

Java/Spring Boot Elasticsearch

Estimated Time: ~ 2–3 days for Assignment A; +1 day for Assignment B (bonus)

Deliverables: A Git repository that includes all source code, configuration files (such as Docker Compose), sample data, and a clear README. Please fill out the Google Form and share all relevant links there – [\[Click Here\]](#).

Overview

Build a small Spring Boot application that:

1. Indexes a set of sample “course” documents into Elasticsearch.
2. Exposes a REST endpoint to **search** courses with multiple filters, pagination, and sorting (Assignment A).
3. (Bonus) Implements **autocomplete suggestions** and **fuzzy matching** for course titles (Assignment B).

Glimpse of Browse Course - [Click Here](#)

Candidates will start from an empty project (e.g., generated via Spring Initializr) and launch Elasticsearch locally using Docker Compose.

Part 1: Elasticsearch Setup

1. Create a **docker-compose.yml** that spins up a single-node Elasticsearch cluster (version 7.x or 8.x).
 2. Ensure the cluster is accessible on **localhost:9200** without authentication.
 3. Include instructions in your README on how to bring Elasticsearch up and verify it's running (e.g., via **curl http://localhost:9200**).
-

Part 2: Prepare Sample Data

1. Create a JSON file named **sample-courses.json** containing at least **50 course objects**. Each object must include fields for:
 - **id** (unique identifier)
 - **title** (short text)
 - **description** (longer text)

- **category** (e.g., “Math,” “Science,” “Art,” etc.)
 - **type** (values: **ONE_TIME**, **COURSE**, or **CLUB**)
 - **gradeRange** (e.g., “1st–3rd”)
 - **minAge** and **maxAge** (numeric)
 - **price** (decimal or double)
 - **nextSessionDate** (ISO-8601 date-time string, e.g., “2025-06-10T15:00:00Z”).
2. Vary the data so you have a mix of categories, age ranges, course types, prices, and session dates spanning different weeks.
 3. Place this file under **src/main/resources** so the application can read it when bootstrapping.
-

Part 3: Assignment A (Required) – Basic Course Search with Filters

3.1. Project Initialization

- Create a **new Spring Boot project** (group/artifact of your choice).
- Include dependencies for:
 - Spring Web (REST controllers)
 - Spring Data Elasticsearch (or the official Elasticsearch client)
 - Your chosen JSON library (e.g., Jackson)
 - (Optional) Lombok or any utility libraries.

3.2. Elasticsearch Configuration

- Configure your application to connect to the local Elasticsearch instance on **localhost:9200**.
- Document in your README how to set any necessary properties (e.g., host, port) so that a reviewer can run the application without modification.

3.3. Define Course Document Structure

- Create an entity (e.g., **CourseDocument**) corresponding to the fields in **sample-courses.json**.
- Ensure each field has an appropriate Elasticsearch mapping type (e.g., **text** for searchable text, **keyword** for exact matches, **date** for session dates, **double** or

`float` for price).

3.4. Bulk-Index Sample Data

- Implement a component (e.g., a service or a startup listener) that reads `sample-courses.json` at application startup and bulk-indexes all course objects into Elasticsearch's `courses` index.
- Document in your README how to trigger or verify this data ingestion.

3.5. Implement the Search Service

- Build a service layer method that executes a search query against the `courses` index, applying:
 - **Full-text search** on `title` and `description` (e.g., multi-match).
 - **Range filters** for `minAge/maxAge` and `minPrice/maxPrice`.
 - **Exact filters** for `category` and `type`.
 - **Date filter** for `nextSessionDate` (to show only courses on or after a given date).
- Implement sorting logic:
 - Default sort: ascending by `nextSessionDate` (soonest upcoming first).
 - If a `sort=priceAsc` parameter is passed, sort by `price` (low to high).
 - If `sort=priceDesc` is passed, sort by `price` (high to low).
- Add pagination support via `page` and `size` parameters.

3.6. Expose the Search Controller

- Expose a REST endpoint, for example: `GET /api/search`
- Accept query parameters:
 - `q` (search keyword)
 - `minAge, maxAge`
 - `category`
 - `type`
 - `minPrice, maxPrice`
 - `startDate` (ISO-8601)
 - `sort` (`upcoming`, `priceAsc`, `priceDesc`)
 - `page` (default 0), `size` (default 10)
- Return a JSON response containing:
 - `total` (total hits)

- `courses` (array of matching course documents, including at least `id`, `title`, `category`, `price`, and `nextSessionDate`).

3.7. Testing & Verification

- Include in your README:
 - Exact `curl` or HTTP examples showing how to call `/api/search` with different combinations of parameters.
 - Expected behavior (e.g., correct `total` count, proper sorting, and filtering).
 - (Optional, but encouraged) Provide a few basic integration tests that:
 - Spin up Elasticsearch (via Testcontainers or a local instance)
 - Index a small subset of courses
 - Verify that specific queries return the expected results.
-

Part 4: Assignment B (Bonus) – Autocomplete Suggestions & Fuzzy Search

If you finish Assignment A early or want extra credit, implement the following:

4.1. Autocomplete (Completion Suggester)

1. Index with a Completion Field

- Create a new Elasticsearch index (or extend the existing one) that includes a “completion” field for `title`.
- Re-index your sample data so that each course document has a “title” and a corresponding “suggest” sub-field used for autocomplete.

2. Autocomplete Endpoint

- Expose an endpoint, for example: `GET /api/search/suggest?q={partialTitle}`
 - Query Elasticsearch using the completion suggester API to return up to 10 matching titles that start with `partialTitle`.
 - Return a JSON array of suggested course titles.

3. README Examples

- Show in the README how to call the suggest endpoint (e.g., `q=phy`) and what responses to expect.

4.2. Fuzzy Search Enhancement

1. Fuzzy Matching in Search

- Adjust your existing search logic so that, when a keyword (`q`) is provided, the `title` field can match with fuzziness (allowing small typos).
- For instance, searching for “dinors” should still match “Dinosaurs 101.”

2. Documentation

- In your README, demonstrate via examples that a fuzzy query with a typo still returns the correct document(s).
-

Part 5: Submission Guidelines

1. Git Repository

- Push all your code to a public or private repo.
- Ensure the commit history is clear and incremental (e.g., separate commits for setup, indexing logic, search implementation, bonus features).

2. README

- Clearly explain how to:
 - Launch Elasticsearch (via `docker-compose up -d`).
 - Build and run the Spring Boot application.
 - Populate the index with sample data.
 - Call each endpoint, with example `curl` or HTTP requests.
 - (Bonus) Trigger autocomplete and fuzzy search tests.

3. Project Structure & Naming

- Follow sensible package and directory conventions (e.g., `config`, `document`, `repository`, `service`, `controller`).
- Use clear class and method names.

4. Tests (Optional but Recommended)

- Include unit or integration tests that verify core search functionality.
- If using Testcontainers, show how to run tests against an ephemeral Elasticsearch instance.

5. Evaluation Criteria

- **Correctness:** Does the search endpoint return accurate results for given filters and sorting?
- **Performance:** Are queries constructed efficiently (using filters rather than scanning all documents)?
- **Code Organization:** Clear separation of concerns (controllers vs. services vs. repositories).

- **Documentation:** Can a reviewer follow the README and get everything running in under 30 minutes?
- **Bonus:** Correct implementation of autocomplete and fuzzy matching if provided.