Este exemplo e texto foram retirados de https://www.javacodegeeks.com/2017/03/elasticsearch-java-developers-elasticsearch-java.html

Along this part of the tutorial we are going learn how to talk to Elasticsearch by means of native Java APIs. Our approach to that would be to code and to work on a couple of Java applications, using Apache Maven for build management, terrific Spring Framework for dependency wiring and inversion of control, and awesome JUnit / AssertJ as test scaffolding.

2. Using Java Client API

Since the early versions, Elasticsearch distributes a dedicated Java client API with each release, also known as transport client. It talks Elasticsearch native transport protocol and as such, imposes the constraint that the version of the client library should at least match the major version of Elasticsearch distribution you are using (ideally, the client should have exactly the same version).

As we are using Elasticsearch version `5.2.0`, it would make sense to add the respective client version dependency to our `pom.xml` file.

```
1 <dependency>
2     <groupId>org.elasticsearch.client</groupId>
3     <artifactId>transport</artifactId>
4     <version>5.2.0</version>
5 </dependency>
```

Since we have chosen Spring Framework to power our application, literally the only thing we need is a transport client configuration.

```
01 @Configuration
02 public class ElasticsearchClientConfiguration {
03     @Bean(destroyMethod = "close")
04     TransportClient transportClient() throws
   UnknownHostException  {
05         return new PreBuiltTransportClient(
06             Settings.builder()-
07                 .put(ClusterName.CLUSTER_NAME_SETTING.getKey(),
   "es-catalog")
08                 .build()
09             )
10             .addTransportAddress(new InetSocketTransportAddress(
11                 InetAddress.getByName("localhost"), 9300));
12     }
13 }
```

The `PreBuiltTransportClient` follows the builder pattern (as most of the classes as we are going to see soon) to construct `TransportClient` instance, and once it is there, we could use the injection techniques supported by Spring Framework to access it:

```
1 @Autowired private TransportClient client;
```

The `CLUSTER_NAME_SETTING` is worth of our attention: it should match exactly the name of the Elasticsearch cluster we are connecting to, which in our case is `es-catalog`.

Great, we have our transport client initialized, so what can we do with it? Essentially, the transport client exposes a whole bunch of methods (following the fluent interface style) to open the access to all Elasticsearch APIs from the Java code. To get one step ahead, it should be noted that transport client has explicit separation between regular APIs and admin APIs. The latter is available by invoking `admin()` method on the transport client instance.

Before rolling the sleeves and getting our hands dirty, it is necessary to mention that Elasticsearch Java APIs are designed to be fully asynchronous and as such, are centered around two key abstractions: `ActionFuture<?>` and `ListenableActionFuture<?>`. In fact, `ActionFuture<?>` is just a plain old Java Future<?> with a couple of handful methods added, stay tuned on that. From the other side, `ListenableActionFuture<?>` is more powerful abstraction with the ability to take the callbacks and notify the caller about the result of the execution.

Picking one style over the other is totally dictated by the needs of your applications, as both of them do have own pros and cons. Without further ado, let us go ahead and make sure our Elasticsearch cluster is healthy and is ready to rock.

```
01 final ClusterHealthResponse response = client
02     .admin()
03     .cluster()
04     .health(
05         Requests
06             .clusterHealthRequest()
07             .waitForGreenStatus()
08             .timeout(TimeValue.timeValueSeconds(5))
09     )
10     .actionGet();
11
12 assertThat(response.isTimedOut())
13     .withFailMessage("The cluster is unhealthy: %s",
   response.getStatus())
14     .isFalse();
```

The example is pretty simple and straightforward. What we do is inquiring Elasticsearch cluster about its status while explicitly asking to wait at most `5 seconds` for the status to become `green` (if it is not the case yet). Under the hood, `client.admin().cluster().health(...)` returns `ActionFuture<?>` so we have to call one of the `actionGet` methods to get the response.

Here is another, slightly different way to use Elasticsearch Java API, this time employing the `prepareXxx` methods family.

```
01 final ClusterHealthResponse response = client
02     .admin()
```

```
03     .cluster()
04     .prepareHealth()
05     .setWaitForGreenStatus()
06     .setTimeout(TimeValue.timeValueSeconds(5))
07     .execute()
08     .actionGet();
09
10 assertThat(response.isTimedOut())
11     .withFailMessage("The cluster is unhealthy: %s",
   response.getStatus())
12     .isFalse();
```

Although both code snippets lead to absolutely identical results, the latter one is calling `client.admin().cluster().prepareHealth().execute()` method at the end of the chain, which returns `ListenableActionFuture<?>`. It does not make a lot of difference in this example but please keep it in mind as we are going to see more interesting use cases where such a detail becomes really a game changer.

And finally, last but not least, the asynchronous nature of any API (and Elasticsearch Java API is not an exception) assumes that invocation of the operation will take some time and it becomes the responsibility of the caller to decide how to deal with that. What we have used so far is just calling `actionGet` on the instance of `ActionFuture<?>`, which effectively transforms the asynchronous execution into a blocking (or, to say it the other way, synchronous) call. Moreover, we did not specify the expectations in terms of how long we would agree to wait for the execution to be completed before giving up. We could do better than that and in the rest of this section we are going to address both of these points.

Once we have our Elasticsearch cluster status all `green`, it is time to create some indices, much like we have done in the previous part of the tutorial but this time using Java APIs only. It would be good idea to ensure that `catalog` index does not exist yet before creating one.

```
1 final IndicesExistsResponse response = client
2     .admin()
3     .indices()
4     .prepareExists("catalog")
5     .get(TimeValue.timeValueMillis(100));
6
7 if (!response.isExists()) {
8     ...
9 }
```

Please notice that in the snippet above we provided the explicit timeout for the operation to complete, `get(TimeValue.timeValueMillis(100))`, which is essentially the shortcut to `execute().actionGet(TimeValue.timeValueMillis(100))`.

For the `catalog` index settings and mapping types we are going to use exactly the same JSON file, `catalog-index.json`, which we had been using in the previous part of the tutorial. We are going to place it into `src/test/resources` folder, following Apache Maven conventions.

```
@Value("classpath:catalog-index.json")
private Resource index;
```

Fortunately Spring Framework simplifies a lot the injection of the classpath resources so not much we need to do here to gain the access to `catalog-index.json` content and feed it directly to Elasticsearch Java API.

```
try (final ByteArrayOutputStream out = new ByteArrayOutputStream()) {
    Streams.copy(index.getInputStream(), out);

    final CreateIndexResponse response = client
        .admin()
        .indices()
        .prepareCreate("catalog")
        .setSource(out.toByteArray())
        .setTimeout(TimeValue.timeValueSeconds(1))
        .get(TimeValue.timeValueSeconds(2));

    assertThat(response.isAcknowledged())
        .withFailMessage("The index creation has not been acknowledged")
        .isTrue();
}
```

This code block illustrates yet another way to approach the Elasticsearch Java APIs by utilizing the `setSource` method call. In a nutshell, we just supply the request payload ourselves in a form of opaque blob (or string) and it is going to be sent to Elasticsearch node(s) as is. However, we could have used a pure Java data structures instead, for example:

```
final CreateIndexResponse response = client
    .admin()
    .indices()
    .prepareCreate("catalog")
    .setSettings(...)
    .setMapping("books", ...)
    .setMapping("authors", ...)
    .setTimeout(TimeValue.timeValueSeconds(1))
    .get(TimeValue.timeValueSeconds(2));
```

Good, with that we are going to conclude the transport client admin APIs and switch over to document and search APIs, as those would be the ones you would use most of the time. As we remember, Elasticsearch speaks JSON so we have to somehow convert books and authors to JSON representation using Java. In fact, Elasticsearch Java API helps with that by supporting the generic abstraction over the content named `XContent`, for example:

```
final XContentBuilder source = JsonXContent
    .contentBuilder()
    .startObject()
```

```
    .field("title", "Elasticsearch: The Definitive Guide. ...")
    .startArray("categories")
        .startObject().field("name", "analytics").endObject()
        .startObject().field("name", "search").endObject()
        .startObject().field("name", "database store").endObject()
    .endArray()
    .field("publisher", "O'Reilly")
    .field("description", "Whether you need full-text search or ...")
    .field("published_date", new LocalDate(2015, 02, 07).toDate())
    .field("isbn", "978-1449358549")
    .field("rating", 4)
    .endObject();
```

Having the document representation, we could send it over to Elasticsearch for indexing. To keep the promises, this time we would like to go truly asynchronous way and do not wait for the response, providing the notification callback in a shape of `ActionListener<IndexResponse>` instead.

```
client
    .prepareIndex("catalog", "books")
    .setId("978-1449358549")
    .setContentType(XContentType.JSON)
    .setSource(source)
    .setOpType(OpType.INDEX)
    .setRefreshPolicy(RefreshPolicy.WAIT_UNTIL)
    .setTimeout(TimeValue.timeValueMillis(100))
    .execute(new ActionListener() {
        @Override
      public void onResponse(IndexResponse response) {
          LOG.info("The document has been indexed with the result: {}",
              response.getResult());
        }

        @Override
        public void onFailure(Exception ex) {
            LOG.error("The document has been not been indexed", ex);
        }
    });
```

Nice, so we have our first document in the `books` collection! What about `authors` though? Well, just as reminder, the book in question has more than one author so it is a perfect occasion to use document bulk indexing.

```
final XContentBuilder clintonGormley = JsonXContent
    .contentBuilder()
    .startObject()
    .field("first_name", "Clinton")
    .field("last_name", "Gormley")
    .endObject();

final XContentBuilder zacharyTong = JsonXContent
    .contentBuilder()
    .startObject()
    .field("first_name", "Zachary")
    .field("last_name", "Tong")
    .endObject();
```

The `XContent` part is clear enough and frankly, you may never use such an option, preferring to model real classes and use one of the terrific Java libraries for automatic to / from JSON conversions. But the following snippet is really interesting.

```
final BulkResponse response = client
    .prepareBulk()
    .add(
        Requests
            .indexRequest("catalog")
            .type("authors")
            .id("1")
            .source(clintonGormley)
            .parent("978-1449358549")
            .opType(OpType.INDEX)
    )
    .add(
        Requests
            .indexRequest("catalog")
            .type("authors")
            .id("2")
            .source(zacharyTong)
            .parent("978-1449358549")
            .opType(OpType.INDEX)
    )
    .setRefreshPolicy(RefreshPolicy.WAIT_UNTIL)
    .setTimeout(TimeValue.timeValueMillis(500))
    .get(TimeValue.timeValueSeconds(1));

assertThat(response.hasFailures())
    .withFailMessage("Bulk operation reported some failures: %s",
```

```
27          response.buildFailureMessage())
28      .isFalse();
```

We are sending two index requests for `authors` collection in one single batch. You might be wondering what this `parent("978-1449358549")` means and to answer this question we have to recall that `books` and `authors` are modeled using parent / child relationships. So the `parent` key in this case is the reference (by the `_id` property) to the respective parent document in `books` collection.

Well done, so we know how to work with indices and how to index the documents using Elasticsearch transport client Java APIs. It is search time now!

```
01 final SearchResponse response = client
02     .prepareSearch("catalog")
03     .setTypes("books")
04     .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
05     .setQuery(QueryBuilders.matchAllQuery())
06     .setFrom(0)
07     .setSize(10)
08     .setTimeout(TimeValue.timeValueMillis(100))
09     .get(TimeValue.timeValueMillis(200));
10
11 assertThat(response.getHits().hits())
12     .withFailMessage("Expecting at least one book to be
   returned")
13     .isNotEmpty();
```

The simplest search criterion one can come up with is to match all documents and this is what we have done in the snippet above (please notice that we explicitly limited the number of results returned to `10` documents).

To our luck, Elasticsearch Java API has full-fledged implementation of Query DSL in a form of `QueryBuilders` and `QueryBuilder` classes so writing (and maintaining) the complex queries is exceptionally easy. As an exercise, we are going to build the same compound query which we came up with in the previous part of the tutorial:

```
01 final QueryBuilder query = QueryBuilders
02     .boolQuery()
03         .must(
04             QueryBuilders
05                 .rangeQuery("rating")
06                 .gte(4)
07         )
08         .must(
09             QueryBuilders
10                 .nestedQuery(
11                     "categories",
12                     QueryBuilders.matchQuery("categories.name",
```

```
    "analytics"),
                    ScoreMode.Total
                )
            )
        .must(
            QueryBuilders
                .hasChildQuery(
                    "authors",
                    QueryBuilders.termQuery("last_name", "Gormley"),
                    ScoreMode.Total
                )
    );
```

The code looks pretty, concise and human-readable. If you are keen on using static imports feature of the Java programming language, the query is going to look even more compact.

```
final SearchResponse response = client
    .prepareSearch("catalog")
    .setTypes("books")
    .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
    .setQuery(query)
    .setFrom(0)
    .setSize(10)
    .setFetchSource(
        new String[] { "title", "publisher" }, /* includes */
        new String[0] /* excludes */
    )
    .setTimeout(TimeValue.timeValueMillis(100))
    .get(TimeValue.timeValueMillis(200));

assertThat(response.getHits().hits())
    .withFailMessage("Expecting at least one book to be returned")
    .extracting("sourceAsString", String.class)
    .hasOnlyOneElementSatisfying(source -> {
        assertThat(source).contains("Elasticsearch: The Definitive Guide.");
    });
```

To keep both versions of the query identical, we also hinted the search request through `setFetchSource` method that we are interested only in returning title and publisher properties of the document source.

The curious readers might be wondering how to use aggregations along with the search requests. This is excellent topic to cover so let us talk about that for a moment. Along with Query DSL, Elasticsearch Java API also supplies aggregations DSL, revolving around

`AggregationBuilders` and `AggregationBuilder` classes. For example, this is how we could build the bucketed aggregation by `publisher` property.

```
final AggregationBuilder aggregation = AggregationBuilders
    .terms("publishers")
    .field("publisher")
    .size(10);
```

Having the aggregations defined, we could inject them into search request using `addAggregation` method call as is shown in the code snippet below:

```
final SearchResponse response = client
    .prepareSearch("catalog")
    .setTypes("books")
    .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
    .setQuery(QueryBuilders.matchAllQuery())
    .addAggregation(aggregation)
    .setFrom(0)
    .setSize(10)
    .setTimeout(TimeValue.timeValueMillis(100))
    .get(TimeValue.timeValueMillis(200));

final StringTerms publishers =
response.getAggregations().get("publishers");
assertThat(publishers.getBuckets())
    .extracting("keyAsString", String.class)
    .contains("O'Reilly");
```

The results of the aggregations are available in the response and could be retrieved by referencing the aggregation name, for example `publishers` in our case. However be cautious and carefully use the proper aggregation types in order to not get surprises in a form of ClassCastException. Because our publishers aggregation has been defined to group terms into buckets, we are safe by casting the one from the response to `StringTerms` class instance.

3. Using Java Rest Client

One of the drawbacks related to the usage of the Elasticsearch Java client API is the requirement to be binary compatible with the version of Elasticsearch (either standalone or cluster) you are running.

Fortunately, since the first release of `5.0.0` branch, Elasticsearch brings another option on the table: Java REST client. It uses HTTP protocol to talk to Elasticsearch by invoking its RESTful API endpoints and is oblivious to the version of Elasticsearch (literally, it is compatible with all Elasticsearch versions).

It should be noted though that Java REST client is pretty low level and is not as convenient to use as Java client API, far from that in fact. However, there are quite a few reasons why one may prefer to use Java REST client over Java client API to communicate with Elasticsearch so it is worth its own

discussion. To start off, let us include the respective dependency into our Apache Maven `pom.xml` file.

```
<dependency>
    <groupId>org.elasticsearch.client</groupId>
    <artifactId>rest</artifactId>
    <version>5.2.0</version>
</dependency>
```

From the configuration perspective we only need to construct the instance of `RestClient` by calling `RestClient.builder` method.

```
@Configuration
public class ElasticsearchClientConfiguration {
    @Bean(destroyMethod = "close")
    RestClient transportClient() {
        return RestClient
            .builder(new HttpHost("localhost", 9200))
            .setRequestConfigCallback(new
RequestConfigCallback() {
                @Override
                public Builder customizeRequestConfig(Builder
builder) {
                    return builder
                        .setConnectTimeout(1000)
                        .setSocketTimeout(5000);
                }
            })
            .build();
    }
}
```

We are jumping a bit ahead here but please pay particular attention to configuration of the proper timeouts because Java REST client does not provide a way (at least, at the moment) to specify those on per-request level basis. With that, we can inject the `RestClient` instance anywhere, using the same wiring techniques Spring Framework is kindly providing to us:

```
@Autowired private RestClient client;
```

To make a fair comparison between Java client API and Java REST client, we are going to dissect a couple of examples we have looked at in the previous section, setting out the stage by checking the Elasticsearch cluster health.

```
@Test
public void esClusterIsHealthy() throws Exception {
    final Response response = client
        .performRequest(HttpGet.METHOD_NAME, "_cluster/health",
emptyMap());

```

```
06     final Object json = defaultConfiguration()
07         .jsonProvider()
08         .parse(EntityUtils.toString(response.getEntity()));
09
10     assertThat(json, hasJsonPath("$.status",
   equalTo("green")));
11 }
```

Indeed, the difference is obvious. As you may guess, Java REST client is actually a thin wrapper around the more generic, well-known and respected Apache Http Client library. The response is returned as a string or byte array and it becomes the responsibility of the caller to transform it to JSON and extract the necessary pieces of data. To deal with that in test assertions, we have on-boarded the wonderful JsonPath library, but you are free to make a choice here.

A family of `performRequest` methods is the typical way for synchronous (or blocking) communication using Java REST client API. Alternatively, there is a family of `performRequestAsync` methods which are supposed to be used in fully asynchronous flows. In the next example we are going to use one of those in order to index the document into `books` collection.

The simplest way to represent JSON-like structure in Java language is using plain old `Map<String, Object>` as it is demonstrated in the code fragment below.

```
01 final Map<String, Object> source = new LinkedHashMap<>();
02 source.put("title", "Elasticsearch: The Definitive
   Guide. ...");
03 source.put("categories",
04     new Map[] {
05         singletonMap("name", "analytics"),
06         singletonMap("name", "search"),
07         singletonMap("name", "database store")
08     }
09 );
10 source.put("publisher", "O'Reilly");
11 source.put("description", "Whether you need full-text search or
   ...");
12 source.put("published_date", "2015-02-07");
13 source.put("isbn", "978-1449358549");
```

Now we need to convert this Java structure into valid JSON string. There are dozens of way to do so but we are going to leverage the json-smart library, for the reason that it is already available as a transitive dependency of JsonPath library.

```
1 final HttpEntity payload = new
  NStringEntity(JSONObject.toJSONString(source),
2     ContentType.APPLICATION_JSON);
```

Having the payload ready, nothing prevents us from invoking Indexing API of the Elasticsearch to add a book into `books` collection.

```
client.performRequestAsync(
    HttpPut.METHOD_NAME,
    "catalog/books/978-1449358549",
    emptyMap(),
    payload,
    new ResponseListener() {
        @Override
        public void onSuccess(Response response) {
            LOG.info("The document has been indexed successfully");
        }

        @Override
        public void onFailure(Exception ex) {
            LOG.error("The document has been not been indexed", ex);
        }
    });
```

This time we decided to not wait for the response but supply a callback (instance of `ResponseListener`) instead, keeping the flow truly asynchronous. To finish up, it would be great to understand what it takes to perform more or less realistic search request and parse the results.

As you may expect, the Java REST client does not provide any fluent APIs around Query DSL so we have to fallback to `Map<String, Object>` one more time in order to construct the search criteria.

```
final Map<String, Object> authors = new LinkedHashMap<>();
authors.put("type", "authors");
authors.put("query",
    singletonMap("term",
        singletonMap("last_name", "Gormley")
    )
);

final Map<String, Object> categories = new LinkedHashMap<>();
categories.put("path", "categories");
categories.put("query",
    singletonMap("match",
        singletonMap("categories.name", "search")
    )
);

final Map<String, Object> query = new LinkedHashMap<>();
query.put("size", 10);
```

```
query.put("_source", new String[] { "title", "publisher" });
query.put("query",
    singletonMap("bool",
        singletonMap("must", new Map[] {
            singletonMap("range",
                singletonMap("rating",
                    singletonMap("gte", 4)
                )
            ),
            singletonMap("has_child", authors),
            singletonMap("nested", categories)
        })
    )
);
```

The price to pay by tackling the problem openly is a lot of cumbersome and error-prone code to write. In this regard, the consistency and conciseness of Java client API really makes a huge difference. You may argue that in reality one may rely on much simpler and safer techniques, like data transfer object, value objects, or even to have JSON search query templates with placeholders, but the point is a little help is offered by Java REST client at the moment.

```
final HttpEntity payload = new
NStringEntity(JSONObject.toJSONString(query),
    ContentType.APPLICATION_JSON);

final Response response = client
    .performRequest(HttpPost.METHOD_NAME,
"catalog/books/_search",
        emptyMap(), payload);

final Object json = defaultConfiguration()
    .jsonProvider()
    .parse(EntityUtils.toString(response.getEntity()));

assertThat(json, hasJsonPath("$.hits.hits[0]._source.title",
    containsString("Elasticsearch: The Definitive Guide.")));
```

Not much to add here, just consult the Search API documentation on the format and extract the details of your interest from the response, like we do by asserting on the `title property` of the document `_source`.

With that, we are wrapping up our discussion about Java REST client. Frankly speaking, you may find it unclear if there are any benefits of using it versus choosing one of the generic HTTP clients Java ecosystem is rich of. Indeed, this is a valid concern but please keep in mind that Java REST client is brand new addition to Elasticsearch family and, hopefully, we are going to see a lot of exciting features pumped into it very soon.

4. Using Testing Kit

As our applications become more complex and distributed, the proper testing becomes as important as never before. For years Elasticsearch provides the superior test harness in order to simplify the testing of the applications which heavily rely on its search and analytics features. More specifically, there are two types of tests which you may need in your projects:

- **Unit tests**: those are testing the individual units (like classes f.e.) in isolation and generally do not require to have running Elasticsearch nodes or clusters. These kinds of tests are backed by `ESTestCase` and `ESTokenStreamTestCase`.
- **Integration tests**: those are testing the complete flows and usually require at least one running Elasticsearch node (or cluster, to stress out more realistic scenarios). These kinds of tests are backed by `ESIntegTestCase`, `ESSingleNodeTestCase` and `ESBackCompatTestCase`.

Let us roll the sleeves one more time and learn how to use the test scaffolding provided by Elasticsearch to develop our own test suites. We are going to start off by declaring our dependencies, still using Apache Maven for that.

```
01 <dependency>
02     <groupId>org.apache.lucene</groupId>
03     <artifactId>lucene-test-framework</artifactId>
04     <version>6.4.0</version>
05     <scope>test</scope>
06 </dependency>
07 
08 <dependency>
09     <groupId>org.elasticsearch.test</groupId>
10     <artifactId>framework</artifactId>
11     <version>5.2.0</version>
12     <scope>test</scope>
13 </dependency>
```

Although this is not strictly necessary, we are also adding the explicit dependency to JUnit, bumping its version to `4.12`.

```
01 <dependency>
02     <groupId>junit</groupId>
03     <artifactId>junit</artifactId>
04     <version>4.12</version>
05     <scope>test</scope>
06     <exclusions>
07         <exclusion>
08             <groupId>org.hamcrest</groupId>
09             <artifactId>hamcrest-core</artifactId>
10         </exclusion>
11     </exclusions>
12 </dependency>
```

We have to sound a note of caution here: the Elasticsearch test framework is exceptionally sensitive to dependencies, making sure your application does not fall into the problem so well known to every Java developer as jar hell. One of the pre-checks Elasticsearch test framework does is ensuring there are no duplicate classes in classpath. It is quite often that you may use other excellent testing libraries along the way but if your Elasticsearch test cases suddenly are starting to fail the initialization phase, it is very likely due to jar hell issues detected and some exclusions have to be applied.

And one more thing, very likely you may need to turn off security manager during the test runs by setting `tests.security.manager` property to `false`. This could be done either by passing `-Dtests.security.manager=false` argument to JVM directly or using Apache Maven plugin configuration.

```
1 <plugin>
2     <groupId>org.apache.maven.plugins</groupId>
3     <artifactId>maven-surefire-plugin</artifactId>
4     <version>2.19.1</version>
5     <configuration>
6         <argLine>-Dtests.security.manager=false</argLine>
7     </configuration>
8 </plugin>
```

Wonderful, all prerequisites are explained and we are all set to start developing our first test cases. The unit tests in context applicable to Elasticsearch are very useful to test your own analyzers, tokenizers, token filters and character filters. We have not done much in this regards, but the integration tests are a very different story. Let us see what it takes to spin up Elasticsearch cluster with 3 nodes.

```
1 @ClusterScope(numDataNodes = 3)
2 public class ElasticsearchClusterTest extends ESIntegTestCase {
3 }
```

And … literally, that is it. Surely, although the cluster is up, it has no indices or whatnot preconfigured. Let us add some test background to create a `catalog` index and its mapping types, using the same `catalog-index.json` file.

```
01 @Before
02 public void setUpCatalog() throws IOException {
03     try (final ByteArrayOutputStream out = new
   ByteArrayOutputStream()) {
04         Streams.copy(getClass().getResourceAsStream("/catalog-
   index.json"),
05             out);
06
07         final CreateIndexResponse response = admin()
08         .indices()
09             .prepareCreate("catalog")
10             .setSource(out.toByteArray())
```

```
11             .get();
12
13         assertAcked(response);
14         ensureGreen("catalog");
15     }
16 }
```

If you recognize this code already it is because we are using the same transport client we have learned about before! Elasticsearch test scaffolding provides it for you behind `client()` or `admin()` methods, along with `getRestClient()` in case you need Java REST client instance. It would be good to clear up the cluster after each test run, luckily we can use `cluster()` method to get access to couple of very useful operations, for example:

```
1 @After
2 public void tearDownCatalog() throws IOException,
  InterruptedException {
3     cluster().wipeIndices("catalog");
4 }
```

Overall, Elasticsearch test harness aims for two goals: simplify the most common tasks (we have already seen `client()`, `admin()`, `cluster()` in action) and easily do the verification, assertions or expectations (for example, `ensureGreen(...)`, `assertAcked(...)`). The official documentation has dedicated sections which go over helper methods and assertions so please take a look.

To begin with, the empty index should have no documents in it so our first test case is going to assert this fact explicitly.

```
01 @Test
02 public void testEmptyCatalogHasNoBooks() {
03     final SearchResponse response = client()
04         .prepareSearch("catalog")
05         .setTypes("books")
06         .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
07         .setQuery(QueryBuilders.matchAllQuery())
08         .setFetchSource(false)
09         .get();
10
11     assertNoSearchHits(response);
12 }
```

Easy one, but what about creating real documents? Elasticsearch test framework has a wide range of helpful methods to generate random values for mostly any type. We can leverage that to create a book, add it into the book `catalog` index and issue the queries against it.

```
01 @Test
02 public void testInsertAndSearchForBook() throws IOException {
03     final XContentBuilder source = JsonXContent
04         .contentBuilder()
```

```
05        .startObject()
06          .field("title", randomAsciiOfLength(100))
07          .startArray("categories")
08              .startObject().field("name",
"analytics").endObject()
09              .startObject().field("name", "search").endObject()
10              .startObject().field("name", "database
store").endObject()
11          .endArray()
12          .field("publisher", randomAsciiOfLength(20))
13          .field("description", randomAsciiOfLength(200))
14          .field("published_date", new LocalDate(2015, 02,
07).toDate())
15          .field("isbn", "978-1449358549")
16          .field("rating", randomInt(5))
17          .endObject();
18
19      index("catalog", "books", "978-1449358549", source);
20      refresh("catalog");
21
22      final QueryBuilder query = QueryBuilders
23          .nestedQuery(
24              "categories",
25              QueryBuilders.matchQuery("categories.name",
"analytics"),
26              ScoreMode.Total
27          );
28
29      final SearchResponse response = client()
30          .prepareSearch("catalog")
31          .setTypes("books")
32          .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
33          .setQuery(query)
34          .setFetchSource(false)
35          .get();
36
37      assertSearchHits(response, "978-1449358549");
38  }
```

As you can see, most of the book properties are generated randomly except the `categories` so we could reliably search by them.

The Elasticsearch testing support opens a lot of interesting opportunities not only to test the successful outcomes but also to simulate realistic cluster behavior and erroneous conditions (the helper methods provided by `internalCluster()` are exceptionally useful here). For such a complex distributed system as Elasticsearch, the value of such tests is priceless so please leverage

the available options to ensure that the code deployed into production is robust and resilient to failures. As a quick example, we could shutdown random data node while running the search requests and assert that they are still being processed.

```
01 @Test
02 public void testClusterNodeIsDown() throws IOException {
03     internalCluster().stopRandomDataNode();
04
05     final SearchResponse response = client()
06         .prepareSearch("catalog")
07         .setTypes("books")
08         .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
09         .setQuery(QueryBuilders.matchAllQuery())
10         .setFetchSource(false)
11         .get();
12
13     assertNoSearchHits(response);
14 }
```

We just scratched the surface of what is possible with Elasticsearch test harness. Hopefully you are practicing test-driven development in your organization and the examples we have looked at could serve you well as a starting point.

5. Conclusions

In this part of the tutorial we have learned about two types of the Java client APIs which Elasticsearch offers out of the box: transport client and REST client. You may find it difficult to make a choice of what Java client APIs favor to use, but by and large, it highly depends on an application. In most cases transport client is the best option however if your project uses just a couple of Elasticsearch APIs (or very limited subset of its features), REST client could be a better alternative. Also, we should not forget that Java REST client is pretty new and will improve in future releases for sure, so keep an eye on it.

While we have been dissecting transport client, the point has been made about its fully asynchronous nature. Although it is good thing by all means, we have seen that it is based on callbacks (more precisely, listeners) which may quickly lead to the problem known as callback hell. It is highly advisable to fight this issue early on (luckily, there are quite a few libraries and alternatives available like RxJava 2 and Project Reactor, with Java 9 catching up as well).

And last but not least, we have glanced over test harness of Elasticsearch and had a chance to appreciate the great helps it provides to Java / JVM developers.