 Unauthorized`.
- Log entries appear in `access.log` and `error.log` reflecting authentication attempts.

5. Implement proxy_cache

Step 1 – Create Cache Directory

Create a directory on the host machine to store cached responses:

```
mkdir cache
```

This directory will be mounted into the NGINX container.

2025-11-17

Step 2 – Mount Cache Directory in Docker Compose

Add a new volume in the NGINX service (example):

`./cache:/var/cache/nginx`

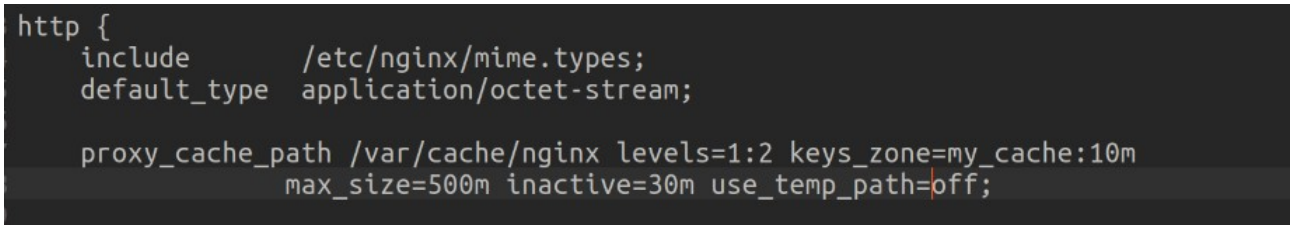
This ensures NGINX has a persistent place to store cached files.

Step 3 – Define Cache Zone in nginx.conf

Inside the `http` block of `nginx.conf`, define:

```
proxy_cache_path /var/cache/nginx levels=1:2 keys_zone=my_cache:10m;
```

This creates a cache zone named `my_cache`.

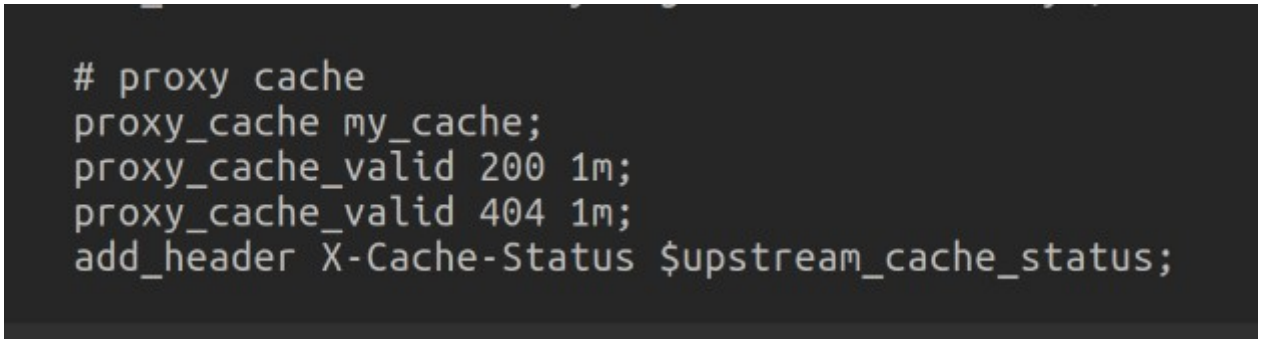


Step 4 – Enable Caching in reverse-proxy Configuration

Inside `reverse-proxy.conf`, within the `/app/` location:

```
proxy_cache my_cache;
proxy_cache_valid 200 1m;
proxy_cache_valid 404 1m;
add_header X-Cache-Status $upstream_cache_status;
```

This enables caching and adds a response header showing HIT/MISS status.



Step 5 – Restart Containers

Apply configuration changes:

```
docker compose down && docker compose up -d
```

Step 6 – Test Caching

First request (expect MISS)

2025-11-17

```
curl -I http://localhost:8080/app/
```

Expected:

```
X-Cache-Status: MISS
```

Second request (expect HIT)

```
curl -I http://localhost:8080/app/
```

Expected:

```
X-Cache-Status: HIT
```

6. Implement rate limiting, connection limits and buffering settings

1. Rate Limiting
Under you http block nside your nginx.conf, add:

```
http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;

    # rate limiting
    limit_req_zone $binary_remote_addr zone=req_limit_zone:10m rate=10r/s;
```

Add the `limit_req` directive to the `server` block in `ssl.conf` to enforce rate limiting on specific locations (e.g., `/app/`). Depending on your config, you can add it into your reverseproxy.conf file as well:

```
# rate limiting
limit_req zone=req_limit_zone burst=10 nodelay;
```

Directives

- `limit_req_zone`:
    - **Purpose**: Defines a shared memory zone to track request rates.
    - **Example**: `limit_req_zone $binary_remote_addr zone=req_limit_zone:10m rate=10r/s;`

    - **Explanation**: Tracks requests by client IP (`$binary_remote_addr`), allows 10 requests per second (`rate=10r/s`), and uses 10MB of memory (`zone=req_limit_zone:10m`).
- `limit_req`:
    - **Purpose**: Applies rate limiting to a specific location.
    - **Example**: `limit_req zone=req_limit_zone burst=20;`
    - **Explanation**: Uses the `req_limit_zone` zone and allows bursts of 20 requests.

2. Connection Limits
Add to **`nginx.conf`** in the **`http`** block

```
http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;

    # connection limits
    limit_conn_zone $binary_remote_addr zone=conn_limit_zone:10m;
```

Add the **`limit_conn`** directive to the **`server`** block in `reverse-proxy.conf` to enforce connection limits:

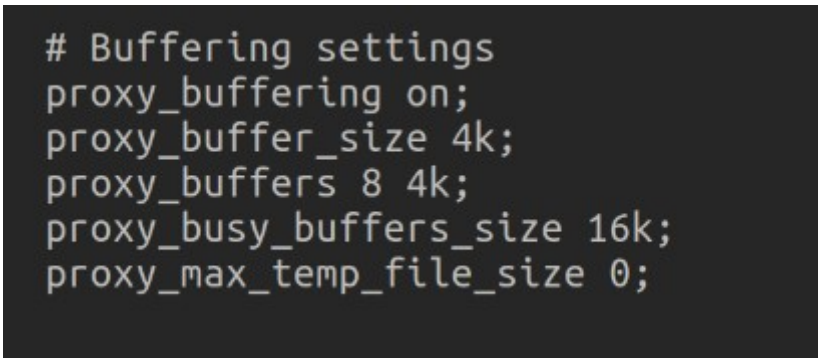
```
    # connection limits
    limit_conn conn_limit_zone 10;
```

Directives
- `limit_conn_zone`:
    - **Purpose**: Defines a shared memory zone to track connections.
    - **Example**: `limit_conn_zone $binary_remote_addr zone=conn_limit_zone:10m;`
    - **Explanation**: Tracks connections by client IP (`$binary_remote_addr`) and uses 10MB of memory (`zone=conn_limit_zone:10m`).
- `limit_conn`:
    - **Purpose**: Applies connection limits to a specific location.
    - **Example**: `limit_conn conn_limit_zone 10;`
    - **Explanation: Limits each client IP to 10 simultaneous connections using the** `conn_limit_zone` **zone.**

3. Buffering Settings

2025-11-17

Add buffering settings to the `location /app/` block in **reverse-proxy.conf**



Directives

- `proxy_buffering`:
  - **Purpose**: Enables or disables buffering of responses from the backend.
  - **Example**: `proxy_buffering on;`
  - **Explanation**: Enables buffering of responses from the backend server.
- `proxy_buffer_size`:
  - **Purpose**: Sets the size of the buffer used for reading the response from the backend.
  - **Example**: `proxy_buffer_size 4k;`
  - **Explanation**: Sets the buffer size to 4KB.
- `proxy_buffers`:
  - **Purpose**: Sets the number and size of the buffers used for reading a response from the backend.
  - **Example**: `proxy_buffers 8 4k;`
  - **Explanation**: Uses 8 buffers, each of size 4KB.
- `proxy_busy_buffers_size`:
  - **Purpose**: Limits the total size of buffers that can be busy sending a response to the client.
  - **Example**: `proxy_busy_buffers_size 16k;`
  - **Explanation**: Limits busy buffers to a total size of 16KB.
- `proxy_max_temp_file_size`:
  - **Purpose**: Sets the maximum size of temporary files used when buffering large responses.
  - **Example**: `proxy_max_temp_file_size 0;`
  - **Explanation: Disables temporary files for buffering by setting the size to 0.**

Apply changes

nginx -t # Test the configuration

systemctl reload nginx

7. Integrate PHP using PHP-FPM and fastcgi_pass

`Steps`

1. Update `docker-compose.yml`
Add a PHP-FPM service to handle PHP processing.

```yaml
php-fpm:
  image: php:fpm-alpine
  container_name: php-fpm
  volumes:
    - ./html:/var/www/html
  networks:
    - nginx-net
  expose:
    - "9000"
```

2. Configure NGINX for PHP-FPM
Create a new configuration file `php.conf` in the `conf.d` directory to handle PHP requests.

```
server {
    listen 80;
    server_name localhost;

    root /var/www/html;
    index index.php index.html index.htm;

    location / {
        try_files $uri $uri/ =404;
    }

    location ~ \.php$ {
        fastcgi_pass php-fpm:9000;
        fastcgi_index index.php;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        include fastcgi_params;
    }
}
```

2025-11-17

3. Create a PHP Test File

Create a simple PHP file in the `html` directory to test the setup.

`html/index.php:`

```
<?php
phpinfo();
?>
```

4. Restart Services

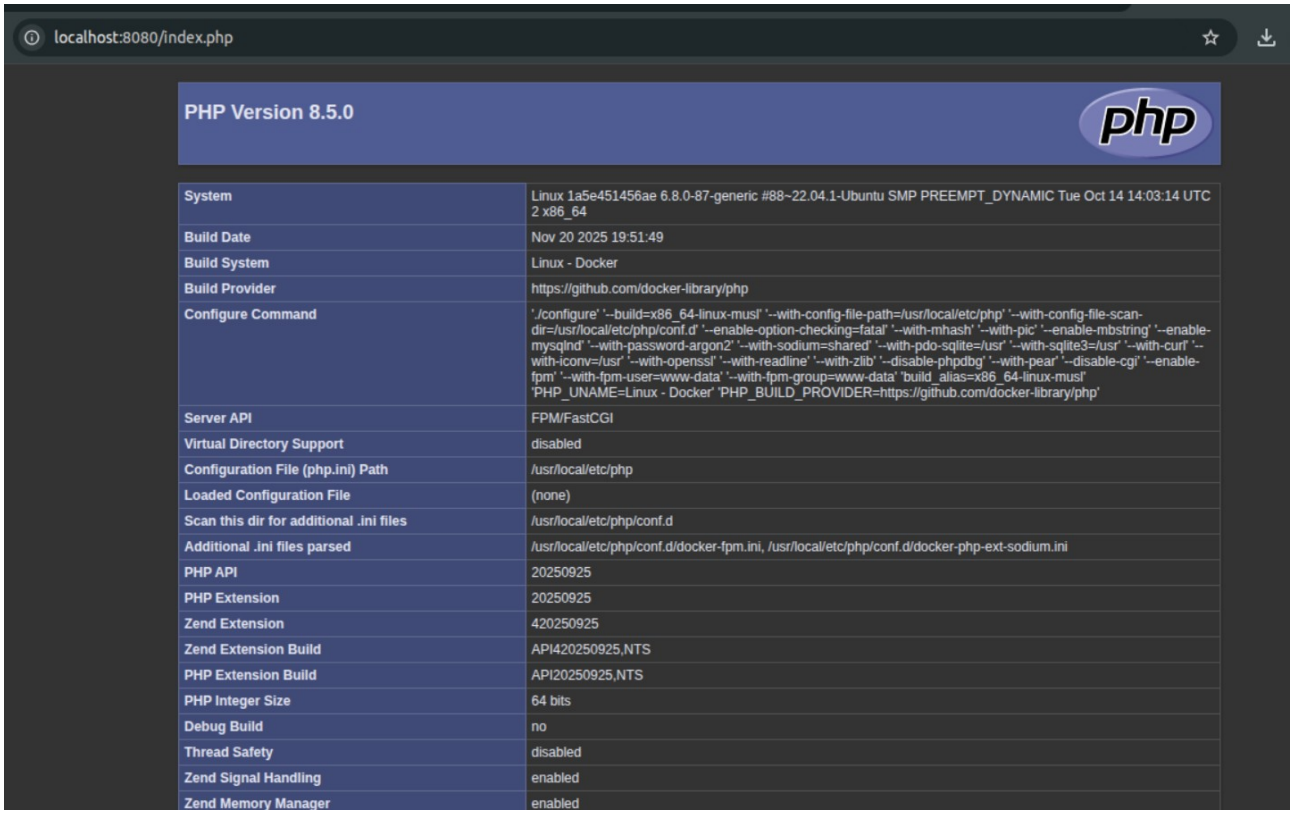
Restart your Docker containers to apply the changes.

```
Docker-compose down && docker-compose up -d
```

5. Test PHP Integration

Access `http://localhost:8080/index.php` in your browser. You should see the PHP info page, confirming that PHP is working.

2025-11-17



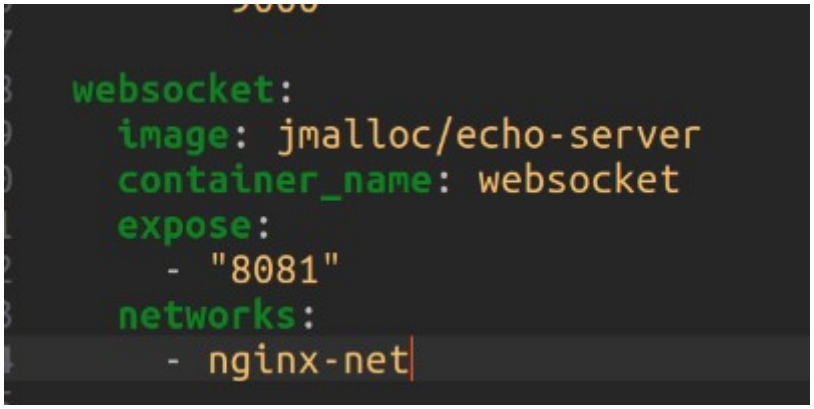
System	Linux 1a5e451456ae 6.8.0-87-generic #88~22.04.1-Ubuntu SMP PREEMPT_DYNAMIC Tue Oct 14 14:03:14 UTC 2 x86_64
Build Date	Nov 20 2025 19:51:49
Build System	Linux - Docker
Build Provider	https://github.com/docker-library/php
Configure Command	'./configure' '--build=x86_64-linux-musl' '--with-config-file-path=/usr/local/etc/php' '--with-config-file-scan-dir=/usr/local/etc/php/conf.d' '--enable-option-checking=fatal' '--with-mhash' '--with-pic' '--enable-mbstring' '--enable-mysqlnd' '--with-password-argon2' '--with-sodium=shared' '--with-pdo-sqlite=/usr' '--with-sqlite3=/usr' '--with-curl' '--with-iconv=/usr' '--with-openssl' '--with-readline' '--with-zlib' '--disable-phpdbg' '--with-pear' '--disable-cgi' '--enable-fpm' '--with-fpm-user=www-data' '--with-fpm-group=www-data' 'build_alias=x86_64-linux-musl' 'PHP_UNAME=Linux - Docker' 'PHP_BUILD_PROVIDER=https://github.com/docker-library/php'
Server API	FPM/FastCGI
Virtual Directory Support	disabled
Configuration File (php.ini) Path	/usr/local/etc/php
Loaded Configuration File	(none)
Scan this dir for additional .ini files	/usr/local/etc/php/conf.d
Additional .ini files parsed	/usr/local/etc/php/conf.d/docker-fpm.ini, /usr/local/etc/php/conf.d/docker-php-ext-sodium.ini
PHP API	20250925
PHP Extension	20250925
Zend Extension	420250925
Zend Extension Build	API420250925,NTS
PHP Extension Build	API20250925,NTS
PHP Integer Size	64 bits
Debug Build	no
Thread Safety	disabled
Zend Signal Handling	enabled
Zend Memory Manager	enabled

8. WebSocket Support in NGINX

WebSockets allow a persistent two-way connection between client and server. Unlike normal HTTP requests, a WebSocket request must upgrade the connection protocol. Therefore, NGINX must explicitly pass the `Upgrade` and `Connection` headers to the backend WebSocket server.

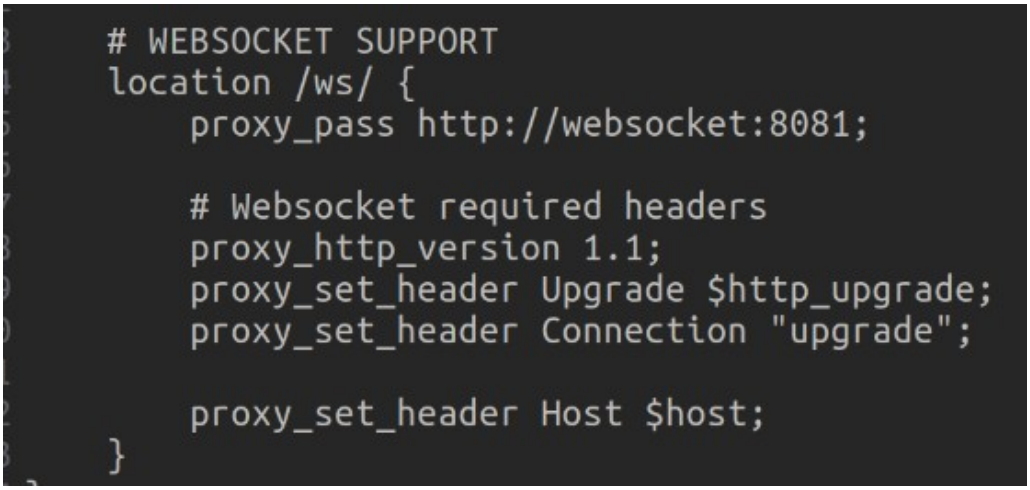
Steps

1. Add a WebSocket server container
A lightweight WebSocket echo server is added to `docker-compose.yml`:



2025-11-17

2. Update `reverse-proxy.conf`
Add a dedicated `/ws/` location block to enable WebSocket headers:



3. Apply Changes
Docker-compose down && docker-compose up -d

9. URL rewriting and redirects

Request routing allows NGINX to:

- Redirect traffic from one URL to another
- Rewrite a URL internally before proxying
- Improve user experience and SEO (clean URLs, fixed paths, etc.)

Steps
Add the required routing rules inside `reverse-proxy.conf`.

2025-11-17

```
server {
    listen 80;
    server_name localhost;

    # 1. Redirect /old → /app/
    location = /old {
        return 301 /app/;
    }

    # 2. Rewrite /api/... to backend/... internally
    location /api/ {
        rewrite ^/api/(.*)$ /backend/$1 break;
        proxy_pass http://backend/;
    }
```

Keep the existing `/app/` proxy block unchanged.

Apply Changes

Docker-compose down && docker-compose up -d

Testing

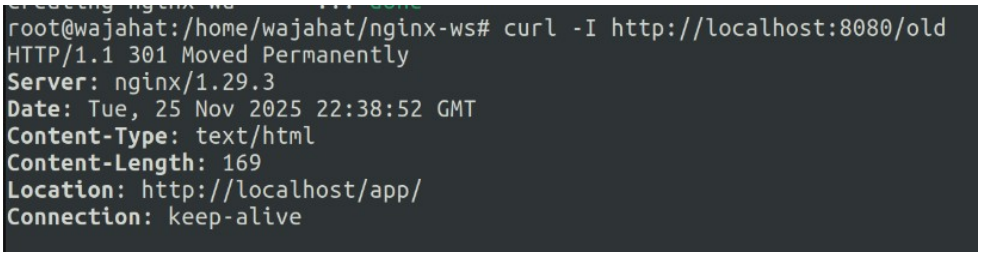
Test 1 : Redirect

curl -I http://localhost:8080/old

Expected:

HTTP/1.1 301 Moved Permanently

Location: /app/



Test 2 : Rewrite

curl -I http://localhost:8080/api/test

If your backend logs show a request for `/backend/test`, rewrite is working. Check your log file:

2025-11-17

```
025/11/25 22:38:30 [notice] 1#1: start worker process 39
025/11/25 22:38:30 [notice] 1#1: start worker process 40
025/11/25 22:38:30 [notice] 1#1: start worker process 41
72.24.0.8 - - [25/Nov/2025:22:39:21 +0000] "HEAD /backend/test HTTP/1.0" 404 0 "-" "curl/7.81.0" "-"
025/11/25 22:39:21 [error] 30#30: *1 open() "/usr/share/nginx/html/backend/test" failed (2: No such file or directory), client: 172
24.0.8, server: localhost, request: "HEAD /backend/test HTTP/1.0", host: "backend"
72.24.0.8 - - [25/Nov/2025:22:39:24 +0000] "HEAD /backend/test HTTP/1.0" 404 0 "-" "curl/7.81.0" "-"
025/11/25 22:39:24 [error] 31#31: *2 open() "/usr/share/nginx/html/backend/test" failed (2: No such file or directory), client: 172
24.0.8, server: localhost, request: "HEAD /backend/test HTTP/1.0", host: "backend"
```

10. Custom access logs

Define a new custom access log format in the `nginx.conf` file under the `http` block. This format includes detailed request and response metrics, such as client IP, request time, upstream connection time, and response time, to provide comprehensive logging for performance analysis.

```
log_format custom_access '$remote_addr - $remote_user [$time_local] '
                         '"$request" $status $body_bytes_sent '
                         '"$http_referer" "$http_user_agent" '
                         'rt=$request_time uct=$upstream_connect_time '
                         'uht=$upstream_header_time urt=$upstream_response_time';
```

Enable the custom log format in the nginx `.conf` configuration file by adding the following directive inside the `server` block (but outside any `location` blocks):

```
# access custom logs
access_log /var/log/nginx/custom_access.log custom_access;
```

Apply Changes
Docker-compose down && docker-compose up -d

Access the log directory and open the access logs, you will the custom logs as

```
172.24.0.1 - - [25/Nov/2025:23:22:43 +0000] "GET /favicon.ico HTTP/1.1" 404 153 "http://localhost:8080/app/" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:144.0
Gecko/20100101 Firefox/144.0" rt=0.000 uct=- uht=- urt=-
172.24.0.1 - - [25/Nov/2025:23:22:44 +0000] "GET /app/ HTTP/1.1" 304 0 "-" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:144.0) Gecko/20100101 Firefox/144.0"
rt=0.001 uct=0.000 uht=0.000 urt=0.000
172.24.0.1 - - [25/Nov/2025:23:22:44 +0000] "GET /app/ HTTP/1.1" 304 0 "-" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:144.0) Gecko/20100101 Firefox/144.0"
rt=0.000 uct=0.001 uht=0.001 urt=0.001
172.24.0.1 - - [25/Nov/2025:23:22:44 +0000] "GET /app/ HTTP/1.1" 304 0 "-" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:144.0) Gecko/20100101 Firefox/144.0"
rt=0.001 uct=0.000 uht=0.000 urt=0.000
172.24.0.1 - - [25/Nov/2025:23:22:45 +0000] "GET /app/ HTTP/1.1" 304 0 "-" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:144.0) Gecko/20100101 Firefox/144.0"
rt=0.001 uct=0.000 uht=0.000 urt=0.000
172.24.0.1 - - [25/Nov/2025:23:22:45 +0000] "GET /app/ HTTP/1.1" 304 0 "-" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:144.0) Gecko/20100101 Firefox/144.0"
rt=0.000 uct=0.000 uht=0.001 urt=0.001
172.24.0.1 - - [25/Nov/2025:23:22:45 +0000] "GET /app/ HTTP/1.1" 304 0 "-" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:144.0) Gecko/20100101 Firefox/144.0"
rt=0.000 uct=0.000 uht=0.001 urt=0.001
172.24.0.1 - - [25/Nov/2025:23:22:45 +0000] "GET /app/ HTTP/1.1" 304 0 "-" "Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:144.0) Gecko/20100101 Firefox/144.0"
rt=0.001 uct=0.000 uht=0.000 urt=0.000
```

11. Custom Error pages

`Steps:`

1. Create the custom 404 page
Inside your `html/` directory:

`html/404.html`

2025-11-17

```
root@wajahat:/home/wajahat/nginx-ws/html# cat 404.html
<html>
<head><title>Page Not Found</title></head>
<body style="font-family: Arial; text-align:center; margin-top: 80px;">
<h1>404 Page Not Found :\\</h1>
<p>The resource you requested does not exist.</p>
</body>
</html>
root@wajahat:/home/wajahat/nginx-ws/html#
```

1. Configure nginx to use it

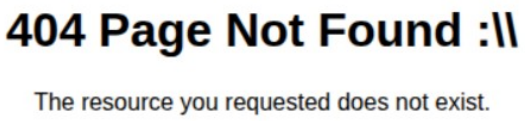
Pick ANY server block you want to activate it on (for example in `reverse-proxy.conf`):

```
# custom 404
error_page 404 /404.html;
location = /404.html {
    root /usr/share/nginx/html;
    internal;
}
```

- `error_page 404 /404.html;` tells NGINX what page to use.
- `internal;` prevents users from directly accessing `/404.html` via browser URL.

Apply Changes
Docker-compose down && docker-compose up -d

Test
Visit something invalid: localhost:/8080/test



12. Host multiple sites with subdomains and separate server blocks

Steps
1. Create separate site folders
html/

2025-11-17

admin/
index.html
dev/
index.html

html/admin/index.html:

```
<h1>Welcome to Admin Portal</h1>
```

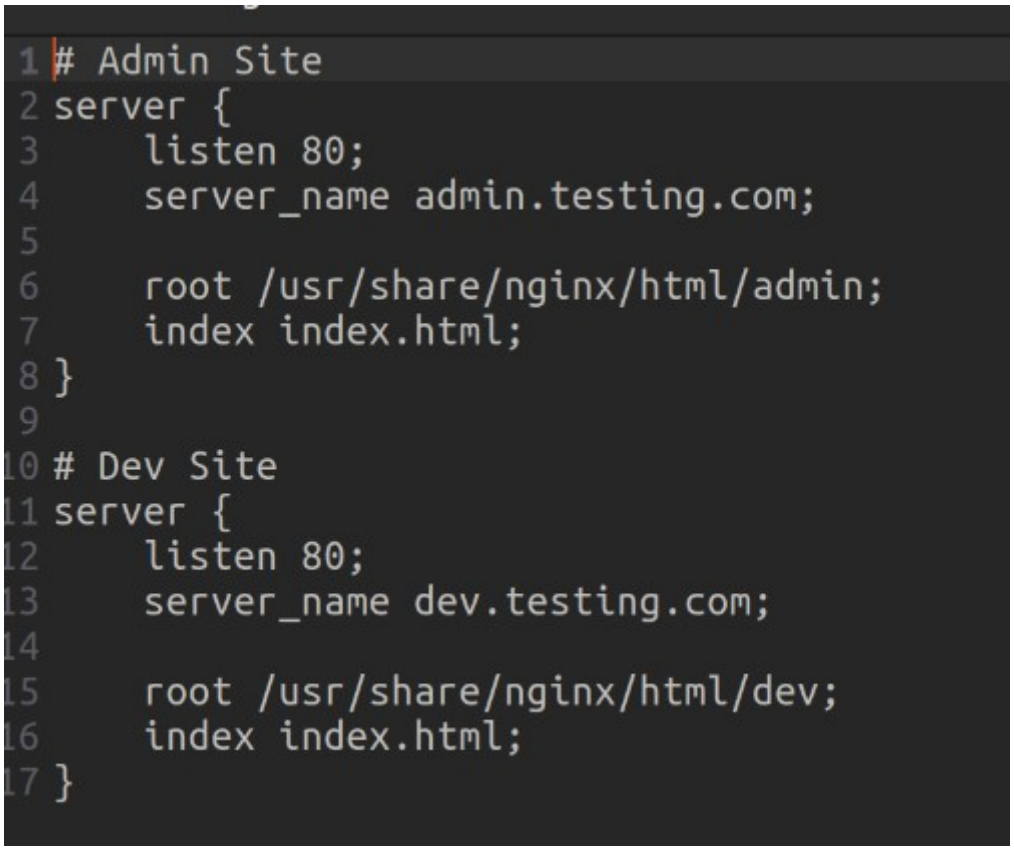
html/dev/index.html:

```
<h1>Welcome to Dev Portal</h1>
```

2. Update nginx config to add two server blocks

Create a new config file like:

conf.d/multisite.conf



```
# Admin Site
server {
    listen 80;
    server_name admin.testing.com;

    root /usr/share/nginx/html/admin;
    index index.html;
}

# Dev Site
server {
    listen 80;
    server_name dev.testing.com;

    root /usr/share/nginx/html/dev;
    index index.html;
}
```

This tells NGINX:

- When user hits `admin.testing.com`, serve content from `/admin/`
- When user hits `dev.testing.com`, serve content from `/dev/`

3. Map hostnames locally

On your host machine, edit `/etc/hosts`:

```
127.0.0.1  admin.testing.com
127.0.0.1  dev.testing.com
```

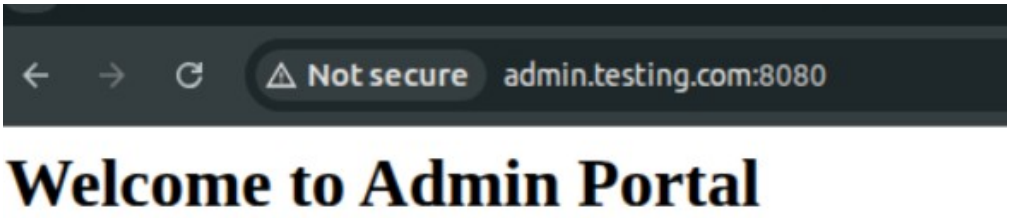
This simulates DNS so the browser resolves those “domains” to your local machine.

2025-11-17

5.Apply Changes
Docker-compose down && docker-compose up -d

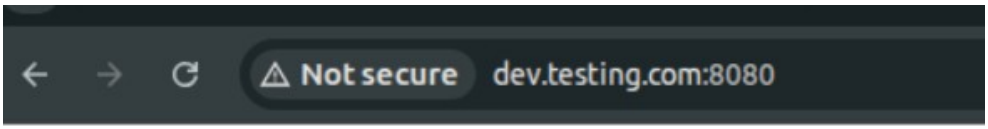
6. Test
On your Browser
Visit:

http://admin.testing.com:8080



Welcome to Admin Portal

http://dev.testing.com:8080



Welcome to Dev Portal

Expected:

- First one shows “Welcome to Admin Portal”
- Second one shows “Welcome to Dev Portal”