**Sample Search Service Using SpringBoot and ElasticSearch**

**Problem Statement and Use-Cases:**

In this post , we shall be learning how to create a Sample Search Service Using SpringBoot and ElasticSearch. We shall be learning how to Ingest Bulky data efficiently into an ElasticSearch Index , how to define analyzers and custom mappings for an ElasticSearch Index and how to use Java’s High Level Rest Client for all communications between SpringBoot and ElasticSearch server. We shall also learn about BooleanQueries in ElasticSearch and how to use Fuzzy queries to optimize the search and get better results.

**Why ElasticSearch over Solr?**

* ElasticSearch is more lightweight and easier to configure than SOLR
* Our dataset contain JSON documents which are easily supported and stored in ElasticSearch indexes.

**Introduction:**

For our sample POC, we shall be building a Simple Search Service by ingesting an employee data into an Elasticsearch index and then sending various types of queries to it and see how efficiently the results match our query string.

**Pre-Requisites:**

1. IntelliJ IDEA configured and Installed.
2. JDK 1.8.
3. ElasticSearch Installed And Running.
4. Basic Knowledge of SpringBoot, Springdata JPA , ElasticSerach.
5. Postman Installed.

**Dataset:**

The dataset that we shall be using for our service is downloaded from :

https://a511e938-a640-4868-939e-6eef06127ca1.mock.pstmn.io/handsets/list

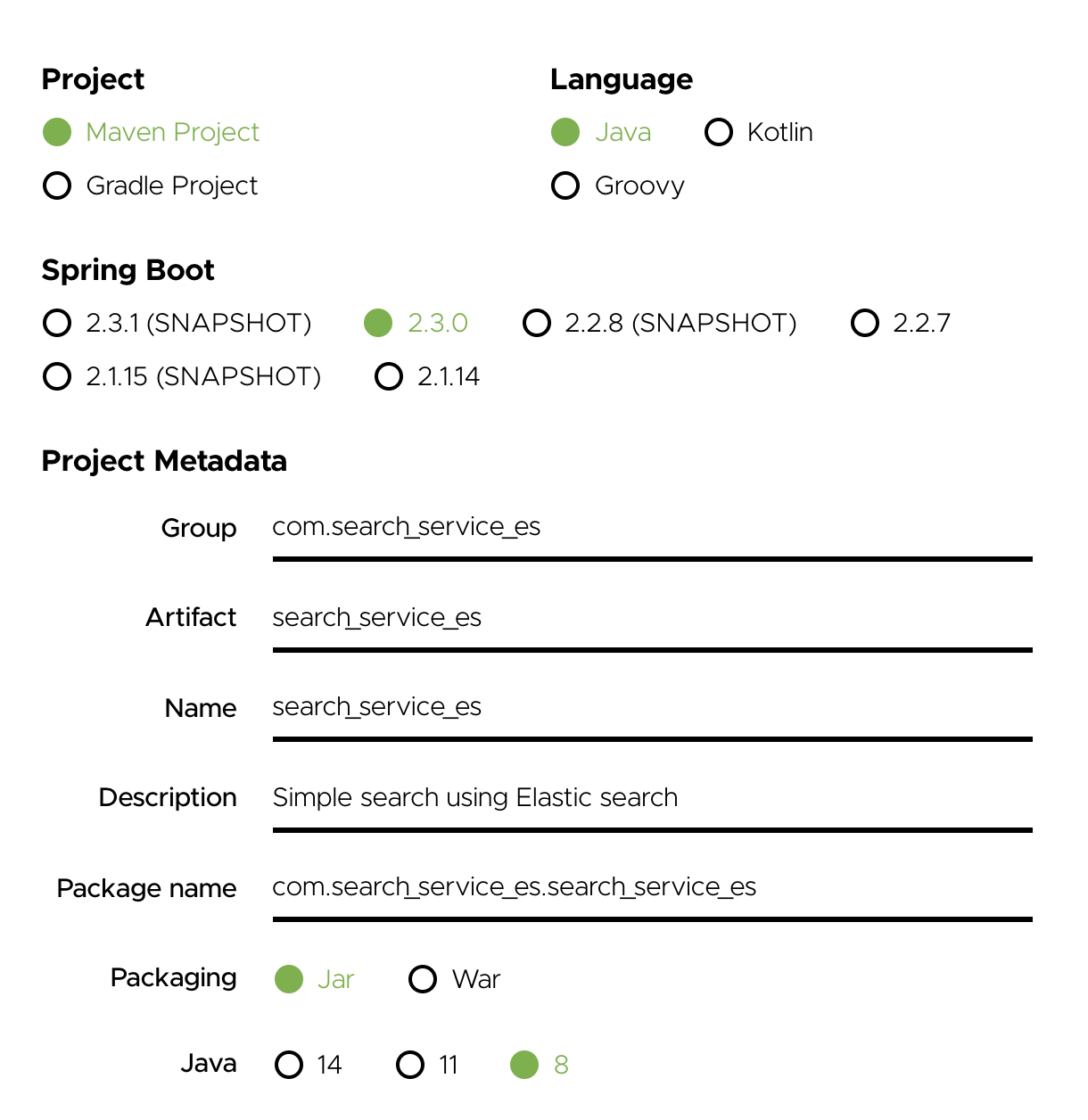
**Preprocessing:**

Before starting off with the configurations, we need to carry out a certain set of preprocessings on the dataset we just downloaded.

Remove all blank lines from the file.

Creating a Sample SpringBoot Project:

Now head to <https://start.spring.io/> and Generate a new SpringBoot project. Here is a snapshot for reference :



**Configurations:**

Before we start writing the Controllers(APIs) it is important to correctly import all the maven dependencies and Configure ElasticSearch with SpringBoot. To correctly do so, carry out the following steps :

Step 1 : Import the project ‘search-service-es’ in IntelliJ IDEA.

Step 2: Head to the ‘pom.xml’ file, clear all the existing contents of the file and add the following dependencies :

<?xml version="1.0" encoding="UTF-8"?>  
<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 https://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
 <parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>2.3.0.RELEASE</version>  
 <relativePath/> <!-- lookup parent from repository -->  
 </parent>  
 <groupId>com.search\_service\_es</groupId>  
 <artifactId>search\_service\_es</artifactId>  
 <version>0.0.1-SNAPSHOT</version>  
 <name>search\_service\_es</name>  
 <description>Simple search using Elastic search</description>  
  
 <properties>  
 <java.version>1.8</java.version>  
 </properties>  
  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-test</artifactId>  
 <scope>test</scope>  
 </dependency>  
 <!-- Elasticsearch Dependencies -->  
 <dependency>  
 <groupId>org.elasticsearch</groupId>  
 <artifactId>elasticsearch</artifactId>  
 <version>7.6.0</version>  
 </dependency>  
 <dependency>  
 <groupId>org.elasticsearch.client</groupId>  
 <artifactId>elasticsearch-rest-high-level-client</artifactId>  
 <version>7.6.0</version>  
 </dependency>  
 <dependency>  
 <groupId>org.elasticsearch.client</groupId>  
 <artifactId>elasticsearch-rest-client</artifactId>  
 <version>7.6.0</version>  
 </dependency>  
 <dependency>  
 <groupId>org.elasticsearch.client</groupId>  
 <artifactId>elasticsearch-rest-client-sniffer</artifactId>  
 <version>7.6.0</version>  
 </dependency>  
 <dependency>  
 <groupId>org.apache.logging.log4j</groupId>  
 <artifactId>log4j-api</artifactId>  
  
 </dependency>  
 <dependency>  
 <groupId>org.apache.logging.log4j</groupId>  
 <artifactId>log4j-core</artifactId>  
 </dependency>  
  
 <dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 </dependency>  
 </dependencies>  
 <build>  
 <plugins>  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 </plugin>  
 </plugins>  
 </build>  
  
</project>

Note : It is a wise choice to Re-import all Maven dependencies before continuing.

Step 3 : Add the following lines in the application.properties file under resources package:

spring.data.elasticsearch.cluster-name=elasticsearch  
spring.data.elasticsearch.cluster-nodes=elasticsearch  
server.port=8181

The application.properties file needs to have properties defined for Local ElasticSearch as well as for the app. By default SpringBoot Uses Apache Tomcat Server running on port 8080 but just for reference we have explicitly defined it to run on port 8181.

**Java Rest Client :**

Java Rest Client is the official ElasticSearch client that is used by Java to connect and communicate to ES. We would be using Java High-Level Rest Client for the same purpose. JHLC is used to expose API-specific method that enables transfer of objects between source and destination as request and response. In simpler words , we can use JHLC to connect to ElasticSearch server and then perform various operations on ElasticSearch Indexes like Insert , Update , Ingestion etc. As you would’ve probably seen , we have included the JHLC dependencies in pom.xml file.

Now create a Java class called “ESConfig” where we shall be telling JHLC which ports to listen to for all ES communications. By default ES runs on Port 9020. Add the following lines inside ESConfig class to do the same:

private static final Logger *LOG* = LoggerFactory.*getLogger*(EsConfig.class);  
@Value("${spring.data.elasticsearch.cluster-nodes}")  
private String clusterNodes;  
@Value("${spring.data.elasticsearch.cluster-name}")  
private String clusterName;  
private RestHighLevelClient restHighLevelClient;  
  
@Bean  
public RestHighLevelClient createInstance() {  
 return buildClient();  
}  
  
private RestHighLevelClient buildClient() {  
 try {  
 restHighLevelClient = new RestHighLevelClient(  
 RestClient.*builder*(  
 new HttpHost("localhost", 9200, "http"),  
 new HttpHost("localhost", 9201, "http")));  
 } catch (Exception e) {  
 *LOG*.error(e.getMessage());  
 throw e;  
 }  
 return restHighLevelClient;  
}

Note : Do import the required libraries.

**Mappings:**

ES stores data as “Json Documents”. Schema in ElasticSearch can be thought of as a “mapping” that describes the various fields in the Json documents along with their data types. So we use mappings in order to define what are the various fields in a JSON document along with their datatypes etc. Read more about mappings here : <https://www.elastic.co/guide/en/elasticsearch/reference/current/mapping.html>

**Analysers:**

In order to understand how ES uses the analysis phase in order to effectively match the required documents, let us first understand how ES indexes the document.

When Documents are inserted in the ES Index, Elastic Search by default uses something called the Standard Analyzer on all fields of the documents whose datatype is defined as “Text” during mappings(Unless custom analysis is explicitly defined). Analysis phase itself consists of 3 internal phases :

1. Character Filters : The character filter has the ability to perform addition , removal or replacement actions on the input text given to them. The most common use case of this is to remove html tags from input text. For example : “html\_strip” character filter shall convert "My <b> name-is </b> Satyam" to "My name-is Satyam"
2. Tokenizer : The transformed text from the character filter is given as an input to the tokenizer ,which would transform the text into series of tokens. For example , the standard tokenizer which is the default tokenizer used by ES provides grammar based tokenization and works well for most use cases. So it shall transform the text "My name-is Satyam" to [“My” , “name” , “is” , “Satyam”].
3. Token-Filter : The token filter takes individual token from the Tokenizer phase and can modify, add or remove them. For example, to facilitate a better search , we can take the individual tokens and add synonyms for that token , convert them to lower case, remove stop words etc. So a lower-case filter shall transform the tokens [“My” , “name” , “is” , “Satyam”] to [“my”,”name”,”is”,”satyam”]

After the Analysis Phase is over during index time , all these individually modified tokens are indexed in another ES index called the “Reverse Index” or “Inverted Index”, indicating which tokens match which documents.

This Inverted Index is used to identify the most relevant documents when a query string is sent.

When a query string is sent during search time, the same analyzer that was used during index time is used on the query string to transform it into tokens and a look-up is performed in the Reverse Index to find out the matching documents in the main index. Of course we can define custom mappings and different analyzers during index-time and search-time Analysis.

For our dataset, we make it searchable on the following fields:

Brand

Phone

Picture

announceDate

priceEur

sim

resolution

audioJack

gps

battery

The custom mappings and custom analyzers that I have used for the same is as follows :

{

"settings": {

"index": {

"number\_of\_shards": "3",

"analysis": {

"filter": {

"filter\_stop": {

"type": "stop"

}

},

"analyzer": {

"my\_analyzer": {

"filter": [

"lowercase"

],

"tokenizer": "my\_tokenizer\_1"

},

"my\_analyzer\_1":{

"filter" : [

"lowercase",

"filter\_stop"

],

"tokenizer" : "my\_tokenizer"

},

"my\_analyzer\_2" : {

"tokenizer" : "lowercase"

}

},

"tokenizer": {

"my\_tokenizer": {

"token\_chars": [

"letter",

"digit"

],

"min\_gram": "2",

"type": "edge\_ngram",

"max\_gram": "20"

},

"my\_tokenizer\_1": {

"type" : "edge\_ngram",

"min\_gram" : 2,

"max\_gram" : 20,

"token\_chars" : [

"letter"

]

}

}

},

"number\_of\_replicas": "2"

}

},

"mappings": {

"dynamic": "true",

"properties": {

"brand": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

},

"Phone": {

"type": " text "

},

"AnnounceDate": {

"type": " text "

},

"sim": {

"type": "keyword"

},

"FirstName": {

"type": "text",

"analyzer" : "my\_analyzer",

"search\_analyzer" : "my\_analyzer\_2"

},

"id": {

"type": "long"

}

}

}

}

Let us call our Index as “device\_index”

Go ahead and copy this entire definition in the request body of Postman and send a PUT request to the ES server at the URL : <http://127.0.0.1:9200/device_index>

An Index with the required mappings should be created at the server.

Note : Download a chrome extension called “elastic-head” to get a better view of the indexes on the server.

**Writing the Controller and Services:**

* Create a Device Class with the following variables(Same as the JSON document fields):

Note : Import all required libraries and add required getters and setters.

* Using Jackson ObjectMapper Class :

ObjectMapper class is the simplest way to parse JSON with Jackson. The Jackson ObjectMapper can parse JSON from a string, stream or file, and create a Java object representing the parsed JSON. Parsing JSON into Java objects is also referred to as to deserialize Java objects from JSON. The Jackson ObjectMapper can also create JSON from Java objects. Generating JSON from Java objects is also referred to as to serialize Java objects into JSON.

So go ahead and read the cleaned dataset line by line in Java and deserialize Java objects from JSON using objectmapper’s readValue method and append them to a list of Device Objects

* Using XContentBuilder:

XContentBuilder is a built-in ElasticSearch Helper that is used to generate JSON documents. Using XContentBuilder’s builder object , we can build the JSON document that is to be indexed, using the following command:

XContentBuilder builder = jsonBuilder()

.startObject()

.field(<field\_name\_1>, <field\_value\_1>)

.field<field\_name\_2>, <field\_value\_2>)

.field(<field\_name\_3>, <field\_value\_3>)

.endObject()

1. Ingestion API using JHLC’s Bulk APIs

It is quite evident that right now, we have a pretty large dataset. JHLC comes to the rescue again by providing a class called the BulkProcessor that can be easily imported and utilized. The BulkProcessor simplifies the usage of the Bulk API by providing a utility class that allows index(insert) and update operations to be transparently executed as they are added to the processor.

Let us understand the important aspects to Ingest Data into ES Index:

Step 1) Using Jackson’s Objectmapper’s convertValue method to convert the employee object into a Java Map ( Key -> Value pairs) .

Step 2) Using the method map.keySet() to get Keys from the Map. Note that these keys are also our field names in the employee\_index mappings we defined earlier.

Step 3) Using the builder.field() of XContentBuilder command as explained above.

Step 4 and 5) As we can see we have used the upsert method of the Updaterequest class. The main purpose of using upsets was that if the document id already exists, the builder that we just created(Step 3) simply updates the document at that id , and if the document id does not exist, then a new document is indexed in the specified index.

Step 6) Simply add the updateRequest to your bulkprocessor. This shall index the documents in batches in ES index.

1. Search API using JHLC’s Search APIs

Now that we have successfully ingested some data into ES index , let us send out some search queries to ElasticSearch and see what results we get. As defined earlier . we shall be searching the query string on the 5 searchable fields. So let us briefly understand what Boolean queries are in ElasticSearch.

Boolean queries are compound queries that consists of Boolean combination of other queries. Boolean queries are of the following types:

must: The clause (query) must appear in matching documents and will contribute to the relevance score.

filter: The clause (query) must appear in matching documents. However unlike ‘must’ the score of the query will be ignored.

should: The clause (query) should appear in the matching document.

must\_not: The clause (query) must not appear in the matching documents.

Lets understand Boolean Queries better with an example:

1. For an Expression of format : (FirstName= “Satyam” AND LastName = “Lal”) the boolquery looks like:

{

"query" : {

"bool" : {

"must": [{

"match": {

"brand": "iPhone"

}

}, {

"match": {

"brand": "Lal"

}

}]

}

}

}

1. For an Expression of format : (FirstName= “Satyam” OR LastName = “Lal”) the boolquery looks like:

{

"query" : {

"bool" : {

"should": [{

"match": {

" brand ": "iPhone"

}

}, {

"match": {

" brand ": " iPhone "

}

}]

}

}}

For our Use case we simply need the following expression format :

(brand=’String’ OR type =’String’ OR sim=’String’ OR picture=’String’ OR resolution=’String’).

In simpler words, the queryString to be searched shall be searched in all of the searchable fields, and based on how closely they match, an overall relevance score shall be calculated for the documents. In the end, a list of documents with the highest relevance scores shall be returned as the response. Lets try to define the Boolquery for search using Java:

As in the above code snippet, we have defined an object of the BoolQueryBuilder class for defining our ES Boolean query. We try to match the ‘search\_string’ with all the searchable fields in our use case and thus build our query. There are some other methods invoked here for better search results, let us try to understand them one by one.

1. termQuery vs matchQuery : TermQuery is used while querying fields that were defined as keywords during mappings. TermQuery deals with exact matches. This means that the query returns true if and only if there is an exact match found in the seach\_string. MatchQuery is used for full-text searches and supports fuzzy searches on text fields.
2. Fuziness(<edit\_dist>) method : When querying text or keyword fields, fuzziness is interpreted as a Levenshtein Edit Distance — the number of one character changes that need to be made to one string to make it the same as another string.

An edit distance is the number of one-character changes needed to turn one term into another. These changes can include:

Changing a character (box → fox)

Removing a character (black → lack)

Inserting a character (sic → sick)

Transposing two adjacent characters (act → cat)

Adding fuzziness to certain fields allows ES to search for matches similar to the queryString rather than exact matches. So a searchString ‘BAX’ can also match documents whose ‘Interest’ field includes ‘BMX’ (edit-distance 1)

1. Boost(float) : Query Time boost allows us to manually decide which field has to be given more weightage/priority as compared to the others. This can be decided based on the use case. TermQueries have a boost of 1.0 by default. In our case, we have given boost randomly to the fields, but it can vary based on use-cases.

To send a Search request using JHLC’s SearchRequest Class , using the following commands:

SearchRequest searchRequest = new SearchRequest();  
searchRequest.indices("employee\_index");  
SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilder();  
searchSourceBuilder.query(boolQueryBuilder);  
searchRequest.source(searchSourceBuilder);

Get the Search Response using JHLC’s SearchResponse Class, and return a list of matching Employee Documents to the controller:

SearchResponse searchResponse = client.search(searchRequest, RequestOptions.*DEFAULT*);  
SearchHit[] searchHit = searchResponse.getHits().getHits();  
List<Employee> employeeList = new ArrayList<>();  
for (SearchHit hit : searchHit) {  
 employeeList.add(objectMapper.convertValue(hit.getSourceAsMap(), Employee.class));  
}  
return employeeList;

And that’s it, we can build different ES Queries using the QueryBuilders and use the same format described above in order to get the most relevant results. For example, to create a match\_all query we can define our BoolQueryBuilder as:

BoolQueryBuilder boolQueryBuilder = QueryBuilders.*boolQuery*();  
boolQueryBuilder.must(QueryBuilders.*matchAllQuery*());

And then send this boolQueryBuilder object as an argument to the searchSourceBuilder object, and repeat the above mentioned steps to get a List of all documents in the ES Index.