Logging & Log Analysis

A Complete 10-Chapter Technical Support Course

Topics covered:

The log-first triage mindset · log levels & severity

Where logs live on Linux & Windows · reading Apache/Nginx logs in depth

Diagnosing login failures, slow responses, and wrong responses

Log aggregation & centralized logging · writing to logs, safely

Capstone: three real support tickets, triaged end to end

Exercises: 30 hands-on scenarios with worked solutions

Format: A4 · Dark-theme code examples

Philip Osztromok · Generated with Claude

Table of Contents

CHAPTER 1 OF 10

Why Logs Matter: The Support Engineer's First Instinct

Logging & Log Analysis

Chapter 1 · Why Logs Matter: The Support Engineer's First Instinct

Two support engineers get the same ticket: "the site is slow." One opens a browser, clicks around, shrugs, and starts guessing — maybe it's the database, maybe it's the network, maybe it's just Tuesday. The other opens the logs first. This course is entirely about becoming the second engineer — not because guessing never works, but because logs turn "maybe" into evidence, and evidence is what actually gets a ticket closed correctly the first time.

The Guessing Trap

Guessing feels faster than it is. A plausible-sounding theory — "it's probably the database" — sends you off to check database metrics, find nothing obviously wrong, move on to the next guess, and repeat. Each wrong guess costs real time and produces no evidence toward the actual answer. Worse, a confident wrong guess can lead to a wrong fix — restarting a service that was never the problem, "just in case" — which doesn't just fail to help, it can actively muddy the diagnostic trail for whoever looks at this next.

The log-first habit inverts this. Instead of theorizing and then checking, you check first and let what you find narrow the theories that are even worth considering. It's slower to start — reading log output isn't as immediately satisfying as taking an action — but it's dramatically faster to actually finish, because every minute spent reading logs is a minute spent on real evidence, not a guess that might be completely wrong.

What a Log Actually Is

Strip away the specifics of any particular system, and a log is simple: **a timestamped record of an event a system was told to record.** A web server logs each request it receives. An authentication system logs each login attempt. An application logs errors when something goes wrong internally. Different systems, same underlying idea.

```
127.0.0.1 - - [08/Aug/2026:14:32:07 +0100] "GET /login HTTP/1.1" 500 1204
```

Even without knowing this exact format yet — Chapter 4 covers it in full — a few things are already readable: something happened at a specific moment (`14:32:07`), it involved a request to `/login` , and the outcome was a `500` , a server-side error. That's already more than a guess would have given you for free.

What Logs Can Actually Tell You

WHAT LOGS ARE GOOD AT	WHY IT MATTERS
Exact timing	When something happened, down to the second — critical for correlating events across different systems
Sequence	What happened before and after a given event, in the order it actually occurred
Error detail	The specific error message or stack trace a system produced, often far more precise than a user's own description of "it's broken"
Volume & frequency	Whether something is a one-off blip or a repeating pattern — a single error looks very different from the same error every 30 seconds

What Logs Can't Tell You — An Honest Limit

Logs are not a complete recording of everything that happened; they're a recording of everything someone *chose* to record. That distinction matters more than it sounds:

- **Silent failures aren't logged at all.** If nobody wrote a log line for a particular failure case, it simply won't appear — the absence of an error in the logs is not proof nothing went wrong
- **A single log line is often missing context.** One line might tell you a request failed, but the actual cause could be recorded in a completely different system's log, at roughly the same timestamp — Chapters 6 and 7 both depend on this kind of cross-referencing
- **Logs can be actively misleading if misconfigured.** A server with its clock set to the wrong timezone will log technically accurate events at confusingly wrong-looking times; an overly generic error message can point you toward the wrong subsystem entirely

> **NEVER DELETE LOGS TO "FIX" A PROBLEM MID-INVESTIGATION**
> Clearing a log file to free up disk space during an active incident is a genuinely common, genuinely damaging mistake — it destroys the exact evidence you're trying to use to diagnose the incident, often permanently. If disk space is a real concern, move or compress the log file instead of deleting it, and only once you're confident you no longer need what's in it.

A Concrete Example: One Symptom, Three Different Causes

A user reports "the site gave me an error." That single symptom could mean genuinely different things underneath — a failed database connection, a bug in the application code, or the server running out of memory under load — and each of those has a completely different fix. Without logs, you're choosing between three guesses. With logs, the error message, the timing, and what else was happening at that exact

moment usually point clearly at just one of them. Chapters 6 and 7 build this exact skill in full, working through a slow-response scenario and an incorrect-response scenario end to end.

READ THE TIMESTAMP BEFORE YOU READ THE MESSAGE

It's tempting to jump straight to an error message and start reasoning about what it means. Checking the timestamp first — and confirming it actually lines up with when the reported problem happened — catches a surprising number of false leads early, before you spend time investigating an error that turns out to be unrelated, from before or after the actual incident.

What This Course Covers

Log levels and severity, where to actually find logs on Linux and Windows, reading web server and authentication logs in real depth, two full diagnostic walkthroughs built around genuinely common support scenarios, an orientation to log aggregation at scale, and — closing the loop — how to write to logs yourself, both appending to an existing application's log and building a well-behaved custom one. The capstone ties all of it together against three realistic support tickets.

Hands-On Exercises

EXERCISE 1

Explain, in your own words, why this chapter argues that checking logs first is actually faster overall than guessing first, even though reading logs takes real time up front.

View solution

EXERCISE 2

A colleague says "there's nothing in the logs, so nothing went wrong." Explain why this chapter would consider that conclusion unsafe, and what a "silent failure" actually is.

View solution

EXERCISE 3

During a live incident, disk space runs low on the server you're investigating. A colleague suggests deleting the log file that's taking up the space. Explain why this chapter says that's a mistake, and what to do instead.

View solution

CHAPTER 1 QUICK REFERENCE

- **Log-first, not guess-first** — checking logs before theorizing turns diagnosis into evidence-gathering rather than trial and error
- A log is a **timestamped record of an event a system was told to record** — nothing more, nothing less
- Logs are good at: exact timing, sequence, error detail, and frequency/volume
- Logs can't guarantee completeness (silent failures), can lack context (needing cross-referencing), and can mislead if misconfigured (e.g. wrong server timezone)
- **Never delete a log file mid-investigation** to free disk space — move or compress it instead
- Always confirm a log entry's **timestamp** actually matches the reported incident before trusting its message
- **Next chapter:** Log Levels & Severity

CHAPTER 2 OF 10

Log Levels & Severity

Logging & Log Analysis

Chapter 2 · Log Levels & Severity

Chapter 1 established that a log only records what someone chose to record. Severity levels are the mechanism behind that choice — a way of tagging each log line with how much it actually matters, so both the system generating logs and the person reading them can filter signal from noise.

The Standard Levels, Quietest to Loudest

LEVEL	MEANING	TYPICAL USE
DEBUG	Fine-grained detail, useful only while actively developing or troubleshooting	"Entered function X with these arguments"
INFO	Normal, expected events — nothing wrong, just a record that something happened	"User 4821 logged in successfully"
WARN / WARNING	Something unexpected, but the system recovered or the request still succeeded	"Retrying database connection (attempt 2 of 3)"
ERROR	A specific operation failed — the request or task genuinely did not complete as intended	"Failed to save user preferences: connection timeout"
CRITICAL / FATAL	The failure is severe enough to affect the whole system, not just one operation	"Database connection pool exhausted — service unavailable"

These five names are the most common convention across application-level logging, but they're a convention, not a universal law — some systems use slightly different names or collapse two of these into one. The relative ordering (quietest/least severe to loudest/most severe) is the part that stays consistent.

Why Levels Exist: Filtering Signal From Noise

Almost every logging system lets you configure a **minimum level** to actually output or store. Set the minimum to `WARN`, and every `DEBUG` and `INFO` line is silently dropped — only warnings and worse get through. This is deliberate: a healthy, correctly-behaving system genuinely doesn't need to permanently store a line for every single normal event, but it does need a way to see everything in detail the moment something is being actively investigated.

THE LEVEL FILTER IS YOUR FIRST, FASTEST TRIAGE TOOL

Before reading log content in detail, filter to `WARN` and above. This alone often reduces a firehose of thousands of lines down to a handful that are actually worth reading — and if nothing shows up at that filter level for the time window in question, that's itself useful information, pointing you toward Chapter 1's own "silent failure" territory rather than a logged one.

The Real-World Mistake: One Level for Everything

Levels only help if they're used with any discipline. Two genuinely common failure patterns undo the entire benefit:

- **Everything logged as INFO (or worse, as ERROR).** If routine, expected events are logged at the same level as genuine failures, filtering by level stops working — you're back to reading everything, exactly the noise levels exist to avoid. A classic version of this: logging "user not found" as `ERROR` during a normal login attempt, when a mistyped password is completely routine, not a system failure.
- **Running with DEBUG enabled permanently in production.** Debug-level detail is enormously verbose by design — enabling it around the clock produces massive log volume, real disk and performance overhead, and buries the genuinely important lines even deeper than having no levels at all would.

"ALERT FATIGUE" IS A DIRECT CONSEQUENCE OF LEVEL MISUSE

Once ERROR-level logs (or alerts built on top of them) routinely include things that aren't actually problems, the people reading them learn — correctly, given the evidence — to stop trusting that an ERROR line means something worth acting on. By the time a genuine, serious error appears, it's just one more line in a stream that's already been mentally tuned out.

A Practical Example: The Same Situation, Two Different Levels

```
[WARN] Database connection attempt 1 failed, retrying (recovered on attempt 2)
[ERROR] Database connection failed after 3 attempts — request could not be completed
```

Same underlying condition — a database connection that isn't responding — logged at two different levels because the outcome was different. The first line describes something that recovered on its own and needed no action; the second describes an operation that genuinely failed. Reading only the level, before even reading the rest of the message, already tells you which of these two deserves attention right now.

Syslog's Own Scheme: Eight Levels, Not Five

Linux's own system-wide logging convention, **syslog**, predates most application-level logging frameworks and uses a finer-grained, numbered scale — worth knowing before Chapter 3 covers where these logs actually live.

SYSLOG LEVEL	NUMERIC VALUE	ROUGHLY MAPS TO
Emergency	0	System is unusable
Alert	1	Immediate action required
Critical	2	CRITICAL / FATAL
Error	3	ERROR
Warning	4	WARN
Notice	5	Normal but significant
Informational	6	INFO
Debug	7	DEBUG

Lower numbers mean higher severity — the opposite direction application-level logging usually reads in, but the same underlying "how much does this matter" idea. You'll see these numeric values directly in real configuration, such as Apache's own `LogLevel warn` directive or a syslog facility/severity pairing like `mail.crit`.

Hands-On Exercises

EXERCISE 1

A team logs a mistyped-password login failure as ERROR, arguing "the login did fail, so it's an error." Explain why this chapter would consider that a misuse of the ERROR level, and what level would fit better.

View solution

EXERCISE 2

Explain what "alert fatigue" means, using this chapter's own explanation, and how consistently misusing the ERROR level directly causes it.

View solution

EXERCISE 3

A syslog entry is tagged with severity 2, and another with severity 5. Explain which one is more severe and why, and name the application-level severity name each one roughly corresponds to.

View solution

CHAPTER 2 QUICK REFERENCE

- **DEBUG → INFO → WARN → ERROR → CRITICAL/FATAL** — the standard application-level severity ladder, quietest to loudest
- A configurable **minimum level** is the first, fastest filter for triage — start at WARN and above
- Common misuse: logging routine events as ERROR, or running DEBUG permanently in production — both destroy the filtering benefit levels exist for
- **Alert fatigue** is the direct, predictable consequence of level misuse — real errors get ignored once false ones have trained people to stop trusting the signal
- **Syslog** uses 8 numbered levels (0 Emergency – 7 Debug), lower number = more severe — the opposite direction from how severity usually reads, but the same underlying idea
- **Next chapter:** Where to Find the Logs

CHAPTER 3 OF 10

Where to Find the Logs

Logging & Log Analysis

Chapter 3 · Where to Find the Logs

Knowing what a log is and how severity works doesn't help much if you can't actually locate one. This chapter is a practical map: where Linux keeps its logs, the modern systemd alternative to plain log files, how Windows does the equivalent job very differently, and what to do when a log genuinely isn't where the defaults say it should be.

Linux: The `/var/log/` Directory

On most Linux systems, `/var/log/` is the traditional home for both system-level and application-level log files. A handful of names show up constantly:

FILE / FOLDER	WHAT IT HOLDS
/var/log/syslog	General system activity — the traditional catch-all on Debian/Ubuntu systems
/var/log/messages	The same kind of general system activity — the equivalent file on RHEL/CentOS-family systems
/var/log/auth.log	Authentication events (logins, sudo usage) — Debian/Ubuntu naming
/var/log/secure	The same authentication events — RHEL/CentOS naming
/var/log/apache2/ or /var/log/httpd/	Apache's own access and error logs — folder name depends on distro (Chapter 4 covers the file contents in depth)
/var/log/dmesg	Kernel ring buffer — hardware and boot-time messages

THE SAME LOG, TWO DIFFERENT FILENAMES — A GENUINELY COMMON SOURCE OF CONFUSION

Debian/Ubuntu-family distros and RHEL/CentOS-family distros use different conventional filenames for the same underlying log — `auth.log` vs. `secure`, `syslog` vs. `messages`, `apache2` vs. `httpd`. Following a guide written for the wrong distro family is a real, common way to end up looking for a file that simply doesn't exist on the system in front of you.

The systemd Journal — `journalctl`

Most modern Linux distributions run **systemd** as their init system, and systemd includes its own logging component, **journald**, which stores structured, binary log data rather than plain text files. It's queried with the `journalctl` command, not a text editor or `cat`.

```
# everything in the journal, oldest first
journalctl

# only logs from one specific service
journalctl -u apache2.service

# follow live, the same idea as `tail -f` for a plain log file
journalctl -f

# only entries from the last hour
journalctl --since "1 hour ago"

# only entries at "err" priority or worse
journalctl -p err
```

> **JOURNALCTL'S -P FLAG USES CHAPTER 2'S OWN SYSLOG PRIORITY NAMES**
>
> `journalctl -p` accepts the exact same syslog severity keywords (and numbers) covered in Chapter 2 — `emerg`, `alert`, `crit`, `err`, `warning`, `notice`, `info`, `debug`. Knowing that scale from Chapter 2 means `journalctl -p err` already makes sense — it's the same "0 through 7, lower is worse" filter applied directly at the command line.

Windows: Event Viewer

Windows takes a genuinely different approach from plain text files: the **Windows Event Log** service stores structured events, each with an Event ID, a source, and a severity level, browsable through the **Event Viewer** GUI (`eventvwr.msc`). The three logs worth knowing first:

LOG	WHAT IT HOLDS
Application	Events logged by applications and services running on the system
System	Events logged by Windows itself and its own drivers/components
Security	Login attempts, permission changes, and other security-relevant events

For a command-line equivalent to browsing Event Viewer's GUI, `Get-WinEvent` (or the older `Get-EventLog`) does the same job from PowerShell — see *PowerShell Fundamentals* and *PowerShell Intermediate/Advanced* for real depth on scripting against these logs directly.

Application-Specific Log Locations

APPLICATION	TYPICAL DEFAULT LOCATION
Apache	`/var/log/apache2/access.log` and `error.log` (or the `httpd` equivalent)
Nginx	`/var/log/nginx/access.log` and `error.log`
MySQL	`/var/log/mysql/error.log` by default, but genuinely configurable via `my.cnf`
Docker containers	Not a plain file at all by default — `docker logs <container>` reads the container's own captured stdout/stderr

When the Log Isn't Where You Expect

Defaults are defaults, not guarantees. When a log file genuinely isn't in its usual location, a few things are worth checking before assuming it doesn't exist at all:

- **The application's own configuration.** Most services let you set a custom log path — the config file is the actual source of truth, not the convention
- **The systemd unit file** for that service, if it's managed by systemd — `StandardOutput` / `StandardError` directives can redirect a service's output somewhere other than the journal entirely
- **Log rotation.** The file you're looking for might have already been rotated — renamed to `access.log.1`, or compressed to `access.log.2.gz` — Chapter 9 covers exactly how and why this happens

Hands-On Exercises

EXERCISE 1

A troubleshooting guide written for Ubuntu tells you to check /var/log/auth.log, but the server you're actually working on is CentOS. Explain what's likely going on, and what file you should check instead.

View solution

EXERCISE 2

Explain what journalctl -p err actually does, and why understanding Chapter 2's syslog severity scale makes this command immediately readable rather than something to memorize separately.

View solution

EXERCISE 3

You expect to find a service's log at its usual default path, but the file isn't there. Name two genuinely different reasons this could be true, according to this chapter, other than "the log doesn't exist."

View solution

CHAPTER 3 QUICK REFERENCE

- **/var/log/** — the traditional home for Linux system/application logs; exact filenames vary by distro family (Debian/Ubuntu vs. RHEL/CentOS)
- **journalctl** — queries systemd's own structured journal; `-u` filters by service, `-f` follows live, `-p` filters by Chapter 2's own syslog severity scale
- **Windows Event Viewer** — Application/System/Security logs, structured events with Event IDs; `Get-WinEvent` is the PowerShell equivalent
- Applications have their own default log paths, but config files can override them — the default is a convention, not a guarantee
- A "missing" log might be redirected (via app config or a systemd unit file) or simply rotated to a numbered/compressed filename
- **Next chapter:** Reading Web Server Logs: Apache & Nginx

CHAPTER 4 OF 10

Reading Web Server Logs: Apache & Nginx

Logging & Log Analysis

Chapter 4 · Reading Web Server Logs: Apache & Nginx

Chapter 3 found the files. This chapter reads what's actually in them — the access log's field-by-field structure, what HTTP status codes really tell you, a genuinely easy-to-miss gap in the default format, and the error log that fills that gap in. Apache and Nginx are covered together throughout, since their conventional log formats are close enough to read with one shared mental model.

The Access Log: Common & Combined Format

Both Apache and Nginx default to a format descended from the same standard, **Common Log Format (CLF)**, almost always extended in practice to **Combined Log Format**, which adds two more fields. A single line looks like this:

```
192.168.1.10 - - [08/Aug/2026:14:32:07 +0100] "GET /login HTTP/1.1" 500 1204 "https://example.com/dashboard" "
```

FIELD	EXAMPLE VALUE	WHAT IT MEANS
Remote host	192.168.1.10	The IP address the request came from
Identity	-	Almost always a literal dash — rarely used in practice
Authenticated user	-	A dash unless the request used HTTP basic authentication
Timestamp	[08/Aug/2026:14:32:07 +0100]	When the request was received, including timezone offset
Request line	"GET /login HTTP/1.1"	Method, path, and protocol version, together
Status code	500	The HTTP status code returned — covered in depth below
Response size	1204	Bytes sent in the response body
Referer	"https://example.com/dashboard"	The page the request was linked from, if any (Combined format only)
User-agent	"Mozilla/5.0..."	The requesting browser/client's own self-reported identity (Combined format only)

HTTP Status Codes: Reading the Middle Field

RANGE	CATEGORY	COMMON EXAMPLES
2xx	Success	200 OK
3xx	Redirection	301/302 (moved, found)
4xx	Client error — the request itself was the problem	404 Not Found, 401 Unauthorized, 403 Forbidden
5xx	Server error — the server failed to handle a valid request	500 Internal Server Error, 502 Bad Gateway, 503 Service Unavailable, 504 Gateway Timeout

That 4xx/5xx split matters for triage: a spike in 404s often means a broken link or a bot scanning for pages that don't exist — rarely a real system problem. A spike in 5xx codes means the server itself is failing to do its job, and is exactly the kind of signal Chapters 6 and 7 build a full diagnostic process around.

The Gap Almost Everyone Assumes Isn't There: Response Time

> **DEFAULT ACCESS LOGS DON'T INCLUDE HOW LONG A REQUEST TOOK**
>
> Neither Common nor Combined Log Format includes response time at all. Diagnosing "the site is slow" from the access log requires that timing actually be logged in the first place — which means checking, before anything else, whether it's been added to the server's own log format configuration.

SERVER	DIRECTIVE TO ADD RESPONSE TIME
Apache	`%D` (microseconds) or `%T` (seconds), added to a custom `LogFormat`
Nginx	`$request_time`, added to a custom `log_format` directive

With `%D` added, the same request from earlier might look like this — the final field, `4523912`, is microseconds, meaning this request took roughly 4.5 seconds:

```
192.168.1.10 - - [08/Aug/2026:14:32:07 +0100] "GET /login HTTP/1.1" 500 1204 4523912
```

The Error Log: Where the "Why" Often Lives

The access log tells you *what* request came in and *what* status went out. It rarely tells you *why* a request failed — that's the error log's job.

```
[Sat Aug 08 14:32:07.123456 2026] [php:error] [pid 12345] [client 192.168.1.10:54321] PHP Fatal error: Uncaugh
```

```
2026/08/08 14:32:07 [error] 12345#0: *67 connect() failed (111: Connection refused) while connecting to upstre
```

Both examples describe the same underlying situation — the web server couldn't reach a backend it depends on — from Apache's PHP module and Nginx's own upstream-proxy perspective respectively. Neither the Apache nor the Nginx access log line for this same request would show anything more specific than a `500` or `502`; the error log is what actually names the cause.

Reading Both Logs Together

The real technique, used throughout Chapters 6 and 7, is straightforward once both formats are familiar: find the access log line for the problem request, note its exact timestamp, then search the error log for entries at that same moment. The access log confirms *that* something failed and roughly how; the error log — when the failure produced one — explains *why*.

> **TIMESTAMPS ARE THE JOIN KEY BETWEEN LOGS**
>
> There's no formal database-style link between an access log line and an error log line — the timestamp is the only thing connecting them. This is exactly why Chapter 1's advice to check the timestamp first, and Chapter 3's note on confirming a server's clock/timezone is correct, both matter well beyond a single log file in isolation.

Hands-On Exercises

EXERCISE 1

A colleague says "I'll just check the access log to see how slow that request was." Explain why this chapter says that might not be possible, and what needs to be true first.

View solution

EXERCISE 2

An access log shows a request returned a 404, and a separate request returned a 500. Explain which one is more likely to indicate a genuine server-side problem worth investigating, and why.

View solution

EXERCISE 3

Explain what actually connects a specific access log line to the relevant error log line for the same failed request, given that this chapter says there's no formal link between the two files.

View solution

CHAPTER 4 QUICK REFERENCE

- **Combined Log Format** — host, identity, user, timestamp, request line, status, size, referer, user-agent, in that order
- **2xx** success, **3xx** redirect, **4xx** client error, **5xx** server error — a 5xx spike is the one worth real investigation
- Response time is **not included by default** — Apache needs `%D` / `%T`, Nginx needs `$request_time`, added explicitly to a custom log format
- The **error log** explains why a request failed; the access log only confirms that it did
- **Timestamps are the only link** between an access log line and its corresponding error log entry — there's no other formal connection
- **Next chapter:** Reading Authentication & Login Logs

CHAPTER 5 OF 10

Reading Authentication & Login Logs

Logging & Log Analysis

Chapter 5 · Reading Authentication & Login Logs

"A user can't log in" has been this course's own running example since Chapter 1. This chapter finally works through it properly — not as one problem, but as four genuinely different layers a request passes through, each with its own log source, each capable of being the actual point of failure.

The Diagnostic Order: Outside In

LAYER	WHAT COULD GO WRONG HERE	WHERE IT SHOWS UP
1. Network/perimeter	The request never reaches this server at all	Nowhere on this server — Chapter 1's own "silent failure" applies directly
2. Blocked before the app	Firewall, rate limiting, or a tool like fail2ban rejects the request first	fail2ban's own log, or a 401/403 in the access log with no matching application log entry at all
3. Application-level auth	Wrong credentials, locked/disabled account, expired session	The application's own log, plus `auth.log` / `secure` from Chapter 3
4. Post-authentication	Login succeeds, but something afterward breaks (redirect, session storage)	A successful auth entry, immediately followed by an unrelated-looking error elsewhere

Working outside-in matters: checking layer 3 first, when the real problem is layer 2, means reading application logs that were never going to show anything, because the request never reached the application to log anything at all.

Layer 1: Never Reaching the Server

If a user reports being unable to log in and *nothing* in any log on this server — not the access log, not the error log, not the application's own log — shows any trace of a request from them at all, the request most likely never arrived. A DNS problem, a network routing issue, or a client-side connectivity problem on the user's own end can all produce exactly this symptom. This server's own logs, by definition, cannot record a request that never reached it.

Layer 2: Blocked Before the Application

A very common real-world tool here is **fail2ban**, which watches logs (often `auth.log` itself) for repeated failed attempts and automatically firewalls off the offending IP for a period of time. Its own actions are logged separately:

```
2026-08-08 14:28:03 fail2ban.actions [12345]: NOTICE  [sshd] Ban 192.168.1.10
```

Once banned, that IP's later requests are rejected at the firewall — before Apache or Nginx ever sees them, meaning they won't appear in the access log either. A legitimate user who mistyped their password a few times too many can end up banned by exactly the same mechanism meant to stop an attacker; `fail2ban-client status <jail>` (or the ban log itself) is how you tell the two apart.

A softer version of the same layer: HTTP basic authentication configured directly on the web server (via `.htaccess` or an equivalent) rejects a request with a `401` before the application underneath ever runs — this does appear in the access log, just without any corresponding application-level log entry.

Layer 3: Authentication Failing in the Application

This is the layer most people assume is the whole problem, and it does cover the most common real causes:

- **Wrong credentials** — routine, per Chapter 2's own example; logged at INFO, not ERROR, in a well-configured application
- **Account locked or disabled** — a deliberate security feature after too many failed attempts on one specific account, distinct from fail2ban's IP-level ban
- **Account doesn't exist** — a typo'd username, or a genuinely deleted/never-created account
- **Session or token expired** — the user was previously logged in, but their session has since lapsed

> **ONE USER, MANY FAILURES VS. MANY USERS, ONE IP — A REAL SECURITY DISTINCTION**
>
> Repeated failed logins for the *same account* from the *same source* usually just means someone forgot their password. Repeated failed logins across *many different accounts* from the *same source* is a genuinely different pattern — often credential stuffing or a brute-force attempt — and worth escalating rather than treating as routine, even though both can look similar at a glance in a raw log.

Layer 4: Authenticated Successfully, Then Something Breaks

The rarest but most confusing case: the application's own log shows a genuinely successful authentication, but the user still ends up unable to log in — because something *after* that succeeded step fails. A session store that's unreachable (Redis down, disk full for file-based sessions) or a broken post-login redirect are both real

examples. The tell is a successful auth log line immediately followed by an apparently unrelated error somewhere else in the same request's timeframe — exactly the kind of cross-log correlation Chapter 4 introduced.

Hands-On Exercises

EXERCISE 1

A user says they can't log in, but there's no trace of their request anywhere in the access log, error log, or application log. Explain what this chapter says is the most likely explanation, and why the application's own logs couldn't have shown anything different.

View solution

EXERCISE 2

Explain the difference between an IP being banned by fail2ban and a single account being locked after too many failed attempts, and why checking application logs first would be the wrong order for diagnosing the fail2ban case.

View solution

EXERCISE 3

Explain why failed logins across many different usernames from the same source IP are treated differently in this chapter than repeated failed logins on a single user's own account, and what each pattern most likely indicates.

View solution

CHAPTER 5 QUICK REFERENCE

- "Can't log in" is **four layers**, diagnosed outside-in: network/perimeter → blocked before the app (fail2ban, basic auth) → application-level auth → post-authentication breakage
- If nothing appears in *any* log, the request likely never reached the server at all — this server's logs can't record what it never received
- **fail2ban** bans an IP after repeated failures; once banned, later requests won't even reach the access log
- Wrong credentials are routine (INFO); account lockouts and session/token expiry are the other common application-level causes
- **Same account repeatedly, same source** — routine. **Many accounts, same source** — a real security pattern worth escalating
- A successful auth entry followed by an unrelated-looking error is Layer 4 — something broke after login succeeded, not during it

- **Next chapter:** Diagnosing "It's Slow" — A Log-Based Troubleshooting Walkthrough

CHAPTER 6 OF 10

Diagnosing "It's Slow": A Log-Based Troubleshooting Walkthrough

Logging & Log Analysis

Chapter 6 · Diagnosing "It's Slow": A Log-Based Troubleshooting Walkthrough

Every chapter so far has built one piece: severity levels, log locations, access/error log fields, response-time logging, cross-log correlation. This chapter puts all of it to work on one realistic ticket, start to finish — the "Apache running slow" scenario this course has referenced since Chapter 1.

The Ticket

"Customers are reporting the checkout page is taking 10+ seconds to load. Started sometime this morning. Nothing else seems affected."

STEP 1

Before anything else: is response time even being logged? Checking the site's Apache config confirms `%D` was added to the `LogFormat` months ago — Chapter 4's own prerequisite is already satisfied, so the access log can actually answer a "how slow" question.

STEP 2

Filter the access log for the affected path and scan the response-time field:

```
grep "/checkout" /var/log/apache2/access.log # showing timestamp and %D (microseconds)

[08/Aug/2026:07:58:12] 312044   # 0.3s — normal
[08/Aug/2026:08:01:47] 298511   # 0.3s — normal
[08/Aug/2026:09:02:03] 11402887 # 11.4s — matches the complaint
[08/Aug/2026:09:02:19] 10887233 # 10.9s
[08/Aug/2026:09:15:41] 287905   # 0.3s — back to normal
[08/Aug/2026:10:02:05] 12109442 # 12.1s — slow again
```

Two real findings already: the slow requests are genuinely confirmed (not just a user's impression), and they're not constant — they cluster into short windows, roughly an hour apart, then return to normal.

STEP 3

Checking the error log for the same timestamps — `09:02` through `09:15` — turns up nothing dramatic, but one recurring line stands out precisely because it's routine and repeating in that exact window:

```
[08/Aug/2026:09:02:15] [warn] [pid 8821] AH01067: Slow database query detected (4.2s): SELECT * FROM cart
```

The application layer is confirming the slowness lives somewhere around the database, but not yet why.

STEP 4

Checking system-level activity for the same window via `journalctl --since "09:00" --until "09:20"` turns up the actual trigger:

```
Aug 08 09:00:01 web01 CRON[9432]: (root) CMD (/usr/local/bin/nightly_backup.sh)
Aug 08 09:00:02 web01 kernel: [warn] CPU load average: 8.42, 6.10, 4.88
```

STEP 5 — CORRELATING THE PATTERN

Lining the three sources up by timestamp tells a complete, consistent story:

- A cron job named `nightly_backup.sh` is starting at **09:00**, **10:00**, and presumably every hour on the hour — despite its name suggesting it should run once, overnight
- It drives CPU load sharply upward, visible in the kernel/system log
- Database queries slow down under that load, visible as the Apache `warn`-level slow-query notice
- The checkout page's own response time spikes to match, visible directly in the access log's `%D` field

No single log source proved this on its own — the access log proved *that* it was slow, the error log hinted at *where*, and the system log finally explained *why*. Correlated by timestamp, the three together point at one specific, fixable cause: a backup script with a misconfigured hourly schedule, not a nightly one.

DON'T STOP AT THE FIRST CORRELATED SIGNAL

The slow-query warning in Step 3 was tempting to treat as the full answer on its own — "the database is slow, case closed." It's true, but incomplete: without Step 4's system-level check, the actual root cause (a misconfigured cron schedule) would have stayed hidden, and any fix attempted at the database layer alone would likely have missed the real trigger entirely.

Hands-On Exercises

EXERCISE 1

Explain why Step 1 of this walkthrough checks whether %D is configured before doing anything else, referencing what this course covered in Chapter 4.

View solution

EXERCISE 2

Explain what each of the three log sources (access log, error log, system/journal log) individually proved in this walkthrough, and why the full picture needed all three rather than any one alone.

View solution

EXERCISE 3

Explain why this chapter warns against treating the Step 3 slow-query warning as a complete diagnosis on its own, even though it was a real, accurate finding.

View solution

CHAPTER 6 QUICK REFERENCE

- Confirm response time is actually logged (Chapter 4) **before** trying to diagnose "slow" from the access log at all
- The access log's `%D` / `$request_time` field confirms and quantifies the symptom — proof, not just a user's impression
- The error log often hints at *where* the slowness lives, without fully explaining *why*
- System-level logs (`journalctl`, kernel messages) frequently reveal the actual trigger — resource exhaustion, a misbehaving scheduled job, and similar
- Correlate all sources by timestamp before concluding a diagnosis is complete — a single confirmed finding can still be an incomplete answer

- **Next chapter:** Diagnosing "It's Returning the Wrong Thing" — A Second Troubleshooting Walkthrough

CHAPTER 7 OF 10

Diagnosing "It's Returning the Wrong Thing": A Second Troubleshooting Walkthrough

Logging & Log Analysis

Chapter 7 · Diagnosing "It's Returning the Wrong Thing": A Second Troubleshooting Walkthrough

Chapter 6 was one thread of evidence, layered until it pointed at a single cause. This chapter is different on purpose: four genuinely plausible causes, each checked and eliminated with real evidence, until only one survives. "Wrong response" problems are rarely obvious from the symptom alone — this is the second half of the user's own original example, worked through directly.

The Ticket

"The product page shows different prices for different users — some see today's updated prices, some still see yesterday's. Started right after this morning's price update went out."

Four candidate explanations are worth considering before touching anything: a bug in the update itself, a caching layer serving stale content, a misconfigured virtual host serving from the wrong source, or a backend that didn't receive the update. The temptation is to jump straight to the one that sounds most familiar — here, that's caching, since "different users see different things" is the classic caching fingerprint. This walkthrough deliberately doesn't skip the other three just because one guess feels obviously right.

STEP 1 — WAS THE UPDATE ACTUALLY APPLIED CORRECTLY? RULED OUT

Checking the application's own log for this morning's price update confirms it completed successfully, updating every affected product:

```
[08/Aug/2026:07:00:04] [INFO] Price update job completed: 1,204 products updated, 0 failures
```

The source of truth is correct. Whatever's serving stale prices, it isn't because the update itself failed or ran incompletely.

STEP 2 — IS A CACHING LAYER SERVING STALE CONTENT? RULED OUT

This is the tempting answer, so it gets checked properly rather than assumed. A quick request with response headers visible shows no cache layer is even in front of this path:

```
curl -I https://example.com/product/4821
HTTP/1.1 200 OK
X-Cache: (header not present at all)
```

No `X-Cache`, no `Age` header, nothing indicating a cache sits in front of this request. The symptom *looked* like caching, but there's no cache configured here to actually be the cause.

STEP 3 — IS THE WRONG VIRTUAL HOST SERVING SOME REQUESTS? RULED OUT

Checking the access log's own recorded `Host` field for both an "old price" and a "new price" request confirms both landed on the same, correct virtual host, serving from the same document root:

```
[08/Aug/2026:09:14:02] Host: shop.example.com 200 # showed old price
[08/Aug/2026:09:14:19] Host: shop.example.com 200 # showed new price
```

Identical host, identical config on both requests. Whatever's different between these two, it isn't which virtual host answered.

STEP 4 — DID EVERY BACKEND ACTUALLY RECEIVE THE DEPLOYMENT? CONFIRMED

This site runs behind an Nginx reverse proxy, load-balancing across two Apache backends — a detail worth checking directly, using an Nginx log format that includes `$upstream_addr`, the actual backend server that handled each request:

```
[08/Aug/2026:09:14:02] upstream: 10.0.0.11:80 # backend-1 — showed old price
[08/Aug/2026:09:14:19] upstream: 10.0.0.12:80 # backend-2 — showed new price
```

Requests alternate between two backend servers, and only one of them is actually serving new prices. The rollout of this morning's deployment reached `backend-2` but never completed on `backend-1` — which explains the exact symptom precisely: which price a user sees depends entirely on which backend the load balancer happens to route them to on that request.

THE RIGHT-SOUNDING ANSWER ISN'T THE SAME AS THE CONFIRMED ANSWER

> Caching was a genuinely reasonable first guess — the symptom pattern really does match how caching problems usually look. The discipline that mattered here wasn't avoiding that guess, it was refusing to stop at it without checking. A pattern match is a starting hypothesis, not a diagnosis, until something in the logs actually confirms it.

Hands-On Exercises

EXERCISE 1

Explain what specifically ruled out caching as the cause in Step 2, and why "different users see different things" wasn't enough evidence on its own to confirm it.

View solution

EXERCISE 2

Explain what $upstream_addr revealed in Step 4, and why this piece of information required a specific log configuration choice rather than being present in a default Nginx access log.

View solution

EXERCISE 3

Explain why this chapter says caching being "a genuinely reasonable first guess" wasn't the actual problem with jumping to it — what was the real mistake this chapter warns against?

View solution

CHAPTER 7 QUICK REFERENCE

- "Wrong response" problems often have several plausible causes — **check and eliminate each one with real evidence**, don't stop at whichever guess feels most familiar
- Confirm the update actually succeeded at the source before assuming it's a delivery/serving problem
- **Caching** is ruled in or out by response headers (`X-Cache`, `Age`) — not by symptom pattern-matching alone
- A misconfigured **virtual host** is ruled out by confirming the `Host` field and served content genuinely match across requests
- In a load-balanced setup, **which backend actually served a request** ($upstream_addr in Nginx) can reveal a partially-completed deployment — a genuinely different cause from all three of the others
- **Next chapter:** Log Aggregation & Centralized Logging

CHAPTER 8 OF 10

Log Aggregation & Centralized Logging

Logging & Log Analysis

Chapter 8 · Log Aggregation & Centralized Logging

Chapters 6 and 7 both worked because everything lived on one server, or a small enough handful that checking each one by hand was realistic. This chapter is a conceptual orientation to what happens once that stops being true — not a deep implementation guide, but enough to recognize the pieces and know what they're for.

The Problem: When "SSH Into Each Server" Stops Working

Two things break the manual approach as a system grows:

- **Scale.** Checking one server's logs by hand is fine. Checking the same thing across dozens or hundreds of servers, one at a time, isn't — the time cost alone makes it impractical during an active incident
- **Ephemerality.** In containerized and cloud environments, individual containers or instances are routinely destroyed and replaced rather than kept running indefinitely. A log that only exists on a container that no longer exists is a log that's gone permanently — Chapter 1's "silent failure" problem, but caused by the infrastructure itself rather than a missing log statement

The Core Idea: Ship Logs Somewhere Central

Rather than leaving each log where it was generated, a copy is forwarded to a central location as it's written. This solves both problems at once: one place to search instead of many, and logs that outlive the server or container that produced them.

syslog & rsyslog Forwarding

The syslog protocol Chapter 2 already introduced was designed to support forwarding to a remote collector, not just writing to a local file. **rsyslog**, the syslog daemon most Linux distributions actually run today, can be configured to forward matching log entries to a remote server with a small addition to its own config:

```
# /etc/rsyslog.conf — forward everything to a remote collector over TCP
*.* @@log-collector.internal:514
```

The double `@@` specifically means TCP (a single `@` means UDP) — a small syntax detail, but one worth getting right, since UDP forwarding can silently drop log messages under load with no error at all.

A Conceptual Tour: The ELK Stack

One of the most common centralized-logging stacks in real production use, built from three (or more) cooperating tools:

PIECE	ROLE
Elasticsearch	Stores and indexes the log data, making it searchable
Logstash (or Filebeat/Fluentd)	Collects and ships logs from each source into Elasticsearch, often reshaping them along the way
Kibana	The web UI for searching, filtering, and visualizing what's in Elasticsearch

A Conceptual Tour: Grafana Loki

A more lightweight alternative, built around a genuinely different tradeoff: rather than indexing the full text of every log line the way Elasticsearch does, **Loki** indexes only a small set of labels (like which service or server a log came from), storing the actual log text compressed and unindexed. That's a real, deliberate resource-usage tradeoff — cheaper to run at scale, at the cost of full-text search being somewhat less immediate than in an ELK-style setup. **Promtail** is Loki's typical log-shipping agent, and **Grafana** — already familiar to many teams from metrics dashboards — is the usual way to query and view Loki's own data.

ELK STACK	GRAFANA LOKI
Indexes full log text	Indexes only labels/metadata, not full text
Heavier resource footprint	Deliberately lighter, cheaper to run at scale
Kibana for visualization	Grafana for visualization — often already in use for metrics

What Doesn't Change

> **AGGREGATION CHANGES WHERE YOU SEARCH, NOT HOW YOU READ**
>
> Every skill from Chapters 1 through 7 still applies once logs are centralized — severity levels, status code categories, access/error log correlation by timestamp, the layered diagnostic approach from Chapter 5. Aggregation is a scale-and-search layer sitting on top of the exact same underlying log content, not a replacement for understanding what's actually in it.

CLOCK SKEW QUIETLY BREAKS TIMESTAMP CORRELATION ONCE CENTRALIZED

Every technique in this course that relies on matching timestamps across log sources depends on those timestamps actually being accurate. Once logs from many servers are pooled together, even a small clock discrepancy between two of those servers can make genuinely related events look minutes apart — or make unrelated events look suspiciously close together. Keeping every server's clock synchronized via NTP isn't a minor detail; it's what makes cross-server correlation trustworthy at all.

Hands-On Exercises

EXERCISE 1

Explain the two separate problems this chapter says log aggregation solves, and why "just SSH into each server" stops being a workable answer to either one as a system grows.

View solution

EXERCISE 2

Explain the core architectural difference between the ELK stack and Grafana Loki, and what real tradeoff that difference represents.

View solution

EXERCISE 3

Explain why clock skew between servers becomes a real problem specifically once logs are centralized, using techniques this course already covered in earlier chapters as part of your answer.

View solution

CHAPTER 8 QUICK REFERENCE

- Manual per-server log checking breaks down for two reasons: **scale** (too many servers to check by hand) and **ephemerality** (containers/instances that no longer exist)
- **rsyslog** can forward logs to a remote collector — `@@host:port` for TCP, a single `@` for UDP (which can silently drop messages under load)
- **ELK stack:** Elasticsearch (storage/search) + Logstash/Filebeat (shipping) + Kibana (visualization) — indexes full text
- **Grafana Loki:** indexes labels only, not full text — lighter, cheaper, paired with Grafana
- Aggregation doesn't change *what* you're reading or *how* you read it — every earlier chapter's skills still apply directly

- **Synchronized clocks (NTP) matter more, not less**, once correlating logs across many servers centrally
- **Next chapter:** Writing to Logs — Existing Files vs. Custom Log Files

CHAPTER 9 OF 10

Writing to Logs: Existing Files vs. Custom Log Files

Logging & Log Analysis

Chapter 9 · Writing to Logs: Existing Files vs. Custom Log Files

Every chapter so far has been about reading logs someone else's system already produced. This one flips the direction: writing them yourself — either adding to a log that already exists, or building a well-behaved new one from scratch.

Two Different Situations

	APPENDING TO AN EXISTING LOG	CREATING A CUSTOM LOG FILE
When	Adding to an application's own established log stream	A standalone script or tool with nothing existing to append to
Main risk	Breaking a format other tools already depend on	Choosing a bad location, or forgetting rotation entirely
Key discipline	Match the existing conventions exactly	Establish good conventions from the start

Appending to an Existing Log Correctly

An application's log file is often already being read by other tools — dashboards, alerting rules, or simply a colleague's own habitual `grep` commands. Writing a new line into it that doesn't match the existing format (different timestamp style, missing the usual level tag, fields in a different order) can silently break every one of those downstream consumers, even though the new line is perfectly readable to a human.

- **Use the application's own logging mechanism** where one exists, rather than writing raw text to the file directly — it already knows the correct format, and often already handles rotation and concurrent-write safety for you
- **Match level conventions** from Chapter 2 — don't introduce a new ad-hoc severity scheme into a file that already has an established one

CONCURRENT WRITES WITHOUT LOCKING CAN CORRUPT LOG LINES

If two processes write to the same file at the same moment without any coordination, their output can interleave mid-line — producing a genuinely corrupted, unparseable entry that's neither one process's line nor the other's. This is

> exactly why using an established logging mechanism (which typically handles this safely) is preferable to raw file writes from multiple sources.

Creating a Custom Log File

For a standalone script or tool with no existing log to append to, a simple, self-written logging function is often enough:

```
log() {
  local level="$1"; shift
  echo "$(date '+%Y-%m-%d %H:%M:%S') [$level] $*" >> /var/log/myapp/custom.log
}

log INFO "Backup completed successfully"
```

Two things this small function already gets right, on purpose: a consistent, sortable timestamp format, and a bracketed level tag matching Chapter 2's own convention — the same discipline that matters for an application's existing log applies just as much to a brand-new one.

Location matters too — a system-wide tool conventionally logs under `/var/log/`, in its own subdirectory, with permissions that match who's actually allowed to read or write it, rather than being placed wherever happens to be convenient at the time.

The Danger of Unbounded Growth

Chapter 1 warned against deleting a log file mid-incident to reclaim disk space. This is the other half of that same problem, addressed in advance: a custom log file with no rotation in place will simply grow forever, until it eventually does fill the disk on its own — turning "we might need to free some space one day" into a real, urgent incident.

`logrotate`: Automating the Solution

`logrotate` is the standard Linux tool for automatically rotating, compressing, and eventually discarding old log files on a schedule — it's what actually produces the `access.log.1` and `access.log.2.gz` files Chapter 3 mentioned in passing.

```
/* /etc/logrotate.d/myapp */
/var/log/myapp/custom.log {
    daily
    rotate 14
    compress
    missingok
    notifempty
    create 0640 myapp myapp
}
```

DIRECTIVE	WHAT IT DOES
daily	Rotate once a day (also commonly `weekly` or size-based)
rotate 14	Keep 14 old rotated copies before deleting the oldest
compress	Gzip rotated files once they're no longer the active log
missingok	Don't error out if the log file happens to be missing
notifempty	Skip rotating a file that's currently empty
create 0640 myapp myapp	Recreate the active log file with these permissions and ownership immediately after rotating

TEST A LOGROTATE CONFIG BEFORE TRUSTING IT LIVE

`logrotate -d /etc/logrotate.d/myapp` runs in debug mode — it shows exactly what logrotate *would* do without actually touching any files, letting you confirm the configuration behaves as intended before it runs for real on a schedule.

Hands-On Exercises

EXERCISE 1

A developer adds a new line format to an existing application log, using their own preferred timestamp style instead of the file's established one. Explain what this chapter says could go wrong, even though the new line is still readable to a human.

View solution

EXERCISE 2

Explain how this chapter's warning about unbounded log growth relates to Chapter 1's own warning about deleting logs mid-incident — how are they two sides of the same underlying problem?

View solution

EXERCISE 3

Explain what rotate 14 and compress each do in a logrotate config, and why testing with logrotate -d before relying on a new config is good practice.

View solution

CHAPTER 9 QUICK REFERENCE

- Appending to an **existing log**: match its format exactly, use the application's own logging mechanism where possible, respect Chapter 2's level conventions
- Concurrent, uncoordinated writes to one file can **interleave and corrupt lines** — a real reason to prefer an established logging mechanism over raw writes
- A **custom log file** needs a consistent format, a sensible location (typically under `/var/log/`), and correct permissions from the start
- Unbounded log growth is the same underlying risk as Chapter 1's "don't delete logs mid-incident" warning, addressed in advance rather than reacted to
- **logrotate** automates rotation, compression, and eventual deletion — `daily` / `rotate N` / `compress` / `create` are the core directives
- Test any new logrotate config with `logrotate -d` before trusting it to run unattended
- **Next chapter:** Capstone — Triaging Three Real Support Tickets

CHAPTER 10 OF 10

Capstone: Triaging Three Real Support Tickets

Logging & Log Analysis

Chapter 10 · Capstone — Triaging Three Real Support Tickets

Nine chapters built the pieces — levels, locations, formats, correlation, layered diagnosis, aggregation, writing your own. This capstone applies all of it to three fresh tickets, worked more briskly than Chapters 6 and 7's own dedicated walkthroughs, since the underlying discipline should already feel familiar by now.

Ticket 1: "I can't log into the admin panel"

An internal support-team member reports being unable to log into the admin panel since this morning, insisting their password is correct.

APPLYING CHAPTER 5'S LAYERED MODEL

- **Layer 1 (network):** their requests appear in the access log — ruled out
- **Layer 2 (blocked before the app):** `fail2ban-client status` shows their IP was never banned — ruled out
- **Layer 3 (application auth):** the app's own log shows the real cause — `[WARN] Login rejected: account status = inactive (user_id 118)`

An overnight account-cleanup job (visible in the same log, timestamped just before midnight) incorrectly flagged this active admin account as inactive during a routine sweep — not a password issue at all. Reactivating the account, and confirming a subsequent login succeeds in the log, closes the ticket.

Ticket 2: "The reporting dashboard is really slow"

Export generation on the reporting dashboard has been taking 20+ seconds since yesterday afternoon.

APPLYING CHAPTERS 4 AND 6'S TECHNIQUE

- **Access log:** `%D` confirms export requests genuinely spiked from ~1s to 20s+, starting right after yesterday's 2pm deployment

- **Error log:** hundreds of individual database query log lines per single export request — a strong signal, not just one slow query
- **Correlation:** yesterday's deployment notes show a code change to the export feature — comparing the before/after query counts confirms the new code queries the database once *per row* instead of once for the whole export

An N+1 query pattern, introduced by yesterday's own deployment — a code-level cause this time, not an infrastructure one, but found the same way: access log confirms the symptom, error log narrows down where, correlating with a recent change explains why.

Ticket 3: "EU customers are seeing US content"

Customers configured for the EU region occasionally see US-region homepage content instead.

APPLYING CHAPTER 7'S ELIMINATION METHOD — WITH A DIFFERENT RESULT THIS TIME

Checking response headers, exactly as Chapter 7 did:

```
curl -I https://example.com/ -H "X-Region: EU"
HTTP/1.1 200 OK
X-Cache: HIT
```

Unlike Chapter 7's own ticket, this time caching genuinely **is** involved — a CDN edge cache was added in front of this page last month for performance, but its own configuration never included the region header in its cache key. The first request to a given edge location gets cached, and every subsequent request to that same edge — regardless of region — receives the cached response. The fix is a caching-config change (varying the cache key on region), not a code change at all.

THE SAME CHECK, TWO HONESTLY DIFFERENT OUTCOMES

Chapter 7 ruled caching out by checking headers. This ticket confirms caching *is* the cause, using the exact same check. Neither result was assumed in advance — the discipline is checking, not guessing which answer is more likely to be right.

Chapter Attribution

TECHNIQUE USED ABOVE	SOURCE CHAPTER
Log-first triage, silent failures	Chapter 1

TECHNIQUE USED ABOVE	SOURCE CHAPTER
Reading severity levels (WARN in Ticket 1)	Chapter 2
Knowing where fail2ban and application logs live	Chapter 3
`%D` response-time field, access/error log correlation	Chapter 4
The four-layer login diagnostic model	Chapter 5
Correlating access log, error log, and a deployment record	Chapter 6
Checking cache headers rather than assuming either way	Chapter 7
(Not directly used this chapter, but the same skills scale to it)	Chapter 8
(Fixing Ticket 1 relies on the app's own logging conventions from)	Chapter 9

Honest Scope Note

WHAT THIS COURSE DELIBERATELY DOESN'T COVER

- No deep dive into commercial log-management platforms (Splunk, Datadog, and similar) — Chapter 8 covers the underlying concepts they're built on, not any specific vendor's own product
- No log-based alerting or monitoring setup — this course is about reading and writing logs, not building the systems that watch them automatically
- No SIEM/security-specific log analysis in depth — a genuinely separate discipline worth its own course
- No custom log-parsing tooling (regex extraction pipelines, structured logging libraries in specific languages) beyond what's needed to read a log by eye

Each is a legitimate, separate topic — not silently assumed solved by what this course actually covers.

Hands-On Exercises

EXERCISE 1

Explain why Ticket 1 turned out not to be a password problem at all, and which specific layer of Chapter 5's model actually contained the real cause.

View solution

EXERCISE 2

Explain what an "N+1 query pattern" means based on Ticket 2's own description, and why comparing the timing to yesterday's deployment was the key step in finding it.

View solution

EXERCISE 3

Explain why this chapter says Ticket 3 and Chapter 7's own ticket used "the exact same check" despite reaching opposite conclusions about caching, and why that isn't a contradiction.

View solution

CHAPTER 10 QUICK REFERENCE — COURSE COMPLETE

- Ticket 1: an apparent password problem was actually an application-level account status flag — Chapter 5's layered model found it directly
- Ticket 2: a code-level N+1 query pattern, found by correlating access-log timing with a recent deployment — the same method as Chapter 6, a different kind of cause
- Ticket 3: caching genuinely was the cause this time — the same header check from Chapter 7, an honestly different result
- The recurring theme across all ten chapters: **check, don't guess** — whichever specific technique that means in the moment
- This closes *Logging & Log Analysis*, 10/10 chapters — the first course under the new Technical Support subject