Java High-Level REST Client — Elasticsearch 7

Suman Das Follow
Aug 3, 2019 · 5 min read

Elasticsearch is an open-source, highly scalable full-text search and analytics engine. It is a distributed NoSQL database and uses documents rather than schema or tables. You can store data in JSON document format and easily store, search, and analyze a large amount of data in real-time. It can be used to search all kinds of documents.

Java REST client is the official client for Elasticsearch. It comes in 2 flavors:

1. ***Java Low-Level REST client*** — It allows communicating with an Elasticsearch cluster through HTTP and leaves requests marshalling & responses un-marshalling to users.
2. ***Java High-Level REST client*** — It is based on low-level client and exposes API specific methods, taking care of requests marshalling and responses un-marshalling.

The ***Java High-Level REST client*** works on top of the ***Java Low-Level REST client***. Its main goal is to expose API specific methods, that accept request objects as an argument and return response objects, so that request marshalling and response un-marshalling is handled by the client itself.

Each API can be called synchronously or asynchronously. The synchronous methods return a response object, while the asynchronous methods, whose names end with the `async` suffix, requires a listener argument that is notified (on the thread pool managed by the low-level client) once a response or an error is received.

We can use the ***Java High-Level REST client*** to perform various operations on **Elasticsearch**.

Implementation using Java High-Level REST client for Elasticsearch

We will use Spring Boot 1.5.9.RELEASE project with the following dependencies:

```
<dependency>
    <groupId>org.elasticsearch</groupId>
    <artifactId>elasticsearch</artifactId>
    <version>7.1.1</version>
</dependency>
<dependency>
    <groupId>org.elasticsearch.client</groupId>
    <artifactId>elasticsearch-rest-high-level-client</artifactId>
    <version>7.1.1</version>
</dependency>
```

Initialization

A `RestHighLevelClient` instance needs a REST low-level client builder to be built as follows:

```
@Bean
public RestHighLevelClient client(){
final CredentialsProvider credentialsProvider =new
BasicCredentialsProvider();

credentialsProvider.setCredentials(AuthScope.ANY,new
UsernamePasswordCredentials("username", "password"));

RestClientBuilder builder =RestClient.builder(new HttpHost(host,
port, "http")).setHttpClientConfigCallback(httpClientBuilder ->
httpClientBuilder.setDefaultCredentialsProvider(credentialsProvider))
;

 RestHighLevelClient client = new RestHighLevelClient(builder);
    return client;
}
```

Here, we can replace the host with the IP address on which Elasticsearch is running. And, **9200** is the port to send REST requests to that node. If the cluster is secured with Basic Authentication then we provide username and password for the same.

The high-level client will internally create the low-level client used to perform requests based on the provided builder. That low-level client maintains a pool of connections and starts some threads so you should close the high-level client when you are well and truly done with it and it will, in turn, close the internal low-level client to free those resources.

The Java High-Level REST Client supports the various APIs. Index, Update, Search, Get, Delete, Bulk are some of those APIs and there are many more.

In ElasticSearch everything starts with an index. During an indexing operation, Elasticsearch converts raw data such as log files or message files into internal documents and stores them in a basic data structure similar to a JSON object.

Let's see how we can create an index. An index is a collection of documents that have somewhat similar characteristics. For example, we can have an index for customer data, another index for a product catalog, and yet another index for order data. An index is identified by a name (that must be all lowercase) and this name is used to refer to the index when performing indexing, search, update, and delete operations against the documents in it. In a single cluster, we can define as many indexes as we want.

Let's see how we can create an Index using `RestHighLevelClient`

Create Index

The below code can be used to create an Index.

```
CreateIndexRequest request = new CreateIndexRequest("users");
request.settings(Settings.builder()
        .put("index.number_of_shards", 1)
        .put("index.number_of_replicas", 2)
);
Map<String, Object> message = new HashMap<>();
message.put("type", "text");

Map<String, Object> properties = new HashMap<>();
properties.put("userId", message);
properties.put("name", message);

Map<String, Object> mapping = new HashMap<>();
mapping.put("properties", properties);
request.mapping(mapping);
CreateIndexResponse indexResponse = client.indices().create(request,
RequestOptions.DEFAULT);
System.out.println("response id: "+indexResponse.index());
```

Some of the core components of creating an index have been explained below.

@CreateIndexRequest: creates an index with a given name. Here we are creating an index with name **users** which consists of one **Shard and two Replicas.** Each index created can have specific settings associated with it. The above index is created with mappings for its document types. Here we have specified document type as the **text** for both fields **userId** and **name**. When executing a `CreateIndexRequest` in the above manner, the client waits for the `CreateIndexResponse` to be returned before continuing with code execution. Synchronous calls may throw an `IOException` in case of either failing to parse the REST response in the high-level REST client, the request times out or similar cases where there is no response coming back from the server.

Executing a `CreateIndexRequest` can also be done in an asynchronous fashion so that the client can return directly. Users need to specify how the response or potential failures will be handled by passing the request and a listener to the asynchronous create-index method:

```
client.indices().createAsync(request, RequestOptions.DEFAULT,
listener);
```

Upsert a Document

The below code can be used to save a document in Elasticsearch.

```
IndexRequest request = new IndexRequest("users");
request.id(user.getUserId());
request.source(new ObjectMapper().writeValueAsString(user),
XContentType.JSON);
IndexResponse indexResponse = client.index(request,
RequestOptions.DEFAULT);
System.out.println("response id: "+indexResponse.getId());
return indexResponse.getResult().name();
```

Some of the core components of creating an index have been explained below.

@IndexRequest: specify the index name in which the document will be stored. Id specifies the document id. Source specifies the document which needs to be stored. The document is stored in JSON format. If the document exists then it will update the document else it will create the document. If the index doesn't exist then it will create the index with the name else will use the index with that name.

Delete a Document

Deleting a document just requires two lines of code. Create a DeleteRequest with index name and document id and then execute it via the REST client.

```
DeleteRequest request = new DeleteRequest("users",id);
DeleteResponse deleteResponse =
client.delete(request,RequestOptions.DEFAULT);
```

Search Documents

The SearchRequest is used for any operation that has to do with searching documents, aggregations, suggestions and also offers ways of requesting highlighting on the resulting documents.
First, we need to create a SearchRequest passing the index name as the argument.

```
SearchRequest searchRequest = new SearchRequest(index_name)
```

After that, we need to create SearchSourceBuilder, which contains the query which we want to execute.

```
QueryBuilder matchQueryBuilder = QueryBuilders.matchQuery("name",
field);
SearchSourceBuilder sourceBuilder = new SearchSourceBuilder();
sourceBuilder.query(matchQueryBuilder);

searchRequest.source(sourceBuilder);
```

Lastly, executing the SearchRequest through the REST client.

```
SearchResponse searchResponse =
client.search(searchRequest,RequestOptions.DEFAULT);
```

There are several other operations we can execute via a High-Level REST client.

For all other search options please refer **Search APIs**

If you would like to refer to the full code, do check https://github.com/sumanentc/elasticsearch.

References & Useful Readings

https://www.elastic.co/guide/en/elasticsearch/client/java-rest/current/java-rest-high.html

https://github.com/elastic/elasticsearch

Elasticsearch