TECHNICAL SUPPORT

Graylog

A practical, query-and-troubleshooting-focused course built directly around real day-to-day usage at logger.daxtra.io — architecture and Lucene syntax, core scoping fields, severity, correlation IDs, service-specific field families, effective query-building, real worked troubleshooting workflows, and avoiding false signals — closing with a full incident investigation from first report to root cause.

10 Chapters | Philip Osztromok | Generated with Claude

CHAPTER 1: WHY GRAYLOG? ARCHITECTURE & THE LUCENE-STYLE SEARCH BOX

Graylog

Chapter 1 · Why Graylog? Architecture & the Lucene-Style Search Box

This course is deliberately practical, not a general Graylog administration guide — it's built around real usage at Daxtra's own Graylog instance, `logger.daxtra.io`, and the actual fields, services, and query patterns a solutions engineer runs into day to day. This opening chapter covers just enough of the underlying architecture and search syntax to make everything that follows make sense.

What Graylog Actually Is

Graylog is a log management platform — something that collects log messages from many different services, stores them centrally, and lets you search across all of them from one place instead of SSHing into a dozen different machines and grepping local files by hand. Under the hood, Graylog is built directly on real, well-established storage technology: it indexes and searches messages using Elasticsearch or OpenSearch (Graylog's own open-source codebase ships storage modules for both), while Graylog's own configuration — dashboards, alert definitions, user accounts — is kept separately in MongoDB. Log messages themselves arrive through **inputs**, real configured listeners that accept messages from applications, containers, or other log-shipping tools.

None of that plumbing needs to be memorized to use Graylog well day to day — but it explains a real, practical fact worth knowing up front: the search box you'll spend most of this course in isn't running a Graylog-invented query language of its own. It's speaking Lucene.

The Search Box Speaks Lucene

Graylog's own search box uses **Lucene-style query syntax** — the same query language family used by Elasticsearch itself and a wide range of other search tools built on it. The core shape of a query is `field:value`:

```
service:scout
```

Combine multiple conditions with real boolean operators — `AND`, `OR`, and `NOT` — and group alternatives inside parentheses:

```
service:scout AND level:(2 OR 3)
```

Wrap free text in double quotes to search for an exact phrase rather than individual words, and use a trailing asterisk for a wildcard match:

SYNTAX	MEANING
field:value	Exact match on a structured field
"exact phrase"	Free-text search for the literal phrase, not individual words
A AND B	Both conditions must match
A OR B	Either condition may match
field:(X OR Y)	Shorthand for field:X OR field:Y on the same field
field:Foo*	Trailing wildcard — matches anything starting with "Foo"

NO LEADING WILDCARDS

A query like `class:*Foo` — a wildcard at the *start* of the value — simply fails. Only trailing wildcards work (`class:Foo*`). This is a real, confirmed limitation of the search syntax at `logger.daxtra.io`, not a one-off bug — don't spend time debugging a leading-wildcard query that returns nothing; rewrite it instead.

THE TIME RANGE LIVES OUTSIDE THE QUERY STRING

Unlike every field covered so far, the time window a search covers isn't part of the Lucene query text at all — it's set separately, through Graylog's own time-range picker in the UI. A perfectly correct query can return zero results simply because the picker is scoped to the wrong window. Chapter 10's own capstone also covers a real (but explicitly unverified) way to encode a time range directly into a shareable URL.

What's Ahead in This Course

Chapters 2 through 6 build up the real field vocabulary used across Daxtra's own services — the core scoping fields every query starts from, severity levels, the correlation IDs that let you follow one request or job across multiple log lines, the HTTP-shaped fields common to the Node-based services, and the field families specific to Flux, the legacy Java monolith, and the older Perl/CGI stack. Chapter 7 turns that vocabulary into genuinely effective queries, Chapter 8 works through real troubleshooting scenarios query by query, and Chapter 9 covers the gotchas that cause a technically-correct query to still mislead you. The capstone ties all of it into one realistic incident investigation, start to finish.

Hands-On Exercises

EXERCISE 1

Explain, using this chapter's own material, why a query like `class:*Controller` would return zero results even if plenty of matching log lines genuinely exist, and rewrite it into a form that would actually work if the class names in question all started with the same known prefix.

View solution

EXERCISE 2

A colleague runs the query `service:scout AND level:(2 OR 3)` and gets zero results, even though they're confident scout has thrown critical or error-level messages recently. Using this chapter's own material, name the most likely explanation that has nothing to do with the query text itself.

View solution

EXERCISE 3

Write a Lucene-style query that matches messages where the `service` field is exactly `ws-lite` and the free-text content contains the exact phrase `connection refused`.

View solution

CHAPTER 1 QUICK REFERENCE

- Graylog stores and indexes messages via **Elasticsearch/OpenSearch**; configuration lives in **MongoDB**; messages arrive via configured **inputs**
- The search box uses real **Lucene-style query syntax**: `field:value`, `AND` / `OR` / `NOT`, `"exact phrase"` quoting, and `field:(X OR Y)` grouping
- **Only trailing wildcards work** — `field:Foo*` is valid, `field:*Foo` is not
- The **time range is set via the UI picker**, entirely separate from the query string itself
- **Next chapter:** Core Scoping Fields — service, account, daxtraResource & source

CHAPTER 2: CORE SCOPING FIELDS: SERVICE, ACCOUNT, DAXTRARESOURCE & SOURCE

Graylog

Chapter 2 · Core Scoping Fields: service, account, daxtraResource & source

Almost every query you'll build in this course starts by narrowing down *where* to look before worrying about *what* went wrong. These four fields are that narrowing toolkit — each answering a genuinely different real question: which application, whose data, which piece of infrastructure, and which exact machine.

service — The Logical App Name

`service` is the field you'll reach for first in nearly every investigation — it names the logical application that produced a given log message, independent of which specific host or container it happened to run on. The known clean values in real use:

```
scout, flux-worker, flux-controller, flux-rest, integrations-perl,
integrations-api, ws-sync, hwsdb, ws-lite, @daxtratech/rms-integration,
search, nginx, location-api, scheduler, libre-convert, spiders,
secrets-manager, match-explainer, cvx, doc-convert
```

Chapters 5 and 6 come back to several of these by name — the Node/Koa-shaped services (scout, search, integrations-api, ws-lite), the Flux pipeline (flux-worker, flux-controller, flux-rest), the Java monolith (hwsdb), and the legacy Perl stack (integrations-perl) each carry their own additional real field families on top of the ones this chapter covers.

account — The Customer/Account Identifier

`account` identifies a specific customer or account, and is mainly populated on the integration/RMS job platform. Real shapes seen in practice vary by which part of the system wrote the value:

```
bullhorn_27515        (underscore separator)
bullhorn-27515        (hyphen separator)
```

```
ti_bh_prd            (mapping-filename style, seen on integrations-perl)
```

EXACT MATCH — THE SEPARATOR HAS TO BE RIGHT

`account` is an exact-match field. Guessing the wrong separator — a hyphen where the real value uses an underscore, or vice versa — doesn't produce a partial match or a warning; it silently returns zero results. If an account query comes back empty, confirm the real, exact value first before concluding the account has no matching log activity at all.

daxtraResource — The Infrastructure Unit

`daxtraResource` names the specific piece of infrastructure a message came from — a cluster or a host, one level more concrete than the logical `service` name. Real examples:

```
scout-cluster-eu-west-1
scout-cluster-eu-west-2
dkr-eu-west-1-100.daxtra.io
es-azeu-107.daxtra.com
rms-integration-cluster-eu-west-1
```

This is the field to reach for when a problem is suspected to be regional or cluster-specific — the same logical service running in two different clusters can behave differently, and `daxtraResource` is what lets you isolate one from the other.

source — The Always-Present Fallback

`source` is the lowest-level identity field — a hostname or container ID — and unlike the three fields above, it's always present on every message, regardless of which service produced it or whether that service bothered to populate the more specific fields. When nothing else is available to scope a search, `source` still is.

FIELD	ANSWERS	ALWAYS PRESENT?
service	Which application?	No — populated per service, but consistently used

FIELD	ANSWERS	ALWAYS PRESENT?
account	Whose data? (mainly integration/RMS platform)	No — mainly integration/RMS platform
daxtraResource	Which cluster or host?	No — infrastructure-dependent
source	Which exact machine/container?	Yes — the real fallback field

COMBINE SERVICE AND ACCOUNT FOR TIGHT, REAL SCOPING

These fields are rarely used alone in practice. A real, common pattern for the legacy Daxtra Search product is combining service and account directly:

```
service:search AND account:bullhorn-27515
```

This immediately narrows a search to one specific application, for one specific customer — usually enough on its own to make the remaining log volume genuinely readable.

Hands-On Exercises

EXERCISE 1

A ticket mentions account "bullhorn_31200" but a query for `account:bullhorn-31200` returns zero results. Using this chapter's own material, explain the most likely cause and how you'd confirm it.

View solution

EXERCISE 2

A query scoped by `service` and `account` returns nothing at all, and you suspect the service in question never populates the account field for this particular log line. Using this chapter's own material, name the one field that should still be usable to identify where the message came from.

View solution

EXERCISE 3

Write a query that scopes to the scout service running specifically on the eu-west-2 cluster, using this chapter's own real `daxtraResource` example value.

View solution

CHAPTER 2 QUICK REFERENCE

- **service** — the logical app name (scout, hwsdb, ws-lite, nginx, and more); the field almost every query starts from
- **account** — customer/account identifier, mainly on the integration/RMS platform; **exact-match** — a wrong separator (hyphen vs. underscore) silently returns zero results
- **daxtraResource** — the specific cluster or host; use it to isolate a regional or cluster-specific problem
- **source** — hostname or container ID; the one field that's **always present**, regardless of service
- Combining `service` + `account` is a real, common scoping pattern
- **Next chapter:** Severity & the level Field

CHAPTER 3: SEVERITY & THE LEVEL FIELD

Graylog

Chapter 3 · Severity & the level Field

Once a query is scoped to the right service, account, or cluster, the next real question is usually "how bad is it?" — and that's exactly what the `level` field answers.

The Real Syslog Severity Scale

`level` holds a real, standard syslog severity value (RFC 5424) — and the numbering is genuinely counter-intuitive the first time you see it: **lower numbers mean more severe**, the opposite of how a 1-to-10 "severity rating" usually works.

VALUE	REAL SYSLOG NAME
0	Emergency
1	Alert
2	Critical
3	Error
4	Warning
5	Notice
6	Informational
7	Debug

IN REAL DAY-TO-DAY USE AT LOGGER.DAXTRA.IO

Daxtra's own services don't populate all eight levels in practice — the ones that actually show up day to day are **2 (critical), 3 (error), 4 (warning), 6 (info), and 7 (debug)**. Levels 0 (Emergency), 1 (Alert), and 5 (Notice) exist in the standard, but aren't part of the real, working vocabulary here — don't spend time searching for them.

Querying by Severity

A single severity value is a plain field match:

```
level:3
```

Group several severities together the same way Chapter 1 covered grouping any other field:

```
level:(2 OR 3)
```

Lucene also supports real range comparisons on numeric fields, using `<` and `>` directly:

```
level:<3
```

Watch the Direction (and the Edge) of the Inequality

LEVEL:<3 DOES NOT INCLUDE ERROR (3) ITSELF

Because lower numbers mean more severe, `level:<3` matches only levels 0, 1, and 2 — Emergency, Alert, and Critical — genuinely excluding level 3 (Error) itself. In real day-to-day use at logger.daxtra.io, where levels 0 and 1 aren't actually populated, `level:<3` effectively means "critical only." If what you actually want is "error or worse," the safer, unambiguous query is the explicit OR-grouping from earlier in this chapter — `level:(2 OR 3)` — or a less-than-or-equal comparison, `level:<=3`. Always double-check which one you actually meant before trusting the results of a level range query; the off-by-one mistake here is easy to make and doesn't produce any kind of warning, just a silently narrower result set than intended.

Hands-On Exercises

EXERCISE 1

Using this chapter's own real severity table, name the syslog severity name for level 4, and state whether it's more or less severe than level 6.

View solution

EXERCISE 2

Write a query, using this chapter's own real syntax, that matches every message at warning level or more severe (levels 2, 3, and 4) for the ws-lite service — using the explicit OR-grouping form, not a range comparison.

View solution

EXERCISE 3

A colleague writes `level:<4` expecting it to include warning-level (4) messages. Using this chapter's own material, explain why it doesn't, and give the corrected query.

View solution

CHAPTER 3 QUICK REFERENCE

- The real syslog severity scale (RFC 5424): 0=Emergency, 1=Alert, **2=Critical, 3=Error, 4=Warning**, 5=Notice, **6=Informational, 7=Debug**
- **Lower number = more severe** — the opposite of a typical 1-10 severity rating
- In real use at logger.daxtra.io, only levels **2, 3, 4, 6, and 7** actually appear — 0, 1, and 5 aren't part of the working vocabulary
- `level:(2 OR 3)` groups severities explicitly; `level:<3` is a range comparison — **watch the direction and the edge**, since `level:<3` excludes level 3 itself
- **Next chapter:** Correlation IDs — Following One Request or Job Across Log Lines

CHAPTER 4: CORRELATION IDS: FOLLOWING ONE REQUEST OR JOB ACROSS LOG LINES

Graylog

Chapter 4 · Correlation IDs: Following One Request or Job Across Log Lines

A single real user action — one API call, one Flux sync, one job run — almost never produces just one log line. It produces a scattered trail across several services, each logging its own small piece of the story. Correlation IDs are what let you pull that scattered trail back together into one coherent sequence, and different fields correlate at genuinely different scopes.

requestId, restRequestId & messageRequestId — One HTTP Request

These three fields all serve the same real purpose: correlating every log line produced while handling one single HTTP request, regardless of how many internal steps that request triggers. Which specific name shows up depends on which part of the stack wrote the log line.

THE CORE REAL WORKFLOW

Once you've found one relevant log line — say, from a candidate-facing error message — copy its `requestId` and search on that alone:

```
requestId:"<id from a candidate message>"
```

This surfaces every other log line tied to that exact same request, across whichever services touched it — often the fastest real way to go from "one confusing error" to "the full sequence of what actually happened."

correlationId — Cross-Service Job Correlation

`correlationId` operates one level up from a single HTTP request — it correlates one item or job as it moves through **cross-service** steps, the real example being a Flux pipeline run that touches several distinct services in sequence. Where `requestId` ties together the log lines from one request, `correlationId` ties together the log lines from one whole multi-step job, potentially spanning several requests.

sessionId — User/Session Flow

`sessionId` correlates at a different scope again — a user's own session, potentially spanning multiple separate requests over time as they interact with a system.

inputId — One Flux Sync Run

`inputId` is Flux-specific, identifying one particular sync run.

jobRunId — The Scheduler's Own ID

`jobRunId` is used by the scheduler service specifically.

JOBRUNID GETS FORWARDED DOWNSTREAM AS REQUESTID

A job that originates in the scheduler doesn't keep its own `jobRunId` name as it propagates into the rest of the system — it's forwarded downstream and shows up there under the name `requestId` instead. If you're tracing a scheduler-originated job and it seems to "disappear" after the scheduler's own log lines, search for the same ID value under `requestId` rather than assuming the trail has genuinely gone cold.

FIELD	CORRELATES
requestId / restRequestId / messageRequestId	One HTTP request, across every service that touches it
correlationId	One item/job across cross-service steps (e.g. Flux)
sessionId	One user/session flow
inputId	One Flux sync run
jobRunId	One scheduler job — forwarded downstream as requestId

Hands-On Exercises

EXERCISE 1

You've found one log line for a scheduler job with `jobRunId:abc123`, but a follow-up search for `jobRunId:abc123` only returns that one line, even though you know the job triggered further downstream processing. Using this chapter's own material, explain what to search for next.

View solution

EXERCISE 2

Explain, using this chapter's own material, the real difference in scope between `requestId` and `correlationId` — which one is broader, and what real example does this chapter give for when the broader one is actually needed?

View solution

EXERCISE 3

Write a query, using this chapter's own real syntax, that finds every log line associated with the request ID `7f2a-91bc`.

View solution

CHAPTER 4 QUICK REFERENCE

- **requestId / restRequestId / messageRequestId** — one HTTP request, across services
- **correlationId** — one item/job across cross-service steps (e.g. Flux)
- **sessionId** — one user/session flow
- **inputId** — one Flux sync run
- **jobRunId** — scheduler-specific; **forwarded downstream as requestId** — search there if the trail seems to end
- **Next chapter:** The HTTP/Koa/Express Field Shape

CHAPTER 5: THE HTTP/KOA/EXPRESS FIELD SHAPE

Graylog

Chapter 5 · The HTTP/Koa/Express Field Shape

A cluster of real services — scout, search, integrations-api, ws-lite, and nginx — all log requests in a shared, consistent shape, since they're all Node-based (Koa/Express) or, in nginx's case, fronting them directly. Learn this one field set once, and it applies across all five services without re-learning anything.

The Field Set

FIELD	HOLDS
requestMethod	The HTTP method — GET, POST, etc.
requestPath	The request path
requestHost	The request's target host
requestIp	The requesting client's IP address
status / requestStatus	The HTTP status code returned
responseSize	Response size, in bytes
gl2_processing_duration_ms	Processing duration, in milliseconds

GL2_ IS A GRAYLOG-INTERNAL PREFIX, NOT AN APP-LOGGED FIELD

Every other field in this table was written by the application itself as part of its own real log message. `gl2_processing_duration_ms` is different — the `gl2_` prefix is Graylog's own reserved namespace for metadata Graylog itself attaches to a message during processing, not something the service's own code chose to log. It's still genuinely useful for spotting slow requests, just worth knowing it comes from a different real source than the rest of this table.

Which Services Use This Shape

This field set applies specifically to **scout, search, integrations-api, ws-lite, and nginx** — real services either built on Koa/Express or, for nginx, sitting in front of them as a reverse proxy and logging the same kind of request/response detail. Chapter 6 covers a genuinely different real field shape used by other services in the same environment — don't assume this one applies everywhere.

A Real Worked Example: HTTP 500s for One Service

Combining `service` (Chapter 2) with `status` from this chapter is one of the most direct real ways to find server-side failures for a specific application:

```
service:ws-lite AND status:500
```

Hands-On Exercises

EXERCISE 1

Write a query, using this chapter's own real field, that finds every POST request to the integrations-api service.

View solution

EXERCISE 2

Explain, using this chapter's own material, why `gl2_processing_duration_ms` is worth treating differently from the other six fields in this chapter's own table when reasoning about where a value actually came from.

View solution

EXERCISE 3

A colleague wants to search for slow, large responses from the scout service — responses over 100,000 bytes. Using this chapter's own real field and Chapter 3's own range-comparison syntax, write a query for responses larger than 100000 bytes on scout.

View solution

CHAPTER 5 QUICK REFERENCE

- Shared HTTP field shape: **requestMethod, requestPath, requestHost, requestIp, status/requestStatus, responseSize, gl2_processing_duration_ms**
- Applies to **scout, search, integrations-api, ws-lite, and nginx** — not universal across every service
- `gl2_` is a **Graylog-internal prefix** — that one field's value comes from Graylog's own processing metadata, not the application's own log line
- Real worked pattern: `service:<name> AND status:500` for server-side failures on one service
- **Next chapter:** Service-Specific Field Families: Flux, hwsdb & Legacy Perl/CGI

CHAPTER 6: SERVICE-SPECIFIC FIELD FAMILIES: FLUX, HWSDB & LEGACY PERL/CGI

Graylog

Chapter 6 · Service-Specific Field Families: Flux, hwsdb & Legacy Perl/CGI

Chapter 5 covered a field shape shared across five services. This chapter covers the opposite — vocabularies that belong to just one part of the stack, and one field, `candidateId`, that genuinely isn't a structured field at all despite looking like one.

integrations-api's Own Fields

`action` and `integration` are structured fields specific to **integrations-api only**.

> **INTEGRATIONS-API ≠ INTEGRATIONS-PERL**
>
> These two service names look like variants of the same thing, but they genuinely aren't — they're different real codebases with different real field vocabularies. `action` and `integration` apply to **integrations-api** and are **not** structured fields on **integrations-perl**, which has its own separate real field family, covered later in this chapter. Double-check which of the two you're actually querying before assuming a field applies.

candidateId — The Free-Text-Only Gotcha

> **CANDIDATEID IS USUALLY NOT A STRUCTURED FIELD**
>
> In most real services, `candidateId` appears only as free text embedded in a log message, not as its own structured field. Searching `candidateId:<id>` as a field will usually miss real matches entirely — search the ID as plain, unquoted-field free text instead:
>
> ```
> <the id itself, as free text, no field prefix>
> ```
>
> This is a genuinely easy mistake, since `candidateId` looks exactly like the other ID-shaped fields covered in Chapter 4 — the difference is that those really are structured, and this one usually isn't.

Flux Pipeline Fields

FIELD	HOLDS
rmsId	An RMS-side identifier
datumType	Candidate or Vacancy
datum	The Mongo ID of the underlying record
requestTime	Time, in milliseconds

These sit alongside Chapter 4's own Flux-relevant correlation fields (`correlationId`, `inputId`) — together they cover both *what* a Flux pipeline step touched (`datumType` / `datum`) and *which run* it belonged to.

hwsdb (Java Monolith) Fields

FIELD	HOLDS
class	The fully qualified class name
pid	Process ID
thread	Thread name

> **A DIRECT ECHO OF CHAPTER 1**
>
> `class` is exactly the field Chapter 1 used to introduce the leading-wildcard limitation — `class:*Foo` fails, only a trailing wildcard like `class:Foo*` works. Anytime you're narrowing an hwsdb investigation by class name and only know part of it, that constraint applies directly.

Legacy Perl/CGI Fields

FIELD	HOLDS
file	Source filename
line	Line number

FIELD	HOLDS
package	Perl package name
subroutine	Subroutine name
categoryName	A dot-joined category, e.g. `CC.Bullhorn.DB`

`categoryName` is worth a second look — its dot-joined structure (`CC.Bullhorn.DB`) means a broader category search can use a trailing wildcard the same way `class` does: `categoryName:CC.Bullhorn.*` would match every subcategory under `CC.Bullhorn` at once.

Hands-On Exercises

EXERCISE 1

A colleague queries `action:sync AND service:integrations-perl` and gets zero results, even though they're confident sync actions are happening on that service right now. Using this chapter's own material, explain the most likely cause.

View solution

EXERCISE 2

You need to find every log line mentioning candidate ID `cand-88213`. Using this chapter's own material, explain why `candidateId:cand-88213` is the wrong query to start with, and write the query you'd actually use instead.

View solution

EXERCISE 3

Write a query, using this chapter's own real field and its trailing-wildcard behavior, that matches every legacy Perl/CGI log line categorized anywhere under `CC.Bullhorn`.

View solution

CHAPTER 6 QUICK REFERENCE

- **action** / **integration** — integrations-api only, genuinely NOT shared with integrations-perl despite the similar name
- **candidateId** — usually free text only, not a structured field; search the ID as plain text instead
- **Flux** — rmsId, datumType (Candidate/Vacancy), datum (Mongo ID), requestTime
- **hwsdb** — class (subject to Chapter 1's leading-wildcard limit), pid, thread
- **Legacy Perl/CGI** — file, line, package, subroutine, categoryName (dot-joined, wildcard-friendly from the right end)
- **Next chapter:** Building Effective Lucene Queries

CHAPTER 7: BUILDING EFFECTIVE LUCENE QUERIES

Graylog

Chapter 7 · Building Effective Lucene Queries

Chapters 2 through 6 built up a real field vocabulary. This chapter turns that vocabulary into genuinely effective queries — how to combine fields well, a real precedence trap worth knowing about straight from Lucene's own documentation, and how to exclude results deliberately rather than just include them.

Combining Fields: The Building-Block Approach

The real, most common query shape in day-to-day use is a chain of `AND`-joined field matches, each one narrowing the result set further. Two real, worked examples:

```
-- Critical/error logs for one account
account:bullhorn-27515 AND level:(2 OR 3)

-- Critical/error logs for one service
service:scout AND level:(2 OR 3)
```

Both queries follow the identical real shape: one field pinning down *where* to look (account or service), combined with `level:(2 OR 3)` pinning down *how bad* — the same real severity grouping covered in Chapter 3.

Operator Precedence: Always Parenthesize Mixed AND/OR

A real, easy-to-miss fact straight from Apache Lucene's own documentation: **OR is the default conjunction**. If two terms sit next to each other with no operator between them at all, Lucene treats that as `OR`, not `AND`.

> **MIXING AND AND OR WITHOUT PARENTHESES IS GENUINELY AMBIGUOUS**
>
> Lucene's own documentation doesn't guarantee a specific precedence when `AND` and `OR` are mixed in one query without grouping. Its own recommended real example: `(jakarta OR apache) AND`

`website` — explicit parentheses "eliminate any confusion" about which combination was actually intended. Applied to this course's own real field vocabulary: don't write `service:scout OR service:search AND level:3` and assume you know what it means — write `(service:scout OR service:search) AND level:3` instead, so the grouping is unambiguous regardless of how the query engine happens to resolve unparenthesized precedence.

Excluding Results: NOT and the Minus Operator

Two real ways to exclude a term. The `NOT` operator removes a specific term, but genuinely requires at least one other term alongside it — `NOT` can't stand alone as a whole query. The minus operator (`-`) does the same job and can be used more freely:

```
service:scout AND NOT level:7
service:scout -level:7
```

Both real forms above do the same thing — scout's own log activity, excluding debug-level (7) noise.

Free Text + Field Combos

Quoted free text (Chapter 1) and field matches combine the same way any two field conditions do — a real, common pattern for chasing one specific known error message without it getting lost across every service that happens to log something similar:

```
"connection timed out" AND service:hwsdb
```

Building a Query Incrementally

In real practice, the most reliable way to build a query is rarely to write the whole thing at once. Start with one broad scoping field (`service` or `account`), run it, confirm there's real log volume to work with, then add `level`, then narrow further with a correlation ID or free-text phrase once you've spotted something specific worth chasing. Each step is independently

checkable, which makes it far easier to tell whether an empty result set means "nothing happened" or "one part of this query is wrong."

GOAL	PATTERN
Scope + severity	field:value AND level:(2 OR 3)
Mixed AND/OR	(field:A OR field:B) AND field:C — always parenthesized
Exclude a value	field:value AND NOT field:other / field:value -field:other
Known error, scoped	"exact phrase" AND service:name

Hands-On Exercises

EXERCISE 1

A query needs to match either service scout or service search, but only at critical or error level (2 or 3). Using this chapter's own real parenthesization rule, write it correctly.

View solution

EXERCISE 2

Write a query, using this chapter's own minus-operator syntax, that finds all ws-lite activity except debug-level (7) messages.

View solution

EXERCISE 3

Explain, in your own words, why building a query incrementally — one field at a time — makes it easier to diagnose an unexpectedly empty result set than writing the whole query at once.

View solution

CHAPTER 7 QUICK REFERENCE

- Real, common pattern: `field:value AND level:(2 OR 3)` — a scoping field plus a severity group
- **OR is Lucene's default conjunction** when no operator is given — mixing AND and OR without parentheses is genuinely ambiguous; always group explicitly
- **NOT** requires at least two terms; the **minus operator (-)** does the same job and can be used more freely
- Free text and field matches combine the same way any two conditions do: `"phrase" AND field:value`
- **Build incrementally** — one field at a time — to make an empty result set easy to diagnose
- **Next chapter:** Real Troubleshooting Workflows, Worked Query by Query

CHAPTER 8: REAL TROUBLESHOOTING WORKFLOWS, WORKED QUERY BY QUERY

Graylog

Chapter 8 · Real Troubleshooting Workflows, Worked Query by Query

No new syntax in this chapter — just three realistic support scenarios, each worked from the first broad query through to a specific answer, using nothing but the fields and techniques Chapters 2 through 7 already covered.

Scenario 1: "A Customer's Search Results Look Wrong"

A ticket reports that account `bullhorn-27515` is seeing unexpected search results on the legacy Daxtra Search product. Following Chapter 7's own incremental-building method:

```
Step 1 — confirm there's real activity for this account on this service
service:search AND account:bullhorn-27515

Step 2 — narrow to anything that actually looks like a problem
service:search AND account:bullhorn-27515 AND level:(2 OR 3)

Step 3 — pick one specific error line and follow its requestId
requestId:"<id copied from that error line>"
```

Step 1 alone confirms the account is genuinely active on the right service — a real, useful sanity check before assuming anything is broken. Step 2 filters straight to anything worth investigating. Step 3 follows Chapter 4's own request-correlation pattern to see the full sequence of what happened around that one specific error.

Scenario 2: "RMS Integration Jobs Are Failing Overnight"

A report comes in that RMS integration jobs failed overnight, with no specific account named yet.

```
Step 1 — scope to the cluster, not a specific service or account yet
daxtraResource:"rms-integration-cluster-eu-west-1" AND level:(2 OR 3)
```

```
Step 2 — once a pattern emerges, pick one affected job and trace it
correlationId:"<id from one failing job's own log line>"
```

Starting at the cluster level (Chapter 2) rather than guessing at a service or account first is the right call here — the ticket doesn't name either, but it does name infrastructure. Once step 1 surfaces a specific failing job, Chapter 4's own `correlationId` — not `requestId` — is the right field to reach for, since this is a cross-service Flux-style job, not a single HTTP request.

Scenario 3: "Scout Errors on One Region"

Monitoring flags an increase in scout errors, specifically on the eu-west-2 cluster.

```
service:scout AND daxtraResource:"scout-cluster-eu-west-2" AND level:<3
```

CONFIRM WHAT "ERRORS" ACTUALLY MEANS BEFORE TRUSTING THIS QUERY

Per Chapter 3's own real warning, `level:<3` here matches only levels 0, 1, and 2 — in real practice at logger.daxtra.io, that's critical-only, since 0 and 1 aren't populated. If the report of "errors" genuinely includes level-3 Error messages, not just level-2 Critical ones, this query is silently too narrow. The corrected version, matching Chapter 3's own fix, is `level:(2 OR 3)` in place of `level:<3` — always worth double-checking which one a report actually means before trusting the result count.

SCENARIO	STARTING FIELD	REFINEMENT FIELD
Search results look wrong	service + account	level, then requestId
RMS jobs failing overnight	daxtraResource (cluster)	level, then correlationId
Scout errors, one region	service + daxtraResource	level range — check < vs. OR-grouping

Hands-On Exercises

EXERCISE 1

In Scenario 1, explain why Step 1 (service + account, no level filter yet) is worth running before Step 2, rather than jumping straight to the level-filtered query.

View solution

EXERCISE 2

In Scenario 2, explain why `correlationId` is the right field to reach for in Step 2, rather than `requestId`.

View solution

EXERCISE 3

Rewrite Scenario 3's own query so that it genuinely includes both critical (2) and error (3) level messages, using this chapter's own corrected form.

View solution

CHAPTER 8 QUICK REFERENCE

- **Scenario 1** — service + account confirms real activity first, then level narrows to problems, then requestId follows one specific error
- **Scenario 2** — daxtraResource (cluster) is the right starting point with no named service/account yet; correlationId (not requestId) traces a cross-service job
- **Scenario 3** — a real, worked reminder that `level:<3` excludes level 3 itself; confirm what "errors" means before trusting the query
- Every scenario reused only fields and techniques already covered in Chapters 2-7 — no new syntax needed
- **Next chapter:** Avoiding False Signals & Performance Pitfalls

CHAPTER 9: AVOIDING FALSE SIGNALS & PERFORMANCE PITFALLS

Graylog

Chapter 9 · Avoiding False Signals & Performance Pitfalls

Every query in this course so far has been about getting real results back. This chapter is about a genuinely different problem: a query that runs cleanly and returns real, matching data can still mislead you, or simply never finish, if you don't know these three real gotchas.

Baseline Noise vs. Real Signal

Finding a real error string in the logs feels like finding the cause — but a common error string appearing is often **baseline noise**, not a genuine signal specific to the incident you're investigating.

> **CHECK KNOWN-HEALTHY ACCOUNTS AND TIME WINDOWS FIRST**
>
> Before treating a matched error as the actual cause of a reported problem, check whether the exact same error also appears for accounts known to be working fine, or during time windows well outside the reported incident. If it does, that error is very likely a real, constant, low-level background noise condition — not evidence of what actually went wrong this time. This connects directly to Chapter 8's own troubleshooting discipline: an incremental, step-by-step query only tells you what matched, never automatically tells you whether what matched is actually relevant.

Free-Text Keywords on Candidate-Search Services

On the candidate-search services — **scout and search** — a generic free-text keyword search behaves very differently from what you might expect. These services log real candidate and resume content as part of their own normal operation, so a plain keyword match is far more likely to hit **resume or candidate content** than any actual code path or error condition.

> **A DIRECT ECHO OF CHAPTER 6**
>
> This is the same real underlying reason Chapter 6's own `candidateId` gotcha exists — these two services carry an unusually large amount of real, unstructured candidate-related text through their logs. The practical fix here is the same discipline: scope with `service` and `daxtraResource` first, and

prefer exception- or status-shaped structured queries (`level:(2 OR 3)` , `status:500`) over generic free-text keyword searches on these two services specifically.

Wide Aggregations & High-Cardinality Timeouts

Aggregating over a real high-cardinality field — one with many distinct values, like `daxtraResource` — across a wide time range (seven or more days) can genuinely time out rather than complete.

NARROW THE TIME RANGE FIRST

If a wide aggregation is timing out, the real fix is narrowing the time range before the query even runs — using Chapter 1's own time-range picker, not the query text itself. A shorter window over the same high-cardinality field is far more likely to complete, and can be widened again incrementally once you know the query itself is reasonable.

Hands-On Exercises

EXERCISE 1

A specific error string is found in the logs for the account reporting a problem. Using this chapter's own material, describe the one check you'd run before concluding this error is the actual cause.

View solution

EXERCISE 2

A colleague searches scout's logs for the free-text keyword "manager" hoping to find code-path errors related to a manager service, but gets thousands of unrelated results. Using this chapter's own material, explain what's most likely happening and how they should rewrite the search.

View solution

EXERCISE 3

A wide aggregation on `daxtraResource` over the last 30 days keeps timing out. Using this chapter's own material, explain the real fix, and why it involves a part of the Graylog UI outside the query box itself.

View solution

CHAPTER 9 QUICK REFERENCE

- **Baseline noise** — check whether a matched error also appears for known-healthy accounts/time windows before treating it as the real cause
- **scout and search** log real candidate/resume content — generic free-text keywords match that content, not code paths; prefer service/daxtraResource scoping plus exception/status-shaped queries
- **Wide aggregations** on high-cardinality fields (e.g. daxtraResource) over 7+ days can time out — narrow the time range (Chapter 1's own picker) first, not the query text
- **Next chapter:** Capstone — Investigating a Real Incident Start to Finish

CHAPTER 10: CAPSTONE — INVESTIGATING A REAL INCIDENT START TO FINISH

Graylog

Chapter 10 (Capstone) · Investigating a Real Incident Start to Finish

No new fields, no new syntax — one realistic incident, traced from a vague first report through to a confirmed root cause, using nothing but the real tools Chapters 1 through 9 already covered.

The Ticket

"Account bullhorn-27515 says their Bullhorn sync stopped working overnight, and separately a couple of their recruiters say search results look off. Might be related, might not."

Step 1 — Confirm Real Activity (Chapter 2)

```
service:search AND account:bullhorn-27515
```

Real activity is present — the account and service are genuinely correct, ruling out a Chapter 2-style separator mistake before anything else.

Step 2 — Narrow by Severity, Correctly (Chapters 3 & 7)

```
service:search AND account:bullhorn-27515 AND level:(2 OR 3)
```

Explicit OR-grouping, not a range comparison — deliberately avoiding Chapter 3's own off-by-one trap. A handful of real error-level messages turn up.

Step 3 — Rule Out Baseline Noise (Chapter 9)

Before trusting that error as the cause, check it against a known-healthy account:

```
"<the exact error text>" AND account:<known-healthy account>
```

It doesn't appear there — this error is genuinely specific to this account, not background noise.

Step 4 — Follow One Request (Chapter 4)

```
requestId:"<id from the error line>"
```

The full sequence around that one request shows a downstream call that itself failed — pointing toward something happening outside the search service entirely.

Step 5 — Pivot to the Integration Cluster (Chapters 2, 4 & 6)

That downstream failure looks Flux-shaped, not search-shaped. Widening scope from a single request to the whole integration cluster:

```
daxtraResource:"rms-integration-cluster-eu-west-1" AND level:(2 OR 3)
```

Several real failures show up on this cluster overnight. Picking one and following its own `correlationId` (not `requestId` — this is a cross-service job, per Chapter 4's own scope distinction) surfaces the same real Flux fields from Chapter 6 — `datumType:Candidate`, a specific `rmsId` — confirming this is the same real sync job the ticket originally described as broken.

Step 6 — Confirm It's Not a Candidate-Content False Positive (Chapter 9)

Since this investigation touched `search` directly, it's worth double-checking that nothing here was actually just a free-text keyword coincidentally matching resume content rather than a genuine code-path error — Chapter 9's own scout/search gotcha. Every query used in this

capstone has been structured (`service`, `account`, `level`, correlation IDs), not generic free text, so this concern doesn't apply here — a real, deliberate design choice, not an afterthought.

Building a Shareable Link

UNVERIFIED — TREAT AS A STARTING GUESS, NOT A CONFIRMED RECIPE

Nobody has confirmed that a link built this way actually loads the expected search when clicked. The real, safer alternative is Graylog's own built-in share/save-search feature in the UI, which generates a working link without any manual encoding. What follows is recorded as a real starting point only, not a trusted shortcut.

```
# Base URL
https://logger.daxtra.io/search

# Query parameters
?q=<URL-encoded query>
    # space -> +, : -> %3A, " -> %22, < -> %3C
&rangetype=relative&from=<seconds-ago>
    # from=300 (5 min), from=900 (15 min), from=3600 (1 hour)

# Worked example, encoding "service:scout AND daxtraResource:"scout-cluster-eu-west-2" /
https://logger.daxtra.io/search?q=service%3Ascout+AND+daxtraResource%3A%22scout-cluster
```

CAPSTONE STEP	BUILT USING
Confirm activity	Chapter 2 — service, account
Narrow by severity	Chapter 3, Chapter 7 — level:(2 OR 3), correct grouping
Rule out noise	Chapter 9 — baseline noise check
Follow one request	Chapter 4 — requestId
Pivot to cluster	Chapter 2, Chapter 6 — daxtraResource, correlationId, Flux fields
Confirm no false positive	Chapter 9 — structured vs. free-text discipline
Shareable link	Real, explicitly unverified recipe from the cheat sheet

Hands-On Exercises

EXERCISE 1

Explain, in your own words, why Step 5 switches from `requestId` to `correlationId` — what specifically changed about the nature of what's being traced?

View solution

EXERCISE 2

Build the query for Step 3 (the baseline-noise check) as a full, real query, substituting a literal exact error phrase, `"upstream request failed"`, and a literal known-healthy account, `bullhorn-40021`.

View solution

EXERCISE 3

Explain why this chapter recommends Graylog's own built-in share/save-search feature over hand-building a URL using the recipe given here, even though the recipe is presented in full.

View solution

CHAPTER 10 (CAPSTONE) QUICK REFERENCE

- A full incident investigation built from every real field and technique in Chapters 1-9, with no new syntax introduced
- The shareable-link URL recipe is **explicitly unverified** — Graylog's own built-in share/save-search feature is the safer real alternative

Course Complete

Graylog is now complete — 10/10 chapters, from the shape of the search box itself through to a full, real incident investigation, grounded throughout in genuine day-to-day usage at


logger.daxtra.io.