Topics ˅   Issues ˅   Downloads ˅   Subscribe

Java magazine

FRAMEWORKS

Easy Searching with Elasticsearch

Using Elasticsearch's high- and low-level APIs to search synchronously and asynchronously
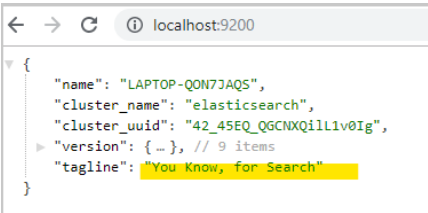
by Henry Naftulin

January 10, 2020

Elasticsearch is an open source search engine built on top of a full-text search library called Apache Lucene. Apache Lucene is a Java library that provides indexing and search technology, spell-checking, and advanced analysis/tokenization capabilities. Lucene dates back to 1999 as a SourceForge project and joined the Apache Software Foundation in 2001. It is the backbone for at least two popular search engines: Solr and Elasticsearch. Both of these search engines are powerful and have their own strengths and weaknesses. Although Solr has been around longer, and historically has better documentation, Elasticsearch is a better choice for applications that require not only text search but also time series search and aggregations. This article concentrates on Elasticsearch only. To learn more about comparing the Solr and Elasticsearch engines, refer to this article.

Learning about Elasticsearch is a large topic. It encompasses search-optimized document design, query and analysis, mappings, cluster management, data ingestion, and security. In this article, I introduce core concepts of Elasticsearch and then explore in depth how to use the Elasticsearch Java API to create, update, delete, and search a document in an index. I describe both the low-level API and the high-level API for performing these operations as well as how to execute these tasks synchronously and asynchronously. I will also discuss how to stream data into an Elasticsearch cluster, which is necessary if you are reading data from a stream, a queue, or another source that is too large to be loaded into memory.

Getting Started

First, download Elasticsearch. Then start it by navigating to the installation `bin` directory and running `elasticsearch.bat`. Once the Elasticsearch engine has started, you will see "started" in the log output. Then you can open `http://localhost:9200/` and you will receive a JSON response letting you know that your single-node cluster is up (see **Figure 1**).

Figure 1. JSON response showing an Elasticsearch cluster is running

Core Concepts

Now that you have Elasticsearch running, let's examine a few core concepts so you can familiarize yourself with the terminology. To make things easier, in **Table 1**, I compare each concept with a similar concept used in database technology. The comparisons are not entirely accurate, but they make learning the new terms a little easier.

Elasticsearch concept	Similar concept in SQL database	Description of Elasticsearch concept
Data type	Data type	A type, such as text, a keyword, an integer, a date, a range, a geographical point, and so on. Applicable to document fields.
Document	Database record	A JSON document. It is stored in Elasticsearch as a first-class citizen. Contains fields, and each field is of a particular data type.
Type		Deprecated. Previously was a convenient way to group similar documents together.
Index	Database table (previously was similar to a database, but now that types are gone, it is similar to a table)	Index that stores and manages documents of the same type.
Node	Node	Single running instance of Elasticsearch.
Cluster	Cluster	Elasticsearch instances that search the same indexes. Even a single node is a cluster in Elasticsearch.
Shard/primary shard		Part of an index that resides in one node of a cluster.
Replica/replica shard		Extra copy of a shard that is stored on a different node for high availability.

Table 1. Comparison of Elasticsearch and database concepts

In this article, I work a lot with catalog items. For my purposes, catalog items have an ID, description, price, and sales rank (a number representing how popular the item is). A catalog item will belong to a category and will be produced by a manufacturer. A category will have a name and a parent category, while a manufacturer has a name and an address.

Low-Level Synchronous CRUD API

Now let's explore the create, read, update, and delete (CRUD) API of the low-level client. The low-level client requires a minimal number of Elasticsearch dependencies, and it mirrors the REST endpoint API provided by Elasticsearch. As such, new releases of Elasticsearch should be backward compatible with the low-level client dependencies. The reason the client is called "low-level" is because you will need to do all the work of creating a JSON object request and also manually parse the response. In an environment where memory is limited, this might be the only solution available to you. The high-level client API is built on top of the low-level API, so it makes sense to start with the low-level API.

To get the low-level Elasticsearch libraries, all you need to do is to import the REST client as shown below for Maven. In my code, I also import libraries that will help serialize and deserialize JSON into the model classes, which are not shown here.

```
<dependency>
  <groupId>org.elasticsearch.client</groupId>
  <artifactId>elasticsearch-rest-client</artifactId>
  <version>7.0.0</version>
</dependency>
```

To build the REST client, I use the REST client builder and point it to the host (or hosts) that the client will communicate with. The client is thread-safe, so it could be used for the entire lifecycle of the application. To release the underlying HTTP client resources, the client needs to be closed when the application is done using it. As an example, I use try-with-resources to initialize the client and once initialized pass it to the `CrudMethodsSynchronous` constructor. I use the

`CrudMethodsSynchronous` class as the wrapper to call Elasticsearch's create/update/delete API:

```java
public static void main(String[] args) {
    try (RestClient client = RestClient.builder(
            new HttpHost("localhost", 9200, "http")).
                CrudMethodsSynchronous scm =
                new CrudMethodsSynchronous(
                    "catalog_item_low_level", client);
    }
}
```

To insert the document into an Elasticsearch index, I create a PUT request and ask the client to execute the request. The important part of the document creation is the HTTP method you use to create it: I chose PUT. You can use a POST method as well; in fact, POST is the preferred method for creating a record while PUT is the preferred method for updating a record. But in my case, because I run this sample program several times and sometimes I don't clean the index, PUT works better. The PUT method is used here as an *upsert* (that is, an insert or update).

The URI is important as well. Let's look at an example:

```
http://localhost:9200/catalog_item_low_level/_doc/1
```

The first part of it is the index that is similar to a database where documents are stored, which in this case is `catalog_item_low_level`. The second part is `_doc`, which indicates that you are dealing with a document. Before Elasticsearch 7, you would have specified the type here, for example, `catalogitem`. But types are no longer used. The last part is the ID of the document. Note that the index name needs to be in lowercase. In this example, I am creating documents one by one; later in this article, I will show how to create several items in bulk.

```java
public void createCatalogItem(List<CatalogItem> iter
    items.stream().forEach(e-> {

    Request request = new Request("PUT",
            String.format("/%s/_doc/%d",
                getIndex(), e.getId()));
    try {
        request.setJsonEntity(
            getObjectMapper().writeValueAsString(e)

        getRestClient().performRequest(request);
    } catch (IOException ex) {
        LOG.warn("Could not post {} to ES", e, ex);
    }
    });
}
```

Once the items are created, I would like to find one item via a full-text search—in this example, I'll look for a flashlight. This search will consider all the fields in the document, and it will return records in which any field has a flashlight as a token. One thing to note here is that Elasticsearch has the concept of both filters and searches. Filters are faster searches that are intended to return results but not rate their relevance, whereas searches return results and rate each result with a relevance score. (In this article, I look at searches only.)

```java
List<CatalogItem> items =
    scm.findCatalogItem("flashlight");
LOG.info("Found {} items: {}", items.size(), items)
```

To run a search with a low-level client, I need to issue a GET request that will run against my index with the following URI:

`/<indexname>/_search`. Because the low-level API uses the Elasticsearch REST interface, I need to construct the REST query object by hand, which in my case is

```
{ "query" : {"query_string" : { "query": "flashlight"
} } }
```
.

After sending the request to Elasticsearch, I will receive a result in a `Response` object. The result contains the return status as well as an entity that represents the JSON response. To get `CatalogItem` results, I need to navigate through the response structure.

```
public List<CatalogItem> findCatalogItem(String text
    Request request = new Request("GET",
            String.format("/%s/_search", getIndex(

    request.setJsonEntity(String.format(SEARCH, text
    try {
        Response response = client.performRequest(re
    if (response.getStatusLine().getStatusCode()==OK
            List<CatalogItem> catalogItems =
                parseResultsFromFullSearch(response

            return catalogItems;
        }
    } catch (IOException ex) {
        LOG.warn("Could not post {} to ES", text, e:
    }
    return Collections.emptyList();
}
```

I first find the documents my search returned, and then I convert the returned JSON documents into my model. As you can see, I have to heavily rely on Elasticsearch REST documentation to both create requests and parse responses. The easiest way to see how to form a request and test what Elasticsearch will return is to use the Advanced REST Client (ARC) plugin to Chrome or the Postman app, or install Kibana.

To change this search so it looks only at a particular field (for example, the category name of a catalog item), all you need to do is to change the previous query to

```
{ "query" : { "match" : { "category.category_name" :
"Home" } } }
```
. Then use the same process to submit the request and parse the results.

These two searches are different from retrieving an item by its ID. Here, I only need to issue a GET request to an index passing the ID, `/<indexname>/_doc/5`, for example, and parsing the return object is different because I am not getting an array of items that were found, but only the one item, if it exists. Here is the code that shows how to use the low-level API to search by ID. As with all low-level API calls, I need to parse the JSON response, skipping over the metadata and extracting `CatalogItem` information, as shown here:

```
public Optional<CatalogItem> getItemById(Integer id
  Request request = new Request("GET",
        String.format("/%s/_doc/%d", getIndex(), id
  try {
    Response response = client.performRequest(reque:
    if (response.getStatusLine().getStatusCode() ==
      String rBody =
          EntityUtils.toString(response.getEntity()
      LOG.debug("find by item id response: {}", rBoc
      int start = rBody.indexOf(_SOURCE);
      int end = rBody.indexOf("}}");
      String json = rBody.substring(
                  start + _SOURCE.length(), end+:
      LOG.debug(json);
      CatalogItem item =
          jsonMapper.readValue(json, CatalogItem.cla
      return Optional.of(item);
    }
```

```
    } catch (IOException ex) {
        LOG.warn("Could not post {} to ES", id, ex);
    }
    return Optional.empty();
}
```

Updating a document. There are a couple of ways to update a document: You can issue an update to an entire document or issue an update that modifies a particular field only. Because `CatalogItem` is a rather small document, I will update it fully:

```
public void updateCatalogItem(CatalogItem item) {
    Request request =
        new Request("POST",
                    String.format("/%s/_update/%d",
                    getIndex(), item.getId()));
    try {
        request.setJsonEntity("{ \"doc\" :" +
                    jsonMapper.writeValueAsString(item)+'
        Response response = client.performRequest(reque
        LOG.debug("update response: {}", response);
    } catch (IOException ex) {
        LOG.warn("Could not post {} to ES", item, ex);
    }
}
```

To update only a particular field, you issue a POST request to the same URI, but instead of sending an object in the request, you send just the field that you need to update, as shown next:

```
public void updateDescription(Integer id, String des
    Request request = new Request("POST",
            String.format("/%s/_update/%d", index,
    try {

        request.setJsonEntity(
            String.format(
                "{ \"doc\" : { \"description\" : \"%s\"
            desc));

        Response response = client.performRequest(reque
        LOG.debug("update response: {}", response);
    } catch (IOException ex) {
        LOG.warn("Could not post {} to ES", id, ex);
    }
}
```

Deleting an item. Deleting an item is also straightforward: All you need to do is send a DELETE request with the index and the document ID, for example, `/<indexname>/_doc/5`.

```
public void deleteCatalogItem(Integer id) {
    Request request = new Request("DELETE",
        String.format("/%s/_doc/%d", getIndex(), id
    try {
        Response response = client.performRequest(reque
        LOG.debug("delete response: {}", response);
    } catch (IOException ex) {
        LOG.warn("Could not post {} to ES", id, ex);
    }
}
```

This completes the overview of CRUD methods in the low-level synchronous client.

Asynchronous calls. To make an asynchronous call using the low-level client, you just need to call the `performRequestAsync` method instead of the `performRequest` method. You must supply a response listener to the asynchronous call. The response listener needs to implement two methods: `onSuccess` and `onFailure`. I show these in lines 5 and 9 of

the following code. In this example, I am upserting several items into an Elasticsearch index asynchronously.

A countdown latch is part of Java concurrent package, and it is used here as a thread synchronization mechanism. The countdown latch is initialized with an integer count, and it will block the thread calling its await() method until the number of countDown calls that are made by other threads are equal to the value it is initialed with.

To do that, I create a countdown latch and use it to make sure createCatalogMethod will not return until all the items are sent and processed by Elasticsearch. In my response listener implementation, I use latch countdown methods upon both success and failure to indicate that Elasticsearch has processed an item.

```java
public void createCatalogItem(List<CatalogItem> iter
  CountDownLatch latch = new CountDownLatch(items.si
  ResponseListener listener = new ResponseListener(
    @Override
    public void onSuccess(Response response) {
      latch.countDown();
    }
    @Override
    public void onFailure(Exception exception) {
      latch.countDown();
      LOG.error(
        "Could not process ES request. ", exception
    }
  };

  itemsToCreate.stream().forEach(e-> {
    Request request = new Request(
              "PUT",
              String.format("/%s/_doc/%d",
                        index, e.getId()));
    try {
      request.setJsonEntity(
          jsonMapper().writeValueAsString(e));
      client.performRequestAsync(request, listener
    } catch (IOException ex) {
      LOG.warn("Could not post {} to ES", e, ex);
    }
  });
  try {
    latch.await(); //wait for all the threads to fin
    LOG.info("Done inserting all the records to the
  } catch (InterruptedException e1) {
    LOG.warn("Got interrupted.",e1);
  }
}
```

Using an asynchronous client is much easier with the high-level REST client.

High-Level REST Client

The high-level REST client is built on top of the low-level client. It adds a few Elasticsearch dependencies to the project, but as you will see, it makes coding much easier and enjoyable for both the synchronous and asynchronous API. One thing to keep in mind when choosing to use the high-level API is that it is recommended to upgrade client dependencies with each major update to the Elasticsearch cluster. This dependency upgrade is not needed when using the low-level API, but you might have to adjust your implementation to compensate for any underlying Elasticsearch API changes. While the high-level client makes coding easier, the low-level client gives you more control and has a smaller binary footprint.

To get the high-level Elasticsearch libraries, all you need to do is import the REST client as shown below for a Maven project.

```xml
<dependency>
  <groupId>org.elasticsearch.client</groupId>
  <artifactId>
    elasticsearch-rest-high-level-client
  </artifactId>
  <version>7.4.2</version>
</dependency>
```

Building a high-level REST client is very similar to building a low-level REST client. The only difference is that I need to wrap my low-level client in a high-level client API:

```java
try(RestHighLevelClient client =
    new RestHighLevelClient(
        RestClient.builder(
            new HttpHost("localhost", 9200, "http")

    CrudMethodsSynchronous scm =
        new CrudMethodsSynchronous(
            "catalog_item_high_level",  client);
```

In the attached code, I have all the same methods the low-level client implemented, but because the high-level model is so much easier to work with, I will describe only two methods here: how to create a document and how to search for a document.

To create a document in the Elasticsearch high-level API, you need to use `IndexRequest` and initialize it with the name of the desired index. Then set the ID on the request and add JSON as a source. Calling the high-level client index API with the request synchronously will return the index response, which could then be used to see if a document was created or updated.

```java
public void createCatalogItem(List<CatalogItem> iter
  items.stream().forEach(e-> {
    IndexRequest request = new IndexRequest(index);
      try {
        request.id(""+e.getId());
        request.source(jsonMapper.writeValueAsString
                XContentType.JSON);
        request.timeout(TimeValue.timeValueSeconds(
        IndexResponse response = client.index(reques
                        RequestOptions.DEFAULT)
        if (response.getResult() ==
                DocWriteResponse.Result.CREATED)
          LOG.info("Added catalog item with id {} "
                  + "to ES index {}",
              e.getId(), response.getIndex());

        } else if (response.getResult() ==
                DocWriteResponse.Result.UPDATED)
          LOG.info("Updated catalog item with id {}
            " to ES index {}, version of the " +
          "object is {} ",
          e.getId(), response.getIndex(),
          response.getVersion());
        }

    } catch (IOException ex) {
      LOG.warn("Could not post {} to ES", e, ex);
    }
  }
  );
}
```

Similarly, a full text search is much easier to read. Here, I create a search request by passing an index and then use a search query builder to construct a full text search. The search response encapsulates the JSON navigation and allows you easy access to the resulting documents via the `SearchHits` array in the following code:

```
public List<CatalogItem> findCatalogItem(String text
    try {
        SearchRequest request = new SearchRequest(i
        SearchSourceBuilder scb = new SearchSourceBu
        SimpleQueryStringBuilder mcb =
        QueryBuilders.simpleQueryStringQuery(text);
        scb.query(mcb);
        request.source(scb);

        SearchResponse response =
            client.search(request, RequestOptions.DI
        SearchHits hits = response.getHits();
        SearchHit[] searchHits = hits.getHits();
        List<CatalogItem> catalogItems =
            Arrays.stream(searchHits)
                .filter(Objects::nonNull)
                .map(e -> toJson(e.getSourceAsStri
                .collect(Collectors.toList());

        return catalogItems;
    } catch (IOException ex) {
        LOG.warn("Could not post {} to ES", text, e:
    }
    return Collections.emptyList();
}
```

In the code above, I searched a specific index in Elasticsearch. To search all indexes, I would need to create a `SearchRequest` without any parameters. What you can see from these two examples is a pattern that spans the rest of the CRUD methods: You first create a specific request, passing it an index and a document ID. Such a request could be an IndexRequest to create a document, a GetRequest to get a document by ID, an UpdateRequest to update the document, and so on. Then, you issue the appropriate request to Elasticsearch, for example, to get, update, or delete and you receive a response that has the status and source objects, if applicable.

Asynchronous calls. Asynchronous calls are a bit less painful to write with the high-level client. To write them, you call a similar synchronous method and add the `Async` postfix and supply either an Elasticsearch ActionListener or a higher-level object, such as a `PlainActionFuture` as a last argument. A `PlainActionFuture` is an Elasticsearch class that is imported by the high-level API dependencies. It implements both an Elasticsearch `ActionListener` interface and Java `Future` interface, making it an ideal choice for response processing.

The following sample code implements all the methods asynchronously: It is identical to the synchronous example, aside from the fact that I create a `PlainActionFuture`, which will hold a search response and which I pass to the `searchAsync` API of the high-level REST client. The caller of this method will then inspect the future and when the search completes, parsing the search response will be done in exactly the same way as with the synchronous API.

The biggest advantage of asynchronous APIs is that you can perform other operations in the working thread until you need the results of the search.

```
public PlainActionFuture<SearchResponse>
                             findItem(String te:

    SearchRequest request = new SearchRequest(getInd
    SearchSourceBuilder ssb = new SearchSourceBuilde
    SimpleQueryStringBuilder mqb =
            QueryBuilders.simpleQueryStringQuery(te
    ssb.query(mqb);
    request.source(ssb);

    PlainActionFuture<SearchResponse> future =
                        new PlainActionFuture<
    client.searchAsync(request,
                    RequestOptions.DEFAULT, futu:
    return future;
}
```

Streaming Data into Elasticsearch

As the last part of this article, I want to show how to stream data into Elasticsearch and I want to introduce bulk operations. Bulk operations allow you to execute multiple index, update, or delete operations using a single request. The advantage of doing a bulk request is that you do everything in only one round trip to the Elasticsearch server instead of doing a round trip for every request. Bulk operations also fit very well for streaming data into Elasticsearch. The only caveat is that you need to figure out how to size a batch to avoid extra latency by ensuring you don't make the batch so big that the entire request times out before fully completing the work.

A batch request can have different operations in it, but in the following example, it will just have an index request to insert data into an Elasticsearch index. The following routine creates one bulk request that adds an index request for each item passed in a batch. It relies on the caller to make sure that the batch size is reasonable. Once all the items are added, a synchronous client bulk request is submitted.

The bulk request call returns a bulk response that contains bulk response items—each of which corresponds to an item in the request, indicating what operation was requested, whether it was successful or not, and so on. I am not using a bulk response in this example for the sake of simplicity.

```
private void sendBatchToElasticSearch(
                List<LineFromShakespeare> linesInBa
    RestHighLevelClient client,
    String indexName) throws IOException {

    BulkRequest request = new BulkRequest();
    linesInBatch.stream().forEach(l -> {
        try {
            request.add(new IndexRequest(indexName)
                    .id(l.getId())
                    .source(jsonMapper.writeValueAsSt
                XContentType.JSON));

        } catch (JsonProcessingException e) {
            LOG.error("Problem mapping object {}", l
        }});
    LOG.info("Sending data to ES");
    client.bulk(request, RequestOptions.DEFAULT);
}
```

The call above is invoked by a procedure that streams a file and makes a batch of 1,000 documents per load into Elasticsearch.

```
public void loadData(String file, String index)
                throws IOException, URISyntaxExceptio
    Path filePath = Paths.get(
            ClassLoader.getSystemResource(file).toURI

    List<String> errors = new ArrayList<>();
    List<LineFromShakespeare> lines = new ArrayList<>
    final int maxLinesInBatch = 1000;
    try(RestHighLevelClient client =
        new RestHighLevelClient(
            RestClient.builder(
             new HttpHost("localhost", 9200, "http")))) {

        Files.lines(file).forEach( e-> {
            try {
                LineFromShakespeare line =
                    jsonMapper.readValue(e, LineFromShal
                //enrich...
                linesInBatch.add(line);
                if (linesInBatch.size() >= maxLinesInBat
                    sendBatchToElasticSearch(linesInBatc
                                            client, ind
                    linesInBatch.clear();
```

```
              }
          } catch (IOException ex) {
            errors.add(e);
            linesInBatch.clear();
          }});

          if (linesInBatch.size() != 0) {
              sendBatchToElasticSearch(linesInBatch,
                                  client, indexName);
              linesInBatch.clear();
          }
      }

      LOG.info("Errors found in {} batches", errors.size
  }
```

Here I have used the Java `Files.lines` API to stream a file line by line, convert the text to an object, enrich it, and add it to the batch to be sent to Elasticsearch. Once the batch size reaches 1,000, I ship the batch to Elasticsearch by using the `sendBatchToElasticSearch` method.

As you can see, using the high-level API simplifies your code and makes it much more readable. So, if the binary footprint is not an issue, and you can live with upgrading dependencies with each major upgrade to the Elasticsearch cluster, I would highly recommend sticking with the high-level API.

Conclusion

In this article, I introduced Elasticsearch and focused on a Java CRUD API used in both a low-level and high-level client, showing most of the needed functions for CRUD applications. The APIs, along with streaming data into Elasticsearch, make up the basic knowledge you need before embarking on an Elasticsearch adventure that includes document design, data analyzers, advanced searching including a multifield search, proximity matching, paging, suggestions, highlighting, result scoring, and different types of data aggregations, geolocation, security, and cluster management. The playing field here is really vast. Enjoy!

Henry Naftulin

Henry Naftulin has been designing Java EE distributed systems for more than 15 years. He is currently leading development of a proprietary award-winning fixed-income trading platform for one of the largest financial companies in the United States.

Share this Page

Contact
US Sales: +1.800.633.0738
Global Contacts
Support Directory
Subscribe to Emails

About Us
Careers
Communities
Company Information
Social Responsibility Emails

Downloads and Trials
Java for Developers
Java Runtime Download
Software Downloads
Try Oracle Cloud

News and Events
Acquisitions
Blogs
Events
Newsroom

ORACLE | Integrated Cloud
Applications & Platform Services

© Oracle | Site Map | Terms of Use & Privacy | Cookie Preferences | Ad Choices