NGINX Cheat Sheet , From Basic to Advanced

A compact reference of common and advanced NGINX configuration snippets, with short explanations and recommended options. Use as a quick reference during workshops or when building configs.

Table of Contents

- Beginner-friendly walkthrough
- What the nginx commands and common directives mean (CLI reference)
- Minimal nginx.conf skeleton
- Basic server block (static site)
- Running NGINX in Docker (optional)
- Reverse proxy
- Upstream / Load balancing
- Proxy tuning
- Caching
- Rate limiting
- Advanced topics (Lua, map, sticky sessions)
- Troubleshooting & debugging

Quick commands

- See the CLI section below: "What the nginx commands and common directives mean" for detailed explanation and examples (test/reload and Docker commands).

What the nginx commands and common directives mean

This short reference explains the CLI commands and the most-used configuration directives in this cheat sheet. It helps you understand *what* each command does, *why* you'd run it, and *what to expect*.

CLI commands

- `nginx -t`
  - Purpose: Test the active NGINX configuration for syntax errors and validate includes.
  - What it does: Parses the configuration files (nginx.conf and included files) and reports syntax correctness and any missing files or invalid directives.
  - When to use: Always run after editing any config and before reloading/restarting NGINX.
  - Expected output: `nginx: the configuration file /etc/nginx/nginx.conf test is successful` on success; errors list file and line on failure.
- `nginx -s reload`
  - Purpose: Gracefully reload NGINX workers with new configuration without dropping connections.
  - What it does: Sends a signal to the master process to reload the configuration; master spawns new worker processes with the new config and gracefully shuts down old workers.
  - When to use: After a successful `nginx -t`, to apply changes without downtime.
  - Note: On systemd systems you can use `systemctl reload nginx` which issues the same graceful reload.

- `docker exec <container> nginx -t` / `docker exec <container> nginx -s reload`
  - Purpose: Run the same `nginx` CLI commands inside a running container.
  - What it does: Executes the command inside the container namespace; useful when NGINX runs in Docker and you don't have a native package-installed binary.
  - When to use: Test/reload containerized NGINX after replacing mounted configs or updating image files.

Common NGINX directives (what they do)

- `server { ... }`
  - Purpose: Define a virtual host (listener) for a combination of IP/port and `server_name`.
  - Goal: Route requests for a domain or port to the relevant `location` blocks and behavior.
- `listen 80;` / `listen 443;`
  - Purpose: Specify which port and (optionally) IP address the server block should accept connections on.
  - Goal: Bind NGINX to that port. Port 80 for HTTP, 443 for HTTPS (TLS) traditionally.
- `server_name example.com;`
  - Purpose: Match the `Host` header to select which server block handles the request.
- `location / { ... }`
  - Purpose: Match request URIs and apply rules (proxying, serving static files, rewrites).
  - Goal: Map URL paths to backends, static content, or special handlers.
- `root /path/to/html;` and `index index.html;`
  - Purpose: Serve static files from the filesystem under `root` and default file names using `index`.

- `try_files $uri $uri/ =404;`
  - Purpose: Try to serve the requested file or fall back to the next expression; `=404` returns 404 if missing.
  - Goal: Provide clean static file serving with fallback behavior.
- `proxy_pass http://backend;`
  - Purpose: Forward the client request to an upstream/backend server or upstream group defined by `upstream`.
  - What to watch: Relative vs absolute URIs can change how the request path is forwarded — see official docs for subtle behavior.
- `upstream backend { server ... }`
  - Purpose: Define a named pool of backend servers and parameters (weights, fail policies).
  - Goal: Use the pool in `proxy_pass http://backend;` for load balancing.
- `proxy_set_header ...`
  - Purpose: Add or replace request headers when proxying to preserve client info (Host, X-Real-IP, X-Forwarded-For, etc.).
  - Goal: Ensure backend apps can see the original client address and host.

- `proxy_http_version 1.1;` and `proxy_set_header Connection "";`
  - Purpose: Use HTTP/1.1 for upstream connections (required for keepalive and WebSocket proxying) and clear `Connection` header to avoid interfering with connection management.
- `proxy_buffering off;`
  - Purpose: Disable proxy response buffering so data streams directly from upstream to client.
  - When to use: Real-time streams, server-sent events, or WebSocket-like flows.
- `limit_req_zone` and `limit_req` / `limit_conn_zone` and `limit_conn`
  - Purpose: Define rate limits and concurrent connection limits by a key (IP, cookie, etc.).
  - Goal: Protect upstreams from abusive traffic and reduce the blast radius of floods.
- `proxy_cache_path` and `proxy_cache` / `proxy_cache_valid`
  - Purpose: Configure on-disk cache for proxied responses to speed up repeated requests.
  - Goal: Reduce backend load and improve response times for cacheable content.

- `add_header NAME "value" always;`
  - Purpose: Add HTTP response headers (security headers, caching headers) to responses.
  - Note: `always` ensures the header is present for error responses too.
- `log_format` and `access_log` and `error_log`
  - Purpose: Define structured log formats and where to write logs. Essential for troubleshooting and metrics.

Beginner-friendly walkthrough (line-by-line explanations)

This section explains the config snippets in plain language for someone who is new to NGINX. Each item below references a short snippet from the cheat sheet and explains the most important lines and why they are there.

1) Minimal `nginx.conf` skeleton

```
user nginx;
worker_processes auto;
error_log /var/log/nginx/error.log warn;
pid /var/run/nginx.pid;

events { worker_connections 1024; }

http {
    include mime.types;
    default_type application/octet-stream;
    sendfile on;
    keepalive_timeout 65;
    include /etc/nginx/conf.d/*.conf;
}
```

- `user nginx;` — the OS user NGINX workers run as for permissions and security. On containers this often maps to a different UID.
- `worker_processes auto;` — how many worker processes NGINX should start. `auto` uses CPU cores automatically.
- `error_log /var/log/nginx/error.log warn;` — write NGINX internal errors to a file (useful for debugging).
- `events { worker_connections 1024; }` — controls how many connections each worker can handle. Multiply by `worker_processes` for total concurrency (rough estimate).

- `http { ... }` — top-level block for HTTP-specific settings. `include` brings in separate site files so you can keep configs organized.
- `sendfile on;` — improves static file performance by using optimized OS calls.

Why this matters: Start with this skeleton. You will add `server {}` blocks under `conf.d/` or `sites-enabled/` for each website or virtual host.

2) Basic server block (static site)

```
server {
    listen 80;
    server_name example.com www.example.com;
    root /var/www/example.com/html;
    index index.html index.htm;
    location / { try_files $uri $uri/ =404; }
}
```

- `listen 80;` — accept HTTP requests on port 80.
- `server_name` — matches the Host header in incoming requests; multiple names can be listed.
- `root` — folder on disk where NGINX looks for files to serve.
- `index` — filenames to try when a client requests a directory (`/` → `index.html`).
- `location / { ... }` — rules for requests to the root path. `try_files` checks for the file and returns 404 if not found.

Beginner note: Create the `index.html` file in the `root` folder and then visit your server IP or mapped port to see it served by NGINX.

Running NGINX in Docker

This section shows common ways to run the official NGINX image, explains the Docker CLI options used, and gives a lightweight `docker-compose` example. Start here if you plan to run NGINX inside a container.

1. Run official image (quick test)

```
docker run --name nginx-test -d -p 8080:80 nginx:stable
```

Explanation of flags:

- `docker run` : start a new container from an image.
- `--name nginx-test` : assign a container name to make it easier to reference.
- `-d` : detach; run the container in the background.
- `-p 8080:80` : map host port 8080 to container port 80 (host:container).
- `nginx:stable` : the image (official NGINX stable release). You can use `nginx:alpine` for a smaller image.

Now visit `http://localhost:8080` to see the default NGINX welcome page.

2. Run with custom config and content (bind mounts)

Prepare a local folder structure (example):

```
./nginx/
    ├─ conf.d/
    │   └─ default.conf
    ├─ html/
    │   └─ index.html
```

Run the container with mount points:

```
docker run -d --name nginx-site \
    -p 80:80 \
    -v $(pwd)/nginx/conf.d:/etc/nginx/conf.d:ro \
    -v $(pwd)/nginx/html:/usr/share/nginx/html:ro \
    nginx:stable
```

Notes:

- `-v <host>:<container>:ro` mounts a host directory into the container as read-only.
- Mounting `conf.d` and the site content lets you edit locally and immediately test by reloading the containerized NGINX.

3. Test and reload config inside the container

Test config:

```
docker exec nginx-site nginx -t
```

Reload gracefully:

```
docker exec nginx-site nginx -s reload
```

If you prefer a single command to replace files and reload, use `docker cp` to copy files into the container and then `nginx -s reload`.

4. Build a custom image (Dockerfile)

If you need modules or pre-installed assets, build a custom image:

```
FROM nginx:stable
COPY conf.d/ /etc/nginx/conf.d/
COPY html/ /usr/share/nginx/html/

# Optional: copy custom mime types or other snippets
COPY snippets/ /etc/nginx/snippets/

EXPOSE 80
```

Build and run:

```
docker build -t my-nginx:local .
docker run -d --name my-nginx -p 80:80 my-nginx:local
```

5. Docker Compose example (recommended for multi-container setups)

`docker-compose.yml`:

```
version: '3.8'
services:
    nginx:
        image: nginx:stable
        container_name: nginx-compose
        ports:
            - "80:80"
        volumes:
            - ./nginx/conf.d:/etc/nginx/conf.d:ro
            - ./nginx/html:/usr/share/nginx/html:ro
        restart: unless-stopped
```

Commands:

- `docker-compose up -d` — start services in background.
- `docker-compose logs -f nginx` — follow NGINX logs.
- `docker-compose exec nginx nginx -t` — test config inside the service.

6. Permissions and SELinux/AppArmor

When bind-mounting, ensure the container user (usually `nginx` with UID 101 in official images) can read mounted files. On SELinux-enabled hosts, add `:z` or `:Z` to volume flags (e.g., `-v ./nginx/conf.d:/etc/nginx/conf.d:ro,z`).

7. Useful Docker commands for maintenance

- `docker ps` — list running containers.
- `docker logs -f nginx-site` — stream logs.
- `docker exec -it nginx-site /bin/bash` (or `/bin/sh` on alpine) — open a shell.
- `docker rm -f nginx-site` — stop and remove a container.

3) Reverse proxy (forwarding requests to an app)

```
location / {
    proxy_pass http://127.0.0.1:3000;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
}
```

- `proxy_pass` — tells NGINX to forward matching requests to another server (a backend application). Here the backend runs on the same machine at port 3000.
- `proxy_set_header Host $host;` — forwards the original Host header so the backend knows the requested domain.
- `proxy_set_header X-Real-IP $remote_addr;` — forwards the client's IP address so the backend can log or apply rate limits.

Beginner tip: Use this when your app listens on a port (e.g., a Node/Express app on 3000). NGINX will accept public requests and forward them to the app.

4) Upstream block (load balancing)

```
upstream backend {
    server 10.0.0.11:8000 weight=3;
    server 10.0.0.12:8000;
}
```

- `upstream` — a named group of backend servers; use `proxy_pass http://backend;` to use them.
- `weight=3` — this server will receive roughly 3x the requests of a server without weight.

Beginner scenario: Start with two app instances and add them here. NGINX will distribute requests (round-robin by default).

5) Proxy tuning (timeouts and buffers)

See the dedicated "Proxy tuning (common recommended options)" section below for the full recommended example and explanations. Start with modest timeouts (connect 5s, read/send 30-60s) and only change buffer sizes when you observe header or response fragmentation.

6) Caching (proxy_cache)

```
proxy_cache_path /var/cache/nginx levels=1:2 keys_zone=STATIC:10m max_size=1g inactive=60m;
proxy_cache STATIC;
proxy_cache_valid 200 302 10m;
```

- `proxy_cache_path` — defines where cached files go on disk and how large the cache is.
- `proxy_cache` — enables caching in a location using the named zone.
- `proxy_cache_valid` — how long to keep cached responses for certain status codes.

Beginner benefit: Caching speeds up responses and reduces load on your application servers for static or rarely-changing content.

7) Rate limiting (protect your app)

```
limit_req_zone $binary_remote_addr zone=one:10m rate=10r/s;
limit_req zone=one burst=20 nodelay;
```

- `limit_req_zone` — creates a shared memory zone keyed by IP to count requests.
- `rate=10r/s` — allow roughly 10 requests per second per IP.
- `burst=20` — allow short spikes up to 20 requests.

Beginner warning: Set conservative limits and test; overly aggressive limits will block real users.

8) WebSocket proxying

```
proxy_http_version 1.1;
proxy_set_header Upgrade $http_upgrade;
proxy_set_header Connection "upgrade";
```

- WebSockets require HTTP/1.1 and the `Upgrade` header to be forwarded to the backend. Without these lines, WebSocket connections will fail.

9) Gzip compression

```
gzip on;
gzip_types text/plain text/css application/json application/javascript;
```

- `gzip on;` — compresses responses to save bandwidth. Useful for text-based responses.

Beginner tip: Test with `curl -I -H "Accept-Encoding: gzip" http://...` and check `Content-Encoding: gzip` in the response.

10) Logging and troubleshooting

See the consolidated "Troubleshooting tips" section later in this file for validation, log locations, and practical `curl` checks to debug virtual hosts and upstreams.

11) Error pages and maintenance mode

`error_page 503 /maintenance.html;` lets you present a friendly page when you intentionally return a 503 (maintenance).

Beginner workflow: edit config -> run `nginx -t` -> `nginx -s reload` -> test URLs. If NGINX won't start, check `error.log` for the first reported error and fix that file/line.

Minimal nginx.conf skeleton

```
user nginx;
worker_processes auto;
error_log /var/log/nginx/error.log warn;
pid /var/run/nginx.pid;

events {
    worker_connections 1024;
}

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

    sendfile        on;
    tcp_nopush      on;
    tcp_nodelay     on;
    keepalive_timeout 65;

    include /etc/nginx/conf.d/*.conf;
    include /etc/nginx/sites-enabled/*;
}
```

Notes: `worker_processes auto` lets NGINX pick the number based on CPU cores. `sendfile` improves static file throughput.

Basic server block (static site)

```
server {
    listen 80;
    server_name example.com www.example.com;

    root /var/www/example.com/html;
    index index.html index.htm;

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

    access_log /var/log/nginx/example.access.log;
}
```

Reverse proxy (simple)

```
server {
    listen 80;
    server_name api.example.com;

    location / {
        proxy_pass http://127.0.0.1:3000;
        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;
    }
}
```

Notes: forward standard headers so backend apps can see original client info.

Upstream block — load balancing

```
upstream backend {
    server 10.0.0.11:8000 weight=3;
    server 10.0.0.12:8000;
    server 10.0.0.13:8000 max_fails=3 fail_timeout=30s;
    # keepalive 16; # enable persistent connections to backends
}

server {
    listen 80;
    server_name app.example.com;

    location / {
        proxy_pass http://backend;
        proxy_set_header Host $host;
        proxy_http_version 1.1;
        proxy_set_header Connection "";
    }
}
```

Notes:

- `weight` changes traffic share.
- `max_fails` and `fail_timeout` form a simple failure policy.
- `keepalive` in upstream enables connection reuse (good for many short requests).

Load balancing algorithms

- Round robin (default): list servers in `upstream {}`
- Least connections: `upstream backend { least_conn; server ... }`
- IP hash (session stickiness by client IP): `upstream backend { ip_hash; server ... }`
- Hash: `hash $request_uri consistent;` (requires the `hash` directive)

Note: Sticky sessions require third-party modules for cookie-based stickiness, or `ip_hash` for IP-based affinity.

Health checks

- NGINX Open Source: no native active health checks; rely on `max_fails` / `fail_timeout` passive checks or use third-party modules (nginx_upstream_check_module) or external tools.
- NGINX Plus: has active health checks (`health_check` directive) and advanced features.

Passive example (in upstream): shown above (`max_fails` , `fail_timeout`).

Proxy tuning (common recommended options)

```
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;
proxy_connect_timeout 5s;
proxy_send_timeout 60s;
proxy_read_timeout 60s;
proxy_buffer_size 16k;
proxy_buffers 4 32k;
proxy_busy_buffers_size 64k;
proxy_temp_file_write_size 64k;
```

Notes: adjust buffers/timeouts for your workload; too small causes fragmentation, too large wastes RAM.

Caching (proxy cache)

```
http {
    proxy_cache_path /var/cache/nginx levels=1:2 keys_zone=STATIC:10m max_size=1g inactive=60m use_temp_path=off;

    server {
        location / {
            proxy_cache STATIC;
            proxy_cache_valid 200 302 10m;
            proxy_cache_valid 404 1m;
            proxy_cache_bypass $http_cache_control; # bypass when client asks
            proxy_pass http://backend;
        }
    }
}
```

Notes: tune `keys_zone` and `max_size` for available RAM/disk. `use_temp_path=off` reduces disk IO on some setups.

Rate limiting (basic)

```
http {
    limit_req_zone $binary_remote_addr zone=one:10m rate=10r/s;

    server {
        location /api/ {
            limit_req zone=one burst=20 nodelay;
            proxy_pass http://backend;
        }
    }
}
```

Notes: `limit_req` protects against request floods (r/s). `burst` allows short spikes.

Connection limiting:

```
limit_conn_zone $binary_remote_addr zone=addr:10m;

location / {
    limit_conn addr 10;
}
```

Gzip compression

```
http {
    gzip on;
    gzip_types text/plain text/css application/json application/javascript text/xml application/xml application/xml+rss text/javascript;
    gzip_min_length 256;
    gzip_proxied any;
}
```

WebSocket proxying

```
location /ws/ {
    proxy_pass http://websocket_backend;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
    proxy_set_header Host $host;
}
```

Notes: must use `proxy_http_version 1.1` and set `Upgrade` / `Connection` headers.

Security headers (example)

```
add_header X-Frame-Options "SAMEORIGIN" always;
add_header X-Content-Type-Options "nosniff" always;
add_header X-XSS-Protection "1; mode=block" always;
add_header Referrer-Policy "no-referrer-when-downgrade" always;
add_header Content-Security-Policy "default-src 'self'; script-src 'self' 'unsafe-inline' 'unsafe-eval' https:" always;
```

Notes: CSP is powerful but can break pages if too strict.

Client body and upload limits

```
client_max_body_size 50M;
client_body_timeout 12s;
```

Timeouts and keepalives

```
keepalive_timeout 65;
send_timeout 30s;
client_header_timeout 10s;
client_body_timeout 10s;
```

Buffering control (for streaming / large responses)

- `proxy_buffering off;` disables buffering (useful for real-time streaming)
- `proxy_buffers`, `proxy_buffer_size`, and `proxy_busy_buffers_size` control memory used for buffered responses

Example to disable buffering for a location:

```
location /stream/ {
    proxy_buffering off;
    proxy_pass http://backend_streamer;
}
```

Logging: custom formats and conditional logs

```
log_format main '$remote_addr - $remote_user [$time_local] "$request" '
                '$status $body_bytes_sent "$http_referer" '
                '"$http_user_agent" "$http_x_forwarded_for"';
access_log /var/log/nginx/access.log main;
```

Conditional logging (skip health checks):

```
map $request_uri $loggable {
    ~^/health$ 0;
    default 1;
}
access_log /var/log/nginx/access.log main if=$loggable;
```

Error pages and maintenance mode

```
error_page 500 502 503 504 /50x.html;
location = /50x.html {
    root /usr/share/nginx/html;
}

# Maintenance
server {
    listen 80;
    server_name example.com;

    if (-f /var/www/maintenance.enable) {
        return 503;
    }

    error_page 503 @maintenance;
    location @maintenance {
        root /var/www/maintenance;
        try_files /index.html =503;
    }
}
```

Split traffic with `map` (A/B testing / feature flag)

```
map $http_cookie $variant {
    ""      0;
    "ab=1"  1;
}

server {
    location / {
        if ($variant = 1) {
            proxy_pass http://blue_backend;
        }
        proxy_pass http://green_backend;
    }
}
```

Notes: `map` is efficient for runtime decisions.

Sticky sessions (cookie based)

- Open-source: cookie-based sticky requires third-party module (nginx-sticky-module-ng) or use the `hash` directive/methods.
- NGINX Plus: has `sticky` directive for stable cookie-based persistence.

Example using `ip_hash` (simpler):

```
upstream backend {
    ip_hash;
    server 10.0.0.11:8000;
    server 10.0.0.12:8000;
}
```

Using `include` for modular configs (recommended layout)

- `/etc/nginx/nginx.conf` includes `/etc/nginx/conf.d/*.conf` and `/etc/nginx/sites-enabled/*`.
- Put site-specific server blocks under `sites-available` and symlink into `sites-enabled`.
- Use `include snippets/` to store reusable pieces (proxy-headers, gzip-params, etc.).

Lua example (embedding business logic)

- Requires `ngx_http_lua_module` (OpenResty or compiled module).

Simple snippet:

```
location /lua/ {
    content_by_lua_block {
        ngx.say("Hello from Lua: ", ngx.var.remote_addr)
    }
}
```

Notes: powerful but use with care (complexity and security considerations).

Advanced: rate-limiting by key (user-based)

```
limit_req_zone $binary_remote_addr zone=perip:10m rate=5r/s;
limit_req_zone $cookie_userid zone=peruser:10m rate=1r/s;

server {
    location /api/ {
        limit_req zone=peruser burst=5 nodelay if=$cookie_userid;
        limit_req zone=perip burst=20;
        proxy_pass http://backend;
    }
}
```

Troubleshooting tips

- Always run `nginx -t` after edits.
- Inspect logs: `/var/log/nginx/error.log` and `/var/log/nginx/access.log`.
- If config changes do not take effect, remember to `nginx -s reload` or `systemctl reload nginx`.
- Use `curl -I -H "Host: example.com" http://127.0.0.1` to test virtual hosts locally.

When to prefer NGINX Plus or modules

- Active upstream health checks, advanced monitoring, dynamic reconfiguration → NGINX Plus
- Cookie-based sticky sessions without third-party modules → NGINX Plus
- Otherwise, open-source + third-party modules (or external tools) suffice for many setups.

Further reading

- Official docs: https://nginx.org/en/docs/
- NGINX Plus features: https://www.nginx.com/products/nginx/
- OpenResty (nginx + lua): https://openresty.org/