ELASTICSEARCH TUTORIAL

Hot Recipes for the Elastic Platform

elastic

ANDRIY REDKO

Java Code Geeks
Java 2 Java Developers Resource Center

Elasticsearch Tutorial

Contents

Preface

Elasticsearch is a search engine based on Lucene. It provides a distributed, multitenant-capable full-text search engine with an HTTP web interface and schema-free JSON documents. Elasticsearch is developed in Java and is released as open source under the terms of the Apache License. Elasticsearch is the most popular enterprise search engine followed by Apache Solr, also based on Lucene.

Elasticsearch can be used to search all kinds of documents. It provides scalable search, has near real-time search, and supports multitenancy. Elasticsearch is distributed, which means that indices can be divided into shards and each shard can have zero or more replicas. Each node hosts one or more shards, and acts as a coordinator to delegate operations to the correct shard(s). Rebalancing and routing are done automatically. Related data is often stored in the same index, which consists of one or more primary shards, and zero or more replica shards. Once an index has been created, the number of primary shards cannot be changed. (Source: https://en.wikipedia.org/wiki/Elasticsearch)

In this ebook, we provide a series of tutorials so that you can develop your own Elasticsearch based applications. We cover a wide range of topics, from installation and operations, to Java API Integration and reporting. With our straightforward tutorials, you will be able to get your own projects up and running in minimum time.

About the Author

Andriy completed his Master Degree in Computer Science at Zhitomir Institute of Engineering and Technologies, Ukraine. For the last fifteen years he has been working as the Consultant/Software Developer/Senior Software Developer/Team Lead for a many successful projects including several huge software systems for customers from North America and Europe.

Through his career Andriy has gained a great experience in enterprise architecture, web development (ASP.NET, Java Server Faces, Play Framework), software development practices (test-driven development, continious integration) and software platforms (Sun JEE, Microsoft .NET), object-oriented analysis and design, development of the rich user interfaces (MFC, Swing, Windows Forms/WPF), relational database management systems (MySQL, SQL Server, PostgreSQL, Oracle), NoSQL solutions (MongoDB, Redis) and operating systems (Linux/Windows).

Andriy has a great experience in development of distributed (multi-tier) software systems, multi-threaded applications, desktop applications, service-oriented architecture and rich Internet applications. Since 2006 he is actively working primarily with JEE / JSE platforms.

As a professional he is always open to continuous learning and self-improvement to be more productive in the job he is really passionate about.

Chapter 1

Introduction to Elasticsearch

1.1 Introduction

Effective, fast and accurate search functionality is an integral part of vast majority of the modern applications and software platforms. Either you are running a small e-commerce web site and need to offer your customers a search over product catalogs, or you are a service provider and need to expose an API to let the developers filter over users and companies, or you are building any kind of messaging application where finding a conversation in the history is a must-have feature from day one ... What is really important is that, delivering as relevant results as fast as possible could be yet another competitive advantage of the product or platform you are developing.

Indeed, the search could have many faces, purposes, goals and different scale. It could be as simple as looking by exact word match or as complex as trying to understand the intent and the contextual meaning of the words one's is looking for (semantic search engines). In terms of scale, it could be as trivial as querying a single database table, or as complex as crunching over billions and billions of web pages in order to deliver the desired results. It is very interesting and flourishing area of research, with many algorithms and papers published over the years.

In case you are a Java / JVM developer, you may have heard about Apache Lucene project, a high-performance, full-featured indexing and search library. It is the first and the best in class choice to unleash the power of full-text search and embed it into your applications. Although it is a terrific library by all means, many developers have found Apache Lucene too low-level and not easy to use. That is one of the reasons why two other great projects, Elasticsearch and Apache Solr, have been born.

In this tutorial, we are going to talk about Elasticsearch, making an emphasis on development side of things rather than operational. We are going to learn the basics of Elasticsearch, get familiarized with the terminology and discuss different ways to run it and communicate with it from within Java / JVM applications or command line. At the very end of the tutorial we are going to talk about Elastic Stack to showcase the ecosystem around Elasticsearch and its amazing capabilities.

If you are a junior or seasoned Java / JVM developer and interested in learning about Elasticsearch, this tutorial is definitely for you.

1.2 Elasticsearch Basics

To get started, it would be great to answer the question: so, what is Elasticsearch, how it can help me and why should I use it?

Elasticsearch is a highly scalable open-source full-text search and analytics engine. It allows you to store, search, and analyze big volumes of data quickly and in near real time. It is generally used as the underlying engine/technology that powers applications that have complex search features and requirements. (https://www.elastic.co/)

Elasticsearch is built on top of Apache Lucene but favors communication over RESTful APIs and advanced in-depth analytics features. The RESTful part makes Elasticsearch particularly easy to learn and use. As of the moment of this writing, the latest stable release branch of the Elasticsearch was `5.2`, with the latest released version being `5.2.0`. We should definitely give Elasticsearch guys the credit for keeping the pace of delivering new releases so often, `5.0.x` /`5.1.x` branches are just a few months old ....

In perspective of Elasticsearch, being RESTful APIs has another advantage: every single piece of data sent to or received from Elasticsearch is itself a human-readable JSON document (although this is not the only protocol Elasticsearch supports as we are going to see later on).

To keep the discussion relevant and practical, we are going to pretend that we are developing the application to manage the catalog of books. The data model will include categories, authors, publisher, book details (like publishing date, ISBN, rating) and brief description.

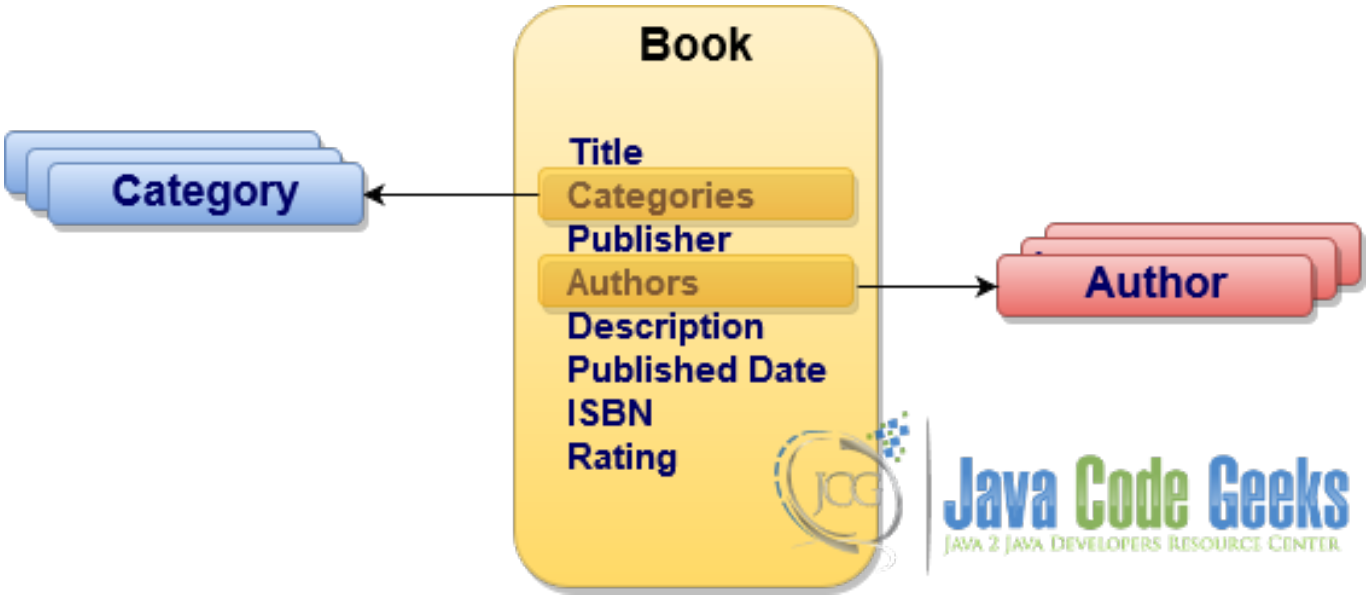

Figure 1.1: Book Catalog

Let us see how we could leverage Elasticsearch to make our book catalog easily searchable but before that we need to get familiarized a bit with the terminology. Although in the next couple of sections we are going to go over most of the concepts behind Elasticsearch, please do not hesitate to consult Elasticsearch official documentation any time.

1.2.1 Documents

To put it simply, in context of Elasticsearch document is just an arbitrary piece of data (usually, structured). It could be absolutely anything which makes sense to your applications (like users, logs, blog posts, articles, products, ...) but this is a basic unit of information which Elasticsearch could manipulate.

1.2.2 Indices

Elasticsearch stores documents inside indices and as such, an index is simply a collection of the documents. To be fair, persisting absolutely different kind of the documents in the same index would be somewhat convenient but quite difficult to work with so every index may have one or more types. The types group documents logically by defining a set of common properties (or fields) every document of such type should have. Types serve as a metadata about documents and are very useful for exploring the structure of the data and constructing meaningful queries and aggregations.

1.2.3 Index Settings

Each index in Elasticsearch could have specific settings associated with it at the time of its creation. The most important ones are number of shards and replication factor. Let us talk about that for a moment.

Elasticsearch has been built from the ground up to operate over massive amount of indexed data which will very likely exceed the memory and/or storage capabilities of a single physical (or virtual) machine instance. As such, Elasticsearch uses sharding as

a mechanism to split the index into several smaller pieces, called shards, and distribute them among many nodes. Please notice, once set the number of shards could not be altered (although this is not entirely true anymore, the index could be shrunk into fewer shards).

Indeed, sharding solves a real problem but it is vulnerable to data loss issues due to individual node failures. To address this problem, Elasticsearch supports high availability by leveraging replication. In this case, depending on replication factor, Elasticsearch maintains one or more copies of each shard and makes sure that each shard's replica is placed on different node.

1.2.4 Mappings

The process of defining the type of the documents and assigning it to a particular index is called index mapping, mapping type or just a mapping. Coming up with a proper type mapping is, probably, one of the most important design exercises you would have to make in order to get most out of Elasticsearch. Let us take some time and talk about mappings in details.

Each mapping consists of optional meta-fields (they usually start from the underscore character like `_index`, `_id`, `_parent`) and regular document fields (or properties). Each field (or property) has a **data type**, which in Elasticsearch could fall into one of those categories:

- Simple data types
  - text - indexes full-text values
  - keyword - indexes structured values
  - date - indexes date/time values
  - long - indexes signed 64-bit integer values
  - integer - indexes signed 32-bit integer values
  - short - indexes signed 16-bit integer values
  - byte - indexes signed 8-bit integer values
  - double - indexes double-precision 64-bit IEEE 754 floating point values
  - float - indexes single-precision 32-bit IEEE 754 floating point values
  - half_float - indexes half-precision 16-bit IEEE 754 floating point values
  - scaled_float - indexes floating point values that is backed by a long and a fixed scaling factor
  - boolean - indexes boolean values (for example, **true/false**, **on/off**, **yes/no**, **1/0**)
  - ip - indexes either IPv4 or IPv6 address values
  - binary - indexes any binary value encoded as a Base64 string
- Composite data types
  - object - indexes inner objects which, in turn, may contain inner objects themselves
  - nested - a specialized version of the object data type that allows to index arrays of objects independently of each other
- Specialized data type
  - geo_point - indexes latitude-longitude pairs
  - geo_shape - indexes an arbitrary geo shapes (such as rectangles and polygons)
  - completion - dedicated data type to back auto-complete/search-as-you-type functionality
  - token_count - dedicated data type to count the number of tokens in a string
  - percolator - specialized data type to store the query which is going to be used by percolate query to match the documents
- Range data types:
  - integer_range - indexes a range of signed 32-bit integers
  - float_range - indexes a range of single-precision 32-bit IEEE 754 floating point values

- long_range - indexes a range of signed 64-bit integers
- double_range - indexes a range of double-precision 64-bit IEEE 754 floating point values
- date_range - indexes a range of date values represented as unsigned 64-bit integer milliseconds elapsed since system epoch

Cannot stress it enough, choosing the proper data type for the fields (properties) of your documents is a key for fast, effective search which delivers really relevant results. There is one catch though: the fields in each mapping type are not entirely independent of each other. The fields with the **same name** and within the **same index** but in **different mapping types must have the same mapping definition**. The reason is that internally those fields are mapped to the **same field**.

Getting back to our application data model, let us try to define the simplest mapping type for `books` collections, utilizing our just acquired knowledge about data types.

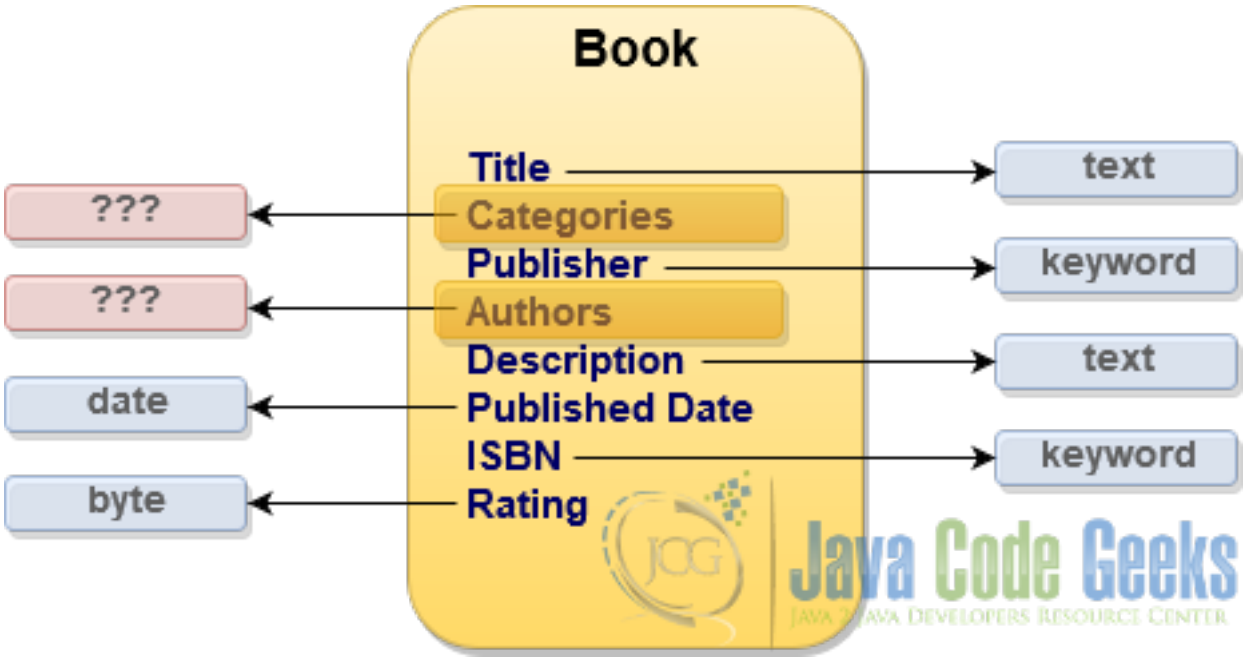

Figure 1.2: Mapping Book Catalog: first attempt

For most of the book properties the mapping data types are pretty straightforward but what about **authors** and `categories`? Those properties essentially contain the collection of values for which Elasticsearch has no direct data type yet, ... or has it?

1.2.5 Advanced Mappings

Interestingly, indeed Elasticsearch has no dedicated array or collection type but by default, any field may contain zero or more values (of its data type).

In case of complex data structures, Elasticsearch supports mapping using object and nested data types as well as establishing parent/child relationships between documents within the same index. There are pros and cons of each approach but in order to learn how to use those techniques let us store `categories` as nested property of the `books` mapping type, while `authors` are going to be represented as a dedicated mapping which refers to `books` as parent.

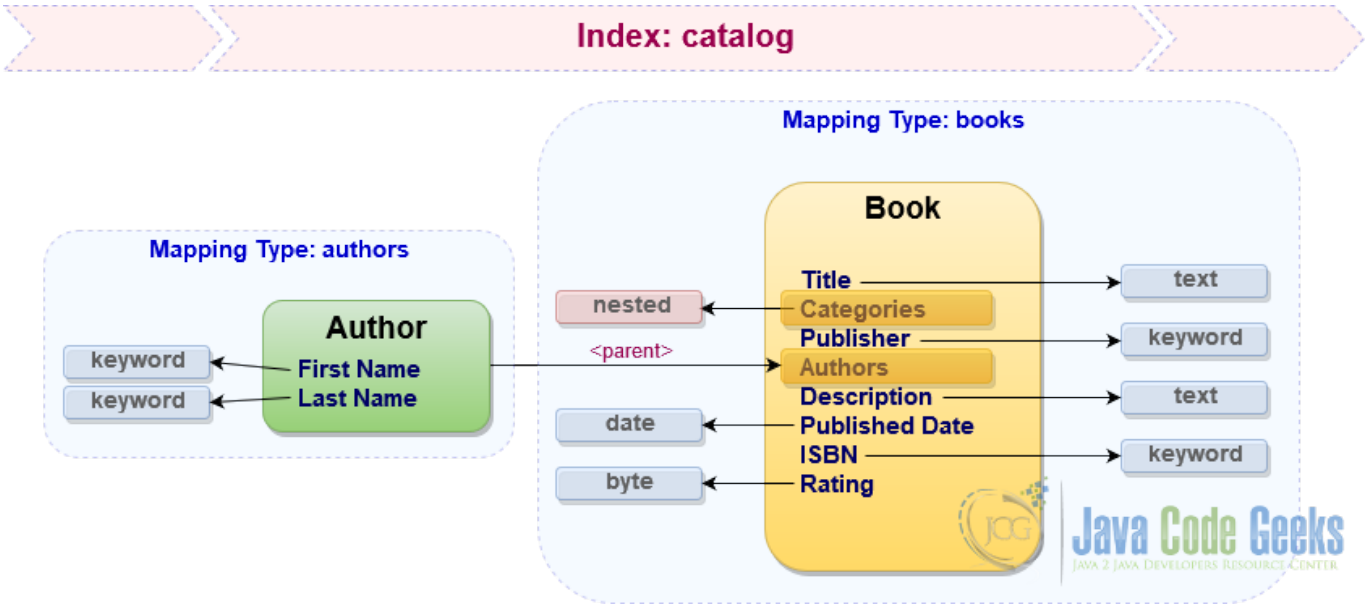

Figure 1.3: Mapping Book Catalog: second (and the last) attempt

These are our close to final mapping types for the `catalog` index. As we already know, JSON is a first class citizen in Elasticsearch, so let us get a feeling of how the typical index mapping looks like in the format Elasticsearch actually understands.

```
{
  "mappings": {
    "books": {
      "_source" : {
        "enabled": true
      },
      "properties": {
        "title": { "type": "text" },
        "categories" : {
          "type": "nested",
          "properties" : {
            "name": { "type": "text" }
          }
        },
        "publisher": { "type": "keyword" },
        "description": { "type": "text" },
        "published_date": { "type": "date" },
        "isbn": { "type": "keyword" },
        "rating": { "type": "byte" }
       }
    },
    "authors": {
      "properties": {
        "first_name": { "type": "keyword" },
        "last_name": { "type": "keyword" }
      },
      "_parent": {
         "type": "books"
       }
     }
  }
}
```

You may be surprised but explicit definition of the fields and mapping types could be omitted. Elasticsearch supports dynamic mapping thereby new mapping types and new field names will be added automatically when document is indexed (in this case Elasticsearch makes a decision what the field data types should be).

Another important detail to mention is that each mapping type can have custom metadata associated with it by using special `_meta` property. It is exceptionally useful technique which will be used by us later on in the tutorial.

1.2.6 Indexing

Once Elasticsearch has all your indices and their mapping types defined (or inferred using dynamic mapping), it is ready to analyze and index the documents. It is quite complex but interesting process which involves at least analyzers, tokenizers, token filters and character filters.

Elasticsearch supports quite a rich number of mapping parameters which let you tailor the indexing, analysis and search phases precisely to your needs. For example, every single field (or property) could be configured to use own index-time and search-time analyzers, support synonyms, apply stemming, filter out stop words and much, much more. By carefully crafting these parameters you may end up with superior search capabilities, however the opposite also holds true, having them loose, and a lot of irrelevant and noisy results may be returned every time.

If you don't need all that, you are good to go with the defaults as we have done in the previous section, omitting the parameters altogether. However, it is rarely the case. To give a realistic example, most of the time our applications have to support multiple languages (and locales). Luckily, Elasticsearch shines here as well.

Before we move on to the next topic, there is an important constraint you have to be aware of. Once the mapping types are configured, in majority of cases they cannot be updated as it automatically assumes that all the documents in the corresponding collections are not up to date anymore and should be re-indexed.

1.2.7 Internalization (i18n)

The process of indexing and analyzing the documents is very sensitive to the native language of the document. By default, Elasticsearch uses standard analyzer if none is specified in the mapping types. It works well for most of the languages but Elasticsearch supplies the dedicated analyzers for Arabic, Armenian, Basque, Brazilian, Bulgarian, Czech, Danish, Dutch, English, Finnish, French, German, Greek, Hindi, Hungarian, Indonesian, Irish, Italian, Latvian, Lithuanian, Norwegian, Persian, Portuguese, Romanian, Russian, Spanish, Swedish, Turkish, Thai and a few more.

There are couple of ways to approach the indexing of the same document in multiple languages, depending on your data model and business case. For example, if document instances physically exist (translated) in multiple languages, than it probably makes sense to have one index per language.

In case when documents are partially translated, Elasticsearch has another interesting option hidden in the sleeves called multi-fields. Multi-fields allow indexing the same document field (property) in different ways to be used for different purposes (like, for example, supporting multiple languages). Getting back to our `books` mapping type, we may have defined the `title` property as a multi-field one, for example:

```
"title": {
  "type": "text",
  "fields": {
    "en": { "type": "text", "analyzer": "english" },
    "fr": { "type": "text", "analyzer": "french" },
    "de": { "type": "text", "analyzer": "german" },
    ...
  }
}
```

Those are not the only options available but they illustrate well enough the flexibility and maturity of the Elasticsearch in fulfilling quite sophisticated demands.

1.3 Running Elasticsearch

Elasticsearch embraces simplicity in many ways and one of those is exceptionally easy way to get started on mostly any platform in just two steps: download and run. In the next couple of sections we are going to talk about quite a few different ways to get your Elasticsearch up and running.

1.3.1 Standalone Instance

Running Elasticsearch as a standalone application (or instance) is the fastest and simplest route to take. Just download the package of your choice and run the shell script on **Linux/Unix/Mac** operating systems:

```
bin/elasticsearch
```

Or from the batch file on **Windows** operating system:

```
bin\elasticsearch.bat
```

And that is it, pretty straightforward, isn't it? However, before we go ahead and talk about more advanced options, it would be useful to get a taste what it actually means to run an instance of Elasticsearch. To be more precise, every time we say we are starting the instance of Elasticsearch, we are actually starting an instance of a `node`. As such, depending on provided configuration (by default, it is stored in `conf/elastisearch.yml` file) there are multiple node types which Elasticsearch supports at the moment. In this regards, every running standalone instance of Elasticsearch could be configured to run as one (or combination) of those node types:

- data node: these kind of nodes are maintaining the data and performing operations over this data (it is controlled by `node.data` configuration setting which is set to `true` by default)
- ingest node: these are special kind of nodes which are able to apply an ingest pipeline in order to transform and enrich the document before indexing it (it is controlled by `node.ingest` configuration setting which is set to `true` by default)

Please take a note that this is not an exhaustive list of node types yet, we are going to learn quite a few more in just a moment.

1.3.2 Clustering

Running Elasticsearch as a standalone instance is good for development, learning or testing purposes but certainly is not an option for production systems. Generally, in most real-world deployments Elasticsearch is configured to run in a cluster: a collection of one or more nodes preferably split across multiple physical instances. Elasticsearch cluster manages all the data and also provides federated indexing, aggregations and search capabilities across all its nodes.

Every Elasticsearch cluster is identified by a unique name which is controlled by `cluster.name` configuration setting (set to `"elasticsearch"` by default). The nodes are joining the cluster by referring to its name so it is quite important piece of configuration. Last but not least, each cluster has a dedicated master node which is responsible for performing cluster-wide actions and operations.

Specifically applicable to the clustered configuration, Elasticsearch supports a couple of more node types, in addition to the ones we already know about:

- master-eligible node: these kind of nodes are marked as eligible to be elected as the master node (it is controlled by `node.master` configuration setting which is set to `true` by default)
- coordinating-only node: these are special kind of nodes which are able to only route requests, handle some search phases, and distribute bulk indexing, essentially behaving as load balancers (the node automatically becomes coordinating-only when `node.master`, `node.data` and `node.ingest` settings are all set to `false`)
- tribe node: these are special kind of coordinating-only nodes that can connect to multiple clusters and execute search or other operations across all of them (it is controlled by `tribe.*` configuration settings)

By default, if configuration is not specified, each Elasticsearch node is configured to be master-eligible, data node and ingest node. Similarly to the standalone instance, Elasticsearch cluster instances could be started quickly from the command line:

```
bin/elasticsearch -Ecluster.name=<cluster-name> -Enode.name=<node-name>
```

Or on Windows platform:

```
bin\elasticsearch.bat -Ecluster.name=<cluster-name> -Enode.name=<node-name>
```

Along with sharding and replication, an Elasticsearch cluster has all the properties of a highly available and scalable system which will organically evolve to meet the needs of your applications. To be noted, despite significant efforts invested into stabilizing Elasticsearch clustering implementation and covering a lot of edge cases related to a different kind of failure scenarios, as of now Elasticsearch is still not recommended to serve as a system of record (or primary storage engine of your data).

1.3.3 Embedding Into Application

Not a long time ago (right till `5.0` release branch) Elasticsearch fully supported the option to be run as part of the application, within the same JVM process (the technique commonly referred as embedding). Although it is certainly not a recommended practice, sometimes it was very useful and saves a lot of effort, for example during integration / system / component test runs.

The situation has changed recently and embedded version of the Elasticsearch is not officially supported nor recommended anymore. Luckily, in case you really need the embedded instance, for example while slowly migrating from older Elasticsearch releases, it is still possible.

```
@Configuration
public class ElasticsearchEmbeddedConfiguration {
  private static class EmbeddedNode extends Node {
    public EmbeddedNode(Settings preparedSettings) {
      super(
          InternalSettingsPreparer.prepareEnvironment(preparedSettings, null),
          Collections.singletonList(Netty4Plugin.class)
        );
    }
  }

  @Bean(initMethod = "start", destroyMethod = "stop")
  Node elasticSearchTestNode() throws NodeValidationException, IOException {
    return new EmbeddedNode(
      Settings
        .builder()
        .put(NetworkModule.TRANSPORT_TYPE_KEY, "netty4")
        .put(NetworkModule.HTTP_TYPE_KEY, "netty4")
        .put(NetworkModule.HTTP_ENABLED.getKey(), "true")
        .put(Environment.PATH_HOME_SETTING.getKey(), home().getAbsolutePath())
        .put(Environment.PATH_DATA_SETTING.getKey(), data().getAbsolutePath())
        .build());
  }

  @Bean
  File home() throws IOException {
    return Files.createTempDirectory("elasticsearch-home-").toFile();
  }

  @Bean
  File data() throws IOException {
    return Files.createTempDirectory("elasticsearch-data-").toFile();
  }

  @PreDestroy
  void destroy() throws IOException {
    FileSystemUtils.deleteRecursively(home());
```

```
    FileSystemUtils.deleteRecursively(data());
  }
}
```

Although this code snippet is based on the terrific Spring Framework, the idea is pretty simple and could be used in any JVM-based application. With that being said, be warned though and reconsider the long-term solution without the need to embed Elasticsearch.

1.3.4 Running As Container

The rise of such tools as Docker, CoreOS and tremendous popularization of the containers and container-based deployments significantly changed our thinking about infrastructure and, in many cases, the development approaches as well.

To say it in other words, there is no need to download Elasticsearch and run it using the shell scripts or batch files. Everything is the container and could be pulled, configured and run using the single `docker` command (thankfully, there is an official Elasticseach Dockerhub repository).

Assuming you have Docker installed on your machine, let us run the single Elasticsearch instance relying on the default configuration:

```
docker run -d -p 9200:9200 -p 9300:9300 elasticsearch:5.2.0
```

Spinning an Elasticsearch cluster is a little bit more complicated but certainly much easier than doing that manually using shell scripts. By and large, Elasticsearch cluster needs multicast support in order for nodes to auto-discover each other, but with Docker you would need to fall back to unicast discovery unfortunately (unless you have subscription to unlock commercial features).

```
docker run -d -p 9200:9200 -p 9300:9300 --name es1 elasticsearch:5.2.0 -E cluster.name=es- ↩
    catalog -E node.name=es1 -E transport.host=0.0.0.0
```

```
docker run -d --name es2 --link=es1 elasticsearch:5.2.0 -E cluster.name=es-catalog -E node. ↩
    name=es2 -E transport.host=0.0.0.0 -E discovery.zen.ping.unicast.hosts=es1
```

```
docker run -d --name es3 --link=es1 elasticsearch:5.2.0 -E cluster.name=es-catalog -E node. ↩
    name=es3 -E transport.host=0.0.0.0 -E discovery.zen.ping.unicast.hosts=es1
```

Once the containers are started, the cluster of three Elasticsearch nodes should be created, with master node accessible at `https://localhost:9200` (in case of native Docker support). If for some reasons you are still on Docker Machine (or even older boot2docker), the master node will be exposed at `https://<docker-machine-ip>:9200` respectively.

If you are actively using Docker Compose, there are certain limitations which will complicate your life at this point. At the moment Elasticsearch images need some arguments to be passed to the entry point (everything you see at the end of the command line as `-E` option) however such feature is not supported by Docker Compose yet (although you may build your own images as a workaround).

Along this tutorial we are going to use only Elasticsearch started as Docker containers, hopefully this is something you have already adopted a long time ago.

1.4 Where Elasticsearch Fits

Search is one of the key features of the Elasticsearch and it does that exceptionally well. But Elasticsearch goes well beyond just search and provides a rich analytics capabilities shaped as aggregations framework which does data aggregations based on a search query. In case you would need to do some analytics around your data, Elasticsearch is a great fit here as well.

Although it may be not immediately apparent, Elasticsearch could be used to manage time series data (for example, metrics, stock prices) and even back search for images. One of the misconceptions about Elasticsearch is that it could be used as a data store. In some extent it is true, it does store the data however it does not provide the same guarantees yet you would expect from the typical data store.

1.5 Conclusion

Although we have talked about many things here, tons of interesting details and useful features of the Elasticsearch have not been covered at all yet. Our focus was kept on development side of things, and as such, the emphasis has been made on understanding the basics of Elasticsearch and starting off quickly. Hopefully, you are already thrilled and excited enough to start reading the official documentation reference right away as more interesting topics are on the way.

1.6 What's next

In the next section we are going to jump right from the discussions into the actions by exploring and playing with the myriads of RESTful APIs exposed by Elasticsearch, armed only with command line and the brilliant curl / http tools.

The source code for this post is available here for download.

Chapter 2

Elasticsearch from the command line

2.1 Introduction

From the previous part of the tutorial we have got a pretty good understanding of what Elasticsearch is, its basic concepts and the power of search capabilities it could bring to our applications. In this section we are jumping right into the battle and going to apply our knowledge in practice. Along this section curl and/or http would be the only tools we are going to use to make friends with Elasticsearch.

To sum up, we have already finalized our book `catalog` index and mapping types so we are going to pick it up from there. In order to keep things as close to reality as possible, we are going to use Elasticsearch cluster with three nodes (all run as Docker containers), while `catalog` index is going to be configured with replication factor of two.

As we are going to see, working with Elasticsearch cluster has quite a few subtleties comparing to standalone instance and it is better to be prepared to deal with them. Hopefully, you still remember from the previous part of the tutorial how to start Elasticsearch as this is going to be the only prerequisite: having the cluster up and running. With that, let us get started!

2.2 Is My Cluster Healthy?

The first thing you would need to know about your Elasticsearch cluster before doing anything with it is its health. There are a couple of ways to gather this information but arguably the easiest and most convenient one is by using Cluster APIs, particularly cluster health endpoint.

```
$ http https://localhost:9200/_cluster/health

HTTP/1.1 200 OK
content-encoding: gzip
content-type: application/json; charset=UTF-8
transfer-encoding: chunked

{
    "active_primary_shards": 0,
    "active_shards": 0,
    "active_shards_percent_as_number": 100.0,
    "cluster_name": "es-catalog",
    "delayed_unassigned_shards": 0,
    "initializing_shards": 0,
    "number_of_data_nodes": 3,
    "number_of_in_flight_fetch": 0,
    "number_of_nodes": 3,
    "number_of_pending_tasks": 0,
    "relocating_shards": 0,
    "status": "green",
```

```
    "task_max_waiting_in_queue_millis": 0,
    "timed_out": false,
    "unassigned_shards": 0
}
```

Among those details we are looking for `status` indicator which should be set to `green`, meaning that that all shards are allocated and cluster is in a good operational shape.

2.3 All About Indices

Our Elasticsearch cluster is all green and ready to rock. The next logical step would be to create a `catalog` index, with the mapping types and settings we have outlined before. But before doing that, let us check if there are any indices already created this time using Indices APIs.

```
$ http https://localhost:9200/_stats

HTTP/1.1 200 OK
content-encoding: gzip
content-type: application/json; charset=UTF-8
transfer-encoding: chunked

{
    "_all": {
        "primaries": {},
        "total": {}
    },
    "_shards": {
        "failed": 0,
        "successful": 0,
        "total": 0
    },
    "indices": {}
}
```

As expected, our cluster has nothing in it yet and we are good to go with creating the index for our book catalog. As we know, Elasticsearch speaks JSON but manipulating ald be said about usag more or less complex JSON document from the command line is somewhat cumbersome. Let us better store the `catalog` settings and mappings in the catalog-index.json document.

```
{
  "settings": {
    "index" : {
      "number_of_shards" : 5,
      "number_of_replicas" : 2
    }
  },
  "mappings": {
    "books": {
      "_source" : {
        "enabled": true
      },
      "properties": {
        "title": { "type": "text" },
        "categories" : {
          "type": "nested",
          "properties" : {
            "name": { "type": "text" }
          }
        },
        "publisher": { "type": "keyword" },
```

```
        "description": { "type": "text" },
        "published_date": { "type": "date" },
        "isbn": { "type": "keyword" },
        "rating": { "type": "byte" }
      }
    },
    "authors": {
      "properties": {
        "first_name": { "type": "keyword" },
        "last_name": { "type": "keyword" }
      },
      "_parent": {
         "type": "books"
       }
     }
   }
}
```

And use this document as an input to create an index API.

```
$ http PUT https://localhost:9200/catalog < catalog-index.json

HTTP/1.1 200 OK
content-encoding: gzip
content-type: application/json; charset=UTF-8
transfer-encoding: chunked

{
    "acknowledged": true,
    "shards_acknowledged": true
}
```

A few words should be said about the usage of `acknowledged` response property across most of the Elasticsearch APIs, especially the ones which apply mutations. In general, this value simply indicates whether the operation completed before the timeout (`true`) or may take an effect sometime soon (`false`). We are going to see more examples of its usage in a different context later on.

That is it and we have brought our `catalog` index live. To ensure the truthiness of this fact, we could ask Elasticsearch to return `catalog` index settings.

```
$ http https://localhost:9200/catalog/_settings

HTTP/1.1 200 OK
content-encoding: gzip
content-type: application/json; charset=UTF-8
transfer-encoding: chunked

{
    "catalog": {
        "settings": {
            "index": {
                "creation_date": "1487428863824",
                "number_of_replicas": "2",
                "number_of_shards": "5",
                "provided_name": "catalog",
                "uuid": "-b63dCesROC5UawbHz8IYw",
                "version": {
                    "created": "5020099"
                }
            }
        }
    }
```

```
}
```

Awesome, exactly what we have ordered. You might wonder how Elasticsearch would react if we would have tried to update the index settings by increasing the number of shards (as we know, not all index settings could be updated once index has been created).

```
$ echo '{"index":{"number_of_shards":6}}' | http PUT https://localhost:9200/catalog/ ↩
    _settings

HTTP/1.1 400 Bad Request
content-encoding: gzip
content-type: application/json; charset=UTF-8
transfer-encoding: chunked

{
    "error": {
        "reason": "can't change the number of shards for an index",
        "root_cause": [
            ...
        ],
        "type": "illegal_argument_exception"
    },
    "status": 400
}
```

The error response comes as no surprise (please notice that the response details have been reduced for illustrative purposes only). Along with settings, it is very easy to get the mapping types for a particular index, for example:

```
$ http https://192.168.99.100:9200/catalog/_mapping

HTTP/1.1 200 OK
content-encoding: gzip
content-type: application/json; charset=UTF-8
transfer-encoding: chunked

{
    "catalog": {
        "mappings": {
            "authors": {
                ...
            },
            "books": {
                ...
            }
        }
    }
}
```

By and large, the index mappings for existing fields cannot be updated; however there are some exceptions of the rule. One of the greatest features of the indices APIs is the ability to perform the analysis process against a particular index mapping type and field without actually sending any documents.

```
$ http https://localhost:9200/catalog/_analyze field=books.title text="Elasticsearch: The  ↩
    Definitive Guide. A Distributed Real-Time Search and Analytics Engine"

HTTP/1.1 200 OK
content-encoding: gzip
content-type: application/json; charset=UTF-8
transfer-encoding: chunked

{
```

```
    "tokens": [
        {
            "end_offset": 13,
            "position": 0,
            "start_offset": 0,
            "token": "elasticsearch",
            "type": ""
        },
        {
            "end_offset": 18,
            "position": 1,
            "start_offset": 15,
            "token": "the",
            "type": ""
        },

        ...

        {
            "end_offset": 88,
            "position": 11,
            "start_offset": 82,
            "token": "engine",
            "type": ""
        }
    ]
}
```

It is exceptionally useful in case you would like to validate your mapping types' parameters before throwing a bunch of data into Elasticsearch for indexing.

And last but not least, there is one important detail about index states. Any particular index could be in `opened` (fully operational) or `closed` (blocked for read/write operations, archived would be a good analogy) states. As for everything else, Elasticsearch provided an APIs for that.

```
$ http POST https://localhost:9200/catalog/_open

HTTP/1.1 200 OK
content-encoding: gzip
content-type: application/json; charset=UTF-8
transfer-encoding: chunked

{
    "acknowledged": true
}
```

2.4 Documents, More Documents, . . .

The empty index without documents is not very useful so let us switch gears from indices APIs to another great one, document APIs. We are going to start exploring it using the simplest single document operations, relying on the following book.json document:

```
{
  "title": "Elasticsearch: The Definitive Guide. A Distributed Real-Time Search and ↩
      Analytics Engine",
  "categories": [
      { "name": "analytics" },
      { "name": "search" },
      { "name": "database store" }
```

```
  ],
  "publisher": "O'Reilly",
  "description": "Whether you need full-text search or real-time analytics of structured ↩
      data-or both-the Elasticsearch distributed search engine is an ideal way to put your ↩
      data to work. This practical guide not only shows you how to search, analyze, and ↩
      explore data with Elasticsearch, but also helps you deal with the complexities of ↩
      human language, geolocation, and relationships.",
  "published_date": "2015-02-07",
  "isbn": "978-1449358549",
  "rating": 4
}
```

Before sending this JSON to Elasticsearch, it would be great to talk a little bit about documents identification. Each document in Elasticsearch has a unique identifier, stored in a special `_id` field. You may provide one while uploading the document to Elasticsearch (like we do in the example below using `isbn` as it is a great example of natural identifier), or it will be generated and assigned by Elasticsearch.

```
$ http PUT https://localhost:9200/catalog/books/978-1449358549 < book.json

HTTP/1.1 201 Created
Location: /catalog/books/978-1449358549
content-encoding: gzip
content-type: application/json; charset=UTF-8
transfer-encoding: chunked

{
    "_id": "978-1449358549",
    "_index": "catalog",
    "_shards": {
        "failed": 0,
        "successful": 3,
        "total": 3
    },
    "_type": "books",
    "_version": 1,
    "created": true,
    "result": "created"
}
```

Our first document just made its way into a `catalog` index, under `books` type. But we also have `authors` type, which is in a parent / child relationship with `books`. Let us complement the book with its authors from authors.json document.

```
[
  {
    "first_name": "Clinton",
    "last_name": "Gormley",
    "_parent": "978-1449358549"
  },
  {
    "first_name": "Zachary",
    "last_name": "Tong",
    "_parent": "978-1449358549"
  }
]
```

The book has more than one author so we still can use the single document API by indexing each author document one by one. However, let us not do that but switch over to bulk document API instead and transform our authors.json document a bit to be compatible with bulk document API format.

```
{ "index" : { "_index" : "catalog", "_type" : "authors", "_id": "1", "_parent": " ↩
    978-1449358549" } }
```

```
{ "first_name": "Clinton", "last_name": "Gormley" }
{ "index" : { "_index" : "catalog", "_type" : "authors", "_id": "2", "_parent": " ↩
    978-1449358549" } }
{ "first_name": "Zachary", "last_name": "Tong" }
```

Done deal, let us save this document as authors-bulk.json and feed it directly into bulk document API endpoint.

```
$ http POST https://localhost:9200/_bulk < authors-bulk.json

HTTP/1.1 200 OK
content-encoding: gzip
content-type: application/json; charset=UTF-8
transfer-encoding: chunked

{
    "errors": false,
    "items": [
        {
            "index": {
                "_id": "1",
                "_index": "catalog",
                "_shards": {
                    "failed": 0,
                    "successful": 3,
                    "total": 3
                },
                "_type": "authors",
                "_version": 5,
                "created": false,
                "result": "updated",
                "status": 200
            }
        },
        {
            "index": {
                "_id": "2",
                "_index": "catalog",
                "_shards": {
                    "failed": 0,
                    "successful": 3,
                    "total": 3
                },
                "_type": "authors",
                "_version": 2,
                "created": true,
                "result": "created",
                "status": 201
            }
        }
    ],
    "took": 105
}
```

And we have book and author documents as the first citizens of the `catalog` index! It is time to fetch those documents back.

```
$ http https://localhost:9200/catalog/books/978-1449358549

HTTP/1.1 200 OK
content-encoding: gzip
content-type: application/json; charset=UTF-8
transfer-encoding: chunked
```

```
{
    "_id": "978-1449358549",
    "_index": "catalog",
    "_source": {
        "categories": [
            { "name": "analytics" },
            { "name": "search"},
            { "name": "database store" }
        ],
        "description": "...",
        "isbn": "978-1449358549",
        "published_date": "2015-02-07",
        "publisher": "O'Reilly",
        "rating": 4,
        "title": "Elasticsearch: The Definitive Guide. A Distributed Real-Time Search and ↩
            Analytics Engine"
    },
    "_type": "books",
    "_version": 1,
    "found": true
}
```

Easy! However to fetch the documents from `authors` collection, which are children of their respective documents from `books` collection, we have to supply the parent identifier along with the document own one, for example:

```
$ http https://localhost:9200/catalog/authors/1?parent=978-1449358549

HTTP/1.1 200 OK
content-encoding: gzip
content-type: application/json; charset=UTF-8
transfer-encoding: chunked

{
    "_id": "1",
    "_index": "catalog",
    "_parent": "978-1449358549",
    "_routing": "978-1449358549",
    "_source": {
        "first_name": "Clinton",
        "last_name": "Gormley"
    },
    "_type": "authors",
    "_version": 1,
    "found": true
}
```

This is one of the specifics working with parent / child relations in Elasticsearch. As it has been already mentioned, you may model such relationships in a simpler way but our goal is to learn how to deal with it if you choose to go this route in your applications.

The delete and update APIs are pretty straightforward so we just leaf them through, please notice that the same rules regarding identifying the child documents apply. You may be surprised, but deleting a parent document does not automatically delete its children, so keep that in mind. We are going to see how to workaround that a bit later.

To finish up, let us take a look at the term vectors API which returns all the details and statistics about terms in the fields of the document, for example (only the small part of the response has been pasted):

```
$ http https://localhost:9200/catalog/books/978-1449358549/_termvectors?fields=description

HTTP/1.1 200 OK
content-encoding: gzip
content-type: application/json; charset=UTF-8
```

```
transfer-encoding: chunked

{
    "_id": "978-1449358549",
    "_index": "catalog",
    "_type": "books",
    "_version": 1,
    "found": true,
    "term_vectors": {
        "description": {
            "field_statistics": {
                "doc_count": 1,
                "sum_doc_freq": 46,
                "sum_ttf": 60
            },
            "terms": {
                "analyze": {
                    "term_freq": 1,
                    "tokens": [ ... ]
                },
                "and": {
                    "term_freq": 2,
                    "tokens": [ ... ]

                },
                "complexities": {
                    "term_freq": 1,
                    "tokens": [ ... ]

                },
                "data": {
                    "term_freq": 3,
                    "tokens": [ ... ]

                },
                ...
            }
        }
    },
    "took": 5
}
```

You may not find yourself using the term vectors API often however it is a terrific tool to troubleshoot why certain documents may not pop up in the search results.

2.5 What if My Mapping Types Are Suboptimal

Very often over time you may discover that your mapping types may not be optimal and could be made better. However, Elasticsearch supports only limited modifications over existing mapping types. Luckily, Elasticsearch is providing a dedicated reindexing API, for example:

```
$ echo '{"source": {"index": "catalog"}, "dest": {"index": "catalog-v2"}}' | http POST  ↩
    https://localhost:9200/_reindex

HTTP/1.1 200 OK
content-encoding: gzip
content-type: application/json; charset=UTF-8
transfer-encoding: chunked

{
```

```
    "batches": 0,
    "created": 200,
    "deleted": 0,
    "failures": [],
    "noops": 0,
    "requests_per_second": -1.0,
    "retries": {
        "bulk": 0,
        "search": 0
    },
    "throttled_millis": 0,
    "throttled_until_millis": 0,
    "timed_out": false,
    "took": 265,
    "total": 200,
    "updated": 0,
    "version_conflicts": 0
}
```

The trick here is to create a new index with updated mapping types, `catalog-v2`, and than just ask Elasticsearch to fetch all documents from old index (`catalog`) and put them into the new one (`catalog-v2`), and finally swap the indices. Please notice, it also works not only for local but remote indices as well.

Although simple, this API is still considered an experimental and may not be suitable in all cases, for example if your index is really massive or your Elasticsearch is experiencing a high load and should prioritize the application requests.

2.6 The Search Time

We have learned how to create indexes, mapping types and index the documents, all important but not really exciting topics. But search is definitely the heart and soul of Elasticsearch, so let us get to know it right away.

In order to demonstrate different search features we would need a couple of more documents, please upload them to your Elasticsearch cluster from books-and-authors-bulk.json using our friend bulk document API.

```
$ http POST https://localhost:9200/_bulk < books-and-authors-bulk.json
```

Having a few documents in our collections we could start off to issue search queries against them using the most accessible form of search API which accepts search criteria in URI by means of query string. For example, let us search for a term `engine` (keeping in mind `search engine` phrase).

```
$ http POST https://localhost:9200/catalog/books/_search?q=engine

HTTP/1.1 200 OK
content-encoding: gzip
content-type: application/json; charset=UTF-8
transfer-encoding: chunked

{
    "_shards": {
        "failed": 0,
        "successful": 5,
        "total": 5
    },
    "hits": {
        "hits": [
            {
                "_id": "978-1449358549",
                "_index": "catalog",
                "_score": 0.7503276,
                "_source": {
```

```
                    "categories": [
                        { "name": "analytics },
                        { "name": "search" },
                        { "name": "database store" }
                    ],
                    "description": " Whether you need full-text search or real-time ...",
                    "isbn": "978-1449358549",
                    "published_date": "2015-02-07",
                    "publisher": "O'Reilly",
                    "rating": 4,
                    "title": " Elasticsearch: The Definitive Guide. ..."
                },
                "_type": "books"
            }
        ],
        "max_score": 0.7503276,
        "total": 1
    },
    "timed_out": false,
    "took": 22
}
```

Good starting point indeed, this API is quite useful for doing quick and shallow searches, but its capabilities are very limited. The search using request body API is a very different beast and reveals the full power of Elasticsearch. It is built on top of JSON-based Query DSL, the concise and intuitive language to construct arbitrarily complex search queries.

There are quite a few query types which Query DSL allows to describe, each have own syntax and parameters. However, there is a set of common parameters, like sort, from, size, stored_fields (the list is really long actually) which are agnostic to query type and could be applied for any of those.

For the next couple of sections we are going to switch over from http to curl as the latter is a bit more convenient when dealing with JSON payloads.

The first query type we are going to try out using Query DSL is the match all query. In some extent, it is not really a query because it just matches all documents. As such, it may return a lot of results and as a general rule, please always annotate your queries with reasonable size limit, here is an example:

```
$ curl -i https://localhost:9200/catalog/books/_search?pretty -d '
{
    "size": 10,
    "query": {
        "match_all" : {
        }
    }
}'

HTTP/1.1 200 OK
content-type: application/json; charset=UTF-8
content-length: 3112
{
  "took" : 13,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 3,
    "max_score" : 1.0,
    "hits" : [
      {
        "_index" : "catalog",
```

```
        "_type" : "books",
        "_id" : "978-1449358549",
        "_score" : 1.0,
        "_source" : {
          "title" : "Elasticsearch: The Definitive Guide ...",
          "categories" : [
            { "name" : "analytics" },
            { "name" : "search" },
            { "name" : "database store" }
          ],
          "publisher" : "O'Reilly",
          "description" : "Whether you need full-text ...",
          "published_date" : "2015-02-07",
          "isbn" : "978-1449358549",
          "rating" : 4
        }
      },
      ...
    ]
  }
}
```

The next one is a real query type and is referred as a class of full text queries which do a search against full text document fields (probably the most widely used ones). In the basic form it does a match against a single document field, like for example book's `description`.

```
$ curl -i https://localhost:9200/catalog/books/_search?pretty -d '
{
    "query": {
        "match" : {
            "description" : "engine"
        }
    }
}'

HTTP/1.1 200 OK
content-type: application/json; charset=UTF-8
content-length: 1271
{
  "took" : 17,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 0.28004453,
    "hits" : [
      {
        "_index" : "catalog",
        "_type" : "books",
        "_id" : "978-1449358549",
        "_score" : 0.28004453,
        "_source" : {
          "title" : "Elasticsearch: The Definitive Guide. ...",
          "categories" : [
            { "name" : "analytics" },
            { "name" : "search" },
            { "name" : "database store" }
          ],
```

```
          "publisher" : "O'Reilly",
          "description" : "Whether you need full-text ...",
          "published_date" : "2015-02-07",
          "isbn" : "978-1449358549",
          "rating" : 4
        }
      }
    ]
  }
}
```

But the full text queries are very powerful and have quite a few other variations, including match_phrase, match_phrase_prefix, multi_match, common_terms, query_string and simple_query_string.

Moving on, we are entering the world of term level queries which operate on the exact terms and usually used for field types like numbers, dates, and keywords. The `publisher` book field is a good candidate to try it out.

```
$ curl -i https://localhost:9200/catalog/books/_search?pretty -d '
{
   "size": 10,
   "_source": [ "title" ],
   "query": {
        "term" : {
            "publisher" : "Manning"
        }
    }
}'

HTTP/1.1 200 OK
content-type: application/json; charset=UTF-8
content-length: 675

{
  "took" : 21,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 2,
    "max_score" : 0.18232156,
    "hits" : [
      {
        "_index" : "catalog",
        "_type" : "books",
        "_id" : "978-1617291623",
        "_score" : 0.18232156,
        "_source" : {
          "title" : "Elasticsearch in Action"
        }
      },
      {
        "_index" : "catalog",
        "_type" : "books",
        "_id" : "978-1617292774",
        "_score" : 0.18232156,
        "_source" : {
          "title" : "Relevant Search: With applications ..."
        }
      }
```

```
    ]
  }
}
```

Please notice how we have limited the properties of document's _source to return `title` field only. The other variations of term level queries include terms, range, exists, prefix, wildcard, regexp, fuzzy, type and ids.

Joining queries are exceptionally interesting ones in the context of our book `catalog` index. Those queries allow to perform the search against nested objects or documents with parent/child relationship. For example, let us find out all the books in the `analytics` category.

```
$ curl -i https://localhost:9200/catalog/books/_search?pretty -d '
{
   "size": 10,
   "_source": [ "title", "categories" ],
   "query": {
        "nested": {
            "path": "categories",
            "query" : {
                "match": {
                    "categories.name" : "analytics"
                }
            }
       }
    }
}'

HTTP/1.1 200 OK
content-type: application/json; charset=UTF-8
content-length: 1177

{
  "took" : 45,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 2,
    "max_score" : 1.3112576,
    "hits" : [
      {
        "_index" : "catalog",
        "_type" : "books",
        "_id" : "978-1617291623",
        "_score" : 1.3112576,
        "_source" : {
          "categories" : [
            { "name" : "analytics" },
            { "name" : "search" },
            { "name" : "database store" }
          ],
          "title" : "Elasticsearch in Action"
        }
      },
      {
        "_index" : "catalog",
        "_type" : "books",
        "_id" : "978-1449358549",
        "_score" : 1.0925692,
```

```
        "_source" : {
          "categories" : [
            { "name" : "analytics" },
            { "name" : "search" },
            { "name" : "database store" }
          ],
          "title" : "Elasticsearch: The Definitive Guide ..."
        }
      }
    ]
  }
}
```

Similarly, we could have searched for all the books authored by Clinton Gormley, leveraging parent/child relationships between `books` and `authors` collections.

```
$ curl -i https://localhost:9200/catalog/books/_search?pretty -d '
{
   "size": 10,
   "_source": [ "title" ],
   "query": {
       "has_child" : {
            "type" : "authors",
            "inner_hits" : {
                "size": 5
            },
            "query" : {
                "term" : {
                    "last_name" : "Gormley"
                }
            }
        }
    }
}'

HTTP/1.1 200 OK
content-type: application/json; charset=UTF-8
content-length: 1084

{
  "took" : 38,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 1.0,
    "hits" : [
      {
        "_index" : "catalog",
        "_type" : "books",
        "_id" : "978-1449358549",
        "_score" : 1.0,
        "_source" : {
          "title" : "Elasticsearch: The Definitive Guide ..."
        },
        "inner_hits" : {
          "authors" : {
            "hits" : {
```

```
            "total" : 1,
            "max_score" : 0.6931472,
            "hits" : [
              {
                "_type" : "authors",
                "_id" : "1",
                "_score" : 0.6931472,
                "_routing" : "978-1449358549",
                "_parent" : "978-1449358549",
                "_source" : {
                  "first_name" : "Clinton",
                  "last_name" : "Gormley"
                }
              }
            ]
          }
        }
      }
    ]
  }
}
```

Please pay attention to the presence of inner_hits query parameter which let the search results to include the inner documents that matched the joining criteria.

The other query types like Geo queries, specialized queries and span queries work in a very similar way so we would just skip over them and finish up by looking into composite queries. The examples we have seen so far included the queries with only one search criteria but Query DSL has a way to construct the compound queries as well. Let us take a look at the example of using bool query which is the composition of some of the query types we have seen already.

```
$ curl -i https://localhost:9200/catalog/books/_search?pretty -d '
{
   "size": 10,
   "_source": [ "title", "publisher" ],
   "query": {
       "bool" : {
          "must" : [
              {
                  "range" : {
                      "rating" : { "gte" : 4 }
                  }
              },
              {
                  "has_child" : {
                      "type" : "authors",
                      "query" : {
                          "term" : {
                              "last_name" : "Gormley"
                          }
                      }
                  }
              },
              {
                  "nested": {
                      "path": "categories",
                      "query" : {
                          "match": {
                              "categories.name" : "search"
                          }
                      }
                  }
```

```
            }
          ]
        }
    }
}'

HTTP/1.1 200 OK
content-type: application/json; charset=UTF-8
content-length: 531

{
  "took" : 79,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 1,
    "max_score" : 3.0925694,
    "hits" : [
      {
        "_index" : "catalog",
        "_type" : "books",
        "_id" : "978-1449358549",
        "_score" : 3.0925694,
        "_source" : {
          "publisher" : "O'Reilly",
          "title" : "Elasticsearch: The Definitive Guide.  ..."
        }
      }
    ]
  }
}
```

It would be fair to say that the search API of Elasticsearch powered by Query DSL is exceptionally flexible, easy to use and expressive. More to that, it is worth to mention that additionally to queries the search API also supports the concept of filters which offer yet another option to exclude the document from the search results.

2.7 Mutations by Query

Surprisingly (or not), queries could be used by Elasticsearch to perform mutations like updates or deletes over the documents in the index. For example, the following snippet will remove all the books which have low rating in our catalog, published by `Manning`.

```
$ curl -i https://localhost:9200/catalog/books/_delete_by_query?pretty -d '
{
   "query": {
      "bool": {
          "must": [
              { "range" : { "rating" : { "lt" : 3 } } }
          ],
          "filter": [
             { "term" :  { "publisher" : "Manning" } }
          ]
      }
   }
}'
```

```
HTTP/1.1 200 OK
content-type: application/json; charset=UTF-8
content-length: 296

{
  "took" : 12,
  "timed_out" : false,
  "total" : 0,
  "deleted" : 0,
  "batches" : 0,
  "version_conflicts" : 0,
  "noops" : 0,
  "retries" : {
    "bulk" : 0,
    "search" : 0
  },
  "throttled_millis" : 0,
  "requests_per_second" : -1.0,
  "throttled_until_millis" : 0,
  "failures" : [ ]
}
```

It uses the same Query DSL and, for the purpose of illustration of how filtering could be used, includes `filter` as part of the query. But instead of returning the matched documents, the update or delete modifications are going to be applied.

The delete by query API could be used to overcome the limitations of regular delete API and remove the child documents in case their parent is deleted.

2.8 Know Your Queries Better

Sometimes you may find out that your search queries are returning the documents in order you don't expect, ranking some documents higher than others. To help you out, Elasticsearch provides two very helpful APIs. One of these is explain API, which computes a score explanation for a query (and a specific document if needed).

The explanation could be received by specifying the `explain` parameter as part of query:

```
$ curl -i https://localhost:9200/catalog/books/_search?pretty -d '
{
   "size": 10,
   "explain": true,
   "query": {
        "term" : {
            "publisher" : "Manning"
        }
    }
}
```

Or using dedicated explain API endpoint and specific document, for example:

```
$ curl -i https://localhost:9200/catalog/books/978-1617292774/_explain?pretty -d '
{
   "query": {
        "term" : {
            "publisher" : "Manning"
        }
    }
}'
```

The responses have not been included intentionally as the tons of useful details are being returned. Another very useful feature of Elasticsearch is validation API which allows to perform a validation of the query without actually executing it, for example:

```
$ curl -i https://localhost:9200/catalog/books/_validate/query?pretty -d ' {
   "query": {
        "term" : {
            "publisher" : "Manning"
        }
    }
}'

HTTP/1.1 200 OK
content-type: application/json; charset=UTF-8
content-length: 98

{
  "valid" : true,
  "_shards" : {
    "total" : 1,
    "successful" : 1,
    "failed" : 0
  }
}
```

Both APIs are very useful to troubleshoot the relevance or analyze potentially impactful search query without executing it on a live Elasticsearch cluster.

2.9 From Search to Insights

Often you may find yourself in the situation when the search is just not enough, you need some kind of aggregations on top of the matches. The great example would be faceting (or as Elasticsearch calls it, terms aggregations), where the search results are grouped into buckets.

```
$ curl -i https://localhost:9200/catalog/books/_search?pretty -d '
{
   "query": {
        "match" : {
            "description" : "elasticsearch"
        }
    },
    "aggs" : {
        "publisher" : {
            "terms" : { "field" : "publisher" }
        }
    }
}'

HTTP/1.1 200 OK
content-type: application/json; charset=UTF-8
content-length: 3447

{
  "took" : 176,
  "timed_out" : false,
  "_shards" : {
    "total" : 5,
    "successful" : 5,
    "failed" : 0
  },
  "hits" : {
    "total" : 3,
    "max_score" : 0.38828257,
```

```
    "hits" : [
      {
          ...
      }
    ]
  },
  "aggregations" : {
    "publisher" : {
      "doc_count_error_upper_bound" : 0,
      "sum_other_doc_count" : 0,
      "buckets" : [
        {
          "key" : "Manning",
          "doc_count" : 2
        },
        {
          "key" : "O'Reilly",
          "doc_count" : 1
        }
      ]
    }
  }
}
```

In this example along with the search query, we have asked Elasticsearch to count the documents by publishers. By and large, search query could be completely omitted and only aggregations could be sent in request body, for example:

```
$ curl -i https://localhost:9200/catalog/books/_search?pretty -d '
{
  "aggs" : {
      "authors": {
        "children": {
          "type" : "authors"
        },
        "aggs": {
          "top-authors": {
            "terms": {
            "script" : {
              "inline": "doc['first_name'].value + ' ' + doc['last_name'].value",
              "lang": "painless"
            },
            "size": 10
          }
        }
      }
    }
  }
}'

HTTP/1.1 200 OK
content-type: application/json; charset=UTF-8
content-length: 1031
{
  "took": 381,
  "timed_out": false,
  "_shards": {
    "total": 5,
    "successful": 5,
    "failed": 0
  },
  "hits": {
    "total": 5,
```

```
    "max_score": 1,
    "hits": [
      ...
    ]
  },
  "aggregations": {
    "authors": {
      "doc_count": 6,
      "top-authors": {
        "doc_count_error_upper_bound": 0,
        "sum_other_doc_count": 0,
        "buckets": [
          {
            "key": "Clinton Gormley",
            "doc_count": 1
          },
          {
            "key": "Doug Turnbull",
            "doc_count": 1
          },
          {
            "key": "Matthew Lee Hinman",
            "doc_count": 1
          },
          {
            "key": "Radu Gheorghe",
            "doc_count": 1
          },
          {
            "key": "Roy Russo",
            "doc_count": 1
          },
          {
            "key": "Zachary Tong",
            "doc_count": 1
          }
        ]
      }
    }
  }
}
```

In this slightly more complicated example we have bucketed over top authors using Elasticsearch scripting support to compose the terms out of the author's first and last name:

```
"script" : {
  "inline": "doc['first_name'].value + ' ' + doc['last_name'].value",
  "lang": "painless"
}
```

The list of supported aggregations is really impressive and includes bucket aggregations (some of those we have tried out already), metrics aggregations, pipeline aggregations, and matrix aggregations. Covering just one class of those would need its own tutorial, so please check them out to understand the purpose of each one in depth.

2.10 Watch Your Cluster Breathing

Elasticsearch clusters are living "creatures" and should be watched and monitored closely in order to proactively spot any issues and react on them quickly. The cluster health endpoint we have seen before is the easiest way to get overall high-level status of the cluster.

```
$ http https://localhost:9200/_cluster/health

HTTP/1.1 200 OK
content-encoding: gzip
content-type: application/json; charset=UTF-8
transfer-encoding: chunked

{
    "active_primary_shards": 5,
    "active_shards": 5,
    "active_shards_percent_as_number": 20.0,
    "cluster_name": "es-catalog",
    "delayed_unassigned_shards": 0,
    "initializing_shards": 0,
    "number_of_data_nodes": 1,
    "number_of_in_flight_fetch": 0,
    "number_of_nodes": 1,
    "number_of_pending_tasks": 0,
    "relocating_shards": 0,
    "status": "red",
    "task_max_waiting_in_queue_millis": 0,
    "timed_out": false,
    "unassigned_shards": 20
}
```

If you cluster goes `red` (like in the example above), there is certainly a problem to fix. To help you out, Elasticsearch has cluster statistics API, cluster state API, cluster node level statistics API and cluster node indices statistics API.

A bit aside stays another exceptionally important group of APIs, the cat APIs. They are different in a sense that the representation is not in JSON but rather text based, with compact and aligned output, suitable for terminals.

2.11 Conclusions

In this section of the tutorial we went through many features of the Elasticsearch by exploring them through RESTful APIs, using command line tools only. By and large, it is just a tiny part of what Elasticsearch offers through the APIs and the official documentation is a great place to learn them all. Hopefully, at this point we are comfortable enough with Elasticsearch and know how to work with it.

2.12 What's next

In the next part of the tutorial we are going to learn the several flavors of native APIs which Elasticsearch has to offer to Java/JVM developers. These APIs are the essential building blocks of any Java/JVM application which leverages Elasticsearch capabilities.

Chapter 3

Elasticsearch from Java

3.1 Introduction

In the previous part of the tutorial we mastered the skills of establishing meaningful conversations with Elasticsearch by leveraging its numerous RESTful APIs, using the command line tools only. It is very handful knowledge, however when you are developing Java / JVM applications, you would need better options than command line. Luckily, Elasticsearch has more than one offering in this area.

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.

3.2 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.

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

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

```
@Configuration
public class ElasticsearchClientConfiguration {
    @Bean(destroyMethod = "close")
    TransportClient transportClient() throws UnknownHostException  {
        return new PreBuiltTransportClient(
            Settings.builder()-
                .put(ClusterName.CLUSTER_NAME_SETTING.getKey(), "es-catalog")
                .build()
            )
            .addTransportAddress(new InetSocketTransportAddress(
                InetAddress.getByName("localhost"), 9300));
    }
}
```

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:

```
@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.

```
final ClusterHealthResponse response = client
    .admin()
    .cluster()
    .health(
        Requests
            .clusterHealthRequest()
            .waitForGreenStatus()
            .timeout(TimeValue.timeValueSeconds(5))
    )
    .actionGet();

assertThat(response.isTimedOut())
    .withFailMessage("The cluster is unhealthy: %s", response.getStatus())
    .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.

```
final ClusterHealthResponse response = client
    .admin()
    .cluster()
    .prepareHealth()
    .setWaitForGreenStatus()
    .setTimeout(TimeValue.timeValueSeconds(5))
    .execute()
    .actionGet();

assertThat(response.isTimedOut())
    .withFailMessage("The cluster is unhealthy: %s", response.getStatus())
    .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.

```
final IndicesExistsResponse response = client
    .admin()
    .indices()
    .prepareExists("catalog")
    .get(TimeValue.timeValueMillis(100));

if (!response.isExists()) {
    ...
}
```

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",
        response.buildFailureMessage())
    .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!

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

assertThat(response.getHits().hits())
    .withFailMessage("Expecting at least one book to be returned")
    .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:

```
final QueryBuilder query = QueryBuilders
    .boolQuery()
        .must(
            QueryBuilders
                .rangeQuery("rating")
                .gte(4)
        )
        .must(
            QueryBuilders
                .nestedQuery(
                    "categories",
                    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.

3.3 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());

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

    assertThat(json, hasJsonPath("$.status", equalTo("green")));
}
```

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.

```
final Map<String, Object> source = new LinkedHashMap<>();
source.put("title", "Elasticsearch: The Definitive Guide. ...");
source.put("categories",
    new Map[] {
        singletonMap("name", "analytics"),
        singletonMap("name", "search"),
        singletonMap("name", "database store")
    }
);
source.put("publisher", "O'Reilly");
source.put("description", "Whether you need full-text search or ...");
source.put("published_date", "2015-02-07");
source.put("isbn", "978-1449358549");
source.put("rating", 4);
```

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.

```
final HttpEntity payload = new NStringEntity(JSONObject.toJSONString(source),
    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.

3.4 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.

```
<dependency>
    <groupId>org.apache.lucene</groupId>
    <artifactId>lucene-test-framework</artifactId>
    <version>6.4.0</version>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.elasticsearch.test</groupId>
    <artifactId>framework</artifactId>
    <version>5.2.0</version>
    <scope>test</scope>
</dependency>
```

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

```
<dependency>
    <groupId>junit</groupId>
```

```
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.hamcrest</groupId>
            <artifactId>hamcrest-core</artifactId>
        </exclusion>
    </exclusions>
</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.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.19.1</version>
    <configuration>
        <argLine>-Dtests.security.manager=false</argLine>
    </configuration>
</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.

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

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.

```
@Before
public void setUpCatalog() throws IOException {
    try (final ByteArrayOutputStream out = new ByteArrayOutputStream()) {
        Streams.copy(getClass().getResourceAsStream("/catalog-index.json"),
            out);

        final CreateIndexResponse response = admin()
                .indices()
            .prepareCreate("catalog")
            .setSource(out.toByteArray())
            .get();

        assertAcked(response);
        ensureGreen("catalog");
    }
}
```

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:

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

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.

```
@Test
public void testEmptyCatalogHasNoBooks() {
    final SearchResponse response = client()
        .prepareSearch("catalog")
        .setTypes("books")
        .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
        .setQuery(QueryBuilders.matchAllQuery())
        .setFetchSource(false)
        .get();

    assertNoSearchHits(response);
}
```

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.

```
@Test
public void testInsertAndSearchForBook() throws IOException {
    final XContentBuilder source = JsonXContent
        .contentBuilder()
          .startObject()
        .field("title", randomAsciiOfLength(100))
        .startArray("categories")
            .startObject().field("name", "analytics").endObject()
            .startObject().field("name", "search").endObject()
            .startObject().field("name", "database store").endObject()
        .endArray()
        .field("publisher", randomAsciiOfLength(20))
        .field("description", randomAsciiOfLength(200))
        .field("published_date", new LocalDate(2015, 02, 07).toDate())
        .field("isbn", "978-1449358549")
        .field("rating", randomInt(5))
        .endObject();

    index("catalog", "books", "978-1449358549", source);
    refresh("catalog");

    final QueryBuilder query = QueryBuilders
        .nestedQuery(
            "categories",
            QueryBuilders.matchQuery("categories.name", "analytics"),
            ScoreMode.Total
        );

    final SearchResponse response = client()
        .prepareSearch("catalog")
```

```
        .setTypes("books")
        .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
        .setQuery(query)
        .setFetchSource(false)
        .get();

    assertSearchHits(response, "978-1449358549");
}
```

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.

```
@Test
public void testClusterNodeIsDown() throws IOException {
    internalCluster().stopRandomDataNode();

    final SearchResponse response = client()
        .prepareSearch("catalog")
        .setTypes("books")
        .setSearchType(SearchType.DFS_QUERY_THEN_FETCH)
        .setQuery(QueryBuilders.matchAllQuery())
        .setFetchSource(false)
        .get();

    assertNoSearchHits(response);
}
```

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.

3.5 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.

3.6 What's next

In the upcoming part, the last one of the tutorial, we are going to talk about ecosystem of the terrific projects centered around Elasticsearch. Hopefully, you will be amazed one more time by Elasticsearch capabilities and features, opening for yourself new horizons of its applicability.

The complete source code for all projects is available for download: elasticsearch-client-rest, elasticsearch-testing, elasticsearch-client-java

Chapter 4

Elasticsearch Ecosystem

4.1 Introduction

In this last part of the tutorial we are going to look around and learn how perfectly Elasticsearch fits into Java ecosystem and inspires many interesting projects. One of the best ways to illustrate that is to take a look at the marriage of Elasticsearch and Hibernate framework, an exceptionally beloved choice among Java developers for managing the persistence layer.

Additionally, at the very end we are going to glance through a tremendously popular suite of the applications, known as Elastic Stack, and what you could do with it. Although it goes well beyond Java applications only, it is hard to overestimate the value it offers to modern, highly distributed software systems.

4.2 Elasticsearch for Hibernate Users

It is practically impossible to find any Java developer out there who has not heard about the Hibernate framework. On the other hand, not so many developers know that there are quite a few projects hidden under the Hibernate umbrella and one of them is a real gem called Hibernate Search.

Hibernate Search transparently indexes your objects and offers fast regular, full-text and geolocation search. Ease of use and easy clustering are core. - https://hibernate.org/search/

Hibernate Search has started as a simple glue layer between Hibernate and Apache Lucene and used to offer a very limited set of supported backends to manage search indices. But the situation is changing for the better with a just recently released final version of the Hibernate Search `5.7.0` along with full-fledged Elasticsearch support (while still bearing an `experimental` label). What it practically means is that if your persistence layer is managed by Hibernate, then by plugging in Hibernate Search you could enrich your data model with full-text search capabilities, all that backed by Elasticsearch. Sounds exciting, right?

To get a feeling of how things work under the hood, let us take a look at our `catalog` data model expressed in terms of JPA entities decorated with Hibernate Search annotations, starting with the `Book` class.

```
@Entity
@Table(name = "BOOKS")
@Indexed(index = "catalog")
public class Book {
    @Id
    @Field(name = "isbn", analyze = Analyze.NO)
    private String id;

    @Field
    @Column(name = "TITLE", nullable = false)
    private String title;

    @IndexedEmbedded(depth = 1)
```

```
    @ElementCollection
    private Set categories = new HashSet<>();

    @Field(analyze = Analyze.NO)
    @Column(name = "PUBLISHER", nullable = false)
    private String publisher;

    @Field
    @Column(name = "DESCRIPTION", nullable = false, length = 4096)
    private String description;

    @Field(name = "published_date", analyze = Analyze.NO)
    @Column(name = "PUBLISHED_DATE", nullable = false)
    @DateBridge(resolution = Resolution.DAY)
    private LocalDate publishedDate;

    @NumericField @Field(name = "rating")
    @Column(name = "RATING", nullable = false)
    private int rating;

    @IndexedEmbedded
    @ManyToMany
    private Set authors = new HashSet();
}
```

For seasoned Java developers it is a familiar piece of code to describe persistent entities, with only a couple of Hibernate Search annotations (like `@Field`, `@DateBridge`, `@IndexedEmbedded`) added on top. We are not going to discuss them here unfortunately, this topic itself is worth of a complete tutorial but please do not hesitate to refer to the official documentation for more details. With that being said, we just move on to the `Category` class.

```
@Embeddable
public class Category {
        @Field(analyze = Analyze.NO)
        @Column(name = "NAME", nullable = false)
        private String name;
}
```

Followed by the `Author` class right after.

```
@Entity
@Indexed(index = "catalog")
@Table(name = "AUTHORS")
public class Author {
    @Id
    private String id;

    @Field(name = "first_name", analyze = Analyze.NO)
    @Column(name = "FIRST_NAME", nullable = false)
    private String firstName;

    @Field(name = "last_name", analyze = Analyze.NO)
    @Column(name = "LAST_NAME", nullable = false)
    private String lastName;
}
```

Thanks to Spring Framework, and particularly to Spring Boot magic, the configuration of the Hibernate and Hibernate Search pointing to Elasticsearch as the search backend is as easy as adding a couple of lines to `application.yml` file.

```
spring:
  jpa:
    properties:
      hibernate:
```

```
        search:
          default:
            indexmanager: elasticsearch
            elasticsearch:
                host: https://localhost:9200
```

Frankly speaking, you may need to tailor this configuration quite a bit to fit the needs of your applications, to our luck the official documentation covers this part pretty well. Every time the instances of the `Book` or `Author` classes are created, modified or deleted, the Hibernate Search takes care of keeping Elasticsearch index in sync, it is completely transparent.

How the search looks like? Well, Hibernate Search does provide its own Query DSL abstraction layer, based on Apache Lucene queries (at the same time leaving a route to use some native Elasticsearch features), for example:

```
@Autowired private EntityManager entityManager;

final FullTextEntityManager fullTextEntityManager = Search
    .getFullTextEntityManager(entityManager);

final QueryBuilder qb = fullTextEntityManager
    .getSearchFactory()
    .buildQueryBuilder()
    .forEntity(Book.class)
    .get();

final FullTextQuery query = fullTextEntityManager
    .createFullTextQuery(
        qb.bool()
            .must(
                qb.keyword()
                    .onField("categories.name")
                    .matching("analytics")
                    .createQuery()
            )
            .must(
                qb.keyword()
                    .onField("authors.last_name")
                    .matching("Tong")
                    .createQuery()
            ).createQuery(), Book.class);

final List books = query.getResultList();
...
```

There are definitely a lot of similarities to Elasticsearch's own Query DSL so this code snippet should look already familiar to you. But before you get too excited, there are several limitations related to Hibernate Search and Elasticsearch integration. First of all, the latest version of Elasticsearch which is supported by Hibernate Search at the moment is `2.4.4`. Not bad but quite far from the current `5.x` release branch, hopefully this is going to be addressed soon. Secondly, indeed, the subset of the Elasticsearch features exposed over the Hibernate Search APIs, in particular Query DSL, is quite limited but frankly speaking, might be enough for many applications.

Anyway, why we are mentioning Hibernate Search in the first place? Simple, if your applications are built on top of Hibernate persistence, using Hibernate Search is probably the fastest and cheapest way to benefit from full-text search capabilities for your data models, leveraging Elasticsearch behind the curtain.

4.3 Elastic Stack: Get It All

If you already run into mysterious ELK abbreviation and were curious what it means than this section would help you to find the answers. ELK is essentially a bundle of products, consisting of (E)lasticsearch, (L)ogstash and (K)ibana, therefore just ELK in

short. Recently, with the addition of the Beats, a new member of this awesome family, the ELK is often referred as Elastic Stack now.

Undoubtedly, Elasticsearch is the heart and soul of ELK so let us talk about what those other products are and why they are useful. Kibana lets you visualize your Elasticsearch data and navigate the Elastic Stack, so you can do anything from learning why you're getting paged at 2:00 a.m. to understanding the impact rain might have on your quarterly numbers. - https://www.elastic.co/-products/kibana Basically, Kibana is just a web application which is capable of creating powerful charts and dashboards based on the data you are having indexed in Elasticsearch. Logstash is an open source, server-side data processing pipeline that ingests data from a multitude of sources simultaneously, transforms it, and then sends it to your favorite "stash", f.e. Elasticsearch - https://www.elastic.co/products/logstash

Consequently, Logstash is a terrific tool, capable of extracting, massaging and feeding the data to Elasticsearch (and a myriad of other sources) which could be visualized using Kibana later on. The Beats are quite close to Logstash but are not yet so powerful.

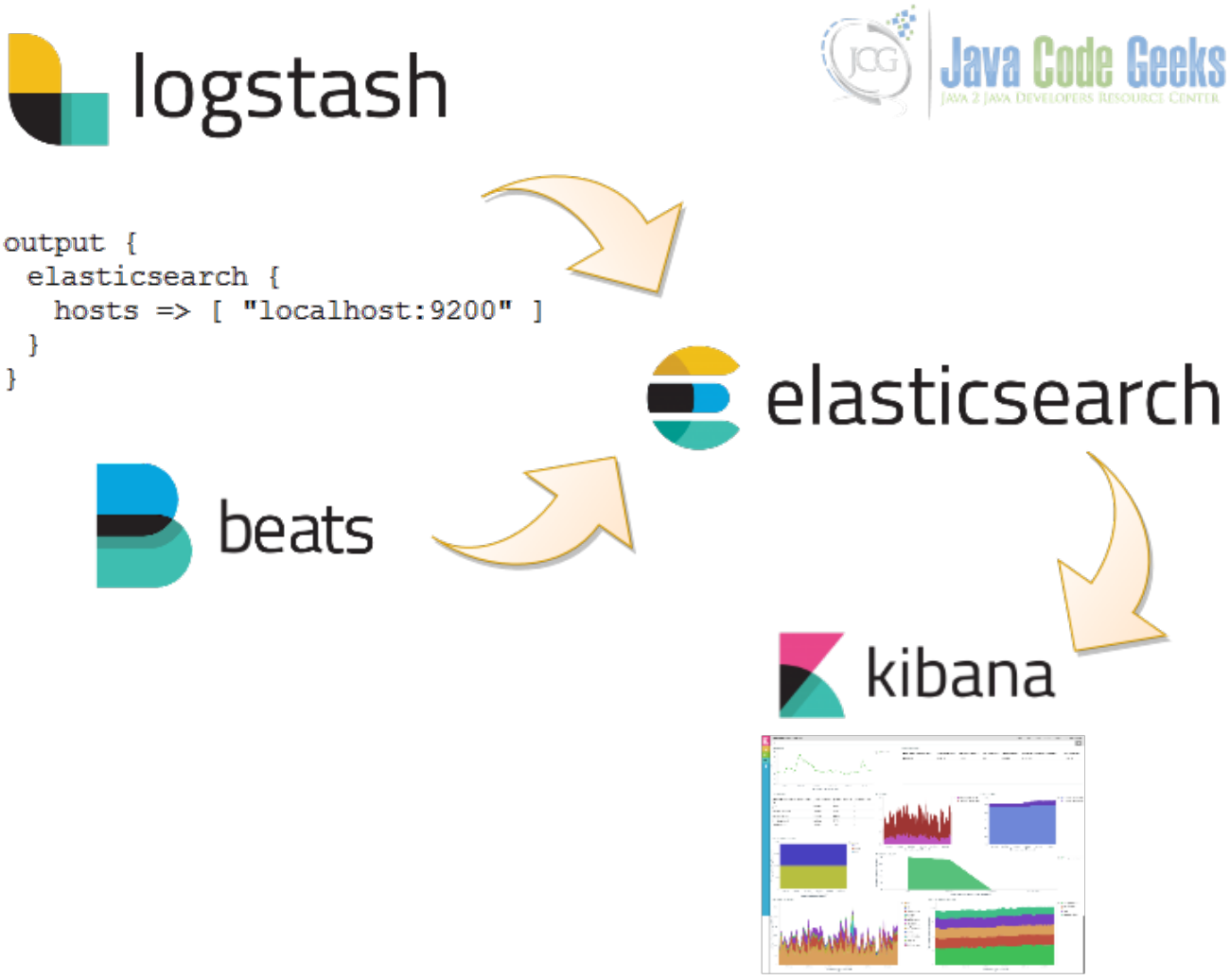

Figure 4.1: Elastic Stack Illustrated

One of the areas where Elastic Stack is exceptionally useful and leads the race is collecting and analyzing the massive amounts of application logs. It may sound not very convincing, why would you need such a complex system in order to tail/grep over a log file? But at scale, when you deal with hundreds or even thousands of the applications (think microservices), the benefits become very obvious: you suddenly have a centralized places where the logs from all the applications are streamed into and could be searched against, analyzed, correlated and visualized.

Without any further talks, let us demonstrate how a typical Spring Boot application could be configured to send its logs to Logstash, which is going to forward them to Elasticsearch as-is, without transformations applied. First, we need Logstash to be

installed, either as Docker container or just running on local machine, the official documentation presents the installation steps exceptionally well.

The only thing we need to tell to Logstash is where to get logs from (using input plugins) and where to send the massaged logs to (using output plugins), all that through `logstash.conf` configuration file.

```
input {
    tcp {
        port => 7760
    }
}

output {
    elasticsearch {
        hosts => [ "localhost:9200" ]
    }
}
```

The number of input and output plugins supported by Logstash is astonishing. To keep the example very simple, we are going to deliver the logs over the TCP socket input plugin and forward straight to Elasticsearch using Elasticsearch output plugin.

It looks great, but how we could send the logs from our Java applications to Logstash? There are many ways to do that and the easiest one is probably by leveraging the capabilities of your logging framework. Most of the Java applications these days rely on the awesome Logback framework and the community has implemented dedicated Logback encoder to be used along with Logstash.

You just have to include an additional dependency into your project, like we do here, for example, using Apache Maven:

```
<dependency>
    <groupId>net.logstash.logback</groupId>
    <artifactId>logstash-logback-encoder</artifactId>
    <version>4.9</version>
</dependency>
```

And then add the Logstash appender into your `logback.xml` configuration file. To be noted, there are several appenders available, the one we are interested in is `LogstashTcpSocketAppender` which talks to Logstash over TCP socket. Please notice that the port under `destination` tag should match to your Logstash input plugin configuration, in our case it is `7760`.

```
<appender name="logstash" class="net.logstash.logback.appender.LogstashTcpSocketAppender">
    <destination>localhost:7760</destination>
    <encoder class="net.logstash.logback.encoder.LogstashEncoder" />
</appender>
```

By and large, this is all we have to do! The logs will be shipped from our application to Elasticsearch and we could explore them using Kibana dashboards. When you download and run Kibana on your local machine, the web UI by default is available at https://localhost:5601:

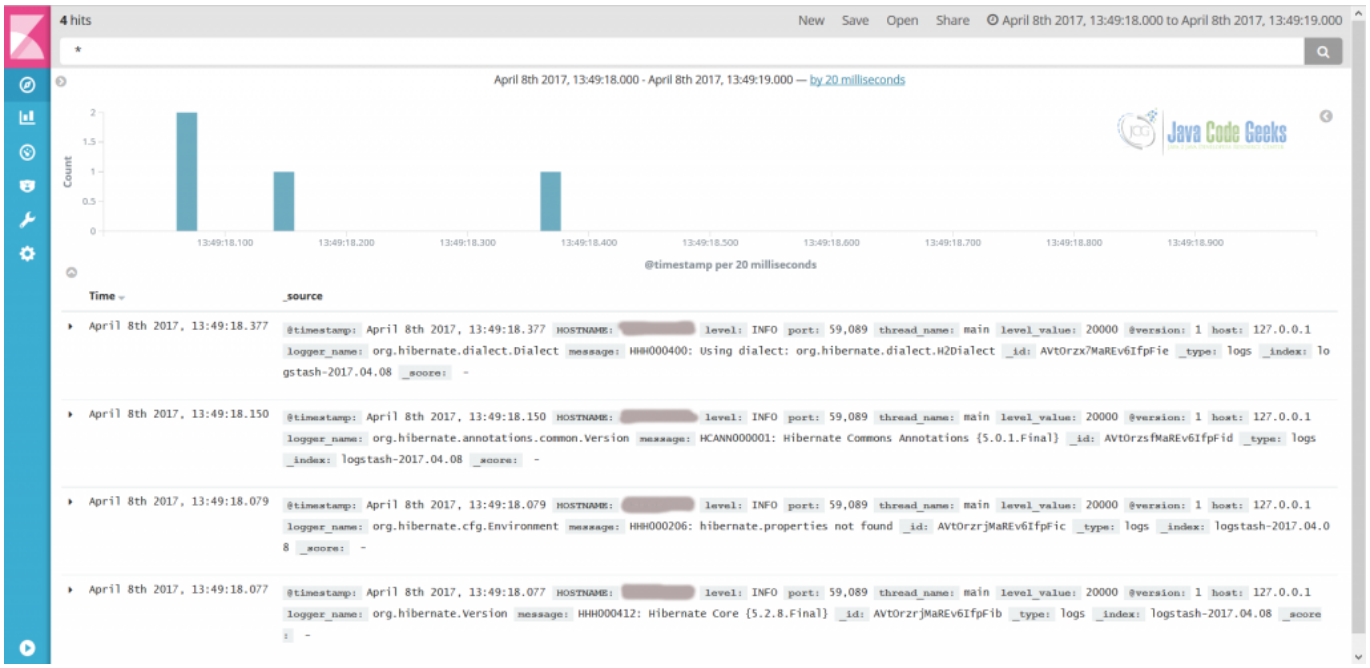

Figure 4.2: Quick Kibana Dashboard

Easy, simple, powerful ... the Elasticsearch way. The only thing to remember is that you should better use the same versions of Elasticsearch , Logstash and Kibana. As we have been using Elasticsearch `5.2.0` along this tutorial, Logstash and Kibana should also be of `5.2.0` release.

If you already using Elasticsearch or are planning to do so, Elastic Stack just opens a whole universe of interesting opportunities for you to discover and benefit from. Moreover, it is being constantly improved and enhanced with new features added every single release.

4.4 Supercharge Elasticsearch with Plugins

Elasticsearch is wonderful but often command line tools or even Java APIs are not the best way to communicate with your clusters. Luckily, Elasticsearch has extensibility built-in from the early days in a form of plugins.

There are many plugins and accompanying tools available at the moment, but it is worth to talk about three of them:

- elasticsearch-head: a web front end for an Elasticsearch cluster
- elasticsearch-HQ: monitoring, management, and querying web interface for Elasticsearch
- search-guard: security for Elasticsearch

The elasticsearch-head is, essentially, a full-fledged web interface to Elasticsearch. Not only you have nice visual representations of indices and shards, you can also browse the documents, play with search queries and navigate through the results easily.

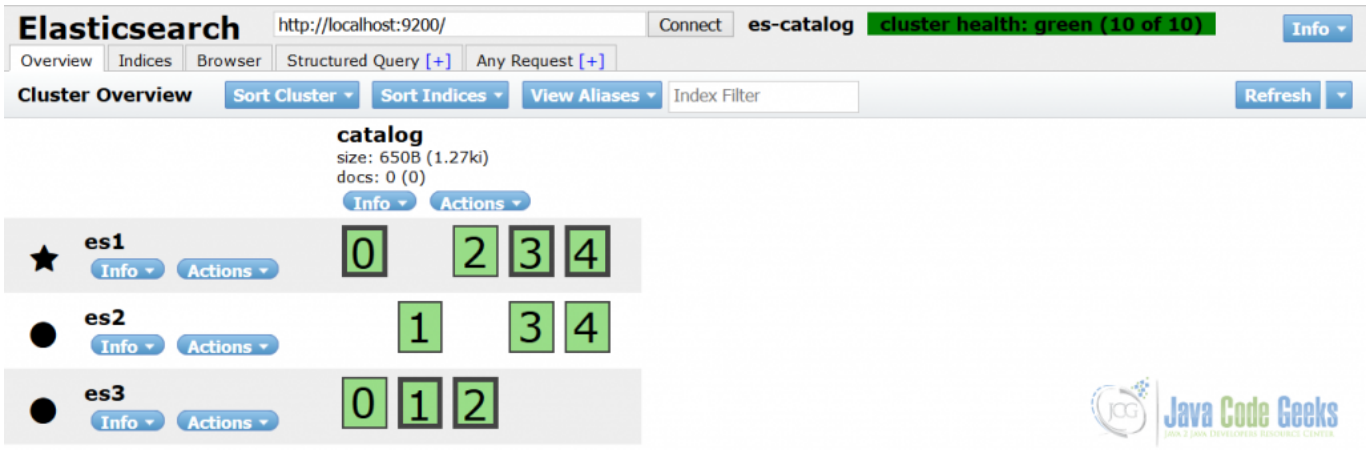

Figure 4.3: An example of elasticsearch-head showing catalog index and cluster status

The ability to run structured or arbitrary queries is very helpful, specifically if your query returns a lot of results and you need a convenient way to go through all of them.

Another very interesting one is elasticsearch-HQ which basically put the focus on exposing operational information about Elasticsearch clusters and nodes. Unfortunately, as of moment of this writing, the elasticsearch-HQ does not support `5.x` release branch of Elasticsearch but the work to make it happen has already begun.

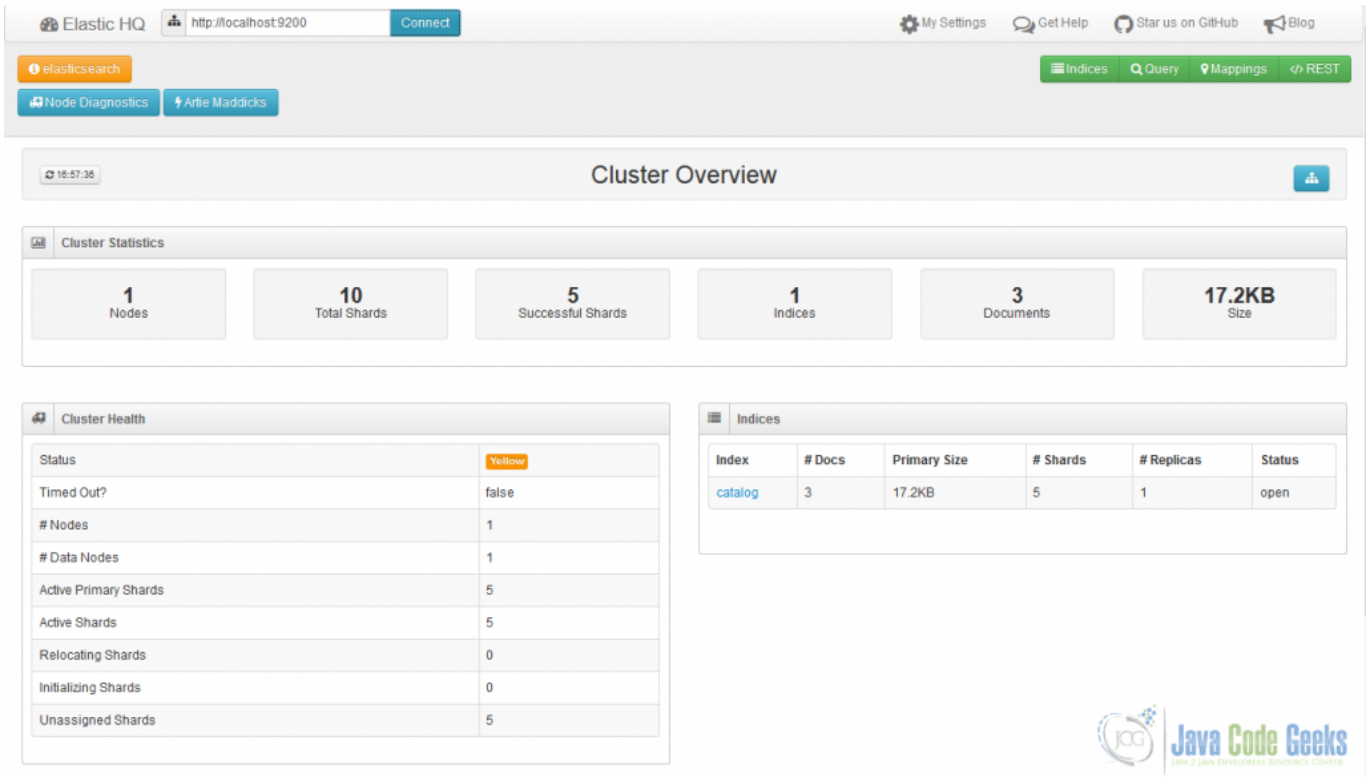

Figure 4.4: An example of elasticsearch-HQ showing catalog index and cluster status

Before we touch upon last plugin, the search-guard, it would be good to talk about the state of security in Elasticsearch. In fact, we have better to say that out of the box Elasticsearch has nothing to offer in terms of security or alike (although this feature, along with many others, is available as part of commercial distribution of Elasticsearch). The search-guard is the oldest

community supported plugin which adds quite a lot of security capabilities to Elasticsearch. Please make sure to take a look at this one, if you are serious about running Elasticsearch in production, luckily it supports all recent Elasticsearch versions.

4.5 Conclusions

With this part, the "Elasticsearch for Java Developers" series is coming to its logical end. Along this tutorial we have learned about Elasticsearch, what it does and how to communicate with it using command line tools and its rich set of RESTful APIs. We have also discussed different flavors of Java APIs available at the moment and briefly debated when to use one or another. And lastly, we have covered a flourishing ecosystem of projects (and products) which has emerged around Elasticsearch and heavily relies on it features.

Hopefully, you have learned something along the way, and if you have hesitated before about giving Elasticsearch a try or not, all your doubts should be cleared out now. It is a great product with terrific potential to solve wide range of hard problems and power your ideas to success.

With that, best of luck on this journey! The complete source code for this article is available here.