Elasticsearch Fundamentals

Day 1 — 4-Day Elasticsearch Course

Elasticsearch 8.18 | Kibana | Docker

Agenda

1. What is Elasticsearch?
2. Core Concepts
3. Setup & Kibana Tour
4. **Practice 1A** — Cluster Exploration
5. CRUD Operations
6. **Practice 1B** — CRUD & Shards
7. Q&A + Day 2 Preview

What is Elasticsearch?

The search engine that powers the modern web

Elasticsearch in One Sentence

A **distributed, RESTful search and analytics engine** built on Apache Lucene, designed for horizontal scalability, near real-time search, and multi-tenancy.

- Open-source core (SSPL / Elastic License)
- Written in Java
- Communicates via **REST API** (JSON over HTTP)
- Schema-free (dynamic mapping) but schema-aware when needed

What Problems Does It Solve?

Problem	How ES Helps
Full-text search	Tokenization, stemming, relevance scoring
Log analytics	Fast aggregations over time-series data
Autocomplete	Edge n-gram tokenizer, suggesters
Geo search	Geo-point/geo-shape queries
Vector / semantic search	dense_vector, ELSER, kNN
Real-time dashboards	Sub-second aggregations with Kibana

Who Uses Elasticsearch?

- **Wikipedia** — full-text search across articles
- **GitHub** — code search across 200M+ repositories
- **Netflix** — monitoring + log analysis
- **Uber** — real-time geospatial search
- **Stack Overflow** — question search + autocomplete

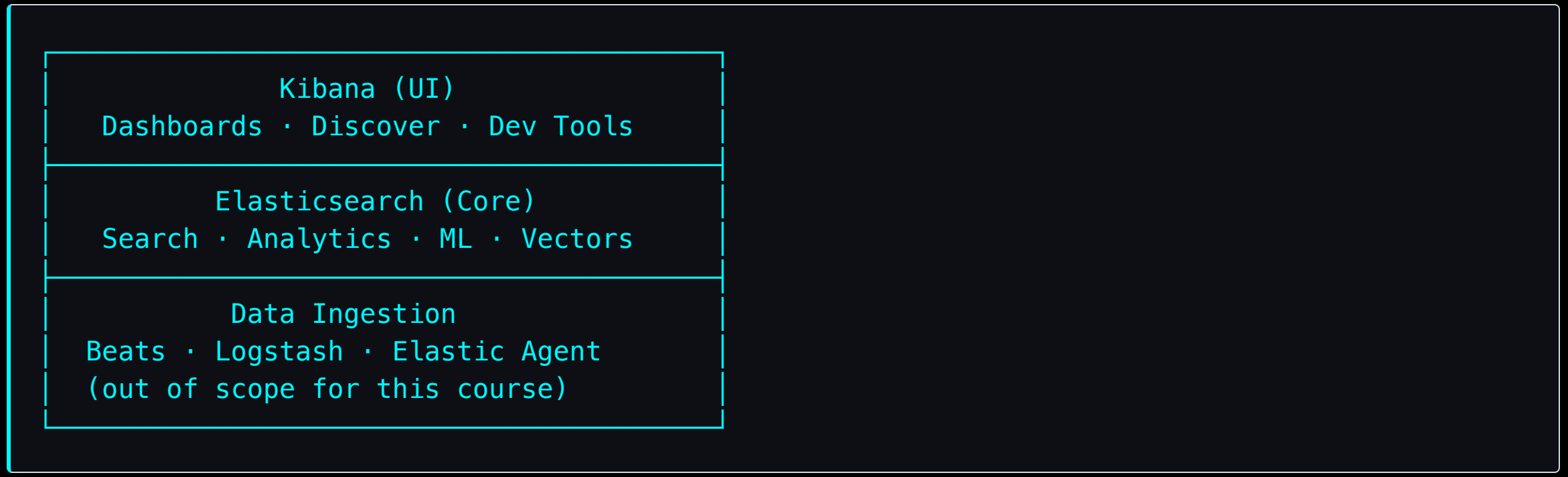
> Every time you search a website, there's a good chance Elasticsearch is behind it.

Elasticsearch vs Relational Databases

	Relational DB	Elasticsearch
Optimized for	Transactions (ACID)	Search & analytics
Schema	Strict (ALTER TABLE)	Flexible (dynamic mapping)
Query language	SQL	Query DSL / ES|QL
Joins	Native (FK)	Limited (denormalize!)
Scaling	Vertical (mostly)	Horizontal (sharding)
Real-time	After COMMIT	Near real-time (~1s refresh)

> ES is **not** a replacement for your database. It's a **complement**.

The Elastic Ecosystem



We'll focus on **Elasticsearch + Kibana** in this course.

Core Concepts

Building blocks of Elasticsearch

Document

The **basic unit** of information in Elasticsearch — a JSON object.

```
{
  "title": "The Shawshank Redemption",
  "overview": "Two imprisoned men bond over years...",
  "genres": ["Drama", "Crime"],
  "vote_average": 8.7,
  "release_date": "1994-09-23"
}
```

- Every document has a unique `_id`
- Stored in an **index**
- Equivalent to a **row** in a relational database

Index

A **collection of documents** with similar structure.

- Equivalent to a **table** in a relational database
- Has a **mapping** (schema) — either dynamic or explicit
- Has **settings** (number of shards, replicas, analyzers)

```
Index: movies
├── Doc 1: The Shawshank Redemption
├── Doc 2: The Godfather
├── Doc 3: Inception
└── ... (thousands more)
```

> Index names must be lowercase, no spaces, no special characters except `-` and `_`.

Cluster & Node

Node — a single Elasticsearch instance (one JVM process)

Cluster — a group of nodes working together

```
Cluster: elastic-cluster
├── Node 1 (master + data)
├── Node 2 (data)
└── Node 3 (data)
```

Node Roles

Role	Purpose
master	Cluster state, index creation/deletion
data	Store data, execute queries
ingest	Pre-process documents (pipelines)
ml	Machine learning (ELSER, inference)

> A single node can have **multiple roles**. Our `elk-single` node is master + data + ingest.

Shards

An index is split into **shards** — each shard is a self-contained Lucene index.

```
Index: movies (5 primary shards)
├── Shard 0  →  Node 1
├── Shard 1  →  Node 2
├── Shard 2  →  Node 1
├── Shard 3  →  Node 3
└── Shard 4  →  Node 2
```

- **Primary shards** — hold the original data
- **Replica shards** — copies for high availability + read throughput
- Default: 1 primary shard, 1 replica (since ES 7.x)

> Number of primary shards is **fixed** at index creation. Plan accordingly!

Why Sharding Matters

Without sharding

```
1 node, 1TB index → single point of failure, limited throughput
```

With sharding

```
5 shards × 2 nodes = parallel search across 5 Lucene indices
```

Benefits:

- **Horizontal scaling** — add nodes, redistribute shards
- **Parallel execution** — queries run on all shards simultaneously
- **Fault tolerance** — replicas on different nodes survive node failure

Replicas

```
Shard 0 (primary)   → Node 1
Shard 0 (replica)   → Node 2    ← automatic failover

Shard 1 (primary)   → Node 2
Shard 1 (replica)   → Node 1
```

- Replicas are **never** on the same node as their primary
- Single-node cluster: replicas stay **UNASSIGNED** (yellow health)
- More replicas = more read throughput (at cost of storage + indexing speed)

The Inverted Index

How Elasticsearch makes full-text search **fast**.

```
Document 1: "The quick brown fox"
Document 2: "The quick brown dog"
Document 3: "The lazy brown fox"
```

Inverted Index Lookup

```
Term      → Documents
──────────────────────
brown     → [1, 2, 3]
dog       → [2]
fox       → [1, 3]
lazy      → [3]
quick     → [1, 2]
the       → [1, 2, 3]
```

Query `"quick fox"` → intersection of `quick[1,2]` ∩ `fox[1,3]` → **Doc 1**

How Text Gets Into the Inverted Index

```
"The Quick Brown FOX!"
    ↓  Character filter (strip HTML, etc.)
"The Quick Brown FOX!"
    ↓  Tokenizer (split into tokens)
["The", "Quick", "Brown", "FOX!"]
    ↓  Token filters (lowercase, stemming, stop words)
["the", "quick", "brown", "fox"]
    ↓  Stored in inverted index
```

This pipeline is called an **analyzer** — we'll deep-dive on Day 3.

BM25 — How Relevance Scoring Works

Every search result has a `_score`. Elasticsearch uses **BM25** (Best Match 25):

```
score(q, d) = Σ  IDF(t) × (tf(t,d) × (k1 + 1)) / (tf(t,d) + k1 × (1 − b + b × |d|/avgdl))
```

Intuition (what matters):

Factor	Meaning
TF (term frequency)	More occurrences in doc → higher score
IDF (inverse doc frequency)	Rare terms across index → higher score
Field length	Shorter fields → higher score (match in title > match in overview)

> "The Shawshank Redemption" scores higher for `"shawshank"` than a 5-page document mentioning it once.

Putting It All Together

```
Cluster: elastic-cluster
│
├── Index: movies
│   ├── Mapping: title(text), genres(keyword), vote_average(float)...
│   ├── Settings: 1 shard, 1 replica
│   │
│   ├── Shard 0 (primary) → Node 1
│   │   ├── Inverted index for "title"
│   │   ├── Doc values for "vote_average"
│   │   └── Documents 1–2500
│   │
│   └── Shard 0 (replica) → Node 2
│
└── Index: .kibana (system index)
```

Setup & Kibana Tour

Getting your hands dirty

Starting Elasticsearch

We use the **elk-single** stack — single node, HTTP, simple auth.

```
# Start the stack
docker compose -f docker/elk-single/docker-compose.yml \
  --env-file .env up -d

# Check containers
docker compose -f docker/elk-single/docker-compose.yml \
  --env-file .env ps
```

Endpoints:

- Elasticsearch: `http://localhost:9200` (auth: `elastic` / `elastic`)
- Kibana: `http://localhost:5601` (same credentials)

Verify: Cluster Health

```
GET _cluster/health
```

Response:

```
{
  "cluster_name": "elastic-cluster",
  "status": "yellow",
  "number_of_nodes": 1,
  "number_of_data_nodes": 1,
  "active_primary_shards": 5,
  "unassigned_shards": 3
}
```

Cluster Health Status

Status	Meaning
green	All shards assigned
yellow	All primaries OK, some replicas unassigned
red	Some primary shards unavailable

> Single-node clusters are always **yellow** — replicas can't be on the same node as their primary.

Useful `_cat` APIs

Compact, human-readable output for cluster inspection:

```
# List all indices
GET _cat/indices?v

# List nodes
GET _cat/nodes?v

# Shard allocation
GET _cat/shards?v

# Cluster health (compact)
GET _cat/health?v

# Disk allocation per node
GET _cat/allocation?v
```

> The `?v` parameter adds column headers.

Kibana Tour: Discover

1. Open **http://localhost:5601**
2. Log in with `elastic` / `elastic`
3. Navigate to **Discover** (hamburger menu → Analytics → Discover)

Discover lets you:

- Browse documents in any index
- Filter by time range and field values
- Write KQL (Kibana Query Language) queries
- Switch to ES|QL mode (Day 2)

Kibana Tour: Dev Tools Console

The most important tool for this course!

1. Menu → **Management** → **Dev Tools**
2. Left panel: write requests
3. Right panel: see responses

```
GET _cluster/health
```

Shortcuts:

- `Ctrl+Enter` — execute request
- `Ctrl+/` — toggle comment
- `Ctrl+Space` — autocomplete

> All code examples in this course are meant for Dev Tools.

Kibana Tour: Dashboard & Stack Management

Dashboards:

- Pre-built visualizations from sample data
- We'll load Kibana sample data in exercises

Stack Management:

- Index Management — view/delete indices
- Data Views — define patterns for Discover
- Saved Objects — export/import dashboards

Practice 1A

Cluster Exploration & Setup

`day1-exercises.md` — Part A (2 tasks) | ~10 min

CRUD Operations

Create, Read, Update, Delete

REST API Conventions

```
<HTTP_METHOD> /<index>/<endpoint>  { "body": "JSON" }
```

Operation	HTTP Method	Endpoint
Create/Replace	`PUT`	`/<index>/_doc/<id>`
Create (auto ID)	`POST`	`/<index>/_doc`
Read	`GET`	`/<index>/_doc/<id>`
Update	`POST`	`/<index>/_update/<id>`
Delete	`DELETE`	`/<index>/_doc/<id>`
Search	`GET/POST`	`/<index>/_search`

Create: With Explicit ID (PUT)

```
PUT /movies/_doc/1
{
  "title": "The Shawshank Redemption",
  "overview": "Two imprisoned men bond over years...",
  "genres": ["Drama", "Crime"],
  "vote_average": 8.7,
  "release_date": "1994-09-23"
}
```

- `PUT` with an ID → creates or **replaces** the document
- Response includes `"result": "created"` or `"result": "updated"`

Create: With Auto-generated ID (POST)

```
POST /movies/_doc
{
  "title": "Inception",
  "overview": "A thief who steals corporate secrets...",
  "genres": ["Action", "Science Fiction"],
  "vote_average": 8.4,
  "release_date": "2010-07-15"
}
```

- `POST` without an ID → Elasticsearch generates a unique `_id`
- Use when you don't need to control the ID

Read: Get a Document

```
GET /movies/_doc/1
```

Response:

```
{
  "_index": "movies",
  "_id": "1",
  "_version": 1,
  "_source": {
    "title": "The Shawshank Redemption",
    "genres": ["Drama", "Crime"],
    "vote_average": 8.7
  }
}
```

- `_source` — the original document you indexed
- `_version` — increments on each update

Update: Partial Update

```
POST /movies/_update/1
{
  "doc": {
    "vote_average": 8.8,
    "tagline": "Fear can hold you prisoner. Hope can set you free."
  }
}
```

- **Merges** new fields into existing document
- Existing fields not in `doc` are **preserved**
- Under the hood: read → merge → reindex (documents are immutable in Lucene)

Delete a Document

```
DELETE /movies/_doc/1
```

Response:

```
{
  "_index": "movies",
  "_id": "1",
  "_version": 2,
  "result": "deleted"
}
```

Loading Our Dataset

Instead of creating documents one by one, let's load the **movies dataset**:

```
# Install dependencies (if not done yet)
pip install -r requirements.txt

# Load 100 movies
python data/load_data.py --dataset movies --size small
```

```
GET movies/_count
// → {"count": 100}
```

This loads 100 curated movies with fields:

- `title`, `overview`, `genres`, `vote_average`, `release_date`

Basic Search

```
GET /movies/_search
{
  "query": {
    "match": {
      "title": "godfather"
    }
  }
}
```

- The `match` query analyzes the search text, then finds matching documents
- Results are ranked by **relevance score** (`_score`)

Search Response Anatomy

```
{
  "took": 5,                                    // Time in ms
  "hits": {
    "total": { "value": 1 },                    // Total matches
    "max_score": 7.12,                          // Best score
    "hits": [{
      "_id": "2", "_score": 7.12,
      "_source": { "title": "The Godfather", ... }
    }]
  }
}
```

- `took` — query execution time in milliseconds
- `hits.total.value` — number of matching documents
- `_score` — BM25 relevance score (higher = more relevant)
- `_source` — original document fields

Search: Multiple Fields

```
GET /movies/_search
{
  "query": {
    "multi_match": {
      "query": "space adventure",
      "fields": ["title", "overview"]
    }
  }
}
```

- Searches across both `title` and `overview`
- Returns documents ranked by **BM25 relevance score**
- We'll explore Query DSL in depth on **Day 2**

Filter by Keyword

```
GET /movies/_search
{
  "query": {
    "term": {
      "genres": "Drama"
    }
  }
}
```

- `term` query does **exact matching** (no analysis)
- Use for `keyword` fields: genres, status, tags
- For full-text search on `text` fields, use `match`

Count & Exists

```
# Count documents in an index
GET /movies/_count

# Check if index exists
HEAD /movies

# Check if document exists
HEAD /movies/_doc/1
```

`HEAD` requests return only HTTP status code — `200` (exists) or `404` (not found).

Bulk API Preview

For loading many documents efficiently:

```
POST /_bulk
{"index": {"_index": "movies", "_id": "100"}}
{"title": "My Movie", "genres": ["Action"], "vote_average": 7.0}
{"index": {"_index": "movies", "_id": "101"}}
{"title": "My Other Movie", "genres": ["Drama"], "vote_average": 6.5}
```

- Each operation is **two lines**: action + document
- Much faster than individual requests (single network roundtrip)
- We'll cover bulk operations in depth on **Day 3**

Cleanup

```
# Delete a single index
DELETE /movies

# Delete multiple indices (careful!)
DELETE /movies,movies-test

# List all indices to verify
GET _cat/indices?v
```

> In production, always use aliases and index lifecycle policies instead of manual deletion.

Near Real-Time Search

Why your document doesn't appear instantly

The Refresh Cycle

```
Index document → In-memory buffer → [refresh] → Searchable segment
                                          ↑
                                    Every 1 second
```

- Documents are **not immediately searchable** after indexing
- The **refresh** operation makes buffered docs searchable
- Default refresh interval: **1 second**
- This is why ES is called "near real-time" (NRT)

```
# Force refresh (don't abuse in production)
POST /movies/_refresh

# Check refresh interval
GET /movies/_settings
```

Practice 1B

CRUD, Mappings & Shards

`day1-exercises.md` — Part B (4 tasks) | ~20 min

Summary

What we learned today

Day 1 Recap

Concept	Key Takeaway
Elasticsearch	Distributed search engine on top of Lucene
Document	JSON object = basic unit of data
Index	Collection of documents (≈ table)
Shard	Partition of an index for horizontal scaling
Inverted Index	Data structure enabling fast full-text search
BM25	Relevance scoring: TF × IDF × field length
CRUD	REST API: PUT/POST/GET/DELETE `_doc`
Refresh	~1s delay before docs become searchable

Day 2 Preview

Tomorrow we'll dive deep into **Query DSL**:

- Full-text queries: `match`, `multi_match`, `match_phrase`
- Term-level queries: `term`, `range`, `exists`
- Compound queries: `bool` (must / filter / should / must_not)
- Pagination and highlighting
- **ES|QL** — Elasticsearch's new SQL-like query language

> Make sure `elk-single` is still running!

Thank You!

Questions?

Day 1 — Elasticsearch Fundamentals