WEB SERVERS

Apache In Depth

MPM Tuning · .htaccess & AllowOverride

Core Modules · Virtual Hosts & SNI

mod_ssl · mod_proxy & Load Balancing

Hardening with mod_security · Performance & Troubleshooting

Capstone: A Production Apache Deployment

Philip Osztromok

Generated with Claude

Table of Contents

Apache 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 Apache material introductory in several specific places. This course exists to go deep on exactly those places, and nowhere else.

What Web Servers Fundamentals Already Covered

Apache shows up in more of that course's chapters than either of its siblings, for a genuine reason worth naming honestly: Apache's per-directory configuration model and its MPM architecture are different enough from Nginx's single centralized config and event loop that nearly every comparative chapter had something real to say about Apache specifically, even while keeping it introductory. Chapter 2 introduced the three MPMs — `prefork`, `worker`, `event` — at a conceptual level, without tuning any of them for a real workload. Chapter 3 introduced the directory-context configuration model and `.htaccess` as Apache's own philosophy, without going deep on the directive precedence rules that make it actually predictable. Chapter 4 covered Apache virtual hosts at a basic name-based level. Chapter 6 introduced `mod_proxy` and `ProxyPass` as the basic reverse-proxy mechanism. Chapter 7 covered TLS termination generically across all three servers rather than Apache's own `mod_ssl` directives. Chapter 8 touched access control and hardening only at a survey level. Chapter 9 covered performance basics — compression, keep-alive — the same way, briefly and comparatively.

Why This Course Is Apache-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 Apache is genuinely the right fit for the job in front of you, or you simply want to understand Apache itself in real depth. Either way, there's no more comparing here — every chapter from this point on is Apache configuration, Apache directives, and Apache-specific behavior.

What This Course Actually Adds

- **Chapter 2** — MPM tuning for a real workload: choosing between `prefork`, `worker`, and `event`, and setting their real directives, not just the conceptual overview
- **Chapter 3** — the directory/location context model and `.htaccess` in depth: directive precedence, merge order, and when `.htaccess` is the right tool versus a liability
- **Chapter 4** — core modules beyond what Fundamentals ever touched: `mod_rewrite`, `mod_headers`, `mod_deflate`
- **Chapter 5** — virtual hosts in real depth: name-based, IP-based, and SNI-based hosting together in one server
- **Chapter 6** — `mod_ssl` in depth: certificate chains, cipher suite selection, OCSP stapling — well beyond Chapter 7's generic TLS-termination treatment
- **Chapter 7** — `mod_proxy` and `mod_proxy_balancer` as a real reverse proxy and load balancer, not just a single `ProxyPass` line
- **Chapter 8** — access control, authentication, and hardening with `mod_security`, well beyond Chapter 8's survey-level basics
- **Chapter 9** — performance tuning, log analysis, and troubleshooting real misconfigurations
- **Chapter 10** — a capstone that designs and hardens one complete, production-shaped Apache deployment

TOPIC	WEB SERVERS FUNDAMENTALS	THIS COURSE
MPM architecture	Conceptual overview (Ch.2)	Real MPM selection and tuning (Ch.2)
Directory config & .htaccess	Philosophy introduced (Ch.3)	Precedence, merge order, real tradeoffs (Ch.3)
TLS	Generic termination coverage (Ch.7)	Full `mod_ssl` directive treatment (Ch.6)
Reverse proxy	`mod_proxy` basics (Ch.6)	`mod_proxy_balancer`, real load balancing (Ch.7)
Comparison to Nginx/IIS	Central focus of the whole course	None — Apache only

WHY SO MUCH SHARED HOSTING STILL RUNS ON APACHE SPECIFICALLY

Budget shared-hosting environments and control panels like cPanel/WHM are overwhelmingly built on Apache, and `.htaccess` is the direct reason why: it lets a hosting provider give each tenant real per-directory configuration power — rewrite rules, custom error pages, access restrictions — without ever handing out access to the server's main configuration file. Nginx's centralized-config model, covered in Web Servers Fundamentals Chapter 3, has no real equivalent to this, which is exactly why that per-directory override model gets a full chapter of its own here (Chapter 3) rather than staying a passing comparison point.

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

This course assumes the basic Apache configuration syntax from Web Servers Fundamentals — `<VirtualHost>` blocks, the general shape of `httpd.conf`, and the idea of directory contexts — 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 Apache material (Chapters 2, 3, 4, 6, 7, 8, and 9 in particular — genuinely more of that course than Nginx In Depth needed) 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 Nginx or IIS comparison anywhere, even though Web Servers Fundamentals covered all three servers side by side.

View solution

EXERCISE 2

This chapter names seven Web Servers Fundamentals chapters that touched Apache directly — more than Nginx In Depth needed. Name at least five of them, what each left deliberately shallow, and explain in one sentence why Apache needed more of Fundamentals' own chapters than Nginx did.

View solution

EXERCISE 3

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

View solution

CHAPTER 1 QUICK REFERENCE

- This course is **Apache-only** — the Nginx/IIS comparison already belongs to Web Servers Fundamentals
- Seven Fundamentals chapters touched Apache directly: **Ch.2** (MPMs), **Ch.3** (directory context/.htaccess), **Ch.4** (virtual hosts), **Ch.6** (reverse proxy basics), **Ch.7** (TLS, generic), **Ch.8** (hardening survey), **Ch.9** (performance basics) — more than Nginx In Depth needed, because Apache's own config model differs so much from Nginx's
- Assumes familiarity with basic Apache config syntax from Web Servers Fundamentals — doesn't re-teach it from scratch
- Nine chapters ahead: MPM tuning, directory context/.htaccess depth, core modules, virtual hosts, `mod_ssl`, `mod_proxy_balancer`, hardening with `mod_security`, performance/troubleshooting, and a capstone

Web Servers Fundamentals Chapter 2 introduced Apache's three Multi-Processing Modules conceptually: Prefork (process-per-connection), Worker (processes plus threads), and Event (the modern default, which offloads idle keep-alive connections). This chapter tunes them for a real workload — the actual directives behind each one, how to check which is compiled in, the capacity math each one runs into, and the one dependency that has historically forced the choice for a huge number of real sites: PHP.

Checking Which MPM Is Actually Compiled In

```
# Debian/Ubuntu
apachectl -V | grep -i mpm

# or list every loaded module and grep for mpm
apachectl -M | grep mpm
```

Unlike Nginx, where the event loop is simply how the one binary works, Apache's MPM is chosen at build/package-install time and only one can be active at once. Most modern distributions ship `mpm_event` as the default package, but it's worth confirming rather than assuming — especially on an older server, or one where `mod_php` was installed at some point, since that package has historically pulled in `mpm_prefork` as a dependency.

Prefork In Depth

```
<IfModule mpm_prefork_module>
StartServers             5
    MinSpareServers          5
    MaxSpareServers         10
    MaxRequestWorkers       256
    MaxConnectionsPerChild    0
</IfModule>
```

Prefork spawns a pool of single-threaded child processes in advance, each handling exactly one connection at a time. `MaxRequestWorkers` (the modern name for the older `MaxClients`) is the real concurrency ceiling: it directly caps the number of simultaneous connections, full stop, since there's no threading underneath to multiply it further. `MaxConnectionsPerChild` set to `0` means a child process is never recycled — convenient, but it means any slow memory leak inside a module a request touches accumulates for the lifetime of the server rather than being cleared periodically.

Worker In Depth

```
<IfModule mpm_worker_module>
StartServers             3
    MinSpareThreads         75
    MaxSpareThreads        250
    ThreadsPerChild         25
    MaxRequestWorkers      400
    ServerLimit             16
</IfModule>
```

Worker runs a smaller pool of processes, each running multiple threads — `ThreadsPerChild` of them — so one process can hold several connections open at once, using less memory per connection than Prefork's one-process-each model. The real ceiling here is a genuine multiplication, the same shape as Nginx's own `worker_processes` × `worker_connections` math from Nginx In Depth Chapter 2: `MaxRequestWorkers` can go no higher than `ServerLimit` × `ThreadsPerChild` — in the defaults above, 16 × 25 = 400, which is exactly the value shown. Setting `MaxRequestWorkers` above that product without also raising `ServerLimit` simply gets silently capped at the lower number.

Event In Depth (the Modern Default)

```
<IfModule mpm_event_module>
StartServers              3
    MinSpareThreads          75
    MaxSpareThreads         250
    ThreadsPerChild          25
    MaxRequestWorkers       400
    ServerLimit              16
    AsyncRequestWorkerFactor  2
</IfModule>
```

Event shares Worker's exact directive set and the same `ServerLimit` × `ThreadsPerChild` math, plus one addition: `AsyncRequestWorkerFactor`. Worker's real weakness was that a thread holding an idle keep-alive connection open couldn't be reused for a new request — Event fixes exactly that, handing idle keep-alive connections off to a small dedicated listener thread instead of tying up a full worker thread. `AsyncRequestWorkerFactor` controls how many of those idle connections a single worker thread can effectively cover; the default of `2` means the real practical connection ceiling under Event can run meaningfully higher than `MaxRequestWorkers` alone would suggest for a workload with a lot of idle keep-alive time.

	PREFORK	WORKER	EVENT
Real ceiling directive	`MaxRequestWorkers` directly	`ServerLimit` × `ThreadsPerChild`	Same product, plus `AsyncRequestWorkerFactor` headroom
Idle keep-alive cost	A full process each	A full thread each	Near-zero — handed to a listener thread
mod_php compatible	Yes — the traditional pairing	No — not thread-safe	No — not thread-safe

mod_php and the Prefork Requirement

This is the one dependency that has forced more real-world MPM choices than any tuning preference: the traditional `mod_php` module runs the PHP interpreter directly inside each Apache worker, and that interpreter was never built to be thread-safe. Loading `mod_php` into a threaded MPM (Worker or Event) risks real crashes and corrupted output under concurrent load, so Apache refuses to load it under anything but Prefork. This single dependency is a large part of why so many long-running shared-hosting and WordPress deployments still run Prefork today, even on hardware that could otherwise easily handle Event's lower per-connection overhead.

The modern way around this is **PHP-FPM** (FastCGI Process Manager) — a separate, standalone PHP process pool that Apache talks to over a socket via `mod_proxy_fcgi`, rather than running PHP inside its own worker at all. Because PHP execution now happens in an entirely separate process, Apache itself is free to run Event (or Worker), getting Event's lower memory overhead under high concurrency on a PHP site — a genuine, concrete reason to migrate away from `mod_php` beyond just "it's older."

WATCHING REAL, LIVE MPM BEHAVIOR — NOT JUST THE CONFIG FILE'S NUMBERS

`mod_status`, enabled with `ExtendedStatus On` and a `/server-status` location, shows the real, live count of busy vs. idle workers or threads, broken down by what each one is currently doing (reading a request, sending a response, keep-alive, etc.). This is the practical way to tell whether the directives above are actually well-tuned for real traffic, rather than guessing from the config file's numbers alone.

A HIGH MAXREQUESTWORKERS NUMBER ISN'T FREE CAPACITY

Exactly the same trap as Nginx's own `worker_connections`, covered in Nginx In Depth Chapter 2: each Prefork process or Worker/Event thread that's actually busy consumes real memory — under Prefork specifically, a full process's worth. Setting `MaxRequestWorkers` to a large number without checking whether `MaxRequestWorkers` × your application's real per-process memory footprint fits inside available RAM just means the server starts swapping heavily, or the OOM killer starts terminating processes, once that real ceiling is hit — regardless of what the config file says is allowed.

Hands-On Exercises

EXERCISE 1

A server runs the Worker MPM with ServerLimit set to 16 and ThreadsPerChild set to 20, but MaxRequestWorkers is left at its old default of 400. Calculate the real effective concurrency ceiling this server will actually run into, and explain why it doesn't match the configured MaxRequestWorkers value.

View solution

EXERCISE 2

A team running Prefork sets MaxRequestWorkers to 2000 on a server with 4GB of RAM, where each PHP process typically uses 50MB, expecting this to significantly raise capacity. Using this chapter's own warning box, explain what will actually happen under real heavy load.

View solution

EXERCISE 3

Explain, in your own words, why a traditional mod_php installation forces Apache onto the Prefork MPM, and how switching to PHP-FPM changes that constraint.

View solution

CHAPTER 2 QUICK REFERENCE

- **Prefork** — one process per connection; real ceiling is `MaxRequestWorkers` directly; the only MPM compatible with traditional `mod_php`
- **Worker** — processes + threads; real ceiling is `ServerLimit` × `ThreadsPerChild`, not `MaxRequestWorkers` alone
- **Event** — same math as Worker, plus `AsyncRequestWorkerFactor` headroom from offloading idle keep-alive connections to a listener thread
- **PHP-FPM** (via `mod_proxy_fcgi`) runs PHP in its own process pool, freeing Apache to run Event even on a PHP site
- `mod_status` / `/server-status` shows real, live busy/idle worker counts — the practical way to check tuning, not just the config file's numbers

Web Servers Fundamentals Chapter 3 introduced `.htaccess` conceptually: a per-directory override file, checked on every request, with a real filesystem-scan cost in exchange for letting someone without access to the main config make changes. This chapter goes deep on the two things that actually determine what a given `.htaccess` file can do and how it interacts with everything else: `AllowOverride`'s real granularity, and the actual merge order across every context type Apache supports.

AllowOverride: What .htaccess Is Actually Allowed to Touch

```
<Directory "/var/www/html">
AllowOverride None
</Directory>
<Directory "/var/www/html/blog">
AllowOverride AuthConfig FileInfo
</Directory>
```

`AllowOverride` is not just an on/off switch — it names exactly which *groups* of directives a `.htaccess` file in that directory is permitted to set. `None` disables `.htaccess` checking entirely for that directory tree (and is the real fix for Fundamentals' own filesystem-scan cost, since Apache then knows not to look at all). `All` permits every group — broad and rarely the right default. In between, five named groups exist: `AuthConfig` (authentication directives — `AuthType`, `AuthName`, `Require`), `FileInfo` (`ErrorDocument`, and — notably — every `mod_rewrite` directive), `Indexes` (`IndexOptions`, `DirectoryIndex`, `IndexIgnore`), `Limit` (`Require`/`Order`/`Allow`/`Deny` access-control directives), and `Options[=...]` (the `Options` directive itself).

"INDEXES" THE ALLOWOVERRIDE GROUP VS. "INDEXES" THE OPTIONS VALUE — A REAL NAMING COLLISION

`AllowOverride Indexes` and `Options Indexes` are two completely different things that happen to share a name. The `AllowOverride` group called `Indexes` controls whether `.htaccess` is even allowed to set directory-listing-related directives at all (`IndexOptions`, `DirectoryIndex`). The `Options` value called `Indexes` is the actual setting that turns directory listing on. Granting `AllowOverride Indexes` does not itself enable directory listing — a `.htaccess` file in that directory would still need its own `Options +Indexes` or `Options -Indexes` line, and that specific line falls under the separate `Options` override group, not this one.

The Real Merge Order: Directory → .htaccess → Files → Location

Apache applies contexts in a fixed order, and later contexts can override directives set by earlier ones — but this is a **merge**, not a wholesale replacement. Directives a later context doesn't mention still carry over from the broader context that set them. The real order: (1) the main server config and any matching `<VirtualHost>`, (2) every `<Directory>` block that matches the requested path, evaluated from the *least specific path to the most specific*, followed by that most specific directory's own `.htaccess` file (if `AllowOverride` permits it), (3) `<Files>`/`<FilesMatch>`, and finally (4) `<Location>`/`<LocationMatch>`.

```
<Directory "/var/www/html">
Options Indexes FollowSymLinks
    AllowOverride None
</Directory>
<Directory "/var/www/html/private">
Options -Indexes
    Require ip 10.0.0.0/8
</Directory>
```

A request under `/private` merges both blocks: the more specific `/private` context isn't a full replacement of the parent, so `FollowSymLinks` is still inherited from the parent since `/private` never mentions it — only `Indexes` is actually touched, and the leading `-` means "remove this specific option," not "replace the whole `Options` list." The end result for `/private` is `FollowSymLinks` on, `Indexes` off, plus the new IP restriction — a genuine merge across two contexts, not the child context overwriting the parent's settings wholesale.

A Worked .htaccess Example

```
# Disable directory listing
Options -Indexes

# Custom 404 page
ErrorDocument 404 /errors/404.html

# Enable URL rewriting
RewriteEngine On
RewriteRule ^article/([0-9]+)$ article.php?id=$1 [L]
```

This one file actually needs two separate `AllowOverride` groups to work at all: `Options` (for the `Options -Indexes` line) and `FileInfo` (which covers both `ErrorDocument` and every `mod_rewrite` directive, including `RewriteEngine` and `RewriteRule`). A directory whose `AllowOverride` is set to just `AuthConfig` would reject every line in this file.

ALLOWOVERRIDE GROUP	COVERS	TYPICAL USE
AuthConfig	`AuthType`, `AuthName`, `Require`	Per-directory login restriction
FileInfo	`ErrorDocument`, all `mod_rewrite` directives	Custom error pages, URL rewriting
Indexes	`IndexOptions`, `DirectoryIndex`, `IndexIgnore`	Directory-listing appearance (not whether it's on)
Limit	`Require` / `Order` / `Allow` / `Deny`	IP-based access restriction
Options[=...]	The `Options` directive itself	Turning directory listing, symlink-following, etc. on/off

WHERE THIS POINTS FORWARD IN THIS COURSE

`AllowOverride None` everywhere it isn't genuinely needed is one of the standard hardening recommendations Chapter 8 covers in depth — it isn't just a performance win from skipping the per-request filesystem scan, it also closes off an entire class of misconfiguration risk from a compromised or careless `.htaccess` file, since there's nothing left for one to override.

Hands-On Exercises

EXERCISE 1

Using the two nested <Directory> blocks in this chapter's own merge-order example (Options Indexes FollowSymLinks on the parent, Options -Indexes on /private), state the effective Options settings a request to /private actually ends up with, and explain why FollowSymLinks survives even though /private never mentions it.

View solution

EXERCISE 2

A directory has AllowOverride set to AuthConfig only, but its .htaccess file (this chapter's own worked example) contains RewriteEngine/RewriteRule and Options directives. Using this chapter's own material, explain what actually happens when a request hits that directory — not just "it won't work," but the specific real behavior.

View solution

EXERCISE 3

Explain, in your own words, why "AllowOverride Indexes" does not by itself turn on directory listing, using this chapter's own warning box about the two different things named "Indexes."

View solution

CHAPTER 3 QUICK REFERENCE

- **AllowOverride** — `None` (no .htaccess checking), `All` (everything), or a named group: `AuthConfig`, `FileInfo`, `Indexes`, `Limit`, `Options`
- The `AllowOverride Indexes` group and the `Options Indexes` value are different things that share a name — one permits, the other enables
- Real merge order: main config/VirtualHost → `<Directory>` (least to most specific) + `.htaccess` → `<Files>` → `<Location>` — a merge, not a full replacement, at every step
- A directive a `.htaccess` file isn't permitted to set by `AllowOverride` doesn't fail silently
- `AllowOverride None` wherever genuinely possible is both a performance and a hardening win — expanded on in Chapter 8

Web Servers Fundamentals never covered these three modules directly — the closest it came was Chapter 9's brief, comparative mention of compression across all three servers, and Chapter 3's `FileInfo` override group naming `mod_rewrite` in passing without explaining it. This chapter is genuinely new territory: real URL rewriting, real response-header control, and the actual Apache mechanism behind that compression Chapter 9 only gestured at.

mod_rewrite: RewriteCond and RewriteRule

```
RewriteEngine On

# Force HTTPS
RewriteCond %{HTTPS} off
RewriteRule ^ https://%{HTTP_HOST}%{REQUEST_URI} [L,R=301]

# Force www
RewriteCond %{HTTP_HOST} !^www\. [NC]
RewriteRule ^ https://www.%{HTTP_HOST}%{REQUEST_URI} [L,R=301]
```

A `RewriteRule` matches and rewrites the request URI; a `RewriteCond` immediately above it is a precondition that must be true first — multiple consecutive `RewriteCond` lines are combined with AND by default. `%{HTTPS}` and `%{HTTP_HOST}` are server variables mod_rewrite exposes, not literal strings. The `[L]` flag means "stop processing further rewrite rules if this one matches" (the same idea as Nginx's own `^~` prefix-match short-circuit from Nginx In Depth Chapter 3), and `[R=301]` sends an actual HTTP redirect back to the client rather than rewriting the URI silently — the exact same external-redirect-vs-internal-rewrite distinction Nginx's own `rewrite ... permanent` flag draws, just spelled differently.

mod_headers: Controlling Response Headers

```
Header set X-Frame-Options "SAMEORIGIN"
Header set Cache-Control "public, max-age=3600"
Header unset X-Powered-By
```

`Header` adds, replaces, or removes headers on the response going back to the *client* — `set` replaces any existing value, `add` appends an additional header of the same name, `unset` removes it entirely. Common real uses: basic security headers like `X-Frame-Options` (full header-based hardening belongs to the site's own XSS and OWASP Top 10 courses, not this one), cache-control tuning for static assets, and stripping headers that leak server/framework details.

> **HEADER VS. REQUESTHEADER — TWO DIRECTIVES THAT SOUND THE SAME**
>
> `Header` modifies the response Apache sends back to the *client*. `RequestHeader` — a separate directive, easy to reach for by mistake — modifies headers on the request Apache forwards to a *backend*, relevant specifically in a reverse-proxy setup (Chapter 7). Using `Header` when the actual goal is to inject or strip a header before it reaches a proxied backend does nothing useful — the client sees the change, the backend never does.

mod_deflate: Compressing Responses

```
<IfModule mod_deflate.c>
AddOutputFilterByType DEFLATE text/html text/plain text/css application/javascript
application/json
    SetEnvIfNoCase Request_URI \.(?:gif|jpe?g|png|zip|gz|mp4)$ no-gzip
    DeflateCompressionLevel 6
</IfModule>
```

`AddOutputFilterByType` applies gzip compression only to the listed MIME types — text-based content compresses well; binary formats that are already compressed don't. That's exactly what the `SetEnvIfNoCase` line guards against: it sets an environment variable mod_deflate itself checks, skipping compression entirely for URIs ending in an already-compressed extension. Running gzip against an already-compressed JPEG or ZIP file burns real CPU for little or no size reduction, and occasionally makes the file slightly larger. `DeflateCompressionLevel` (1–9, default 6) trades CPU time for compression ratio — this is Apache's own concrete mechanism behind the "compression" line Web Servers Fundamentals Chapter 9 only mentioned in passing, the direct counterpart to Nginx's `gzip` directive.

MODULE	KEY DIRECTIVE	ALLOWOVERRIDE GROUP
mod_rewrite	`RewriteRule` / `RewriteCond`	FileInfo
mod_headers	`Header`	FileInfo
mod_deflate	`AddOutputFilterByType`	FileInfo

WHY FILEINFO IS SUCH A HEAVILY-USED ALLOWOVERRIDE GROUP

All three modules in this chapter share the exact same `AllowOverride` group from Chapter 3: `FileInfo`. That's not a coincidence — `FileInfo` was specifically designed to cover directives that shape how a response is *delivered* (rewritten, headered, compressed) rather than directives touching authentication or raw filesystem access. A single `AllowOverride FileInfo` line is enough to permit everything in this entire chapter inside a `.htaccess` file, with none of Chapter 3's `AuthConfig` or `Limit` groups needed at all.

Hands-On Exercises

EXERCISE 1

Write a mod_rewrite rule (RewriteCond + RewriteRule) that redirects any request to /old-blog/anything to https://example.com/blog/anything, preserving whatever came after /old-blog/, using a permanent external redirect rather than an internal rewrite.

View solution

EXERCISE 2

A developer wants to add a custom header to every request Apache forwards to a proxied backend application, so the backend can identify which Apache instance sent it. They write "Header set X-Proxy-Source apache1" in the config. Using this chapter's own warning box, explain why this won't achieve what they want, and what directive they should have used instead.

View solution

EXERCISE 3

Explain, in your own words, why applying mod_deflate's compression to a directory full of JPEG images would be a waste of CPU, and how the SetEnvIfNoCase line in this chapter's own mod_deflate example prevents it.

View solution

CHAPTER 4 QUICK REFERENCE

- **mod_rewrite** — `RewriteCond` gates a `RewriteRule`; `[L]` stops further rules, `[R=301]` sends a real external redirect instead of an internal rewrite
- **mod_headers** — `Header` touches the response to the client; `RequestHeader` (a separate directive) touches the request to a proxied backend
- **mod_deflate** — compress by MIME type with `AddOutputFilterByType`; skip already-compressed formats with `SetEnvIfNoCase ... no-gzip`; Apache's own counterpart to Nginx's `gzip`
- All three modules' directives fall under the same `AllowOverride FileInfo` group from Chapter 3

Web Servers Fundamentals Chapter 4 showed one `<VirtualHost>` block matched by `ServerName` / `ServerAlias`, on a single IP, over plain HTTP. That example quietly skipped three real questions: what happens when a server has more than one IP address, how HTTPS makes this whole mechanism dramatically harder, and what actually happens when a request's `Host` header doesn't match anything at all.

Name-Based Virtual Hosting (What Fundamentals Already Showed)

Every distinct `<VirtualHost>` sharing the same IP:port is matched purely by the `Host` header, exactly as Chapter 4 showed. One historical footnote worth knowing if working from an older tutorial: pre-2.4 Apache required an explicit `NameVirtualHost *:80` line to turn this matching on at all. Apache 2.4 removed that requirement — any `IP:port` combination shared by more than one `<VirtualHost>` automatically becomes name-based. A leftover `NameVirtualHost` line from an old config isn't fatal on 2.4 — Apache logs a deprecation warning at startup and ignores it — but it's dead weight worth removing.

IP-Based Virtual Hosting

```
<VirtualHost 192.0.2.10:80>
ServerName site-a.example
    DocumentRoot /var/www/site-a
</VirtualHost>
<VirtualHost 192.0.2.11:80>
ServerName site-b.example
    DocumentRoot /var/www/site-b
</VirtualHost>
```

IP-based hosting is a genuinely different mechanism, not just a variant of the same one: it distinguishes requests by *which IP address they arrived on*, not by the `Host` header at all — this requires the server to actually have multiple IP addresses bound to it (multiple NICs, or IP aliases on one interface). It's rare in modern deployments for the exact reason covered next: the historical problem that made IP-based hosting genuinely necessary for HTTPS has since been solved.

The Historical Problem: HTTPS Broke Name-Based Hosting

A TLS handshake — negotiating encryption and presenting a certificate — happens *before* any HTTP request, including the `Host` header, is ever sent; it has to, since the request itself needs to be encrypted using whatever the handshake just negotiated. That created a real chicken-and-egg problem for HTTPS name-based hosting: with several `<VirtualHost>` blocks sharing one IP, each potentially needing a different TLS certificate, Apache had no way to know *which* certificate to present during the handshake, since the only thing that would have told it which site was intended — the `Host` header — hadn't arrived yet. For years, the only real fix was IP-based hosting: one dedicated IP address per HTTPS domain, so the IP itself told Apache which certificate to present.

SNI: How Modern HTTPS Solves This

Server Name Indication (SNI) is a TLS extension that adds the target hostname into the very first message of the handshake — the `ClientHello` — in plain text, before encryption is even established. That's enough information for Apache to pick the correct `<VirtualHost>`, and therefore the correct certificate, before the rest of the handshake proceeds — solving the historical problem directly, and making genuinely name-based HTTPS hosting on a single IP the normal case today. Client support is effectively universal now (only truly ancient or specialized embedded clients lack it), which is exactly what makes Chapter 6's per-`<VirtualHost>` `mod_ssl` configuration possible in the first place.

The Default/Catch-All VirtualHost

When a request's `Host` header matches *none* of the `ServerName`/`ServerAlias` entries for the matching `IP:port`, Apache doesn't return an error — it silently serves whichever `<VirtualHost>` was defined *first* for that `IP:port`, in the order the config files were parsed (commonly alphabetical, for a typical `sites-enabled/` setup). This is one of the most common real-world Apache surprises: automated scanners and bots routinely probe a server by raw IP address rather than any real domain name, and whatever site happens to load first in the config silently answers every one of those requests.

> **AN UNMATCHED HOST HEADER ISN'T AN ERROR — IT'S WHICHEVER SITE LOADS FIRST**
>
> It's easy to assume a request for an unrecognized domain name simply fails, the way a typo'd URL might. It doesn't — it's served by the first-defined `<VirtualHost>` for that `IP:port`, with no warning that anything unusual happened. A common, deliberate fix is defining an explicit dummy `<VirtualHost>` block *first* in the load order — matching nothing meaningful, and configured to return a `403` or close the connection — specifically so an internal, staging, or simply the "wrong" site never becomes the accidental public face of unmatched traffic.

	NAME-BASED	IP-BASED
Distinguishes by	`Host` header	Which IP address the request arrived on
Requires	Nothing extra — one IP is enough	Multiple IP addresses on the server
Works with HTTPS on one IP?	Yes, via SNI	Yes, always — no SNI needed at all
Common today?	Yes — the normal case	Rare — mostly legacy or specialized isolation needs

CHECKING THE REAL MATCHING ORDER, NOT JUST READING THE CONFIG

`apachectl -S` dumps Apache's own fully-parsed virtual host configuration — every `IP:port` combination, which `<VirtualHost>` block matches each one, and critically, which one is currently acting as the default for unmatched requests. This is the practical way to confirm the real matching order this chapter describes, rather than manually tracing through include order across multiple config files by hand.

Hands-On Exercises

EXERCISE 1

Two <VirtualHost 203.0.113.5:80> blocks share the same IP:port — site-a.example is defined first in the config, site-b.example second. A request arrives with Host: unknown-domain.example. Which site serves this request, and why?

View solution

EXERCISE 2

Explain, in your own words, why hosting multiple HTTPS domains on a single IP address was impossible before SNI existed, and specifically what information SNI adds, and when, to solve it.

View solution

EXERCISE 3

A team migrating an old Apache 2.2 tutorial's config to a modern Apache 2.4 server includes a leftover "NameVirtualHost *:80" line. Explain what actually happens when Apache starts up with this line present — is it a fatal error, and is it still doing anything useful?

View solution

CHAPTER 5 QUICK REFERENCE

- **Name-based** — matched by `Host` header; the normal, modern case; needs SNI to work over HTTPS with multiple certs on one IP
- **IP-based** — matched by which IP the request arrived on; needs multiple real IPs; rare today
- **SNI** — the hostname is sent in the TLS `ClientHello`, before encryption, letting Apache pick the right certificate before the handshake finishes
- An unmatched `Host` header is silently served by the *first-defined* `<VirtualHost>` for that `IP:port` — not an error
- `apachectl -S` shows the real, fully-parsed matching order
- `NameVirtualHost` is deprecated/ignored (with a warning) on Apache 2.4+, not required or fatal

Web Servers Fundamentals Chapter 7 paired `SSLEngine on` with `SSLCertificateFile` / `SSLCertificateKeyFile` and deliberately stopped there — it assumes HTTPS/TLS Fundamentals already covers what a certificate or a TLS handshake actually is, and this chapter keeps that same assumption. What neither of those courses covered is the set of `mod_ssl` directives that actually determine how secure that connection really is: which TLS versions are even allowed, which ciphers get used, and how the client confirms the certificate hasn't been revoked without slowing every connection down.

SSLCertificateFile Can Now Hold the Full Chain

```
<VirtualHost *:443>
ServerName example.com
    SSLEngine on
    SSLCertificateFile /etc/ssl/certs/example.com-fullchain.crt
    SSLCertificateKeyFile /etc/ssl/private/example.com.key
</VirtualHost>
```

Since Apache 2.4.8, `SSLCertificateFile` accepts a single file containing the leaf certificate *and* the full intermediate chain, concatenated in order — the separate `SSLCertificateChainFile` directive Fundamentals never mentioned still works for older setups, but a modern config typically doesn't need it at all. This is exactly the kind of detail that changes across Apache versions and is easy to find outdated advice about.

SSLProtocol: Restricting Which TLS Versions Are Allowed

```
SSLProtocol -all +TLSv1.2 +TLSv1.3
```

`SSLProtocol` controls which TLS (and legacy SSL) versions Apache will negotiate at all. The `-all` then `+TLSv1.2 +TLSv1.3` pattern — deny everything, then explicitly re-allow only what's wanted — is the recommended form specifically because it fails safe: if a future TLS version is ever added to Apache's supported list by default, a config written this way doesn't silently start allowing it. A config that instead lists only positives (`+TLSv1.2 +TLSv1.3` with no leading `-all`) can leave older, weaker protocol versions still enabled underneath, depending on Apache's own compiled-in defaults.

SSLCipherSuite & SSLHonorCipherOrder

```
SSLCipherSuite HIGH:!aNULL:!MD5:!3DES
SSLHonorCipherOrder on
```

`SSLCipherSuite` takes a colon-separated cipher string: `HIGH` selects strong ciphers as a starting set, and each `!`-prefixed entry explicitly excludes a specific weak option (`aNULL` — no authentication at all, `MD5` and `3DES` — both cryptographically weak by modern standards) that might otherwise sneak in under a broader category. `SSLHonorCipherOrder on` tells Apache to enforce its own preferred cipher order rather than deferring to whatever order the connecting client offers — mainly relevant for TLS 1.2 and earlier, since TLS 1.3 negotiates from a small, fixed, already-strong cipher list and doesn't need this kind of manual curation.

OCSP Stapling

```
SSLUseStapling on
SSLStaplingCache "shmcb:/var/run/ocsp(128000)"
```

OCSP (Online Certificate Status Protocol) is how a client checks whether a certificate has been revoked since issuance. Without stapling, that check is the client's own job: the browser itself contacts the certificate authority's OCSP responder on every connection — a real, live network round-trip that adds latency, can fail if the CA's responder is slow or unreachable, and quietly tells the CA which sites a given client is visiting. **OCSP stapling** moves that job to the server: Apache itself periodically fetches a signed, time-stamped "this certificate is still valid" response from the CA (cached and refreshed on its own schedule, not per-request), and attaches — staples — that response directly into the TLS handshake. The client gets the same proof of validity without ever contacting the CA itself.

DIRECTIVE	CONTROLS	COVERED IN FUNDAMENTALS?
SSLCertificateFile / SSLCertificateKeyFile	Certificate and key location	Yes — basic pairing only
SSLProtocol	Which TLS versions are allowed	No
SSLCipherSuite / SSLHonorCipherOrder	Cipher selection and priority	No
SSLUseStapling / SSLStaplingCache	OCSP stapling	No

WHERE THIS HEADS NEXT

Every directive in this chapter is exactly the kind of thing a real production `<VirtualHost :443>` block combines into one config — Chapter 10's capstone assembles a complete, hardened Apache deployment, and this chapter's own directives are what that block's TLS section will actually contain.

RESTRICTING PROTOCOLS TOO AGGRESSIVELY HAS A REAL COST

Disabling every TLS version except the newest isn't automatically the right call for every site. A small population of genuinely old clients — outdated Android versions, old embedded devices, some corporate environments stuck on legacy software — may not support TLS 1.3 or even 1.2 in some rare cases. Restricting `SSLProtocol` to only the newest versions is the right default for most sites today, but it's a real trade-off between security posture and compatibility, not a setting with no downside at all — worth a deliberate decision based on who actually needs to reach the site, not just applied reflexively.

Hands-On Exercises

EXERCISE 1

Write an SSLProtocol line that disables every protocol version except TLS 1.2 and TLS 1.3, using the fail-safe pattern this chapter recommends rather than a positive-only list.

View solution

EXERCISE 2

Explain, in your own words, what problem OCSP stapling actually solves, and specifically what changes about who contacts the certificate authority, and when, once stapling is enabled.

View solution

EXERCISE 3

A junior admin proposes restricting SSLProtocol to TLS 1.3 only, on every site the company runs, "for maximum security." Using this chapter's own warning box, explain what real trade-off this decision involves, and what question should actually be asked before applying it everywhere.

View solution

CHAPTER 6 QUICK REFERENCE

- Modern `SSLCertificateFile` can hold the full chain in one file since Apache 2.4.8 — `SSLCertificateChainFile` is mostly legacy now
- **SSLProtocol** — use `-all +TLSv1.2 +TLSv1.3` (deny-then-allow), not a positive-only list, to fail safe against future protocol additions
- **SSLCipherSuite**/**SSLHonorCipherOrder** — curate and enforce cipher choice, mainly relevant pre-TLS 1.3
- **OCSP stapling** — the server, not the client, fetches and caches proof of certificate validity, then attaches it to the handshake directly
- Restricting TLS versions is a real security/compatibility trade-off, not a setting to maximize without checking who still needs to connect

Web Servers Fundamentals Chapter 6 showed a single `ProxyPass` / `ProxyPassReverse` pair pointing at one backend, and named `mod_proxy_balancer` without ever configuring it. This chapter is the real thing: a genuine multi-backend pool, the algorithms that decide which member handles each request, how Apache detects a failed backend, session affinity, and one very easy-to-miss directive that breaks a surprising number of real proxied applications.

A Real Backend Pool: <Proxy balancer://...>

```
<Proxy "balancer://mycluster">
BalancerMember "http://10.0.0.11:5000"
BalancerMember "http://10.0.0.12:5000"
ProxySet lbmethod=byrequests
</Proxy>
ProxyPass "/app" "balancer://mycluster/"
ProxyPassReverse "/app" "balancer://mycluster/"
```

`<Proxy balancer://mycluster>` names a pool the same way Nginx's own `upstream {}` block does (Nginx In Depth Chapter 4) — each `BalancerMember` line is one backend, and `ProxyPass` / `ProxyPassReverse` now point at the named balancer instead of a single address. `ProxySet lbmethod=byrequests` picks the load-balancing algorithm.

Load Balancing Algorithms

Apache ships three real `lbmethod` options: `byrequests` (the default) distributes requests in round-robin fashion, optionally weighted per member. `bytraffic` balances based on bytes actually transferred rather than raw request count — a better fit when requests vary hugely in payload size, since ten tiny requests and one huge upload aren't equivalent load. `bybusyness` routes each new request to whichever member currently has the fewest active in-flight requests — the right choice when backend response times vary widely, since a member stuck on one slow request won't keep receiving new ones under round robin's blind rotation.

Health Checks and Member Status

When a `BalancerMember` fails to respond, Apache automatically marks it in an error state and stops routing new requests to it — no external health-check tool required for this basic case. By default it retries that member again after 60 seconds (configurable via a member's own `retry=` parameter), giving a recovering backend a chance to rejoin the pool automatically rather than needing a manual restart of the whole proxy layer.

Sticky Sessions

```
<Proxy "balancer://mycluster">
BalancerMember "http://10.0.0.11:5000" route=1
    BalancerMember "http://10.0.0.12:5000" route=2
    ProxySet stickysession=ROUTEID
</Proxy>
```

A backend that keeps session state in its own local memory — rather than in a shared, external store — needs every request from the same client to land on the same backend instance, or that state effectively disappears mid-session. `stickysession` (matched against a cookie or URL parameter, here `ROUTEID`) pairs each client with one specific `route=`-tagged member and keeps routing that client there. This is a real trade-off, not a free feature: it works against true load balancing, since a client's traffic no longer distributes freely once it's pinned. Backends that instead keep session state in a shared external store (e.g. Redis) don't need sticky sessions at all — the state is available to whichever member happens to handle the next request.

PROXYPRESERVEHOST — A SMALL DIRECTIVE THAT BREAKS A SURPRISING NUMBER OF APPS

By default, the `Host` header Apache forwards to the backend comes from the `ProxyPass` target URL itself (e.g. `10.0.0.11:5000`), not the original `Host` header the client actually sent. A backend application that builds absolute URLs, checks its own configured hostname, or does any name-based routing of its own will silently misbehave — generating links pointing at the internal backend address instead of the public domain. `ProxyPreserveHost On` fixes this by forwarding the client's original `Host` header through unchanged, and is worth treating as a near-default whenever the backend app cares about its own hostname at all.

LBMETHOD	BALANCES BASED ON	BEST FOR
byrequests	Request count (weighted round robin)	Uniform, similarly-costly requests
bytraffic	Bytes actually transferred	Requests with widely varying payload sizes
bybusyness	Currently active requests per member	Backends with widely varying response times

WATCHING THE REAL, LIVE POOL STATE

`mod_status`'s companion, `balancer-manager` (enabled at its own dedicated location, e.g. `/balancer-manager`), shows every `BalancerMember`'s live status, current load, and whether Apache currently considers it healthy — the third real, live-introspection tool this course has covered, after Chapter 2's `mod_status` and Chapter 5's `apachectl -S`. All three share the same underlying lesson: check what Apache is actually doing right now, not just what the config file says it should do.

Hands-On Exercises

EXERCISE 1

Write a <Proxy balancer://...> block with two BalancerMembers for a backend pool where response times vary widely between requests — some backends can end up busy far longer than others on a single slow request. Choose and set the lbmethod that best fits this scenario, and explain why.

View solution

EXERCISE 2

A backend application proxied through Apache generates password-reset emails containing links back to the site, but the links it generates point at "http://10.0.0.11:5000/reset" instead of the site's real public domain. Using this chapter's own warning box, explain what's causing this and how to fix it.

View solution

EXERCISE 3

Explain, in your own words, the real trade-off sticky sessions introduce, what specific problem they solve, and describe one alternative approach that avoids needing sticky sessions at all.

View solution

CHAPTER 7 QUICK REFERENCE

- **<Proxy balancer://...>** + **BalancerMember** — Apache's real equivalent of Nginx's `upstream {}` block
- **lbmethod** — `byrequests` (default), `bytraffic`, `bybusyness`
- A failed `BalancerMember` is automatically marked down and retried after 60s by default — no external health checker needed for the basic case
- **stickysession** pins a client to one backend for local session state, at the cost of true load distribution — a shared external session store avoids needing it
- **ProxyPreserveHost On** forwards the client's real `Host` header to the backend instead of the proxy target's own address
- **balancer-manager** shows live pool health — the third live-introspection tool in this course, after `mod_status` and `apachectl -S`

Web Servers Fundamentals Chapter 8 named `Require` / `mod_authz_host` for IP restriction and `.htpasswd`-backed Basic Auth without ever showing a full working config, and stopped at the web-server layer entirely. This chapter completes both — the real, composable `Require` logic and a genuine working Basic Auth setup — then goes one layer deeper than Fundamentals ever went at all: `mod_security`, a real Web Application Firewall that inspects *what's actually being sent*, not just who's allowed to send it.

The Modern Require Syntax In Depth

```
<Directory "/var/www/html/admin">
Require ip 203.0.113.10
    Require ip 10.0.0.0/8
</Directory>
```

Multiple `Require` lines at the same level combine with an implicit **OR** by default — the block above grants access if the request comes from *either* IP condition, not both. Genuine AND logic needs an explicit `<RequireAll>` wrapper:

```
<RequireAll>
Require ip 10.0.0.0/8
    Require user admin
</RequireAll>
```

This forces both conditions — the correct internal IP range *and* the correct authenticated username — rather than accepting either alone. `<RequireNone>` exists too, for an explicit blacklist condition inside a larger logic group.

A Real Basic Authentication Setup

```
# create the password file (bcrypt-hashed, not plaintext)
htpasswd -c /etc/apache2/.htpasswd admin

# httpd.conf / a .htaccess file under AllowOverride AuthConfig
<Directory "/var/www/html/admin">
AuthType Basic
    AuthName "Restricted Area"
AuthUserFile /etc/apache2/.htpasswd
    Require valid-user
</Directory>
```

`htpasswd` manages the password file itself, hashing each password rather than storing it in plain text. `AuthUserFile` points Apache at that file, and `Require valid-user` accepts any correctly-authenticated username in it — swap in `Require user admin` to restrict to one specific account instead, combinable with the IP logic above via `<RequireAll>`. Per Chapter 3's own `AllowOverride` material, these directives fall under the `AuthConfig` group if placed in a `.htaccess` file rather than the main config.

> **BASIC AUTH CREDENTIALS ARE ENCODED, NOT ENCRYPTED**
>
> Basic Authentication sends the username and password base64-encoded on every single request — base64 is trivially reversible, not real encryption. Over plain HTTP, anyone able to observe the traffic can recover the credentials directly. Basic Auth is only genuinely safe paired with HTTPS (Chapter 6's `mod_ssl` material), which is doing the actual protective work here — Basic Auth itself provides no confidentiality of its own.

mod_security: A Real Web Application Firewall

Everything so far in this chapter controls **who** can reach a path. `mod_security` is a fundamentally different layer: it inspects **what's actually being sent** — request parameters, headers, body content — against a rule set, regardless of who's sending it or whether they've authenticated at all, looking for patterns that resemble known attack techniques (SQL injection, XSS payloads, path traversal attempts).

```
SecRuleEngine DetectionOnly

SecRule ARGS "@contains <script>" "id:1000,deny,status:403,msg:'Possible XSS
attempt'"
```

`SecRuleEngine` has three modes: `Off`, `On` (actively blocking matched requests), and `DetectionOnly` (logging what *would* have been blocked, without actually blocking anything). A hand-written `SecRule` follows a fixed shape — variables to inspect (`ARGS` here, meaning request parameters), an operator (`@contains`), and actions (an `id`, what to do on a match, and a log message). In practice, almost nobody hand-writes a full rule set from scratch — the **OWASP Core Rule Set (CRS)** is the standard, actively-maintained rule set nearly every real `mod_security` deployment pairs it with, covering the same attack categories the site's own XSS and OWASP Top 10 courses cover conceptually, now enforced directly at the web-server layer.

LAYER	CONTROLS	EXAMPLE
Require / mod_authz_host	**Who** can reach a path (IP)	`Require ip 10.0.0.0/8`
Basic Auth	**Who**, via credentials	`AuthType Basic` + `Require valid-user`
mod_security	**What's** in the request itself	`SecRule ARGS "@contains <script>" ...`

WHY DETECTIONONLY IS THE RECOMMENDED FIRST STEP, NOT ON

Rolling `mod_security` — especially the full CRS — straight into `SecRuleEngine On` on an existing production site risks blocking genuine, legitimate traffic that happens to trip an aggressive rule (a search box containing the word "select", for instance). Starting with `DetectionOnly` and reviewing what the rule set would have blocked, over real production traffic, before ever switching to `On` is the standard safe rollout — the same "match the machinery to what's actually needed, verify before enforcing" caution this course has already applied to TLS version restriction (Chapter 6) and worker capacity planning (Chapter 2).

A WAF IS NOT A FIRE-AND-FORGET CONTROL

The Core Rule Set is deliberately broad and, by design, produces real false positives against some legitimate applications — a rich-text editor submitting HTML-like content, or an API accepting genuinely unusual input, can each trip rules meant for a different, malicious pattern. Real `mod_security`/CRS deployments routinely need per-application rule exclusions tuned over time. Treating it as a one-time install rather than an ongoing tuning responsibility is a common, avoidable source of either false-positive outages or a WAF that's been quietly disabled entirely to make the outages stop.

Hands-On Exercises

EXERCISE 1

Write a <Directory> block that requires a request to come from the IP range 172.16.0.0/12 AND be authenticated as a valid user from an AuthUserFile — both conditions must be true, not either one alone.

View solution

EXERCISE 2

A site has Require ip restricting /admin/ to the office IP range, and separately has mod_security with the OWASP Core Rule Set enabled site-wide. An attacker from outside the office IP range sends a SQL-injection payload to the public /search endpoint (not /admin/). Explain which of these two controls is actually relevant here, and why the other one doesn't apply to this request at all.

View solution

EXERCISE 3

Explain, in your own words, why SecRuleEngine DetectionOnly is the recommended way to first deploy mod_security with the OWASP Core Rule Set on an existing production site, rather than enabling SecRuleEngine On immediately.

View solution

CHAPTER 8 QUICK REFERENCE

- Multiple `Require` lines default to OR — use `<RequireAll>` for genuine AND logic, `<RequireNone>` for a blacklist condition
- **Basic Auth** — `htpasswd` creates the hashed password file; `Require valid-user` vs. `Require user <name>`; base64-encoded, not encrypted — HTTPS-only in practice
- **mod_security** — inspects request content itself, not who's sending it; `SecRuleEngine Off` / `DetectionOnly` / `On`
- The **OWASP Core Rule Set (CRS)** is the standard rule set almost everyone pairs with `mod_security` rather than hand-writing rules
- Start new WAF deployments with `DetectionOnly`, and treat rule tuning as ongoing, not a one-time install

Web Servers Fundamentals Chapter 9 named `KeepAlive` / `KeepAliveTimeout` and `mod_cache` without real values or directives, and closed with a serious, correct warning: caching personalized content by URL alone can leak one user's data to another. This chapter goes deep on all three — KeepAlive tuning tied directly back to Chapter 2's own capacity math, the actual `mod_cache_disk` mechanism that prevents exactly the leak Fundamentals warned about, and the practical logging and troubleshooting tools that round out this course before its capstone.

KeepAlive Tuning — Tied to Chapter 2's Own Capacity Math

```
KeepAlive On
KeepAliveTimeout 5
MaxKeepAliveRequests 100
```

`KeepAliveTimeout` isn't a free setting to raise generously — it directly connects to Chapter 2's own `MaxRequestWorkers` capacity math. Under Prefork, a connection sitting idle during its keep-alive window still occupies an entire process; under Worker, it still occupies an entire thread — either way, it counts against the same real ceiling a genuinely busy request would, for as long as the timeout keeps it open. A `KeepAliveTimeout` set too high (a common instinct: "longer must be better for reuse") can quietly starve real capacity with connections doing nothing at all. This is exactly the problem Chapter 2's own **Event** MPM solves — its `AsyncRequestWorkerFactor` handling of idle keep-alive connections is precisely what makes a longer `KeepAliveTimeout` far cheaper under Event than under Prefork or Worker. A short window (5–15 seconds is typical) is the usual safe default outside of Event.

mod_cache_disk In Depth — Actually Preventing Fundamentals' Own Warning

```
CacheRoot /var/cache/apache2/mod_cache_disk
CacheEnable disk /static/
CacheDirLevels 2
CacheDirLength 1
```

`CacheRoot` sets where cached responses are stored on disk; `CacheEnable disk /static/` turns caching on for that path specifically, rather than site-wide; `CacheDirLevels` / `CacheDirLength` control how cache files are spread across subdirectories to avoid one directory holding an unmanageable number of files. The part that actually matters for Fundamentals' own warning is what `mod_cache` does *by default*: it respects `Cache-Control` headers from the backend, refusing to cache anything marked `private` or `no-store`. For content that legitimately does vary per user but is still worth caching, the backend's own `Vary` response header — e.g. `Vary: Cookie` — tells `mod_cache` to key its stored copies separately per distinct value of that header, rather than serving one shared cached copy to every visitor of the same URL.

> **A CORRECTLY CONFIGURED VARY HEADER CAN STILL MEAN "NO REAL CACHE HIT"**
>
> `Vary: Cookie` genuinely prevents the cross-user leak Fundamentals warned about — but on a page where every logged-in user has a unique session cookie, keying the cache by that header means each user effectively gets their own single-visitor cache bucket, which almost never produces a real hit on a second request. Configuring `Vary` correctly can be a technically correct fix that still delivers zero real performance benefit. For a genuinely personalized page like an account dashboard, the honest fix is usually not "cache it correctly" at all — it's respecting the backend's own `Cache-Control: private` and simply not caching that response at the server/proxy layer in the first place.

Custom Log Formats

```
LogFormat "%h %l %u %t \"%r\" %>s %b \"%{Referer}i\" \"%{User-Agent}i\"" combined
CustomLog /var/log/apache2/access.log combined
```

Fundamentals mentioned `LogFormat` exists without ever showing it; the standard `combined` format above is worth being able to read directly: `%h` is the client IP, `%l` is the (almost always unused) identd field, `%u` the authenticated username if any, `%t` the timestamp, `%r` the full request line, `%>s` the final HTTP status code, `%b` the response size in bytes, and the two `%{...}i` tokens pull specific request headers directly — `Referer` and `User-Agent` here, but any header name works the same way.

Troubleshooting: configtest Before Every Reload

```
apachectl configtest
# or, equivalently
apachectl -t
```

This validates the entire configuration's syntax without actually reloading or restarting anything — critical to run before every real reload, since applying a broken config live can take the whole site down with it. Two of the most common real error-log patterns worth recognizing on sight: `AH00526: Syntax error on line N` (a straightforward config typo, pointing at the exact line), and `(98)Address already in use: AH00072: make_sock: could not bind to address` (something else — often a previous Apache process that didn't fully stop — is already holding the port). A `403` logged as `AH01630: client denied by server configuration` traces directly back to Chapter 8's own `Require`/`RequireAll` rules rejecting the request.

TOOL	CHECKS	INTRODUCED
apachectl configtest / -t	Config syntax validity, before applying it	This chapter
mod_status	Live worker/thread busy-idle state	Chapter 2
apachectl -S	Parsed VirtualHost matching order	Chapter 5
balancer-manager	Live backend pool health	Chapter 7

THE FOURTH AND FINAL INSTANCE OF THIS COURSE'S OWN RECURRING LESSON

`apachectl configtest` completes a pattern this course has now shown four separate times: `mod_status` for live capacity, `apachectl -S` for real routing order, `balancer-manager` for live pool health, and now `configtest` for config validity before it's even applied. Every one of them exists for the same underlying reason — check what Apache is actually doing, or about to do, rather than trusting what the config file says or assumes.

Hands-On Exercises

EXERCISE 1

A server runs Prefork with MaxRequestWorkers set to 100, and a team sets KeepAliveTimeout to 300 seconds thinking longer is simply better for connection reuse. Explain, using this chapter's own material and Chapter 2's capacity math, why this could genuinely reduce the server's real usable capacity under real traffic.

View solution

EXERCISE 2

A team configures Vary: Cookie on their logged-in account dashboard specifically to make caching it "safe," then is confused when their cache hit rate for that page stays at almost 0%. Explain what's happening, and what the more honest fix usually is for a page like this.

View solution

EXERCISE 3

A team edits their Apache config and immediately runs a full restart, and the entire site goes down with no traffic served at all. Using this chapter's own material, what single command should they have run first, and what would it likely have told them before they ever restarted the live server?

View solution

CHAPTER 9 QUICK REFERENCE

- **KeepAliveTimeout** — too high genuinely costs real capacity under Prefork/Worker (each idle connection still holds a process/thread); Event handles this far more cheaply
- **mod_cache_disk** — respects `Cache-Control` by default; `Vary` keys the cache per-header value to avoid cross-user leaks, but can still yield near-zero real hit rate on highly personalized pages
- **LogFormat combined** — `%h` IP, `%u` user, `%t` time, `%r` request line, `%>s` status, `%b` size, `%{Header}i` for any request header
- **apachectl configtest** (`-t`) — validate syntax before every reload; never reload/restart on an unchecked config
- The fourth live-introspection tool in this course, after `mod_status`, `apachectl -S`, and `balancer-manager`

Web Servers Fundamentals' own capstone compared three servers across several scenarios. This one is different, by design — the same shape Nginx In Depth's own capstone used: 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

One Apache instance hosts two real things: a customer-facing PHP web application at `www.example.com`, and a small internal API — a separate Node.js service — proxied at `api.example.com`. Both need HTTPS, sensible performance, and hardening that reflects everything this course has covered.

Step 1 — MPM Choice (Chapter 2)

```
<IfModule mpm_event_module>
ServerLimit          16
    ThreadsPerChild      25
    MaxRequestWorkers   400
</IfModule>
```

The PHP application runs on **PHP-FPM**, not traditional `mod_php` — per Chapter 2, that's exactly what frees Apache to run **Event** here instead of being forced onto Prefork, gaining Event's lower per-connection overhead for both the PHP site and the proxied API sharing this same instance.

Step 2 — Locking Down .htaccess (Chapter 3)

```
<Directory "/var/www">
AllowOverride None
</Directory>
```

Every piece of configuration for both sites lives in version-controlled main config files, not scattered `.htaccess` overrides — so `AllowOverride None` applies globally, per Chapter 3's own hardening payoff: this removes the per-request filesystem scan entirely and closes off an entire class of misconfiguration risk from a stray or compromised `.htaccess` file, with nothing lost since neither site actually needs one.

Step 3 — Core Modules for the PHP App (Chapter 4)

```
RewriteEngine On
RewriteRule ^article/([0-9]+)$ article.php?id=$1 [L]

Header set X-Frame-Options "SAMEORIGIN"
<IfModule mod_deflate.c>
AddOutputFilterByType DEFLATE text/html text/css application/javascript
</IfModule>
```

Clean article URLs via `mod_rewrite`, a baseline security header via `mod_headers`, and compression for the PHP app's own text-based output via `mod_deflate` — the three Chapter 4 modules, all falling under the same `FileInfo` group Step 2 didn't need to grant anywhere else.

Step 4 — Two Domains, One Apache Instance (Chapter 5)

```
<VirtualHost *:443>
ServerName www.example.com
    DocumentRoot /var/www/site
</VirtualHost>
<VirtualHost *:443>
ServerName api.example.com
</VirtualHost>
```

Both domains share the same IP and port, distinguished purely by `ServerName` — modern name-based hosting over HTTPS, made possible entirely by **SNI** (Chapter 5), with no separate IP needed per domain the way it would have before SNI existed.

Step 5 — TLS Hardening (Chapter 6)

```
SSLEngine on
SSLCertificateFile /etc/ssl/certs/example.com-fullchain.crt
SSLCertificateKeyFile /etc/ssl/private/example.com.key
SSLProtocol -all +TLSv1.2 +TLSv1.3
SSLCipherSuite HIGH:!aNULL:!MD5:!3DES
SSLUseStapling on
```

Applied inside each `<VirtualHost *:443>` block: the modern combined-chain certificate file, a fail-safe `SSLProtocol` line, a curated cipher list, and OCSP stapling so clients confirm certificate validity without a live round-trip to the CA — all four directly from Chapter 6.

Step 6 — Proxying the Internal API (Chapter 7)

```
<Proxy "balancer://apipool">
BalancerMember "http://10.0.0.11:3000"
BalancerMember "http://10.0.0.12:3000"
ProxySet lbmethod=bybusyness
</Proxy>
ProxyPreserveHost On
ProxyPass "/" "balancer://apipool/"
ProxyPassReverse "/" "balancer://apipool/"
```

Inside `api.example.com`'s own `<VirtualHost>`: a two-member balanced pool using `bybusyness`, since this API's own endpoints vary widely in response time, and `ProxyPreserveHost On` so the Node.js service sees the real requested hostname rather than the internal pool address — exactly the gotcha Chapter 7's own warning box covered.

Step 7 — Access Control & mod_security (Chapter 8)

```
<Location "/internal-metrics">
Require ip 10.0.0.0/8
</Location>
SecRuleEngine DetectionOnly
```

The API's own internal metrics endpoint is IP-restricted to the private network via `Require ip` — controlling *who* can reach it. Site-wide, `mod_security` with the OWASP Core Rule Set starts in `DetectionOnly`, per Chapter 8's own recommended rollout, rather than risking false-positive outages by enabling `On` from day one.

Step 8 — Performance, Logging & Pre-Flight Checks (Chapter 9)

```
KeepAliveTimeout 15

CacheRoot /var/cache/apache2/mod_cache_disk
CacheEnable disk /static/

LogFormat "%h %l %u %t \"%r\" %>s %b" combined
CustomLog /var/log/apache2/access.log combined

# Before every reload:
# apachectl configtest
```

`KeepAliveTimeout 15` is comfortably affordable specifically because Step 1 chose Event, not Prefork — the same setting would have cost real `MaxRequestWorkers` capacity under Prefork, per Chapter 2's own math. Caching is scoped to `/static/` only, deliberately excluding the PHP app's own logged-in pages, per Chapter 9's own warning about personalized content and cache hit rate. `apachectl configtest` is the last, non-negotiable step before any of this ever gets reloaded live.

CAPSTONE STEP	CHAPTER IT DRAWS FROM
Step 1 — MPM choice	Chapter 2
Step 2 — AllowOverride None	Chapter 3
Step 3 — mod_rewrite/headers/deflate	Chapter 4
Step 4 — Name-based hosting via SNI	Chapter 5
Step 5 — TLS hardening	Chapter 6
Step 6 — Proxying & load balancing	Chapter 7
Step 7 — Access control & mod_security	Chapter 8
Step 8 — Performance, caching & logging	Chapter 9

TWO CAPSTONES, ONE FULL JOURNEY

Web Servers Fundamentals' own capstone answered "which server fits this job." This one answers "given Apache was already chosen, how do you actually configure it well" — the same relationship Nginx In Depth's own capstone has to that same earlier course. Together, all three capstones across the Web Servers subject cover the full journey from choosing a web server to running a genuinely production-shaped configuration of whichever one was chosen.

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 (`apachectl configtest`) before every reload, a gradual rollout rather than switching all traffic at once, and real alerting configured on top of whatever monitoring is in place — well beyond `mod_status`'s or `balancer-manager`'s own live snapshots. 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 applies AllowOverride None globally in Step 2, even though earlier chapters spent real time explaining how .htaccess and AllowOverride's named groups work. Explain why this is not a contradiction, referencing this chapter's own reasoning.

View solution

EXERCISE 2

A new /account/ endpoint is added to the PHP app, showing each logged-in user their own personalized order history. Should it be added to the CacheEnable disk zone the same way /static/ was in Step 8? 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

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

CHAPTER 10 QUICK REFERENCE

- This capstone builds **one cohesive production config** for two real domains, layering in Chapters 2 through 9 step by step — the same deliberate shape as Nginx In Depth's own capstone, and a different one from Web Servers Fundamentals' own multi-scenario decision framework
- PHP-FPM (not `mod_php`) is what makes choosing Event over Prefork possible in Step 1 — a payoff planted all the way back in Chapter 2
- Only **non-personalized** content belongs in a `mod_cache_disk` zone; personalized pages need `Cache-Control: private` respected instead, not a cleverly-keyed cache (Chapter 9)
- A working config is a **foundation**, not a complete production rollout — `apachectl configtest`, gradual deployment, and real alerting still matter beyond this chapter's own scope