Nginx Course

Flavio Copes

Updated August 3, 2026

Contents

Learn to serve websites and reverse proxy applications with Nginx, HTTPS, safe configuration changes, access controls, logs, and production checks.

What you'll learn: Configure and operate an Nginx server that serves static files, proxies an application, terminates HTTPS, and fails safely.

Server foundations

Install Nginx, understand its processes and configuration, and make safe changes.

Install and inspect Nginx

Install Nginx from Ubuntu packages and verify the package, service, listener, and first HTTP response.

Nginx is a web server and reverse proxy. It sits in front of your files and your applications, and it handles the raw HTTP traffic from the internet.

On Ubuntu, install it through APT so package updates and systemd integration remain visible to the server administrator:

```
sudo apt update
sudo apt install nginx
```

The package puts the configuration under `/etc/nginx/`, installs a systemd unit, and starts the service right away with a default welcome page.

Now prove it's actually working. I use three checks, and I run them in this order because each one tests a different layer.

Is the service running?

```
systemctl status nginx
```

Look for `active (running)` in the output. This tells you systemd started the process and it didn't crash. It says nothing about the network yet.

Is something listening on port 80?

```
sudo ss -lntp | grep nginx
# LISTEN 0  511  0.0.0.0:80   0.0.0.0:*  users:(("nginx",pid=1204,fd=6),...)
```

`ss -lntp` shows listening TCP sockets with the owning process. You want an `nginx` entry bound to port 80. Note the PID and the address: `0.0.0.0` means it accepts connections on every network interface.

Does it answer HTTP?

```
curl -I http://127.0.0.1
# HTTP/1.1 200 OK
# Server: nginx/1.24.0 (Ubuntu)
```

`curl -I http://127.0.0.1` sends a real request and prints only the response headers. A `200 OK` from `Server: nginx` closes the loop: the service runs, the socket is open, and HTTP works end to end.

Keep these three checks separate in your head. A running service with no listener points at a configuration problem. A listener that never answers points at a firewall or a hung process. When something breaks later, the first check that fails tells you where to look.

The most common installation failure is a port conflict. If Apache or another server already owns port 80, Nginx refuses to start, and `journalctl -u nginx` shows a line like `bind() to 0.0.0.0:80`

`failed (98: Address already in use)`. Stop the other server or change the `listen` port, then start Nginx again.

Run the three checks on a disposable Ubuntu server. Record which process listens, which address and port it uses, and the returned status code.

Understand master and worker processes

Separate privileged configuration and lifecycle work from the worker processes that handle connections.

Nginx normally runs one **master process** and one or more **worker processes**. The split exists for privilege and for uptime.

The master runs as root. It reads the configuration, opens the listening sockets (binding to port 80 requires privileges), and manages the workers. It never handles a client request itself.

The workers run as an unprivileged user, `www-data` on Ubuntu. Each worker handles many connections at once with an event-driven model, so a handful of workers can serve thousands of clients.

Look at the process tree with `ps`:

```
ps -ef --forest | grep [n]ginx
# root      1204     1  nginx: master process /usr/sbin/nginx -g daemon on; master_process on
# www-data  1205  1204  nginx: worker process
# www-data  1206  1204  nginx: worker process
```

The master's parent is PID 1, and every worker's parent is the master. Nginx labels each process in its command line, so you can tell them apart at a glance.

The number of workers comes from the `worker_processes` directive. The default packaged configuration sets it to `auto`, which means one worker per CPU core. You rarely need to change it.

Why this matters for reloads

When you reload the configuration, the master reads the new files and starts a fresh set of workers with the new settings. The old workers stop accepting new connections but keep serving the requests they already have, then exit. Clients never see a dropped connection.

This also means a reload does not instantly kill long-lived connections. An old worker holding an open download or WebSocket sticks around until that work finishes. If you see stale workers in `ps` after a reload, that's usually why, not a bug.

Cross-check with systemd:

```
systemctl show nginx --property=MainPID
# MainPID=1204
```

The `MainPID` matches the master. Signals like reload go to the master, and it coordinates the workers.

One realistic mistake: killing a worker process directly to ”fix” something. The master immediately spawns a replacement, so nothing changes, and you may have cut off in-flight requests. If a worker misbehaves, read the error log and act on the master, not on the workers.

Inspect the Nginx process tree on your test server. Identify the master PID, worker PIDs, users,

and the process that systemd considers the main process.

Read configuration contexts

Follow directives through main, events, http, server, and location contexts without losing their scope.

Nginx configuration is a tree of **contexts**. A context is a named block delimited by braces, and every directive is valid only in the contexts documented for it.

Here is the skeleton of a minimal configuration:

```
user www-data;
worker_processes auto;

events {
  worker_connections 768;
}

http {
  include /etc/nginx/mime.types;

  server {
    listen 80;
    server_name example.com;

    location / {
      root /var/www/html;
    }
  }
}
```

Directives outside any block belong to the **main** context and describe the process itself: which user workers run as, how many workers to start. The `events` context configures connection handling. The `http` context contains HTTP-wide settings and server blocks. Each `server` block is one virtual server with its own policy, and locations inside it refine how individual requests are handled.

The `include` directive inserts another file's text at that exact point. That's how Ubuntu splits the configuration: the main file `/etc/nginx/nginx.conf` includes everything in `conf.d/` and `sites-enabled/` from inside the `http` block, which is why those files can start directly with `server { ... }`.

Most directives inherit downward. Set `gzip on;` in `http` and every server and location gets it, unless a lower level overrides it. One catch worth remembering: multi-value directives like `proxy_set_header` don't merge across levels. The moment a location defines one of its own, it stops inheriting all the others from above.

See the whole tree at once

With includes spread over many files, the fastest way to read the effective configuration is:

```
sudo nginx -T
```

This validates the configuration and prints every file, fully assembled, in include order. When you're

debugging on someone else's server, this beats hunting through directories.

The classic mistake is placing a directive in the wrong context. Put `server_name` at the `http` level and validation fails immediately:

```
sudo nginx -t
# nginx: [emerg] "server_name" directive is not allowed here in /etc/nginx/nginx.conf:12
```

The error names the file and line, so read it literally and move the directive into the context the documentation lists for it.

Run `sudo nginx -T` on a test server. Find the main file, the included files, one server block, and the location that handles `/`.

Test, reload, and roll back

Apply configuration changes without replacing a working server with a syntax error or unverified behavior.

Always test the complete effective configuration before asking Nginx to reload it.

`sudo nginx -t` checks syntax and referenced files. A successful reload preserves existing connections. Keep the previous file until HTTP and application health checks pass.

Make a harmless header change. Test, reload, check the response, then restore the previous file and repeat the same safe sequence.

Never reload an untested configuration:

```
sudo nginx -t
sudo systemctl reload nginx
sudo systemctl status nginx --no-pager
```

Keep the previous configuration available before editing. After reload, request the exact hostname and path with curl. A successful syntax check cannot detect a wrong server block, upstream port, certificate name, or application response, so the external smoke test is part of the change.

Check your understanding: server foundations

Check installation, processes, contexts, inheritance, includes, syntax tests, reloads, and rollback.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Requests and static files

Match servers and locations, map URIs to files, and control common responses.

Select a server block

Understand how listen addresses and servername choose the virtual server for a request.

One Nginx instance can host many sites. Each site is a `server` block, and Nginx picks one block per request in two steps.

First it matches the connection against the `listen` directives: which address and port did the client connect to? Then, among the server blocks on that socket, it compares the request's Host header (or the TLS server name) against each `server_name`. The block whose name matches wins.

If no name matches, Nginx uses the **default server** for that listen socket. You mark it explicitly:

```
server {
  listen 80 default_server;
  server_name _;
  return 444;
}

server {
  listen 80;
  server_name blog.example.com;
  root /var/www/blog;
}
```

Here `blog.example.com` gets the blog, and any other hostname — including requests made directly to the IP address — hits the default server. Status `444` is an Nginx special: it closes the connection without a response, a reasonable answer for traffic that isn't addressed to any of your names.

One thing that trips people up: DNS alone does not configure Nginx. Pointing a new domain at your server changes nothing until a server block names it. The reverse is also true — you can test a server block before DNS exists.

Prove which block answers

`curl --resolve` lets you fake DNS for a single request, so you can test each hostname against the real server:

```
curl --resolve blog.example.com:80:203.0.113.10 -I http://blog.example.com/
# HTTP/1.1 200 OK

curl --resolve shop.example.com:80:203.0.113.10 -I http://shop.example.com/
# curl: (52) Empty reply from server   <- the 444 default
```

The first request carries `Host: blog.example.com` and lands in the blog block. The second carries an unknown name and gets the default server's treatment.

The classic mistake is forgetting `default_server` entirely. Then Nginx silently uses the first server block in configuration order as the default, and file ordering in `sites-enabled/` decides which site strangers see when they scan your IP. That's how an internal admin panel ends up answering requests for random hostnames. Always define an explicit catch-all.

Create two test hostnames pointing to one server. Use `curl --resolve` to prove which server block handles each name and an unknown name.

Match location blocks

Predict prefix, exact, and regular-expression location selection before adding nested request rules.

Inside a server block, the request URI selects one `location`. A request is handled by the location selected through Nginx matching rules, not just the first location in the file. Learn the rules once,

and configurations stop surprising you.

There are four kinds of location:

```
location = /health { }      # exact match
location ^~ /assets/ { }    # prefix match, regex-proof
location ~ \.php$ { }       # case-sensitive regex
location /docs/ { }         # plain prefix match
```

The selection algorithm works like this. Exact matches use `=` and win immediately when the URI is identical. Otherwise Nginx finds the **longest matching prefix** among the prefix locations. If that prefix is marked `^~`, it wins and regexes are skipped. Otherwise Nginx checks the regex locations in the order they appear in the file, and the first regex that matches wins. Only when no regex matches does the remembered longest prefix get used.

Read that last part again, because it's the surprise: **a regex beats a longer prefix**. A `location ~ \.png$` defined anywhere in the server block will capture `/assets/logo.png` even if you wrote a `location /assets/` that looks more specific. When you want a prefix to be immune to regexes, mark it `^~`.

Here is a small server that covers the common cases:

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

  location = /health {
    return 200 "ok\n";
  }

  location ^~ /assets/ {
    root /var/www/app/public;
  }

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

Exactly `/health` answers directly. Everything under `/assets/` is served from disk, protected from any regex added later. The remaining application paths fall through to `location /`, which matches every URI as the shortest possible prefix.

Verify each rule, including a near miss:

```
curl -s http://app.example.com/health      # ok
curl -I http://app.example.com/assets/app.css   # 200 from disk
curl -I http://app.example.com/healthz     # proxied to the app, not the health check
```

The `/healthz` case matters. The `=` location matches only the exact URI, so `/healthz` skips it and lands in `location /`. If you expected prefix behavior there, this test catches it.

My advice is to keep the location set small enough that you can predict the winner for any URI in your head. When you can't, run the curl tests before you trust the config.

Serve static files safely

Map request URIs to files with root, alias, index, and tryfiles while avoiding accidental exposure.

Serving files is about one mapping: how does a request URI become a filesystem path? Nginx gives you two directives for it, and they behave differently.

`root` appends the full request URI to a directory. `alias` replaces the matching location prefix with another path.

```
server {
  listen 80;
  server_name static.example.com;
  root /var/www/site/public;

  location / {
    index index.html;
    try_files $uri $uri/ =404;
  }

  location /downloads/ {
    alias /srv/files/;
  }
}
```

With `root`, a request for `/css/site.css` reads `/var/www/site/public/css/site.css` — directory plus the whole URI. With `alias`, a request for `/downloads/report.pdf` reads `/srv/files/report.pdf` — the `/downloads/` prefix is swapped for `/srv/files/`. If you had used `root /srv/files/` there, Nginx would look for `/srv/files/downloads/report.pdf`, which is almost never what you meant.

`try_files $uri $uri/ =404` checks the exact file first, then a directory of that name (which triggers `index`), then gives up with a 404. It's the standard pattern for static sites because it never invents paths: the request either maps to a real file or fails cleanly.

Directory listing stays off unless you have a deliberate reason to enable it. `autoindex` defaults to off, so a directory without an index file returns 403 instead of exposing its contents. Leave it that way.

Keep private files out of the root

The root directory is a public boundary. Anything inside it can be requested by URL, so keep secrets, dotfiles, and build artifacts outside. Publish a `public/` directory and keep `.env`, source code, and backups one level up where no URI can reach them.

Verify the behavior with three requests:

```
curl -I http://static.example.com/index.html   # 200
curl -I http://static.example.com/missing.css  # 404
curl -I http://static.example.com/.env         # 404 - lives outside the root
```

The failure mode to watch for is a misplaced trailing slash with `alias`. `location /downloads/` paired with `alias /srv/files` (no trailing slash) produces paths like `/srv/filesreport.pdf`, and every request 404s. Keep both the location prefix and the alias path ending in `/`, then re-test with

a known file.

Serve a small public directory containing `index.html` and one asset. Verify a missing path returns 404 and a sibling private file cannot be requested.

Return redirects and errors

Use explicit responses and redirects, preserve methods when required, and avoid redirect loops.

Nginx can answer a request without reading a file or contacting an upstream. The `return` directive produces the response directly, and it's the right tool for redirects, health checks, and hard rejections.

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

  location = /health {
    return 200 "ok\n";
  }

  location = /old-pricing {
    return 301 /pricing;
  }

  location = /promo-2025 {
    return 302 /pricing;
  }
}
```

The `/health` location answers with a body and no filesystem access at all. Monitoring systems can hit it thousands of times a day without touching your application.

For redirects, the status code is a real decision. `301` means permanent: browsers and search engines cache it, often aggressively, and there is no practical way to un-teach a browser that has seen it. `302` means temporary and stays revisitable. My advice is to ship a `302` first, watch it behave for a few days, then upgrade to `301` only when the mapping is stable.

There is a second axis: the request method. With `301` and `302`, most clients resend the follow-up request as a GET, even if the original was a POST. Use `307` (temporary) or `308` (permanent) when the request method must be preserved — an API endpoint that moved, for example, where turning a POST into a GET would silently break clients.

Verify status and destination with `curl -I`:

```
curl -I http://www.example.com/old-pricing
# HTTP/1.1 301 Moved Permanently
# Location: /pricing

curl -s http://www.example.com/health
# ok
```

Then check the destination doesn't bounce back. Redirect loops happen when the target location itself redirects — often through a rewrite rule or an HTTPS redirect added later in another block. Follow the chain end to end:

```
curl -sIL -o /dev/null -w "%{num_redirects} redirects -> %{url_effective}\n" http://www.examp
# 1 redirects -> http://www.example.com/pricing
```

More than one hop deserves a look. Twenty hops means a loop: curl gives up with `Maximum (50) redirects followed`, and browsers show "too many redirects".

Add a temporary old-path redirect and a plain `/health` response to your test server. Inspect status and Location headers with `curl -I` before choosing permanent caching.

Check your understanding: requests and static files

Check server selection, location matching, root and alias, index files, tryfiles, redirects, and response controls.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Reverse proxy applications

Proxy requests, preserve useful metadata, handle upgrades, and set timeouts deliberately.

Proxy to an application

Send requests to a local application and understand how the proxypass URI changes upstream paths.

A reverse proxy accepts the public request and makes a separate request to an upstream application.

`proxy_pass` can target a TCP address or Unix socket. A trailing URI changes how Nginx replaces the matched location prefix. Test the exact upstream path instead of guessing.

Run a local test application that prints its request path. Proxy `/app/` to it and compare `proxy_pass` forms with and without a trailing slash.

Proxy one location to a local application:

```
location /api/ {
  proxy_pass http://127.0.0.1:3000/;
  proxy_set_header Host $host;
  proxy_set_header X-Forwarded-Proto $scheme;
}
```

Test the application directly first, then through Nginx. Notice the trailing slash behavior in proxy_pass. Inspect both access logs and application logs so you can follow one request across the proxy boundary.

Forward client request details

Pass the original host, client address, scheme, and request identifiers through a trusted proxy boundary.

The upstream sees Nginx as its direct client unless Nginx forwards selected request metadata. Without help, your application thinks every visitor is `127.0.0.1` making plain HTTP requests to the

wrong hostname. Anything that depends on the real client — logging, rate limits, redirect URLs, "secure cookie" decisions — breaks quietly.

`proxy_set_header` sets the headers Nginx sends upstream. This is the standard set:

```
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;
}
```

`Host` carries the hostname the visitor actually requested, so the application builds correct absolute URLs. `X-Real-IP` is the address of the TCP connection Nginx accepted. `X-Forwarded-For` is the hop chain: `$proxy_add_x_forwarded_for` takes any incoming `X-Forwarded-For` and appends `$remote_addr` to it. `X-Forwarded-Proto` says whether the visitor used `https`, which the app needs for secure cookies and redirect schemes since the proxy leg is often plain HTTP.

The trust problem

Any client can send these headers themselves. Watch:

```
curl http://app.example.com/whoami -H "X-Forwarded-For: 1.2.3.4"
```

With `$proxy_add_x_forwarded_for`, the upstream receives `X-Forwarded-For: 1.2.3.4, 203.0.113.50` — the spoofed value survives as the first entry, followed by the real address Nginx observed. That's fine only if the application knows to trust the last hop, not the first.

So the rule is: the application should trust forwarding headers only from known proxies, and only as many hops deep as your real topology. If Nginx is the sole proxy and you want zero ambiguity, replace untrusted inbound values at the edge instead of appending:

```
proxy_set_header X-Forwarded-For $remote_addr;
```

Now whatever the client claimed is discarded and the upstream sees exactly one trustworthy address.

One Nginx-specific pitfall: `proxy_set_header` directives don't merge across contexts. If a location defines even one of its own, it stops inheriting all the ones set at the `server` or `http` level. Adding a single header in a nested location can silently drop `Host` and `X-Forwarded-Proto`. Keep the full set together, or repeat it.

Make your test application print the headers it receives. Send spoofed forwarding headers from curl and verify the application receives only the policy Nginx intends.

Set timeouts and buffering

Choose connection, read, send, body-size, and buffering limits from application behavior.

Timeouts describe different stages of a proxied request, and Nginx lets you bound each stage separately. A single large value can hide a slow or stuck upstream for too long, so it pays to know which knob covers which phase.

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

```
  proxy_connect_timeout 5s;
  proxy_send_timeout    15s;
  proxy_read_timeout    30s;
  client_max_body_size  10m;
}
```

`proxy_connect_timeout` bounds how long Nginx waits to open the TCP connection to the upstream. A healthy local application connects in milliseconds, so 5 seconds is already generous — if connecting takes longer, the process is down or drowning, and waiting more won't fix it.

`proxy_read_timeout` bounds the gap between two successive reads from the upstream while Nginx waits for the response. This is the one that fires when your application hangs on a slow query. When it expires, the client gets a **504 Gateway Time-out**. `proxy_send_timeout` is the same idea in the other direction, for writing the request to the upstream.

`client_max_body_size` caps the request body. Anything larger is rejected with **413 Request Entity Too Large** before it reaches the application. The default is only 1 megabyte, which is the classic reason file uploads fail behind a fresh Nginx.

Pick the values from measured behavior, not guesses. List one normal and one worst-case duration for the endpoint — say checkout takes 800ms normally and 12s at the p99 — then set the read timeout a little above the worst case, not at 300s "to be safe". A generous timeout keeps a dead request occupying an upstream process long after the user gave up.

Test the failure response on purpose:

```
curl -i https://api.example.com/api/slow-report
# HTTP/1.1 504 Gateway Time-out     <- upstream exceeded proxy_read_timeout

curl -i -X POST https://api.example.com/api/upload --data-binary @big-video.mp4
# HTTP/1.1 413 Request Entity Too Large
```

Proxy buffering is the related setting people forget. With `proxy_buffering on` (the default), Nginx reads the upstream response quickly into memory and disk buffers, freeing the upstream to move on while Nginx trickles bytes to a slow client. Keep it on for normal responses. Turn it off per-location with `proxy_buffering off;` only for streaming endpoints where the client must see bytes as the upstream produces them — otherwise progress events sit in a buffer instead of reaching the browser.

Proxy WebSockets and streams

Forward protocol upgrades and long-lived responses without applying ordinary request assumptions blindly.

WebSocket connections begin as HTTP and then upgrade to a different protocol on the same connection. The client sends a normal-looking request with `Upgrade: websocket` and `Connection: Upgrade` headers, the server answers `101 Switching Protocols`, and from that moment the connection carries WebSocket frames instead of HTTP.

A default proxy configuration breaks this in two ways. Nginx speaks HTTP/1.0 to upstreams unless told otherwise, and HTTP/1.0 has no upgrade mechanism. And `Upgrade` is a hop-by-hop header, so Nginx doesn't forward it on its own. You opt in per location:

```
location /ws/ {
```

```
  proxy_pass http://127.0.0.1:4000;
  proxy_http_version 1.1;
  proxy_set_header Upgrade $http_upgrade;
  proxy_set_header Connection "upgrade";
  proxy_read_timeout 300s;
}
```

`proxy_http_version 1.1` makes the upstream leg capable of upgrading. The two `proxy_set_header` lines pass the client's `Upgrade` value through and force `Connection` to request the upgrade.

The `proxy_read_timeout` matters more here than anywhere else. For a WebSocket it counts the time between two reads on an established connection, so a quiet-but-healthy socket gets closed by Nginx after the timeout — the default is only 60 seconds. Raise it for the WebSocket location, or make the application send periodic pings inside the timeout window.

Verify the handshake with curl:

```
curl -i http://chat.example.com/ws/ \
  -H "Connection: Upgrade" -H "Upgrade: websocket" \
  -H "Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==" \
  -H "Sec-WebSocket-Version: 13"
# HTTP/1.1 101 Switching Protocols
# Upgrade: websocket
# Connection: upgrade
```

The status to look for is `101` with the upgrade headers echoed back. If you get a `200` or a `400` from the application instead, the upgrade headers didn't survive the proxy — almost always a missing `proxy_http_version 1.1` or a location that overrode `proxy_set_header` and dropped the pair.

Server-sent events and other long-lived streaming responses share part of this story. They stay ordinary HTTP, so no upgrade headers, but they need `proxy_buffering off;` in the location and the same deliberate read timeout, or events pile up in Nginx's buffer while the browser sees nothing.

Two operational notes. Capacity planning must include connections that stay open — a thousand idle WebSockets still occupy a thousand connections. And during a reload, old workers keep already-established sockets alive until they close, so long-lived connections delay old workers from exiting. Watch what happens to an open connection while you reload.

Check your understanding: reverse proxy applications

Check proxypass URI behavior, forwarded headers, Unix sockets, timeouts, buffering, and WebSocket upgrades.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

HTTPS and access

Install certificates, redirect HTTP, choose TLS settings, and protect private paths.

Install a certificate chain

Configure the certificate, private key, names, permissions, and chain required for a trusted HTTPS server.

TLS needs a certificate matching the requested hostname and a private key readable by the Nginx master process. In Nginx that's two directives on the HTTPS server block:

```
server {
  listen 443 ssl;
  server_name shop.example.com;

  ssl_certificate     /etc/letsencrypt/live/shop.example.com/fullchain.pem;
  ssl_certificate_key /etc/letsencrypt/live/shop.example.com/privkey.pem;

  root /var/www/shop;
}
```

The word **fullchain** is the important one. Clients validate your certificate by walking a chain: your leaf certificate, signed by an intermediate, signed by a root the client already trusts. The client has the root but usually not the intermediate, so your server must send the leaf and the intermediates together. `fullchain.pem` is exactly that concatenation. Serve the complete certificate chain expected by clients, in that order — leaf first.

Protect the private key. It should be readable by root only:

```
sudo chmod 600 /etc/letsencrypt/live/shop.example.com/privkey.pem
```

That's enough for Nginx because the master process runs as root and reads the key at startup and reload; the unprivileged workers never open the file themselves. A key that other users can read is a key you should consider leaked.

Verify the live handshake

Don't trust the files — test what the server actually presents. Inspect the site with `openssl s_client -connect host:443 -servername host`:

```
openssl s_client -connect shop.example.com:443 -servername shop.example.com </dev/null
# Certificate chain
#  0 s:CN = shop.example.com
#    i:C = US, O = Let's Encrypt, CN = R11
#  1 s:C = US, O = Let's Encrypt, CN = R11
#    i:C = US, O = Internet Security Research Group, CN = ISRG Root X1
# ...
# Verify return code: 0 (ok)
```

Read three things: the subject of certificate 0 matches your hostname, the chain includes the intermediate, and the verify return code is 0. The `-servername` flag matters because it drives SNI — without it a multi-site server may present a different certificate than browsers see. Check the expiry dates too, and repeat this test after every renewal or path change. None of this prints private key material, which is how it should be.

The classic failure is pointing `ssl_certificate` at the bare `cert.pem` instead of `fullchain.pem`. Your browser may still show a padlock, because browsers cache intermediates or fetch them on the

fly. But curl, mobile apps, and API clients fail with `unable to get local issuer certificate`, and `s_client` reports `Verify return code: 21`. If browsers work while scripts fail, check the chain first.

Redirect HTTP to HTTPS

Keep the port 80 server small and redirect each valid hostname to its HTTPS equivalent without reflecting arbitrary hosts.

Once your site serves HTTPS, the HTTP listener on port 80 doesn't disappear. It stays for two jobs: redirecting humans who typed the plain URL, and answering the HTTP-based certificate validation challenges your ACME client uses for renewals.

Keep that server block small and boring:

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

  location /.well-known/acme-challenge/ {
    root /var/www/certbot;
  }

  location / {
    return 301 https://www.example.com$request_uri;
  }
}
```

Two details carry the weight here. The ACME challenge location comes first as a real file path, so certificate renewals keep working over plain HTTP even though everything else redirects. And the redirect target is a **known canonical hostname**, written out literally.

Why not `https://$host$request_uri`? Because `$host` comes from the client's Host header, and building a redirect from unvalidated client input means anyone can make your server issue redirects to a name you never intended. This block only matches your declared `server_name` values anyway, so hardcoding the canonical name costs nothing and removes the question. Keep unrelated traffic out entirely with a separate default server:

```
server {
  listen 80 default_server;
  server_name _;
  return 444;
}
```

Requests carrying random or hostile Host values hit this block and get the connection closed, instead of bouncing through your redirect.

Test all three cases

```
curl -I http://www.example.com/pricing
# HTTP/1.1 301 Moved Permanently
# Location: https://www.example.com/pricing
```

```
curl -I http://example.com/pricing
# Location: https://www.example.com/pricing   <- apex funnels to canonical

curl -I http://203.0.113.10/ -H "Host: evil.example.net"
# curl: (52) Empty reply from server          <- default server, no redirect
```

Confirm each response reaches the intended boundary: canonical host redirects to itself, the alternate host funnels to the canonical one, and an unknown Host value gets nothing useful.

The pitfall that bites in production is redirecting the ACME path along with everything else. Renewals then fail weeks later with a challenge error, long after the config change that caused it, and the certificate quietly expires. After adding or reshuffling redirects, verify the challenge path still returns 404 from the file root rather than a 301:

```
curl -I http://www.example.com/.well-known/acme-challenge/test
# HTTP/1.1 404 Not Found   <- served from disk, not redirected
```

Choose TLS and HSTS policy

Use current protocol settings, secure session behavior, and HSTS only after HTTPS is complete across the domain.

TLS policy changes over time, so start from current Nginx and certificate-provider guidance rather than old copied snippets. A config pasted from a 2015 blog post ships protocol versions that scanners now flag and browsers now warn about.

Today the defensible baseline is short:

```
ssl_protocols TLSv1.2 TLSv1.3;
ssl_prefer_server_ciphers off;
ssl_session_cache shared:SSL:10m;
ssl_session_timeout 1d;
```

`ssl_protocols` disables the obsolete versions — TLS 1.0 and 1.1 have no legitimate audience left. With only modern protocols enabled, `ssl_prefer_server_ciphers off` lets clients pick the cipher best suited to their hardware. The shared session cache lets returning clients resume sessions instead of paying for a full handshake, which is a measurable latency win.

Alongside the protocol settings, keep certificate renewal automated and monitor expiry. An expired certificate takes the site down more reliably than any cipher choice ever will.

HSTS is a commitment, not a checkbox

HSTS (HTTP Strict Transport Security) tells browsers to require HTTPS for your domain for a period:

```
add_header Strict-Transport-Security "max-age=31536000" always;
```

Once a browser sees this header, it refuses plain HTTP for `max-age` seconds — a year, here — and there's no server-side undo. That's the point, and that's the risk. So roll it out in stages: start with a small `max-age` like `300`, confirm nothing on the domain still needs HTTP, then raise it.

The `includeSubDomains` token extends the promise to every subdomain. Add it only when every subdomain is ready. The classic disaster is enabling it on `example.com` while `intranet.example.com` still runs plain HTTP: every browser that visited the main site now re-

fuses to open the intranet, for the full max-age, and you can't fix it remotely.

Verify what you're serving:

```
curl -sI https://www.example.com/ | grep -i strict
# strict-transport-security: max-age=31536000
```

Then test the whole TLS configuration with a current external scanner such as SSL Labs. It checks protocol versions, chain, and HSTS from the outside and tells you what real clients experience. Repeat the scan after any TLS change.

Document your supported TLS versions and your HSTS decision, including the max-age schedule. Preserve a rollback path for compatibility changes — protocols you can re-enable in one line, and an HSTS max-age you grew gradually instead of jumping straight to a year.

Protect private locations

Restrict health, metrics, and administrative paths by network or authentication without treating obscurity as access control.

Private endpoints often expose operational data or powerful actions: metrics, debug pages, admin panels, internal status. A hard-to-guess URL is not a security boundary. Scanners enumerate paths all day, and one leaked log line reveals the ”secret” URL forever. Put a real control in front instead.

Nginx gives you two that compose well: network restrictions and authentication.

Restrict by network

Use `allow` and `deny` for trusted network ranges:

```
location = /status {
  allow 10.0.0.0/8;
  allow 127.0.0.1;
  deny all;
  proxy_pass http://127.0.0.1:3000;
}
```

Rules are evaluated top to bottom and the first match wins, so the pattern is: list the trusted ranges, end with `deny all`. Anyone outside the ranges gets a `403 Forbidden` before the request ever reaches the application.

Restrict by authentication

For access from anywhere, add HTTP basic auth over HTTPS:

```
sudo apt install apache2-utils
sudo htpasswd -c /etc/nginx/.htpasswd flavio
```

```
location /metrics {
  auth_basic "Private area";
  auth_basic_user_file /etc/nginx/.htpasswd;
  proxy_pass http://127.0.0.1:9100;
}
```

Unauthenticated requests get `401 Unauthorized`. Basic auth sends credentials on every request, so it belongs behind TLS only — never expose it on a plain HTTP listener.

Verify both boundaries from the outside:

```
curl -I https://app.example.com/status          # from an untrusted network
# HTTP/1.1 403 Forbidden

curl -I https://app.example.com/metrics
# HTTP/1.1 401 Unauthorized

curl -I -u flavio:the-password https://app.example.com/metrics
# HTTP/1.1 200 OK
```

Test the failure cases with the same care as the success case. An endpoint you believe is restricted deserves proof from a network that shouldn't reach it.

Now the trap: **the apparent client address changes behind another trusted proxy**. If a load balancer or CDN sits in front of Nginx, `allow` and `deny` evaluate the proxy's address, not the visitor's. Depending on the ranges you wrote, that either locks everyone out or — worse — lets everyone in because the proxy's internal address falls inside your allowed range. In that topology you need the real IP restored from a forwarding header (the realip module), configured to trust only your proxy's addresses, before address rules mean anything.

Create a harmless private status path on your test server. Verify access from an allowed address and rejection from a disallowed one, including the proxy-address assumptions.

Check your understanding: https and access

Check certificate names and chains, HTTP redirects, TLS policy, HSTS, private locations, and client addresses.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Operate and troubleshoot

Read logs, control traffic, tune buffering and caching, and diagnose production failures.

Read access and error logs

Correlate request records and error messages by time, host, URI, status, upstream, and request identifier.

Nginx writes two different stories. The access log records completed requests, one line each: who asked for what and what they got. The error log records operational details at configured severity levels: failed upstream connections, permission problems, configuration warnings. Debugging usually means reading both and connecting them.

On Ubuntu they live at `/var/log/nginx/access.log` and `/var/log/nginx/error.log`.

The default access format tells you the status code but not why a request was slow. Define a format that includes upstream timing and request identifiers:

```
log_format timed '$remote_addr [$time_local] "$request" $status '
                 'rt=$request_time urt=$upstream_response_time rid=$request_id';
```

```
access_log /var/log/nginx/access.log timed;
```

`$request_time` is the full time Nginx spent on the request, client included. `$upstream_response_time` is how long the application took. Comparing the two answers the eternal question "is it us or the app?" — a request with `rt=9.1 urt=0.2` was slow because of the client or the network, not the upstream. `$request_id` is a unique value per request; forward it upstream as a header and you can match one Nginx line to one application log entry exactly.

Be deliberate about what you don't log. Avoid logging secrets in query strings or headers — tokens in URLs, `Authorization` values, session cookies. Logs get shipped, backed up, and read by more people than the live database.

Correlate the two logs

Make one successful and one failing request against a test server whose upstream is stopped, then read both files:

```
tail -n 2 /var/log/nginx/access.log
# 203.0.113.7 [03/Aug/2026:17:02:11 +0000] "GET /health HTTP/1.1" 200 rt=0.000 urt=- rid=7f1c
# 203.0.113.7 [03/Aug/2026:17:02:15 +0000] "GET /api/orders HTTP/1.1" 502 rt=0.001 urt=0.001

grep "connect() failed" /var/log/nginx/error.log | tail -n 1
# 2026/08/03 17:02:15 [error] 1205#1205: *18 connect() failed (111: Connection refused)
#   while connecting to upstream, ... upstream: "http://127.0.0.1:3000/api/orders"
```

The access line says a 502 happened. The error line, at the same timestamp, says why: connection refused on port 3000. That pairing — status from the access log, cause from the error log — is the core troubleshooting move.

During an incident, narrow before you read: a tight time window plus a request ID beats scrolling. One config note: the error log's level is set by `error_log /var/log/nginx/error.log warn;` — leave it at `warn` or `error` normally, and remember that a quiet error log at level `error` may hide warnings that explain tomorrow's outage.

Control rate and connections

Limit abusive request rates and concurrent work while preserving normal bursts and clear failure behavior.

Nginx can limit request rate per key and concurrent connections per key. This protects expensive endpoints from one aggressive client — a runaway script, a scraper, a brute-force attempt — without touching application code.

Rate limiting takes two directives. In the `http` context you define a zone; in the location you apply it:

```
limit_req_zone $binary_remote_addr zone=api:10m rate=10r/s;

server {
  location /api/ {
    limit_req zone=api burst=20 nodelay;
    limit_req_status 429;
    proxy_pass http://127.0.0.1:3000;
```

```
  }
}
```

The **key** is `$binary_remote_addr`, the client IP in compact form, so each address gets its own budget of 10 requests per second. The 10-megabyte zone holds the counters for roughly 160,000 addresses.

The **burst** is what keeps real users happy. A browser opening a page fires a burst of requests in the same instant — HTML, then a volley of assets and API calls. Without `burst`, everything past the steady rate in that instant is rejected, and ordinary page loads break. `burst=20` allows 20 requests to exceed the rate momentarily, and `nodelay` serves them immediately instead of queueing them.

`limit_req_status 429` makes rejections return **429 Too Many Requests**. The default is 503, which monitoring systems read as "server broken" rather than "client throttled" — set 429 so your dashboards stay honest.

Concurrent connections are a separate limit for long-held work like downloads:

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

location /downloads/ {
  limit_conn perip 5;
}
```

Test both behaviors separately: normal traffic first, then an intentional excess:

```
for i in $(seq 1 40); do
  curl -s -o /dev/null -w "%{http_code}\n" https://app.example.com/api/ping
done
# 200 x ~30, then 429 for the rest
```

The first ~30 requests pass (rate plus burst), then rejections start. Each rejection also writes a `limiting requests` line to the error log, which is how you'll spot throttling in production.

Choose a key that represents the client boundary you trust. Behind a CDN or load balancer, `$binary_remote_addr` is the proxy's address, so all your users share one bucket and throttle each other — restore the real client IP first, or key on something else. And keep perspective: edge limits complement application authorization and business rules; they do not replace them.

Cache deliberately

Cache only responses with a clear key, lifetime, bypass policy, and invalidation or versioning strategy.

Caching can reduce upstream work dramatically, but a wrong cache key can serve one user another user's data. That's not a performance bug, it's a data leak. So caching in Nginx is a policy decision you write down before you write config.

Start with public immutable assets or explicitly public responses — a product listing, a rendered docs page, an image. Never start with anything behind a login.

The setup takes a `proxy_cache_path` in the `http` context and a few directives in the location:

```
proxy_cache_path /var/cache/nginx keys_zone=app:10m max_size=1g inactive=60m;

server {
  location /products/ {
```

```
    proxy_cache app;
    proxy_cache_valid 200 10m;
    proxy_cache_bypass $http_authorization $cookie_session;
    proxy_no_cache $http_authorization $cookie_session;
    add_header X-Cache-Status $upstream_cache_status;
    proxy_pass http://127.0.0.1:3000;
  }
}
```

Walk through the policy this encodes. The **key** defaults to `$scheme$proxy_host$request_uri`, which already includes the query string. Include every representation-changing input in the key — if the response varies by anything else, like a language header, add it via `proxy_cache_key` or don't cache. The **lifetime** is explicit: 200 responses live 10 minutes. The **bypass policy** is the safety net: requests carrying an `Authorization` header or a session cookie skip the cache both ways — `proxy_cache_bypass` stops them reading a cached copy, `proxy_no_cache` stops their responses from being stored. You need both, and forgetting the second one is how private pages end up cached.

The `X-Cache-Status` header exposes `$upstream_cache_status`, which makes the whole thing testable:

```
curl -sI https://shop.example.com/products/ | grep -i x-cache
# X-Cache-Status: MISS
curl -sI https://shop.example.com/products/ | grep -i x-cache
# X-Cache-Status: HIT

curl -sI https://shop.example.com/products/ -H "Authorization: Bearer abc123" | grep -i x-cac
# X-Cache-Status: BYPASS
```

MISS then HIT proves the cache works. BYPASS on the authorized request proves the safety policy works. That third check is the test that proves private responses cannot enter the cache — run it before shipping, not after a report.

For invalidation, prefer **versioning** over purging: fingerprinted asset URLs like `/assets/app.9f31c2.css` never need invalidation because a new deploy produces a new URL. Where you must refresh in place, a short `proxy_cache_valid` lifetime is the honest tool.

Choose one safe cache candidate. Write its key, freshness lifetime, purge or versioning method, and the bypass test — then implement exactly that.

Diagnose a production request

Trace one failing request through DNS, listener, server selection, location, files or upstream, TLS, and response logs.

A 502, a timeout, a TLS error, and a wrong virtual host are different failures. Each one points at a different layer, so the job is to find the first boundary that does not match expectations — then stop and look there.

Learn the two status codes that get confused constantly. **502 Bad Gateway** means Nginx reached for the upstream and got a broken answer: connection refused, connection reset, or garbage instead of HTTP. The process is down or crashing. **504 Gateway Time-out** means the upstream accepted the connection but didn't respond within `proxy_read_timeout`. The process is up but stuck or slow.

Same symptom for the user, opposite investigations for you.

Work the chain in order, from the outside in.

Reproduce with the exact hostname, because server selection depends on it:

```
curl -sv https://api.example.com/orders -o /dev/null
```

The verbose output shows DNS resolution, the TLS handshake with the certificate names, and the final status. Half your diagnosis is in this one command: wrong IP means DNS, handshake failure means certificates, wrong content means a server-selection problem.

Then confirm the layers on the server itself:

```
sudo ss -lntp | grep nginx                    # is the listener up on 443?
sudo nginx -T | grep -n "api.example.com"     # which server block owns this name?
sudo tail -n 20 /var/log/nginx/error.log      # what does Nginx say happened?
```

`nginx -T` prints the effective configuration, which beats reading files that might not even be included. The error log names the failing upstream and the reason: `connect() failed (111: Connection refused)` reads as 502, `upstream timed out (110: Connection timed out)` reads as 504.

Then step past Nginx and test the upstream directly:

```
curl -i http://127.0.0.1:3000/orders
```

If this fails the same way, Nginx is innocent and the problem is the application. If this works while the proxied request fails, the problem lives in the proxy configuration between them — headers, timeouts, or the `proxy_pass` URI.

The discipline that makes this fast: change one layer only after evidence points there. Restarting the app, editing the config, and flushing DNS all at once tells you nothing about which one mattered.

Practice on a disposable failure. Stop the upstream service, watch the 502 appear, and record the client result, the Nginx error line, the direct upstream test, the repair, and the final health check. Ten minutes of practice buys you calm at 3am.

Check your understanding: operate and troubleshoot

Check access and error logs, rate and connection controls, caching, file limits, upstream failures, and final review.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/nginx/.