WEB SERVERS

Nginx In Depth

Worker Tuning · Upstream Pools & Load Balancing

Reverse Proxy Headers · Caching & Rate Limiting

Performance Tuning & Troubleshooting

Capstone: A Production Reverse-Proxy Tier

Philip Osztromok

Generated with Claude

Table of Contents

Nginx In Depth — 10 Chapters

Web Servers Fundamentals compared Apache, Nginx, and IIS across architecture, configuration philosophy, virtual hosting, TLS, and performance — but deliberately kept its own Nginx material introductory in three specific places. This course exists to go deep on exactly those three places, and nowhere else.

What Web Servers Fundamentals Already Covered

Three chapters there touched Nginx directly, each one explicitly scoped as foundational rather than exhaustive: Chapter 2 introduced Nginx's event-driven architecture at a conceptual level — worker processes, a non-blocking event loop — without tuning any of it for a real workload. Chapter 6 introduced `proxy_pass` and round-robin load balancing as the basic reverse-proxy mechanism, explicitly deferring the real depth to this course. Chapter 9 covered `gzip`, keep-alive, and a first mention of `proxy_cache` at a surface level, as one part of a three-server comparison rather than a serious performance-tuning treatment.

Why This Course Is Nginx-Only

The comparative question — Apache vs. Nginx vs. IIS, and which one actually fits a given job — already belongs to Web Servers Fundamentals, and its own capstone decision framework is the right place to answer it. This course assumes that question is already settled, one way or another: either Nginx is genuinely the right fit for the job in front of you, or you simply want to understand Nginx itself in real depth. Either way, there's no more comparing here — every chapter from this point on is Nginx configuration, Nginx directives, and Nginx-specific behavior.

What This Course Actually Adds

- **Chapter 2** — worker process and worker connection tuning for a real workload, not just the conceptual architecture overview
- **Chapter 3** — location block matching precedence, `try_files`, and `rewrite` rules in depth
- **Chapter 4** — `upstream {}` blocks: defining a real pool of backend servers
- **Chapter 5** — load balancing algorithms beyond round robin: `least_conn`, `ip_hash`, weighted balancing
- **Chapter 6** — the proxy headers a backend actually needs to know who the real client is
- **Chapter 7** — `proxy_cache` zones, cache keys, and invalidation, well beyond Chapter 9's brief mention
- **Chapter 8** — rate limiting and connection limiting to protect a backend from abuse or overload
- **Chapter 9** — performance tuning and troubleshooting: buffers, `sendfile`, `stub_status`, diagnosing real misconfigurations
- **Chapter 10** — a capstone that designs one complete, production-shaped Nginx reverse-proxy tier

TOPIC	WEB SERVERS FUNDAMENTALS	THIS COURSE
Architecture	Conceptual overview (Ch.2)	Real worker/connection tuning (Ch.2)
Reverse proxy	`proxy_pass` basics, round robin (Ch.6)	Upstream pools, multiple algorithms, headers (Ch.4-6)
Caching & performance	Brief mentions (Ch.9)	Full `proxy_cache` and tuning treatment (Ch.7, 9)
Comparison to Apache/IIS	Central focus of the whole course	None — Nginx only

WHERE THIS SHOWS UP BEYOND A STANDALONE SERVER

Kubernetes' own Ingress controllers are very commonly built on exactly the mechanisms this course covers — upstream pools, load balancing algorithms, and reverse-proxy headers, just generated and managed by Kubernetes itself rather than hand-edited. Understanding this course's material directly demystifies a large part of what an Ingress controller is actually doing under the hood.

"IN DEPTH" DOESN'T MEAN "START FROM ZERO, BUT HARDER"

This course assumes the basic Nginx configuration syntax from Web Servers Fundamentals — `server {}` blocks, `location` blocks, the general shape of `nginx.conf` — is already familiar. It doesn't re-teach that syntax from scratch; it builds directly on top of it. Anyone who hasn't gone through Web Servers Fundamentals' own Nginx material (Chapters 2, 4, 6, and 9 in particular) may want to revisit those chapters first, rather than starting here expecting a ground-up introduction.

Hands-On Exercises

EXERCISE 1

In your own words, explain why this course doesn't include an Apache or IIS comparison anywhere, even though Web Servers Fundamentals covered all three servers side by side.

View solution

EXERCISE 2

Name the three specific chapters in Web Servers Fundamentals this course directly extends, and what each one left deliberately shallow.

View solution

EXERCISE 3

A reader with no prior exposure to Nginx at all wants to start with this course directly, skipping Web Servers Fundamentals entirely. Using this chapter's own warning box, explain why that's likely to go poorly, and what you'd recommend instead.

View solution

CHAPTER 1 QUICK REFERENCE

- This course is **Nginx-only** — the Apache/IIS comparison already belongs to Web Servers Fundamentals
- Three chapters this course directly extends: **Ch.2** (architecture), **Ch.6** (reverse proxy basics), **Ch.9** (performance/caching basics)
- Assumes familiarity with basic Nginx config syntax from Web Servers Fundamentals — doesn't re-teach it from scratch
- Ten chapters ahead: worker tuning, location routing, upstream pools, load balancing algorithms, proxy headers, caching, rate limiting, performance/troubleshooting, and a capstone

Web Servers Fundamentals Chapter 2 described Nginx's event-driven model conceptually: a small number of worker processes, each an event loop handling many connections without a thread per connection. This chapter tunes that model for a real workload — the two directives that actually control it, the OS-level mechanism underneath, and the real ceiling those directives run into.

worker_processes

```
worker_processes auto;
```

`worker_processes` sets how many worker processes Nginx runs. Setting it to `auto` matches it to the number of available CPU cores — generally the right starting point, since each worker's own event loop runs most efficiently with its own dedicated core. Setting it too low under-uses available hardware; setting it meaningfully higher than the core count doesn't add real capacity and just adds context-switching overhead between more processes than the machine can genuinely run in parallel.

worker_connections

```
events {
    worker_connections 1024;
}
```

`worker_connections`, set inside the `events {}` block, caps how many simultaneous connections a single worker can hold open. The real theoretical maximum number of simultaneous connections Nginx can handle is `worker_processes` × `worker_connections` — but this number is also capped by the operating system's own file descriptor limit, since every open connection consumes one file descriptor. Raising `worker_connections` without also raising the OS limit (`worker_rlimit_nofile`, and the OS's own `ulimit`) simply hits that ceiling instead of the number configured in Nginx.

The Event Loop in Practice

On Linux, Nginx's event loop is built on **epoll**, a kernel mechanism that lets a single process ask "which of these thousands of connections actually has new data ready?" without checking each one individually — the technical foundation underneath Web Servers Fundamentals' own conceptual description of "non-blocking I/O." This is what lets one worker process track thousands of mostly-idle connections cheaply, rather than needing a dedicated thread watching each one.

multi_accept and accept_mutex

Two lesser-known but real tuning directives, both inside `events {}`: `multi_accept` controls whether a worker accepts every waiting new connection in one pass or just one per event-loop cycle; `accept_mutex` historically serialized which worker was allowed to accept a new incoming connection, to avoid all workers waking up for the same single new connection (a "thundering herd"). Modern Nginx defaults `accept_mutex` to `off`, since modern kernels already handle this efficiently on their own — a good example of a once-important tuning knob that's since become mostly unnecessary.

DIRECTIVE	DEFAULT	CONTROLS
`worker_processes`	`auto` (recent versions)	How many worker processes run
`worker_connections`	512	Max simultaneous connections per worker
`worker_rlimit_nofile`	Unset (inherits OS limit)	Max open file descriptors per worker
`accept_mutex`	`off`	Whether workers serialize accepting new connections

DIFFERENT TUNING PHILOSOPHIES FOR DIFFERENT ENVIRONMENTS

On a single dedicated VPS, tuning `worker_processes` / `worker_connections` upward to use all available hardware makes sense. Inside a Kubernetes deployment (per this course's own Ingress-controller connection from Chapter 1), a single Nginx instance is often deliberately kept small and constrained — scaling out with more Pods running more Nginx instances, rather than scaling one instance up vertically, is frequently the preferred approach in that environment.

A HUGE WORKER_CONNECTIONS NUMBER ISN'T FREE PERFORMANCE

It's tempting to assume setting `worker_connections` to some very large number guarantees more capacity. Each held-open connection consumes real memory and one real file descriptor — a number set far beyond what the machine's own RAM and OS file-descriptor limit (`ulimit -n`) can actually support does nothing useful; connections simply fail once that real ceiling is hit, regardless of what the config file says. The actual capacity ceiling is `worker_processes` × `worker_connections`, capped by whatever the operating system genuinely allows — not an arbitrarily large number typed into a config file.

Hands-On Exercises

EXERCISE 1

A server has 4 CPU cores and worker_connections set to 1024. Calculate the theoretical maximum number of simultaneous connections this configuration allows, and name the one other real-world limit that could still cap this number lower.

View solution

EXERCISE 2

A team sets worker_connections to 100000 on a small VPS with a default OS file-descriptor limit of 1024 per process, expecting this to significantly raise their server's capacity. Using this chapter's own warning box, explain what will actually happen.

View solution

EXERCISE 3

Explain, in your own words, why epoll allows a single Nginx worker to efficiently track thousands of mostly-idle connections, rather than needing a dedicated thread to check each connection individually.

View solution

CHAPTER 2 QUICK REFERENCE

- **worker_processes** — usually `auto`, matching CPU core count
- **worker_connections** — max simultaneous connections per worker; theoretical total capacity is `worker_processes` × `worker_connections`
- Real capacity is also capped by the OS's own file descriptor limit (`worker_rlimit_nofile` / `ulimit`) — not just the config file's own numbers
- **epoll** is the Linux kernel mechanism underneath Nginx's own non-blocking event loop
- **accept_mutex** — off by default on modern kernels, a once-important knob that's now mostly unnecessary

Web Servers Fundamentals Chapter 4 introduced a plain `location /path/ {}` block as a way to match a request path. In practice, Nginx supports several different kinds of location match, each with its own precedence — and getting that precedence wrong is one of the most common real-world sources of "why isn't this location matching what I expect" confusion.

Location Matching Types

```
location = /exact       { /* exact match */ }
location ^~ /assets/    { /* prefix, stop checking regex if matched */ }
location ~ \.php$       { /* regex, case-sensitive */ }
location ~* \.(jpg|png)$ { /* regex, case-insensitive */ }
location /blog/         { /* plain prefix match */ }
```

Nginx supports five distinct forms: an **exact match** (`=`), a **prefix match that stops further checking** (`^~`), a **case-sensitive regex** (`~`), a **case-insensitive regex** (`~*`), and a plain **prefix match** with no modifier at all. Each of these is a different kind of test against the request path, not just a different way of writing the same thing.

Matching Precedence: The Actual Order Nginx Checks

Nginx evaluates these in a fixed order, regardless of what order they're written in the config file: first, any **exact match** (`=`) — if one matches, Nginx stops immediately. Next, the **longest matching prefix** among plain and `^~` prefixes is found; if that longest match used the `^~` modifier, Nginx stops there too, skipping regex entirely. Otherwise, every **regex** location (`~`/`~*`) is checked *in the order they appear in the config file*, and the first one that matches wins. Only if no regex matches at all does Nginx fall back to that earlier plain-prefix match. Regex order genuinely matters — prefix order does not.

try_files

```
location / {
    try_files $uri $uri/ /index.html;
}
```

`try_files` checks a list of candidates in order and serves the first one that actually exists — conceptually the same idea as Web Servers Fundamentals Chapter 5's directory-index list, but applied to serving a request generally, not just a bare-directory request specifically. The line above is the standard single-page-application pattern: try the exact URI as a real file, then as a directory, and if neither exists, fall back to serving `index.html` and letting the client-side app's own router take over from there.

rewrite

```
rewrite ^/old-path/(.*)$ /new-path/$1 permanent;
```

`rewrite` uses a regular expression to rewrite the request URI itself. Adding `permanent` sends the client an actual HTTP 301 redirect to the new URL; `redirect` sends a 302; omitting both flags rewrites the URI *internally* — Nginx serves the new path directly without the client ever seeing a redirect at all, which is a very different, easy-to-miss outcome from what the directive's name might suggest.

SYNTAX	TYPE	PRECEDENCE
`location = /path`	Exact match	Checked first, wins immediately
`location ^~ /path/`	Prefix, stops regex checking	Checked second, wins if longest prefix
`location ~ pattern`	Regex, case-sensitive	Checked in file order, first match wins
`location ~* pattern`	Regex, case-insensitive	Same tier as `~`, file order
`location /path/`	Plain prefix	Fallback if no regex matches

WHERE TRY_FILES SHOWS UP IN A REAL DEPLOYMENT

Every single-page application built in React, Vue, or Svelte (this site's own frontend framework courses) needs exactly the `try_files $uri $uri/ /index.html;` pattern shown above once deployed behind a real Nginx server — without it, refreshing the browser on any client-side route other than the homepage returns a 404, since that path genuinely doesn't exist as a file on disk and only the app's own JavaScript router knows how to handle it.

LOCATION BLOCKS ARE NOT CHECKED TOP-TO-BOTTOM LIKE A FIREWALL RULE LIST

It's a very common and reasonable-sounding assumption that Nginx checks `location` blocks strictly in the order they're written, the way a firewall's rule list works. That's not what happens: per this chapter's own precedence section, exact matches and the longest prefix are found algorithmically regardless of file order, and only *among regex blocks specifically* does file order actually decide a tie. Writing location blocks assuming top-to-bottom evaluation is one of the most common real sources of "my more specific rule isn't matching" bugs.

Hands-On Exercises

EXERCISE 1

Write the Nginx location block(s) needed to serve a single-page application: any request that matches a real file or directory should be served directly, and everything else should fall back to /index.html.

View solution

EXERCISE 2

A config has both `location /images/ { ... }` and `location ~* \.(jpg|png)$ { ... }`, with the plain prefix block written first in the file. A request comes in for /images/photo.jpg. Explain which block actually handles it, and why the order they're written in doesn't decide the outcome here.

View solution

EXERCISE 3

Explain the practical difference between `rewrite ^/old/(.*)$ /new/$1 permanent;` and the same rewrite with no flag at all, from the perspective of what the client's browser actually sees.

View solution

CHAPTER 3 QUICK REFERENCE

- Five location match types: **exact (=)**, **priority prefix (^~)**, **regex (~/~*)**, and plain **prefix**
- Precedence order: exact match → longest prefix (stopping if `^~`) → regex blocks in file order → fallback to the longest plain prefix
- **Regex order matters**; prefix order does not — a common source of routing bugs
- **try_files** checks candidates in order, serving the first that exists — the standard SPA fallback pattern is `try_files $uri $uri/ /index.html;`
- **rewrite** with `permanent` / `redirect` sends a real HTTP redirect; with no flag, it rewrites the URI internally with no redirect visible to the client

Web Servers Fundamentals Chapter 6 showed `proxy_pass` pointing at a single backend address. Once there's more than one backend instance to distribute traffic across, that single address is replaced with an `upstream {}` block — a named pool of servers, with real per-server behavior configurable on top.

The upstream {} Block

```
upstream backend_pool {
    server 10.0.0.11:3000;
    server 10.0.0.12:3000;
}

server {
    location / {
        proxy_pass http://backend_pool;
    }
}
```

`upstream` names a group of backend servers, listed by their own address and port. That name — `backend_pool` here — is then used in place of a single address inside `proxy_pass`. Nginx distributes incoming requests across every `server` line inside that block, using round robin by default (Chapter 5 covers the other algorithms available).

server Entries Inside upstream

```
upstream backend_pool {
    server 10.0.0.11:3000 weight=3;
    server 10.0.0.12:3000;
    server 10.0.0.13:3000 max_fails=3 fail_timeout=30s;
    server 10.0.0.14:3000 backup;
}
```

Each `server` line can carry its own parameters: `weight` shifts the distribution — a server with `weight=3` receives roughly three times the traffic of a default `weight=1` server, useful when backend instances have genuinely different capacity. `max_fails` / `fail_timeout` mark a server temporarily unavailable after a set number of failed attempts within a given time window, automatically routing around it until that window passes. `backup` marks a server as only used once every non-backup server is unavailable. `down` manually takes a server out of rotation without deleting its line — useful for planned maintenance, since the config still documents that server's existence.

Keepalive to Upstream Servers

```
upstream backend_pool {
    server 10.0.0.11:3000;
    server 10.0.0.12:3000;
    keepalive 32;
}
```

Web Servers Fundamentals Chapter 9 covered keep-alive between the *client and Nginx*. This `keepalive` directive is a separate concept: it lets Nginx reuse its own already-open connections to a backend server across multiple proxied requests, rather than opening a fresh TCP connection to the backend for every single request it forwards — a real performance win on the backend-facing side, distinct from anything covered on the client-facing side.

PARAMETER	CONTROLS
`weight`	Proportional share of traffic this server receives
`max_fails` / `fail_timeout`	Temporarily removing a failing server from rotation
`backup`	Only used once every primary server is unavailable
`down`	Manually excluded, kept documented in the config
`keepalive` **(in upstream block)**	Reusing Nginx's own connections to the backend pool

A FAMILIAR SHAPE IF YOU'VE USED KUBERNETES

An `upstream` block naming a pool of backend addresses is conceptually similar to what a Kubernetes Service abstracts automatically over a set of Pods (Kubernetes Fundamentals' own territory) — both exist to let something in front of a backend pool treat "many instances" as one logical destination, just configured by hand here rather than generated automatically by an orchestrator.

MAX_FAILS/FAIL_TIMEOUT IS NOT A REAL HEALTH CHECK

It's easy to assume `max_fails`/`fail_timeout` gives real backend health monitoring. In open-source Nginx, this mechanism is entirely **reactive** — it only reacts after real client requests have already failed against a server, rather than actively probing backend health ahead of time the way a dedicated health-check system does. (Nginx Plus, the paid commercial version, does add active health checks that proactively probe backends — but that's a separate, paid capability, not something open-source Nginx's own `max_fails` mechanism provides.)

Hands-On Exercises

EXERCISE 1

Write an upstream block named api_pool containing three backend servers on 10.0.0.21:8080, 10.0.0.22:8080, and 10.0.0.23:8080, where the third server should receive twice as much traffic as the other two.

View solution

EXERCISE 2

A team relies entirely on max_fails/fail_timeout and considers their backend pool "actively monitored for health" as a result. Using this chapter's own warning box, explain what's actually true and what's missing from this claim.

View solution

EXERCISE 3

Explain why the keepalive directive inside an upstream block is a genuinely different concept from the client-facing keep-alive settings covered in Web Servers Fundamentals, even though both are called "keep-alive."

View solution

CHAPTER 4 QUICK REFERENCE

- **upstream {}** — names a pool of backend servers, referenced by name inside `proxy_pass`
- **weight** — proportional traffic share; **max_fails**/**fail_timeout** — reactive failure tracking; **backup** — last-resort server; **down** — documented manual exclusion
- **keepalive** inside `upstream {}` reuses Nginx's own connections to the backend — a separate concept from client-facing keep-alive
- Open-source Nginx's failure tracking is **reactive only** — not a substitute for a real, proactive health-check system

Web Servers Fundamentals Chapter 6 introduced only plain round robin, and Chapter 4 of this course added weighting on top of it. Nginx supports two genuinely different algorithms beyond that — each solving a different real problem round robin doesn't handle well.

Round Robin, Revisited

Round robin (Nginx's own default, with no algorithm directive needed) cycles through the servers in an `upstream` block in order. With `weight` (Chapter 4), it becomes **weighted round robin** — the cycle still goes in order, but a higher-weighted server appears more often within that cycle. Round robin distributes request *count* evenly (adjusted by weight); it has no idea whether any individual request is fast or slow.

least_conn

```
upstream backend_pool {
    least_conn;
    server 10.0.0.11:3000;
    server 10.0.0.12:3000;
}
```

`least_conn` routes each new request to whichever backend currently has the **fewest active connections**, rather than blindly cycling through the list. This matters when requests take meaningfully different amounts of time to process: round robin can pile several long-running requests onto one server purely by coincidence of ordering, leaving another server comparatively idle, while `least_conn` actively avoids sending new work to an already-busy server.

ip_hash

```
upstream backend_pool {
    ip_hash;
    server 10.0.0.11:3000;
    server 10.0.0.12:3000;
}
```

`ip_hash` routes based on a hash of the client's own IP address, so the same client consistently lands on the same backend across requests — solving **session persistence** ("sticky sessions") for an application that stores session state locally in each backend's own memory rather than in a shared store. The trade-off: if many clients share one IP address (common behind a large corporate NAT or proxy), those clients all get hashed to the same backend, producing an uneven distribution that has nothing to do with actual load.

The General hash Directive

```
upstream backend_pool {
    hash $cookie_session_id consistent;
    server 10.0.0.11:3000;
    server 10.0.0.12:3000;
}
```

`hash` generalizes `ip_hash`'s own idea to any key, not just client IP — a session cookie value, as shown above, or a URL, letting the same key consistently map to the same backend regardless of which client IP it came from. The optional `consistent` parameter uses a consistent-hashing algorithm, meaning adding or removing a backend server reshuffles only a small fraction of existing key-to-backend mappings, rather than potentially reshuffling nearly all of them.

ALGORITHM	DIRECTIVE	BEST FIT
Round robin (weighted)	*default* + `weight`	Uniform, short-lived requests
Least connections	`least_conn`	Requests with variable processing time
IP hash	`ip_hash`	Simple sticky sessions by client IP
General hash	`hash`	Sticky sessions or cache-locality by any key

A REAL SCENARIO WHERE THIS STOPS BEING OPTIONAL

A Node.js application storing session data purely in each server process's own memory (no shared session store) genuinely needs `ip_hash` or `hash` to function correctly under load balancing at all — without it, a user's session data could exist only on the one backend instance that happened to handle their first request, with every subsequent request potentially landing on a different instance that has never heard of that session.

STICKY SESSIONS ARE A WORKAROUND, NOT A SUBSTITUTE FOR SHARED SESSION STORAGE

It's tempting to treat `ip_hash` / `hash` -based sticky sessions as just as good as storing session state in a shared store like Redis. It's a real, useful workaround, but it has genuine limitations this chapter's own material already exposed: uneven distribution behind a large NAT, and — more seriously — if the one backend holding a client's session data goes down, that session is lost entirely, with no other backend able to pick it up. Externalizing session state into a shared store instead makes every backend genuinely stateless, so *any* load-balancing algorithm — including plain round robin — works safely, and a backend failing doesn't destroy anyone's session.

Hands-On Exercises

EXERCISE 1

A backend pool handles requests that sometimes take 50ms and sometimes take 8 seconds, with no predictable pattern to which requests are slow. Which load balancing algorithm from this chapter would you recommend, and why would plain round robin be a worse fit here?

View solution

EXERCISE 2

A company using ip_hash notices that a disproportionate share of their traffic always lands on the same one or two backend servers, even though total request volume is evenly spread across many different users. Using this chapter's own material, explain the likely cause.

View solution

EXERCISE 3

A team relies on ip_hash for session persistence and considers this "just as reliable" as using a shared session store like Redis. Using this chapter's own warning box, explain what real risk this team is exposed to that a shared session store would eliminate.

View solution

CHAPTER 5 QUICK REFERENCE

- **Round robin** (default, + `weight`) — distributes request count evenly, blind to how long each request takes
- **least_conn** — routes to whichever backend has the fewest active connections; better for variable-duration requests
- **ip_hash** — sticky sessions by client IP; risks uneven distribution behind a shared-IP NAT
- **hash** (optionally `consistent`) — sticky sessions or cache-locality on any key, not just IP
- Sticky sessions are a workaround — a shared session store makes backends genuinely stateless and safe under any algorithm

Web Servers Fundamentals Chapter 6 showed `proxy_pass` forwarding a request to a backend, but never addressed what the backend actually sees once that happens. By default, it's less than you'd expect — and fixing that is this chapter's whole job.

The Problem: The Backend Only Sees Nginx

Once Nginx proxies a request, the TCP connection the backend application actually receives comes from Nginx's own IP address, not the real client's. Left unaddressed, this silently breaks anything at the application layer that depends on knowing the real client — IP-based rate limiting, geolocation, access logs that should show real visitor IPs — because as far as the backend can tell, every single request came from the same one machine: Nginx itself.

X-Forwarded-For

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

`X-Forwarded-For` is the de facto standard header carrying the original client's IP address to the backend. Using Nginx's own `$proxy_add_x_forwarded_for` variable — rather than just `$remote_addr` — matters specifically when a request passes through more than one proxy hop: it *appends* the current client IP to any existing `X-Forwarded-For` value already on the request, building a comma-separated chain, rather than overwriting whatever a previous proxy already added.

X-Forwarded-Proto and X-Forwarded-Host

```
proxy_set_header X-Forwarded-Proto $scheme;
proxy_set_header X-Forwarded-Host $host;
```

`X-Forwarded-Proto` tells the backend whether the original client's request was HTTP or HTTPS — critical when TLS is terminated at Nginx itself (Web Servers Fundamentals Chapter 7), since Nginx's own internal connection to the backend is very often plain HTTP regardless of what the original client used. Without this header, a backend can incorrectly conclude the whole request was insecure, which can break secure-cookie logic or redirect behavior that depends on knowing HTTPS was actually used. `X-Forwarded-Host` similarly preserves the original `Host` header the client actually requested, in case Nginx's own internal request to the backend uses a different hostname.

X-Real-IP

```
proxy_set_header X-Real-IP $remote_addr;
```

`X-Real-IP`, popularized by Nginx itself, is a simpler alternative to `X-Forwarded-For`: just the immediate client's IP address as a single value, with no comma-separated chain to parse. Many applications support reading either header; `X-Real-IP` is convenient specifically when there's exactly one proxy hop and no chain to represent.

proxy_set_header

`proxy_set_header` is the general directive underlying every example above — it sets (or overrides) a header on the request Nginx sends to the backend. It's worth knowing this directive doesn't behave like some others when nested inside a `location` block, which the next section covers directly.

HEADER	PURPOSE
`X-Forwarded-For`	Chain of client IPs across multiple proxy hops
`X-Real-IP`	The immediate client's single IP address
`X-Forwarded-Proto`	Whether the original request was HTTP or HTTPS
`X-Forwarded-Host`	The original Host header the client requested

CLOSING THE GAP WEB SERVERS FUNDAMENTALS RAISED

Web Servers Fundamentals Chapter 7's own warning box flagged that terminating TLS at the web server doesn't automatically mean the hop to a backend stays encrypted — and that a backend has no automatic way of knowing the original connection was HTTPS at all. `X-Forwarded-Proto` is precisely the mechanism that closes that specific gap, letting the backend correctly recognize a request as having arrived over HTTPS even though its own direct connection to Nginx is plain HTTP.

A LOCATION BLOCK WITH ITS OWN PROXY_SET_HEADER STOPS INHERITING ALL OF THEM

This is a well-known and easy-to-miss Nginx gotcha: if an outer `server {}` block sets several `proxy_set_header` directives, and a nested `location {}` block then sets even **one** `proxy_set_header` of its own, that location block stops inheriting *every* `proxy_set_header` from the parent context — not just the one it explicitly overrode. Every header actually needed inside that location block then has to be repeated there explicitly, or it silently disappears from requests that go through it, with no error or warning to flag the omission.

Hands-On Exercises

EXERCISE 1

Write the three proxy_set_header directives needed so a backend receives the original client's IP address chain, the original protocol (HTTP or HTTPS), and the original Host header.

View solution

EXERCISE 2

A backend application redirects users to an insecure http:// URL even though every client reaches the site over HTTPS, with TLS terminated at Nginx. Using this chapter's own material, explain the likely missing piece and how to fix it.

View solution

EXERCISE 3

A server {} block sets X-Forwarded-For and X-Forwarded-Proto. A location /api/ {} block inside it adds one more header, X-Api-Version, via its own proxy_set_header. Explain what happens to X-Forwarded-For and X-Forwarded-Proto for requests handled by that location block, and how to fix it.

View solution

CHAPTER 6 QUICK REFERENCE

- By default, a proxied backend only sees **Nginx's own IP**, not the real client's — breaking anything that depends on the real client identity
- **X-Forwarded-For** — client IP chain via `$proxy_add_x_forwarded_for`; **X-Real-IP** — single immediate client IP
- **X-Forwarded-Proto/X-Forwarded-Host** — preserve the original scheme and Host header, closing the gap left by TLS termination at Nginx
- **A location block with its own proxy_set_header stops inheriting ALL parent headers**, not just the one it overrides — every needed header must be repeated there explicitly

Web Servers Fundamentals Chapter 9 mentioned `proxy_cache` only briefly, and its own warning box named a real, serious risk: caching personalized content incorrectly can leak one user's data to another. This chapter covers the actual mechanism behind that risk — and how to configure caching correctly rather than just being warned about it.

proxy_cache_path: Defining a Cache Zone

```
proxy_cache_path /var/cache/nginx levels=1:2 keys_zone=my_cache:10m max_size=1g
inactive=60m;
```

`proxy_cache_path`, set once at the `http {}` level, defines a cache zone: `/var/cache/nginx` is where cached response bodies are actually stored on disk; `keys_zone=my_cache:10m` names this zone `my_cache` and reserves 10MB of shared memory for storing cache keys and metadata (not the cached content itself); `max_size=1g` caps the total cache size; `inactive=60m` removes any cache entry that hasn't been accessed in 60 minutes, regardless of how long ago it was originally cached.

Enabling Caching: proxy_cache

```
location /api/ {
    proxy_cache my_cache;
    proxy_pass http://backend_pool;
}
```

Defining a cache zone doesn't turn caching on by itself — `proxy_cache`, referencing that zone's name, actually enables caching for a specific `location` or `server` block.

The Cache Key: What Actually Gets Cached

```
proxy_cache_key "$scheme$proxy_host$request_uri$cookie_session_id";
```

Every cached response is stored under a **cache key** — by default, a combination of the scheme, backend host, and request URI (`$scheme$proxy_host$request_uri`). This default key is exactly the mechanism behind Web Servers Fundamentals' own warning box: if two different users request the identical URL, and personalization (a logged-in dashboard, per-user content) depends on something the default cache key doesn't account for — a session cookie, an `Authorization` header — both users' requests hash to the *same* cache key, and whichever response gets cached first is served to both. The fix shown above adds the session cookie's own value into the key itself, so different users genuinely produce different cache keys and never collide.

Cache Validity & Bypassing

```
proxy_cache_valid 200 10m;
proxy_cache_bypass $cookie_logged_in;
proxy_no_cache $cookie_logged_in;
```

`proxy_cache_valid` sets how long a response with a given status code stays cached (here, successful 200 responses for 10 minutes). `proxy_cache_bypass` skips reading from the cache for a request matching a given condition — here, whenever a `logged_in` cookie is present; `proxy_no_cache` similarly prevents a matching response from being *written* to the cache at all. Both are commonly set together, for the same condition, to keep logged-in traffic entirely out of the cache in both directions.

Cache Invalidation Strategies

Open-source Nginx has no built-in "purge this specific cached URL right now" command. Real invalidation strategies instead rely on: keeping `proxy_cache_valid` times short enough that stale content ages out naturally, deliberately changing the cache key itself when content changes in a way that should be reflected immediately, or upgrading to the paid Nginx Plus edition, which adds a genuine `proxy_cache_purge` directive. For debugging what's actually happening, adding `add_header X-Cache-Status $upstream_cache_status;` to a response reveals whether a given request was a cache `HIT`, `MISS`, or `BYPASS`.

DIRECTIVE	PURPOSE
`proxy_cache_path`	Defines a named cache zone: disk location, size, expiry
`proxy_cache`	Turns caching on for a specific location/server block
`proxy_cache_key`	What actually distinguishes one cached entry from another
`proxy_cache_valid`	How long a given response status stays cached
`proxy_cache_bypass` / `proxy_no_cache`	Skip reading/writing the cache under a given condition

THIS IS THE MECHANISM BEHIND WEB SERVERS FUNDAMENTALS' OWN WARNING

That course's Chapter 9 warned, in the abstract, that "caching dynamic or personalized content incorrectly... is a real, well-documented class of caching bug." This chapter's own cache-key material is the concrete mechanism underneath that warning: the bug happens specifically when the cache key doesn't include whatever actually varies the response per user, causing two genuinely different responses to collide under one identical key.

PROXY_CACHE_BYPASS ALONE IS NECESSARY, BUT NOT SUFFICIENT

It's tempting to assume adding `proxy_cache_bypass` for logged-in requests fully solves the personalization problem. It only stops a bypassed request from *reading* a cached response — it does nothing to prevent an *earlier*, non-personalized-looking request from having already written a stale, wrongly-shared response into the cache under a key that doesn't actually vary by user identity. Genuinely personalized content needs either its own identity-aware cache key component (as shown above) or `proxy_no_cache` to keep it out of the cache entirely — bypassing reads alone leaves the underlying cache-key collision risk fully in place.

Hands-On Exercises

EXERCISE 1

Write a proxy_cache_path directive defining a zone named product_cache, storing up to 500MB, with cache keys/metadata using 5MB of shared memory, and removing entries unused for 30 minutes.

View solution

EXERCISE 2

A site caches a per-user "My Orders" page using the default cache key. Explain, using this chapter's own material, exactly why two different logged-in users can end up seeing each other's order history, and write the proxy_cache_key fix.

View solution

EXERCISE 3

A team adds proxy_cache_bypass $cookie_logged_in; and considers their personalized-content caching bug fully fixed. Using this chapter's own warning box, explain what risk still remains.

View solution

CHAPTER 7 QUICK REFERENCE

- **proxy_cache_path** defines a zone (disk location, key/metadata memory, size, expiry); **proxy_cache** turns it on for a location/server block
- The **cache key** (default: scheme + host + URI) is the actual mechanism behind Web Servers Fundamentals' own personalized-caching warning — anything that varies the response but isn't in the key can collide across users
- **proxy_cache_valid** — expiry per status code; **proxy_cache_bypass**/**proxy_no_cache** — skip reading/writing under a condition
- Open-source Nginx has **no built-in purge-by-URL** — invalidation relies on short validity times, changing the cache key, or Nginx Plus's paid `proxy_cache_purge`
- **proxy_cache_bypass alone is not sufficient** — genuinely personalized content needs an identity-aware cache key or `proxy_no_cache`

Chapter 4's own `max_fails` mechanism only reacts after a backend has already started failing. This chapter covers a genuinely different, proactive protection: limiting how much traffic any single client can send in the first place, before a backend is ever put under real strain.

limit_req_zone & limit_req

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

location /api/ {
    limit_req zone=req_limit burst=20 nodelay;
    proxy_pass http://backend_pool;
}
```

`limit_req_zone`, set at the `http {}` level, defines a shared-memory zone tracking request rate per key — here, per client IP (`$binary_remote_addr`), capped at 10 requests per second. `limit_req` then applies that zone inside a specific location. `burst=20` allows up to 20 requests above the steady rate to be queued rather than immediately rejected, absorbing short traffic spikes; `nodelay` processes those burst requests immediately instead of queuing/delaying them, at the cost of allowing a brief burst above the strict steady rate.

limit_conn_zone & limit_conn

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

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

`limit_conn` is a distinct mechanism from `limit_req`: rather than capping requests-per-second over time, it caps how many **simultaneous open connections** a single key (again, commonly the client IP) can hold at once — useful for preventing one client from monopolizing a meaningful share of the available worker connections (Chapter 2), for example by opening many concurrent large file downloads at once.

Choosing a Key

`$binary_remote_addr` — the client's IP in binary form, more memory-efficient than the plain-text IP — is the most common key for both directives. The same caveat from Chapter 5's own `ip_hash` material applies here too: many real, distinct users sharing one visible IP address (behind a large corporate NAT) will all share the identical rate or connection limit, since the key can't distinguish between them.

What Happens When the Limit Is Hit

By default, a request exceeding either limit receives an HTTP **503 Service Unavailable** — customizable via `limit_req_status` / `limit_conn_status`. The `burst` + `nodelay` combination specifically changes this behavior: without `nodelay`, requests within the burst allowance are queued and processed with an added delay rather than rejected outright, which can look like the site simply slowed down rather than an explicit block — worth knowing when diagnosing "why did this request take longer than expected" rather than assuming rate limiting only ever produces an outright rejection.

DIRECTIVE	PROTECTS AGAINST
`limit_req`	Too many requests per second from one key
`limit_conn`	Too many simultaneous open connections from one key
`burst` / `nodelay`	Tuning how short traffic spikes are handled

A REAL, PRACTICAL SECURITY CONTROL

Rate limiting is one of the most direct, practical defenses against brute-force login attempts and basic scraping/DoS traffic — directly relevant to Authentication & Session Security's own material on protecting login endpoints. A tight `limit_req` specifically on a login path is a genuinely useful, low-effort layer of defense against automated credential-stuffing attempts.

RATE LIMITING AT NGINX IS ONE LAYER, NOT A COMPLETE DOS DEFENSE

It's tempting to treat Nginx-level rate limiting as a complete defense against denial-of-service traffic. It genuinely protects against one or a few IPs overwhelming a single Nginx instance — but it does nothing against a truly distributed attack spread across many real, different IP addresses, each staying comfortably under the per-IP limit, and nothing against traffic that never reaches Nginx at all (a sufficiently large volumetric attack can saturate network capacity before Nginx's own rate limiting ever gets a chance to act). It's also, per this chapter's own key-selection section, subject to the same shared-IP/NAT caveat as `ip_hash` — a rate limit applied by IP can accidentally throttle an entire legitimate corporate office sharing one address. Real protection combines this with other layers, not this alone.

Hands-On Exercises

EXERCISE 1

Write the limit_req_zone and limit_req directives to limit a /login endpoint to 5 requests per second per client IP, allowing a burst of up to 10 requests processed without added delay.

View solution

EXERCISE 2

A team configures limit_req with burst=20 but no nodelay flag, then is confused when requests within that burst allowance still appear to succeed but take noticeably longer than usual, rather than being rejected outright. Explain why, using this chapter's own material.

View solution

EXERCISE 3

A company believes their Nginx-level rate limiting fully protects them from any denial-of-service attack. Using this chapter's own warning box, describe one realistic attack scenario this rate limiting would NOT stop.

View solution

CHAPTER 8 QUICK REFERENCE

- **limit_req_zone**/**limit_req** — caps requests per second per key; **burst** absorbs spikes; **nodelay** processes burst requests immediately instead of queuing them
- **limit_conn_zone**/**limit_conn** — caps simultaneous open connections per key, a distinct concern from request rate
- Both commonly key on **$binary_remote_addr** — subject to the same shared-IP/NAT caveat as Chapter 5's `ip_hash`
- Default rejection is a **503**; without `nodelay`, burst requests are delayed rather than rejected — a real source of "why did this slow down" confusion
- A genuinely useful layer against brute-force/basic DoS traffic, but **not a complete defense** against a distributed attack or network-saturating traffic

Web Servers Fundamentals Chapter 9 covered keep-alive and gzip at a general, three-server level. This chapter is Nginx-specific: how it actually moves bytes off disk and onto the network, how it manages a backend response before handing it to a client, and the tools to find out what's actually happening when something's slow.

sendfile, tcp_nopush & tcp_nodelay

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

`sendfile` lets the kernel copy a static file's data directly to the network socket, without that data passing through Nginx's own userspace process memory first — a real, meaningful win specifically for serving static files. `tcp_nopush` tells the kernel to wait and accumulate a full network packet's worth of data before sending, working alongside `sendfile` to batch data efficiently. `tcp_nodelay` does the seemingly opposite thing — disabling Nagle's algorithm so small packets are sent immediately rather than held back to be batched. The pairing isn't actually contradictory: Nginx uses `tcp_nopush` to batch data efficiently through most of a response, then `tcp_nodelay` to flush the final packet immediately rather than waiting — both enabled together is the correct, common configuration, not a conflict to resolve.

Proxy Buffering

```
proxy_buffering on;
proxy_buffer_size 4k;
proxy_buffers 8 4k;
```

By default, Nginx buffers a backend's response before sending it on to the client — letting the backend connection close quickly even if the client itself is slow to receive the data, rather than holding that backend connection open for the full duration of a slow client transfer. `proxy_buffer_size` sets the buffer for the response headers specifically; `proxy_buffers` sets the number and size of buffers used for the response body. A response larger than the configured buffer space spills over to a temporary file on disk, which can introduce real disk I/O the config didn't anticipate if buffers are sized too small for typical response sizes.

stub_status: A Real-Time Window Into Nginx

```
location /nginx_status {
    stub_status;
    allow 127.0.0.1;
    deny all;
}
```

The `stub_status` module exposes a simple plain-text endpoint showing active connections right now, and running totals for accepted/handled/total requests, plus how many connections are currently reading, writing, or waiting. It's a lightweight, immediate diagnostic tool — genuinely useful as a first stop, though it's not a substitute for a real observability stack (Observability's own material) that retains history, alerts, and correlates across multiple systems.

Diagnosing Common Misconfigurations

- **Check the error log first** (Web Servers Fundamentals Chapter 8) — many misconfigurations announce themselves there directly.
- **Check** `stub_status` to see whether Nginx itself is under load, or whether the real bottleneck is a slow backend it's simply waiting on.
- **Check** `X-Cache-Status` (Chapter 7) to rule caching in or out of a given slow response.
- **Check whether** `worker_connections` **or the OS file-descriptor limit** (Chapter 2) is silently being hit, rather than assuming the problem is purely about traffic volume.

DIRECTIVE	PURPOSE
`sendfile`	Kernel sends file data directly, bypassing userspace copy
`tcp_nopush` / `tcp_nodelay`	Batch most of a response, flush the final packet immediately
`proxy_buffering`	Buffer a backend response so a slow client doesn't hold the backend connection open
`stub_status`	Real-time connection/request counters for quick diagnosis

STUB_STATUS IS A FIRST STOP, NOT A DESTINATION

`stub_status`'s value is its immediacy — a quick, no-setup snapshot of what Nginx is doing right now. Observability's own material covers what comes next: retaining that data over time, alerting when it crosses a threshold, and correlating it against the rest of a system's own metrics — none of which `stub_status` itself does.

TURNING PROXY_BUFFERING OFF IS NOT AUTOMATICALLY "FASTER"

It's tempting to assume disabling buffering removes a layer of overhead and is therefore simply faster. In practice, `proxy_buffering off` means Nginx passes each chunk of the backend's response straight through to the client as it arrives — if the client is slow to receive data, the connection to the *backend* stays open for the entire duration of that slow transfer, tying up a backend connection (and the resources behind it) for much longer than buffering would have. The real trade-off is backend resource protection (buffering on) against marginally lower latency for an already-fast client (buffering off) — not a straightforward "off is obviously better" choice.

Hands-On Exercises

EXERCISE 1

Write the three directives that enable sendfile and the correct paired TCP behavior for serving static files efficiently.

View solution

EXERCISE 2

A backend response is under heavy load and its connections seem to stay open unusually long whenever a client on a slow mobile connection requests a large file. A teammate suggests turning proxy_buffering off to "speed things up." Using this chapter's own warning box, explain why this is likely to make the actual problem worse, not better.

View solution

EXERCISE 3

A site is running slowly. Using this chapter's own diagnostic checklist, list the order of checks you'd perform to figure out whether the bottleneck is Nginx itself, the backend, caching, or a connection limit — and briefly explain what each check would tell you.

View solution

CHAPTER 9 QUICK REFERENCE

- **sendfile** — kernel sends file data directly; **tcp_nopush**/**tcp_nodelay** — batch most of a response, flush the last packet immediately, both on together (not conflicting)
- **proxy_buffering** — protects backend connections from slow clients; turning it off shifts that cost onto the backend, not automatically "faster"
- **stub_status** — a lightweight, real-time first diagnostic stop; not a substitute for a real observability stack
- Diagnostic order: **error log** → **stub_status** → **X-Cache-Status** → **worker_connections/file-descriptor limits**

Web Servers Fundamentals' own capstone compared three servers across several scenarios. This one is different, by design: a single, cohesive worked example, layering in each of this course's own eight prior chapters, one at a time, until one complete production-shaped configuration exists.

The Scenario

A containerized API service runs three backend instances. It needs traffic distributed across them sensibly, protection from slow clients and abusive traffic, caching for its read-heavy public endpoints, correct client-identity forwarding, TLS termination, and basic real-time visibility — every one of this course's own chapters, applied to one real job.

Step 1 — Worker Tuning (Chapter 2)

```
worker_processes auto;

events {
    worker_connections 2048;
}
```

`worker_processes auto` matches the container's own allotted CPU cores; `worker_connections 2048` is set generously, on the assumption the OS file-descriptor limit has already been raised to match, per Chapter 2's own warning about the real ceiling being whichever of the two numbers is lower.

Step 2 — The Upstream Pool (Chapter 4)

```
upstream api_backend {
    least_conn;
    server api-1:3000;
    server api-2:3000;
    server api-3:3000;
    keepalive 32;
}
```

Three backend instances form the pool, with `keepalive 32` letting Nginx reuse its own connections to them (Chapter 4).

Step 3 — Choosing a Load Balancing Algorithm (Chapter 5)

`least_conn`, already shown above, is chosen deliberately: this API's own request times vary meaningfully by endpoint (Chapter 5's own case for preferring it over plain round robin), so routing toward whichever backend is currently least busy fits better than blindly cycling through the pool.

Step 4 — Reverse Proxy Headers (Chapter 6)

```
proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
proxy_set_header X-Forwarded-Proto $scheme;
proxy_set_header X-Real-IP $remote_addr;
```

These live in the outer `server {}` block, so every `location` nested inside it inherits them automatically — a deliberate choice, since Chapter 6's own warning box showed that any nested block adding its own `proxy_set_header` would otherwise silently lose all of these.

Step 5 — TLS Termination

TLS itself is configured exactly per Web Servers Fundamentals Chapter 7 — `listen 443 ssl`, `ssl_certificate` / `ssl_certificate_key` — not repeated here since that chapter already covers it fully; this capstone assumes it's already in place ahead of everything else below.

Step 6 — Caching Read-Heavy Endpoints (Chapter 7)

```
proxy_cache_path /var/cache/nginx keys_zone=api_cache:10m max_size=1g inactive=30m;

location /api/public/ {
    proxy_cache api_cache;
    proxy_cache_valid 200 5m;
    proxy_cache_bypass $cookie_session_id;
    proxy_no_cache $cookie_session_id;
}
```

Only the public, read-heavy `/api/public/` path is cached — `proxy_cache_bypass` / `proxy_no_cache` on the session cookie keep any authenticated request out of the cache in both directions, exactly the discipline Chapter 7's own warning box insisted on.

Step 7 — Rate Limiting (Chapter 8)

```
limit_req_zone $binary_remote_addr zone=api_limit:10m rate=20r/s;

location /api/ {
    limit_req zone=api_limit burst=40 nodelay;
    proxy_pass http://api_backend;
}
```

A per-IP rate limit protects the backend pool from any single abusive client, with a burst allowance absorbing normal traffic spikes without rejecting legitimate short bursts.

Step 8 — Performance & Monitoring (Chapter 9)

```
sendfile on;
tcp_nopush on;
tcp_nodelay on;
proxy_buffering on;

location /nginx_status {
    stub_status;
    allow 127.0.0.1;
    deny all;
}
```

`proxy_buffering` stays on deliberately (Chapter 9's own warning box), and `stub_status` is restricted to internal access only, giving ops visibility without exposing it publicly.

CAPSTONE STEP	CHAPTER IT DRAWS FROM
Step 1 — Worker tuning	Chapter 2
Step 2 — Upstream pool	Chapter 4
Step 3 — least_conn algorithm	Chapter 5
Step 4 — Proxy headers	Chapter 6
Step 5 — TLS termination	Web Servers Fundamentals Chapter 7
Step 6 — Caching	Chapter 7
Step 7 — Rate limiting	Chapter 8
Step 8 — Performance & monitoring	Chapter 9

TWO CAPSTONES, ONE FULL JOURNEY

Web Servers Fundamentals' own capstone answered "which server fits this job" — this one answers "given Nginx was already chosen, how do you actually configure it well." Together, the two capstones cover the full journey from choosing a web server to running a genuinely production-shaped configuration of it.

THIS CONFIG IS A SOLID FOUNDATION, NOT A COMPLETE GO-LIVE CHECKLIST

Every directive above reflects real material from this course, but a genuine production rollout still needs steps beyond any single config file: validating the config (`nginx -t`) before reloading, a gradual rollout rather than switching all traffic at once, and real alerting configured on top of whatever monitoring is in place (Observability's own material, well beyond `stub_status`'s own simple snapshot). Treat this chapter's config as a strong, correct starting point — not a substitute for an actual deployment process.

Hands-On Exercises

EXERCISE 1

This capstone's config places X-Forwarded-For, X-Forwarded-Proto, and X-Real-IP in the outer server {} block rather than inside the /api/ location block. Explain why this placement was a deliberate choice, referencing the relevant earlier chapter.

View solution

EXERCISE 2

A new /api/private/ endpoint is added, requiring authentication, returning genuinely personalized data per user. Should it be added to the api_cache zone the same way /api/public/ was? Explain your answer using this course's own material.

View solution

EXERCISE 3

Write a short chapter-attribution summary (2-3 sentences) explaining how this capstone's own shape differs from Web Servers Fundamentals' own capstone, and why that difference makes sense given what each course actually covers.

View solution

COURSE COMPLETE

Nginx In Depth — 10 of 10 chapters complete. The Web Servers subject now has both its comparative foundation and its Nginx deep dive.

CHAPTER 10 QUICK REFERENCE

- This capstone builds **one cohesive production config**, layering in Chapters 2, 4, 5, 6, 7, 8, and 9 step by step — a deliberately different shape from Web Servers Fundamentals' own multi-scenario capstone
- Proxy headers belong in the **outer server block** so every nested location inherits them (Chapter 6)
- Only **non-personalized, read-heavy** endpoints belong in a cache zone; authenticated/personalized endpoints need `proxy_no_cache` or their own identity-aware cache key (Chapter 7)
- A working config is a **foundation**, not a complete production rollout — validation, gradual deployment, and real alerting still matter beyond this chapter's own scope