Quarkus - Connecting to an Elasticsearch cluster

Elasticsearch is a well known full text search engine and NoSQL datastore.

In this guide, we will see how you can get your REST services to use an Elasticsearch cluster.

Quarkus provides two ways of accessing Elasticsearch: via the lower level `RestClient` or via the `RestHighLevelClient` we will call them the low level and the high level clients.

> This technology is considered preview.
>
> In *preview*, backward compatibility and presence in the ecosystem is not guaranteed. Specific improvements might require to change configuration or APIs and plans to become *stable* are under way. Feedback is welcome on our mailing list or as issues in our GitHub issue tracker.
>
> For a full list of possible extension statuses, check our FAQ entry.

Prerequisites

To complete this guide, you need:

- less than 15 minutes

- an IDE

- JDK 1.8+ installed with `JAVA_HOME` configured appropriately

- Apache Maven 3.6.3

- Elasticsearch installed or Docker installed

Architecture

The application built in this guide is quite simple: the user can add elements in a list using a form and the list is updated.

All the information between the browser and the server is formatted as JSON.

The elements are stored in Elasticsearch.

Solution

We recommend that you follow the instructions in the next sections and create the application step by step. However, you can go right to the completed example.

Clone the Git repository: `git clone https://github.com/quarkusio/quarkus-quickstarts.git`, or download an archive.

The solution for the low level client is located in the `elasticsearch-rest-client-quickstart` directory.

The solution for the high level client is located in the `elasticsearch-rest-high-level-client-quickstart` directory.

Creating the Maven project

First, we need a new project. Create a new project with the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.8.0.CR1:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=elasticsearch-quickstart \
    -DclassName="org.acme.elasticsearch.FruitResource" \
    -Dpath="/fruits" \
    -Dextensions="resteasy-jackson,elasticsearch-rest-client"
cd elasticsearch-quickstart
```

This command generates a Maven structure importing the RESTEasy/JAX-RS, Jackson, and the Elasticsearch low level client extensions. After this, the `quarkus-elasticsearch-rest-client` extension has been added to your `pom.xml`.

If you want to use the high level client instead, replace the `elasticsearch-rest-client` extension by the `elasticsearch-rest-high-level-client` extension.

> We use the `resteasy-jackson` extension here and not the JSON-B variant because we will use the Vert.x `JsonObject` helper to serialize/deserialize our objects to/from Elasticsearch and it uses Jackson under the hood.

If you don't want to generate a new project, add the following dependencies to your pom.xml.

For the Elasticsearch low level client, add:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-elasticsearch-rest-client</artifactId>
</dependency>
```

For the Elasticsearch high level client, add:

```
<dependency>
    <groupId>io.quarkus</groupId>
    <artifactId>quarkus-elasticsearch-rest-high-level-
client</artifactId>
</dependency>
```

Creating your first JSON REST service

In this example, we will create an application to manage a list of fruits.

First, let's create the Fruit bean as follows:

```
package org.acme.elasticsearch;

public class Fruit {
    public String id;
    public String name;
    public String color;
}
```

Nothing fancy. One important thing to note is that having a default constructor is required by the JSON serialization layer.

Now create a org.acme.elasticsearch.FruitService that will be the business layer of our application and store/load the fruits from the Elasticsearch instance. Here we use the low level client, if you want to use the high level client instead follow the instructions in the Using the High Level REST Client paragraph instead.

```
package org.acme.elasticsearch;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;

import org.apache.http.util.EntityUtils;
import org.elasticsearch.client.Request;
import org.elasticsearch.client.Response;
import org.elasticsearch.client.RestClient;

import io.vertx.core.json.JsonArray;
import io.vertx.core.json.JsonObject;

@ApplicationScoped
```

```java
public class FruitService {
    @Inject
    RestClient restClient; ①

    public void index(Fruit fruit) throws IOException {
        Request request = new Request(
                "PUT",
                "/fruits/_doc/" + fruit.id); ②

request.setJsonEntity(JsonObject.mapFrom(fruit).toString()); ③
        restClient.performRequest(request); ④
    }

    public Fruit get(String id) throws IOException {
        Request request = new Request(
                "GET",
                "/fruits/_doc/" + id);
        Response response = restClient.performRequest(request);
        String responseBody =
EntityUtils.toString(response.getEntity());
        JsonObject json = new JsonObject(responseBody); ⑤
        return json.getJsonObject("_source").mapTo(Fruit.class);
    }

    public List<Fruit> searchByColor(String color) throws
IOException {
        return search("color", color);
    }

    public List<Fruit> searchByName(String name) throws IOException
{
        return search("name", name);
    }

    private List<Fruit> search(String term, String match) throws
IOException {
        Request request = new Request(
                "GET",
                "/fruits/_search");
        //construct a JSON query like {"query": {"match":
{"<term>": "<match"}}
        JsonObject termJson = new JsonObject().put(term, match);
        JsonObject matchJson = new JsonObject().put("match",
termJson);
        JsonObject queryJson = new JsonObject().put("query",
matchJson);
        request.setJsonEntity(queryJson.encode());
        Response response = restClient.performRequest(request);
        String responseBody =
```

```
EntityUtils.toString(response.getEntity());

        JsonObject json = new JsonObject(responseBody);
        JsonArray hits =
json.getJsonObject("hits").getJsonArray("hits");
        List<Fruit> results = new ArrayList<>(hits.size());
        for (int i = 0; i < hits.size(); i++) {
            JsonObject hit = hits.getJsonObject(i);
            Fruit fruit =
hit.getJsonObject("_source").mapTo(Fruit.class);
            results.add(fruit);
        }
        return results;
    }
}
```

In this example you can note the following:

1. We inject an Elasticsearch low level `RestClient` into our service.

2. We create an Elasticsearch request.

3. We use Vert.x `JsonObject` to serialize the object before sending it to Elasticsearch, you can use whatever you want to serialize to JSON.

4. We send the request (indexing request here) to Elasticsearch.

5. In order to deserialize the object from Elasticsearch, we again use Vert.x `JsonObject`.

Now, edit the `org.acme.elasticsearch.FruitResource` class as follows:

```java
@Path("/fruits")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class FruitResource {
    @Inject
    FruitService fruitService;

    @POST
    public Response index(Fruit fruit) throws IOException {
        if (fruit.id == null) {
            fruit.id = UUID.randomUUID().toString();
        }
        fruitService.index(fruit);
        return Response.created(URI.create("/fruits/" +
fruit.id)).build();
    }

    @GET
    @Path("/{id}")
    public Fruit get(@PathParam("id") String id) throws IOException
{
        return fruitService.get(id);
    }

    @GET
    @Path("/search")
    public List<Fruit> search(@QueryParam("name") String name,
@QueryParam("color") String color) throws IOException {
        if (name != null) {
            return fruitService.searchByName(name);
        } else if (color != null) {
            return fruitService.searchByColor(color);
        } else {
            throw new BadRequestException("Should provide name or
color query parameter");
        }
    }

}
```

The implementation is pretty straightforward and you just need to define your endpoints using the JAX-RS annotations and use the `FruitService` to list/add new fruits.

Configuring Elasticsearch

The main property to configure is the URL to connect to the Elasticsearch cluster.

A sample configuration should look like this:

```
# configure the Elasticsearch client for a cluster of two nodes
quarkus.elasticsearch.hosts =
elasticsearch1:9200,elasticsearch2:9200
```

In this example, we are using a single instance running on localhost:

```
# configure the Elasticsearch client for a single instance on
localhost
quarkus.elasticsearch.hosts = localhost:9200
```

If you need a more advanced configuration, you can find the comprehensive list of supported configuration properties at the end of this guide.

Running an Elasticsearch cluster

As by default, the Elasticsearch client is configured to access a local Elasticsearch cluster on port 9200 (the default Elasticsearch port), if you have a local running instance on this port, there is nothing more to do before being able to test it!

If you want to use Docker to run an Elasticsearch instance, you can use the following command to launch one:

```
docker run --name elasticsearch  -e "discovery.type=single-node" -e
"ES_JAVA_OPTS=-Xms512m -Xmx512m"\
       --rm -p 9200:9200
docker.elastic.co/elasticsearch/elasticsearch-oss:7.8.0
```

Creating a frontend

Now let's add a simple web page to interact with our `FruitResource`. Quarkus automatically serves static resources located under the `META-INF/resources` directory. In the `src/main/resources/META-INF/resources` directory, add a `fruits.html` file with the content from this fruits.html file in it.

You can now interact with your REST service:

- start Quarkus with `./mvnw quarkus:dev`

- open a browser to `http://localhost:8080/fruits.html`

- add new fruits to the list via the 'Add fruit' form

- search for fruits by name or color via the 'Search Fruit' form

Using the High Level REST Client

Quarkus provides support for the Elasticsearch High Level REST Client but keep in mind that it comes with some caveats:

- It drags a lot of dependencies - especially Lucene -, which doesn't fit well with Quarkus philosophy. The Elasticsearch team is aware of this issue and it might improve sometime in the future.

- It is tied to a certain version of the Elasticsearch server: you cannot use a High Level REST Client version 7 to access a server version 6.

Here is a version of the `FruitService` using the high level client instead of the low level one:

```java
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

import javax.enterprise.context.ApplicationScoped;
import javax.inject.Inject;

import org.elasticsearch.action.get.GetRequest;
import org.elasticsearch.action.get.GetResponse;
import org.elasticsearch.action.index.IndexRequest;
import org.elasticsearch.action.search.SearchRequest;
import org.elasticsearch.action.search.SearchResponse;
import org.elasticsearch.client.RequestOptions;
import org.elasticsearch.client.RestHighLevelClient;
import org.elasticsearch.common.xcontent.XContentType;
import org.elasticsearch.index.query.QueryBuilders;
import org.elasticsearch.search.SearchHit;
import org.elasticsearch.search.SearchHits;
import org.elasticsearch.search.builder.SearchSourceBuilder;

import io.vertx.core.json.JsonObject;

@ApplicationScoped
public class FruitService {
    @Inject
    RestHighLevelClient restHighLevelClient; ①

    public void index(Fruit fruit) throws IOException {
        IndexRequest request = new IndexRequest("fruits"); ②
        request.id(fruit.id);
        request.source(JsonObject.mapFrom(fruit).toString(),
XContentType.JSON); ③
        restHighLevelClient.index(request, RequestOptions.DEFAULT);
④
    }
```

```java
    public Fruit get(String id) throws IOException {
        GetRequest getRequest = new GetRequest("fruits", id);
        GetResponse getResponse =
restHighLevelClient.get(getRequest, RequestOptions.DEFAULT);
        if (getResponse.isExists()) {
            String sourceAsString =
getResponse.getSourceAsString();
            JsonObject json = new JsonObject(sourceAsString); ⑤
            return json.mapTo(Fruit.class);
        }
        return null;
    }

    public List<Fruit> searchByColor(String color) throws
IOException {
        return search("color", color);
    }

    public List<Fruit> searchByName(String name) throws IOException
{
        return search("name", name);
    }

    private List<Fruit> search(String term, String match) throws
IOException {
        SearchRequest searchRequest = new SearchRequest("fruits");
        SearchSourceBuilder searchSourceBuilder = new
SearchSourceBuilder();
        searchSourceBuilder.query(QueryBuilders.matchQuery(term,
match));
        searchRequest.source(searchSourceBuilder);

        SearchResponse searchResponse =
restHighLevelClient.search(searchRequest, RequestOptions.DEFAULT);
        SearchHits hits = searchResponse.getHits();
        List<Fruit> results = new
ArrayList<>(hits.getHits().length);
        for (SearchHit hit : hits.getHits()) {
            String sourceAsString = hit.getSourceAsString();
            JsonObject json = new JsonObject(sourceAsString);
            results.add(json.mapTo(Fruit.class));
        }
        return results;
    }
}
```

In this example you can note the following:

1. We inject an Elasticsearch `RestHighLevelClient` inside the service.

2. We create an Elasticsearch index request.

3. We use Vert.x `JsonObject` to serialize the object before sending it to Elasticsearch, you can use whatever you want to serialize to JSON.

4. We send the request to Elasticsearch.

5. In order to deserialize the object from Elasticsearch, we again use Vert.x `JsonObject`.

Hibernate Search Elasticsearch

Quarkus supports Hibernate Search with Elasticsearch via the `hibernate-search-elasticsearch` extension.

Hibernate Search Elasticsearch allows to synchronize your JPA entities to an Elasticsearch cluster and offers a way to query your Elasticsearch cluster using the Hibernate Search API.

If you're interested in it, you can read the Hibernate Search with Elasticsearch guide.

Cluster Health Check

If you are using the `quarkus-smallrye-health` extension, both the extension will automatically add a readiness health check to validate the health of the cluster.

So when you access the `/health/ready` endpoint of your application you will have information about the cluster status. It uses the cluster health endpoint, the check will be down if the status of the cluster is **red**, or the cluster is not available.

This behavior can be disabled by setting the `quarkus.elasticsearch.health.enabled` property to `false` in your `application.properties`.

Building a native executable

You can use both clients in a native executable.

You can build a native executable with the usual command `./mvnw package -Pnative`.

Running it is as simple as executing `./target/elasticsearch-low-level-client-quickstart-1.0-SNAPSHOT-runner`.

You can then point your browser to `http://localhost:8080/fruits.html` and use your application.

Conclusion

Accessing an Elasticsearch cluster from a low level or a high level client is easy with Quarkus as it provides easy configuration, CDI integration and native support for it.

Configuration Reference

🔒 Configuration property fixed at build time - All other configuration properties are overridable at runtime

Configuration property	Type	Default
🔒 `quarkus.elasticsearch.health.enabled` Whether or not an health check is published in case the smallrye-health extension is present.	boolean	`true`
`quarkus.elasticsearch.hosts` The list of hosts of the Elasticsearch servers.	list of host:port	`localhost:9200`
`quarkus.elasticsearch.protocol` The protocol to use when contacting Elasticsearch servers. Set to "https" to enable SSL/TLS.	string	`http`
`quarkus.elasticsearch.username` The username for basic HTTP authentication.	string	
`quarkus.elasticsearch.password` The password for basic HTTP authentication.	string	
`quarkus.elasticsearch.connection-timeout` The connection timeout.	Duration ❓	`1S`
`quarkus.elasticsearch.socket-timeout` The socket timeout.	Duration ❓	`30S`
`quarkus.elasticsearch.max-connections` The maximum number of connections to all the Elasticsearch servers.	int	`20`
`quarkus.elasticsearch.max-connections-per-route` The maximum number of connections per Elasticsearch server.	int	`10`

`quarkus.elasticsearch.io-thread-counts` The number of IO thread. By default, this is the number of locally detected processors. Thread counts higher than the number of processors should not be necessary because the I/O threads rely on non-blocking operations, but you may want to use a thread count lower than the number of processors.	int	
`quarkus.elasticsearch.discovery.enabled` Defines if automatic discovery is enabled.	boolean	`false`
`quarkus.elasticsearch.discovery.refresh-interval` Refresh interval of the node list.	Duration ❓	5M

> ℹ️ *About the Duration format*
>
> The format for durations uses the standard `java.time.Duration` format. You can learn more about it in the Duration#parse() javadoc.
>
> You can also provide duration values starting with a number. In this case, if the value consists only of a number, the converter treats the value as seconds. Otherwise, PT is implicitly prepended to the value to obtain a standard `java.time.Duration` format.