Use Elasticsearch in your Java applications

A quick-start guide to the high-performance RESTful search engine and document store

Matt C. Tyson
Architect
Freelance

March 08, 2016

Elasticsearch is taking the full-text search world by storm by combining an easy-to-use REST API with automated cluster scaling. Get a hands-on introduction to using Elasticsearch from a command shell and from within a Java™ application.

If you ever used Apache Lucene or Apache Solr, you know that the experience can be intense. Especially if you needed to scale your Lucene- or Solr-based solution, you understand the motivation behind the Elasticsearch project. Elasticsearch (which is built on top of Lucene) provides high-performance, full-text search capabilities in a simple-to-manage package that supports clustered scaling out of the box. You can interact with Elasticsearch through a standard REST API or from programming-language-specific client libraries.

This tutorial shows how Elasticsearch works in practice. Learn the basics of the REST API first by accessing it from the command line. Then set up a local Elasticsearch server and interact with it from a simple Java application. See Download to get the sample code.

Prerequisites

To work through all of this tutorial's examples, you need Elasticsearch on your system. Download the latest Elastic Search bundle for your platform. Decompress the package to a convenient location. On UNIX or Linux, spin up the instance with:

```
/elastic-search-dir/bin/elasticsearch
```

On Windows, run

```
/elastic-search-dir/bin/elasticsearch.bat
```

When you see the logging message `started`, the node is ready to accept requests.

For the Java example, you also need Eclipse and Apache Maven. Download and install both if you don't already have them on your system.

You also need cURL. On Microsoft Windows, I use the Git Bash shell to run cURL.

Using cURL for REST commands

You can issue cURL requests against Elasticsearch, which makes the framework easy to try out from a command-line shell.

> *“ Elasticsearch is schemaless. It can eat anything you feed it and process it for later querying. ”*

Elasticsearch is schemaless, which means that it can eat anything you feed it and process it for later querying. Everything in Elasticsearch is stored as a *document*, so your first exercise is to store a document consisting of song lyrics. Start by creating an *index*, which is a container for all of your document types — analogous to a database in a relational database such as MySQL. Then insert a document into the index so that you can query the document's data.

Create an index

The general format for Elasticsearch commands is: `REST VERBHOST:9200/index/doc-type` — where `REST VERB` is `PUT`, `GET`, or `DELETE`. (Use the cURL `-X` verb prefix to specify the HTTP method explicitly.)

To create an index, run this command in your shell:

```
curl -XPUT "http://localhost:9200/music/"
```

Schema-optional

Although Elasticsearch is schemaless, under the hood it uses Lucene, which uses schemas. But Elasticsearch hides this complexity from you. In practice, you can treat Elasticsearch doc types as simple subindexes, or table names. But you can specify a schema if you want, so you could consider it a schema-optional data store.

Insert a document

To create a type under the `/music` index, insert a document. In this first example, your document consists of data — including a line of lyrics — about "Deck the Halls," a traditional Christmas tune first written down by Welsh poet John Ceirog Hughes in 1885.

To insert a document for "Deck the Halls" into the index, run this command (type this and the tutorial's other cURL commands as a single line):

```
curl -XPUT "http://localhost:9200/music/songs/1" -d '
{ "name": "Deck the Halls", "year": 1885, "lyrics": "Fa la la la la" }'
```

The preceding command uses the `PUT` verb to add a document to the `/songs` document type and gives the document the ID of 1. The URL path indicates *index/doctype/ID*.

View a document

To see the document, use a simple `GET` command:

```
curl -XGET "http://localhost:9200/music/songs/1"
```

Elasticsearch responds with the JSON content that you `PUT` in the index previously:

```
{"_index":"music","_type":"songs","_id":"1","_version":1,"found":true,"_source":
{ "name": "Deck the Halls", "year": 1885, "lyrics": "Fa la la la la" }}
```

Update a document

What if you realize that the date is wrong, and you want to change it to 1886? Update the document by running:

```
curl -XPUT "http://localhost:9200/music/lyrics/1" -d '{ "name":
"Deck the Halls", "year": 1886, "lyrics": "Fa la la la la" }'
```

Because this command uses the same unique ID of 1, the document is updated.

Delete a document (but not yet)

Don't delete the document yet, but know that to delete it:

```
curl -XDELETE "http://localhost:9200/music/lyrics/1"
```

Insert a document from a file

Here's another trick. You can insert a document from the command line by using the contents of a file. Try this method to add a document for another traditional song, "Ballad of Casey Jones." Copy Listing 1 into a file named caseyjones.json; alternatively, use the caseyjones.json file from the sample code package (see Download). Put the file anywhere that's convenient for running the cURL command against it. (In the code download, the file is in the root directory.)

Listing 1. JSON doc of "Ballad of Casey Jones"

```
{
  "artist": "Wallace Saunders",
  "year": 1909,
  "styles": ["traditional"],
  "album": "Unknown",
  "name": "Ballad of Casey Jones",
  "lyrics": "Come all you rounders if you want to hear
The story of a brave engineer
Casey Jones was the rounder's name....
Come all you rounders if you want to hear
The story of a brave engineer
Casey Jones was the rounder's name
On the six-eight wheeler, boys, he won his fame
The caller called Casey at half past four
He kissed his wife at the station door
He mounted to the cabin with the orders in his hand
And he took his farewell trip to that promis'd land

Chorus:
Casey Jones--mounted to his cabin
Casey Jones--with his orders in his hand
```

```
Casey Jones--mounted to his cabin
And he took his... land"
}
```

`PUT` this document in your `music` index by running:

```
$ curl -XPUT "http://localhost:9200/music/lyrics/2" -d @caseyjones.json
```

While you're at it, save contents of Listing 2 (for "Walking Boss," another folk song) into a walking.json file.

Listing 2. "Walking Boss" JSON

```
{
  "artist": "Clarence Ashley",
  "year": 1920
  "name": "Walking Boss",
  "styles": ["folk","protest"],
  "album": "Traditional",
  "lyrics": "Walkin' boss
Walkin' boss
Walkin' boss
I don't belong to you

I belong
I belong
I belong
To that steel driving crew

Well you work one day
Work one day
Work one day
Then go lay around the shanty two"
}
```

Push this document into the index:

```
$ curl -XPUT "http://localhost:9200/music/lyrics/3" -d @walking.json
```

Search the REST API

It's time to run a basic query that does more than the simple `GET` that you ran to find the "Get the Halls" document. The document URL has a built-in `_search` endpoint for this purpose. To find all songs with the word *you* in the lyrics:

```
curl -XGET "http://localhost:9200/music/lyrics/_search?q=lyrics:'you'"
```

The `q` parameter denotes a query.

The response is:

```
{"took":107,"timed_out":false,"_shards":{"total":5,"successful":5,"failed":0},"hits":{"total":2,"max
_score":0.15625,"hits":[{"_index":"music","_type":"songs","_id":"2","_
score":0.15625,"_source":{"artist": "Wallace Saunders","year": 1909,"styles":
["traditional"],"album": "Unknown","name": "Ballad of Casey Jones","lyrics": "Come all you rounders
if you want to hear The story of a brave engineer Casey Jones was the rounder's name.... Come all
you rounders if you want to hear The story of a brave engineer Casey Jones was the rounder's name
On the six-eight wheeler, boys, he won his fame The caller called Casey at half past four He kissed
his wife at the station door He mounted to the cabin with the orders in his hand And he took his
farewell trip to that promis'd land Chorus: Casey Jones--mounted to his cabin Casey Jones--with his
orders in his hand Casey Jones--mounted to his cabin And he took his... land"
}},{"_index":"music","_type":"songs","_id":"3","_score":0.06780553,"_source":{"artist": "Clarence
Ashley","year": 1920,"name": "Walking Boss","styles": ["folk","protest"],"album":
"Traditional","lyrics": "Walkin' boss Walkin' boss Walkin' boss I don't belong to you I belong I
belong I belong To that steel driving crew Well you work one day Work one day Work one day Then go
lay around the shanty two"}}]}}
```


Use other comparators

Various other comparators are also available. For example, to find all songs written before 1900:

```
curl -XGET "http://localhost:9200/music/lyrics/_search?q=year:<1900
```

This query returns the full "Casey Jones" and "Walking Boss" documents.

Restrict fields

To limit the fields that you see in the results add the `fields` parameter to your query:

```
curl -XGET "http://localhost:9200/music/lyrics/_search?q=year:>1900&fields=year"
```

Examine the search return objects

Listing 3 shows the data that Elasticsearch returns from the preceding query.

Listing 3. Query results

```
{
    "took": 6,
    "timed_out": false,
    "_shards": {
        "total": 5,
        "successful": 5,
        "failed": 0
    },
    "hits": {
        "total": 2,
        "max_score": 1.0,
        "hits": [{
            "_index": "music",
            "_type": "lyrics",
            "_id": "1",
            "_score": 1.0,
            "fields": {
                "year": [1920]
            }
        }, {
            "_index": "music",
            "_type": "lyrics",
            "_id": "3",
            "_score": 1.0,
            "fields": {
                "year": [1909]
```

```
            }
        }]
    }
}
```

In the results, Elasticsearch gives you several JSON objects. The first object contains metadata about the request: See how many milliseconds the requests took (`took`) and if it timed out (`timed_out`). The `_shards` field has to do with the fact that Elasticsearch is a clustered service. Even on this single-node local deployment, Elasticsearch is logically clustered into shards.

Continuing with the search results in Listing 3, observe that the `hits` object contains:

- The `total` field, which tells you how many results were obtained
- `max_score`, which comes into play for full-text search
- The actual results

The actual results consist of the `fields` property because you added the `fields` parameter to the query. Otherwise, the results would consist of `source` and contain the matching documents in their entirety. The `_index`, `_type`, and `_id` are self-explanatory; `_score` refers to the full-text search hit strength. These four fields are always returned in the results.

Use the JSON query DSL

Query-string based search gets complicated fast. For more-advanced querying, Elasticsearch offers a whole JSON-based domain-specific language (DSL). For example, to search for every song for which the `album` value is `traditional`, create a query.json file that contains:

```
{
    "query" : {
        "match" : {
            "album" : "Traditional"
        }
    }
}
```

Then run:

```
curl -XGET "http://localhost:9200/music/lyrics/_search" -d @query.json
```

Using Elasticsearch from Java code

> *“ The full power of Elasticsearch comes from using it via a language API. ”*

The full power of Elasticsearch comes from using it via a language API. Now I'll show you the Java API, and you'll do some searching from an application. The application uses the Spark microframework, so setting up the app is quick.

Sample app

Create a directory for a new project, and then run (type the command on a single line):

```
mvn archetype:generate -DgroupId=com.dw -DartifactId=es-demo
-DarchetypeArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

To generate a project for you to use in Eclipse, `cd` into the project directory that Maven created and run `mvn eclipse:eclipse`.

In Eclipse, select **File** > **Import** > **Existing Project into Workspace**. Navigate to the folder where you used Maven, select the project, and click `Finish`.

In Eclipse you can see a basic Java project layout, including the pom.xml file in the root, and a com.dw.App.java main class file. Add the dependencies that you need into the pom.xml file. Listing 4 shows the complete pom.xml file.

Listing 4. Complete pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.dw</groupId>
  <artifactId>es-demo</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>es-demo</name>
  <url>http://maven.apache.org</url>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <compilerVersion>1.8</compilerVersion>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <dependency>
    <groupId>com.sparkjava</groupId>
    <artifactId>spark-core</artifactId>
    <version>2.3</version>
</dependency>
<dependency>
    <groupId>com.sparkjava</groupId>
    <artifactId>spark-template-freemarker</artifactId>
    <version>2.3</version>
</dependency>
<dependency>
    <groupId>org.elasticsearch</groupId>
    <artifactId>elasticsearch</artifactId>
    <version>2.1.1</version>
</dependency>
  </dependencies>
</project>
```

The dependencies in Listing 4 get the Spark framework core, Spark Freemarker templating support, and Elasticsearch. Also notice that I set the `<source>` version to Java 8, which Spark requires (because it makes heavy use of lambdas).

I don't know about you, but I've built many RESTful apps lately, so for a change of pace you'll give the app a more traditional submit-and-load UI.

In Eclipse, right-click the project in the navigator, and select **Configure** > **Convert to Maven Project** so that Eclipse can resolve the Maven dependencies. Go to the project, right-click, and select **Maven** > **Update Project**.

Java client config

The Java client for Elasticsearch is powerful; it can spin up an embedded instance and run administrative tasks if necessary. But here I focus on running application tasks against the node that you already have running.

When you run a Java app with Elasticsearch, two modes of operation are available. The application can take either a more active or a more passive role in the Elasticsearch cluster. In the more-active case, called Node Client, the application instance receives requests from the cluster and determines which node should handle the request, as a normal node does. (The app can even host indexes and serve requests.) The other mode, known as a Transport Client, simply forwards any requests to another Elasticsearch node, which determines the final destination.

Obtaining the Transport Client

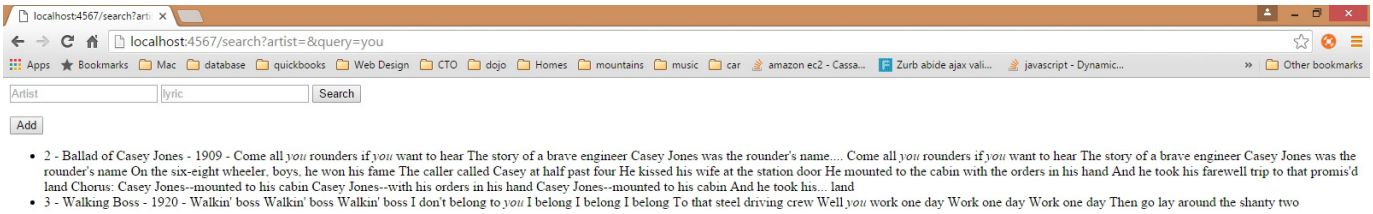
For the demo app, opt (via the initialization that occurs in App.java) for the Transport Client and keep the Elasticsearch processing to a minimum:

```
Client client = TransportClient.builder().build()
   .addTransportAddress(new InetSocketTransportAddress(InetAddress.getByName("localhost"), 9300));
```

The builder can accept multiple addresses if you're connecting to a Elasticsearch cluster. (In this case you have only the single localhost node.) Connect to port 9300, instead of 9200 as you did from cURL for the REST API. The Java client uses this special port, and using 9200 won't work. (Other Elasticsearch clients — the Python client for one — do use 9200 for accessing the REST API.)

Create the client when the server starts and use it throughout the request processing. Spark renders the page with a Java implementation of the Mustache templating engine, and Spark defines the request endpoints — but I won't comment much on these simple use cases.

The app's index page showcases the Java client's capabilities:



The UI:

- Renders the list of existing songs
- Offers a button to add a song
- Makes it possible to search by artist and lyric
- Returns results with matches highlighted

Searching and handling results

In Listing 5, the root URL of `/` is mapped to the index.mustache page.

Listing 5. Basic search

```
Spark.get("/", (request, response) -> {
        SearchResponse searchResponse =
            client.prepareSearch("music").setTypes("lyrics").execute().actionGet();
        SearchHit[] hits = searchResponse.getHits().getHits();

            Map<String, Object> attributes = new HashMap<>();
            attributes.put("songs", hits);

            return new ModelAndView(attributes, "index.mustache");
        }, new MustacheTemplateEngine());
```

The interesting bits in Listing 5 begin with:

```
SearchResponse searchResponse = client.prepareSearch("music").setTypes("lyrics").execute().actionGet();
```

This one line shows a simple use of the search API. Use the `prepareSearch` method to specify an index — in this case, `music`— and then execute the query. The query essentially says, "Give me all of the records in the `music` index." Also, set the document type to `lyrics`, although that's not necessary in this simple case because the index contains only one document type. In a larger app, that setting would be called for. This API call is analogous to the `curl -XGET "http://localhost:9200/music/lyrics/_search"` call that you saw earlier.

The `SearchResponse` object contains interesting functionality — hit counts and scores, for example — but right now, you want only an array of the results, which you obtain with `searchResponse.getHits().getHits();`.

Finally, add the array of results to the view context and let Mustache render it. The Mustache template is shown below:

Listing 6. index.mustache

```
<html>
<body>
<form name="" action="/search">
  <input type="text" name="artist" placeholder="Artist"></input>
  <input type="text" name="query" placeholder="lyric"></input>
  <button type="submit">Search</button>
</form>
<button onclick="window.location='/add'">Add</button>
<ul>
{{#songs}}
  <li>{{id}} - {{getSource.name}} - {{getSource.year}}
    {{#getHighlightFields}} -
      {{#lyrics.getFragments}}
        {{#.}}{{{.}}}{{/.}}
```

```
      {{/lyrics.getFragments}}
    {{/getHighlightFields}}
  </li>
{{/songs}}
</ul>

</body>
</html>
```

Advanced querying and match highlighting

To support more-advanced querying and match highlighting, use `/search` as shown here:

Listing 7. Searching and highlighting

```
Spark.get("/search", (request, response) -> {
        SearchRequestBuilder srb = client.prepareSearch("music").setTypes("lyrics");

        String lyricParam = request.queryParams("query");
        QueryBuilder lyricQuery = null;
        if (lyricParam != null && lyricParam.trim().length() > 0){
            lyricQuery = QueryBuilders.matchQuery("lyrics", lyricParam);
        }
        String artistParam = request.queryParams("artist");
        QueryBuilder artistQuery = null;
        if (artistParam != null && artistParam.trim().length() > 0){
          artistQuery = QueryBuilders.matchQuery("artist", artistParam);
        }

        if (lyricQuery != null && artistQuery == null){
          srb.setQuery(lyricQuery).addHighlightedField("lyrics", 0, 0);
        } else if (lyricQuery == null && artistQuery != null){
          srb.setQuery(artistQuery);
        } else if (lyricQuery != null && artistQuery != null){
          srb.setQuery(QueryBuilders.andQuery(artistQuery,
              lyricQuery)).addHighlightedField("lyrics", 0, 0);
        }

        SearchResponse searchResponse = srb.execute().actionGet();

SearchHit[] hits = searchResponse.getHits().getHits();

    Map<String, Object> attributes = new HashMap<>();
    attributes.put("songs", hits);

    return new ModelAndView(attributes, "index.mustache");
}, new MustacheTemplateEngine());
```

The first interesting API use to note in Listing 7 is `QueryBuilders.matchQuery("lyrics", lyricParam);`. This is where you set the query for the `lyrics` field. Also of note is `QueryBuilders.andQuery(artistQuery, lyricQuery)`, which is how to join the `artist` and `lyrics` parts of the query together in an AND query.

The `.addHighlightedField("lyrics", 0, 0);` call tells Elasticsearch to generate a search-hit highlight result on the `lyrics` field. The second and third parameters specify unlimited size fragments and an unlimited number of fragments, in that order.

When the search results are rendered, place the highlight results into the HTML. Elasticsearch is kind enough to generate valid HTML that uses `<em>` tags to highlight where the matching strings occur.

Inserting documents

Let's take a look at inserting into the index programmatically. Listing 8 shows the add handling.

Listing 8. Inserting into index

```
Spark.post("/save", (request, response) -> {
      StringBuilder json = new StringBuilder("{");
      json.append("\"name\":\""+request.raw().getParameter("name")+"\",");
      json.append("\"artist\":\""+request.raw().getParameter("artist")+"\",");
      json.append("\"year\":"+request.raw().getParameter("year")+",");
      json.append("\"album\":\""+request.raw().getParameter("album")+"\",");
      json.append("\"lyrics\":\""+request.raw().getParameter("lyrics")+"\"}");

      IndexRequest indexRequest = new IndexRequest("music", "lyrics",
          UUID.randomUUID().toString());
      indexRequest.source(json.toString());
      IndexResponse esResponse = client.index(indexRequest).actionGet();

      Map<String, Object> attributes = new HashMap<>();
      return new ModelAndView(attributes, "index.mustache");
    }, new MustacheTemplateEngine());
```

Here you create a JSON string by directly generating it with a `StringBuilder`. In a production app, use a library like Boon or Jackson.

The part that does the Elasticsearch work is:

```
IndexRequest indexRequest = new IndexRequest("music", "lyrics", UUID.randomUUID().toString());
```

In this case, use a UUID to generate an ID.

Conclusion

You're on the fast track to using Elasticsearch from a command shell and in a Java app. You're familiar now with indexing, querying, highlighting, and multifield search. Elasticsearch gives you an impressive amount of capability in a relatively simple-to-use package. Elasticsearch as a project has some interesting outgrowths that might also interest you. In particular, the so-called ELK stack — Elasticsearch, Logstash (for logging management), and Kibana (for reporting/visualization) — is gaining traction.

Related topics Elasticsearch home page Elasticsearch on GitHub Elasticsearch Java Centralize logs for IBM® Bluemix™ apps using the ELK Stack Jump into Java microframeworks

Downloads

Description	Name	Size
Sample code	es-demo.zip	15KB

About the author

Matt C. Tyson

Whether he's hiking the Himalayan mountains of Tibet or exploring the latest software technologies, Matt Tyson sees it as a spiritual adventure. Matt has done full-stack development for more than a decade, and he is the author of several articles, including "Jump into Java microframeworks" at JavaWorld.