**Describe Elasticsearch?**

**Answer: -** 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.

**Here are a few sample use-cases that Elasticsearch could be used for: -**

* You run an online web store where you allow your customers to search for products that you sell. In this case, you can use Elasticsearch to store your entire product catalog and inventory and provide search and autocomplete suggestions for them.
* You want to collect log or transaction data and you want to analyze and mine this data to look for trends, statistics, summarizations, or anomalies. In this case, you can use Logstash (part of the Elasticsearch/Logstash/Kibana stack) to collect, aggregate, and parse your data, and then have Logstash feed this data into Elasticsearch. Once the data is in Elasticsearch, you can run searches and aggregations to mine any information that is of interest to you.
* You run a price alerting platform which allows price-savvy customers to specify a rule like "I am interested in buying a specific electronic gadget and I want to be notified if the price of gadget falls below $X from any vendor within the next month". In this case you can scrape vendor prices, push them into Elasticsearch and use its reverse-search (Percolator) capability to match price movements against customer queries and eventually push the alerts out to the customer once matches are found.
* You have analytics/business-intelligence needs and want to quickly investigate, analyze, visualize, and ask ad-hoc questions on a lot of data (think millions or billions of records). In this case, you can use Elasticsearch to store your data and then use Kibana (part of the Elasticsearch/Logstash/Kibana stack) to build custom dashboards that can visualize aspects of your data that are important to you. Additionally, you can use the Elasticsearch aggregations functionality to perform complex business intelligence queries against your data.

**Basic Concepts**

There are a few concepts that are core to Elasticsearch. Understanding these concepts from the outset will tremendously help ease the learning process.

**Near Realtime (NRT)**

Elasticsearch is a near-Realtime search platform. What this means is there is a slight latency (normally one second) from the time you index a document until the time it becomes searchable.

**Cluster**

A cluster is a collection of one or more nodes (servers) that together holds your entire data and provides federated indexing and search capabilities across all nodes. A cluster is identified by a unique name which by default is "elasticsearch". This name is important because a node can only be part of a cluster if the node is set up to join the cluster by its name.

Make sure that you don’t reuse the same cluster names in different environments, otherwise you might end up with nodes joining the wrong cluster. For instance, you could use **logging-dev, logging-stage, and logging-prod** for the development, staging, and production clusters.

**Note** that it is valid and perfectly fine to have a cluster with only a single node in it. Furthermore, you may also have multiple independent clusters each with its own unique cluster name.

**Node**

A node is a single server that is part of your cluster, stores your data, and participates in the cluster’s indexing and search capabilities. Just like a cluster, a node is identified by a name which by default is a random Universally Unique IDentifier (UUID) that is assigned to the node at startup. You can define any node name you want if you do not want the default. This name is important for administration purposes where you want to identify which servers in your network correspond to which nodes in your Elasticsearch cluster.

A node can be configured to join a specific cluster by the cluster name. By default, each node is set up to join a cluster named elasticsearch which means that if you start up a number of nodes on your network and—assuming they can discover each other—they will all automatically form and join a single cluster named elasticsearch.

In a single cluster, you can have as many nodes as you want. Furthermore, if there are no other Elasticsearch nodes currently running on your network, starting a single node will by default form a new single-node cluster named elasticsearch.

|  |
| --- |
| spring:  profiles.active: development    ---  spring:  profiles: development  zshop.datasource.mysql:  url: jdbc:mysql://10.146.2.32:3306/zshop\_dropship  username: devuser  password: devuser@1234  driverClassName: com.mysql.jdbc.Driver  elasticsearch:  cluster:  nodes:  - nodeHost: 35.154.234.162  nodePort: 9300  name: uat-elkstack    ---  spring:  profiles: uat  zshop.datasource.mysql:  url: jdbc:mysql://10.147.10.121:3306/zshop\_dropship  username: INTGZshopWU  password: Tfa25eAPSM2s5hqP  driverClassName: com.mysql.jdbc.Driver  elasticsearch:  cluster:  nodes:  - nodeHost: 10.147.10.103  nodePort: 9300  name: uat-elkstack  ---  spring:  profiles: production  zshop.datasource.mysql:  url: jdbc:mysql://masterdb.prod.ezmall.com:3306/zshop  username: zeeshopWU  password: hhd639dKJHut  driverClassName: com.mysql.jdbc.Driver  elasticsearch:  cluster:  nodes:  - nodeHost: mon08.prod.ezmall.com  nodePort: 9300  - nodeHost: mon09.prod.ezmall.com  nodePort: 9300  name: production-elkstack |

**Index**

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

**Type (Deprecated in 6.0.0.)**

A type used to be a logical category/partition of your index to allow you to store different types of documents in the same index, e.g. one type for users, another type for blog posts. It is no longer possible to create multiple types in an index, and the whole concept of types will be removed in a later version. See Removal of mapping types for more.

**Document**

A document is a basic unit of information that can be indexed. For example, you can have a document for a single customer, another document for a single product, and yet another for a single order. This document is expressed in JSON (JavaScript Object Notation) which is a ubiquitous internet data interchange format.

Within an index/type, you can store as many documents as you want. Note that although a document physically resides in an index, a document actually must be indexed/assigned to a type inside an index.

**Shards & Replicas**

An index can potentially store a large amount of data that can exceed the hardware limits of a single node. For example, a single index of a billion documents taking up 1TB of disk space may not fit on the disk of a single node or may be too slow to serve search requests from a single node alone.

To solve this problem, Elasticsearch provides the ability to subdivide your index into multiple pieces called shards. When you create an index, you can simply define the number of shards that you want. Each shard is in itself a fully-functional and independent "index" that can be hosted on any node in the cluster.

**Sharding is important for two primary reasons: -**

* It allows you to horizontally split/scale your content volume
* It allows you to distribute and parallelize operations across shards (potentially on multiple nodes) thus increasing performance/throughput.

The mechanics of how a shard is distributed and also how its documents are aggregated back into search requests are completely managed by Elasticsearch and is transparent to you as the user.

In a network/cloud environment where failures can be expected anytime, it is very useful and highly recommended to have a failover mechanism in case a shard/node somehow goes offline or disappears for whatever reason. To this end, Elasticsearch allows you to make one or more copies of your index’s shards into what are called replica shards, or replicas for short.

**Replication is important for two primary reasons:**

* It provides high availability in case a shard/node fails. For this reason, it is important to note that a replica shard is never allocated on the same node as the original/primary shard that it was copied from.
* It allows you to scale out your search volume/throughput since searches can be executed on all replicas in parallel.

**To summarize,** each index can be split into multiple shards. An index can also be replicated zero (meaning no replicas) or more times. Once replicated, each index will have primary shards (the original shards that were replicated from) and replica shards (the copies of the primary shards).

The number of shards and replicas can be defined per index at the time the index is created. After the index is created, you may also change the number of replicas dynamically anytime. You can change the number of shards for an existing index using the \_shrink and \_split APIs, however this is not a trivial task and pre-planning for the correct number of shards is the optimal approach.

By default, each index in Elasticsearch is allocated 5 primary shards and 1 replica which means that if you have at least two nodes in your cluster, your index will have 5 primary shards and another 5 replica shards (1 complete replica) for a total of 10 shards per index.

**Note: -** Each Elasticsearch shard is a Lucene index. There is a maximum number of documents you can have in a single Lucene index. As of LUCENE-5843, the limit is 2,147,483,519 (= Integer.MAX\_VALUE - 128) documents. You can monitor shard sizes using the \_cat/shards API.

**Installation Steps.**

* Download and unzip Elasticsearch.
* Run bin/elasticsearch (or bin\elasticsearch.bat on Windows)
* invoke <http://localhost:9200>

**Exploring Your Cluster**

**The REST API**

Now that we have our node (and cluster) up and running, the next step is to understand how to communicate with it. Fortunately, Elasticsearch provides a very comprehensive and powerful REST API that you can use to interact with your cluster. Among the few things that can be done with the API are as follows:

* Check your cluster, node, and index health, status, and statistics
* Administer your cluster, node, and index data and metadata
* Perform CRUD (Create, Read, Update, and Delete) and search operations against your indexes
* Execute advanced search operations such as paging, sorting, filtering, scripting, aggregations, and many others.

**Cluster Health**

**GET /\_cat/health?v**

Whenever we ask for the cluster health, we either get green, yellow, or red.

* Green - everything is good (cluster is fully functional)
* Yellow - all data is available, but some replicas are not yet allocated (cluster is fully functional)
* Red - some data is not available for whatever reason (cluster is partially functional).

**Note:** When a cluster is red, it will continue to serve search requests from the available shards but you will likely need to fix it ASAP since there are unassigned shards.

**GET /\_cat/nodes?v**

Get a list of nodes in our cluster.

**GET /\_cat/indices?v**

List All Indices

**Create an Index**

**PUT /customer?pretty**

The above command creates the index named "customer" using the PUT verb. We simply append pretty to the end of the call to tell it to pretty-print the JSON response (if any).

Response get after creating index customer :-

health status index uuid pri rep docs.count docs.deleted store.size pri.store.size

green open .kibana\_1 MEYOY-NaTTy\_HQRXLLf3dg 1 0 4 0 17.2kb 17.2kb

green open .kibana\_task\_manager DpsOIYhQSKimnaw6uiFqdg 1 0 2 0 12.5kb 12.5kb

yellow open customer GKLDFh64Rni8jMlvccDeMg 5 1 0 0 1.1kb 1.1kb

**Note: -** The results tell us that we now have **1 index** named customer and it has **5 primary shards** and **1 replica (the defaults)** and it contains **0 documents** in it.

You might also notice that the customer index has a **yellow health** tagged to it. Recall from our previous discussion that yellow means that some replicas are not (yet) allocated. The reason this happens for this index is because Elasticsearch by default created one replica for this index. Since we only have one node running at the moment, that one replica cannot yet be allocated (for high availability) until a later point in time when another node joins the cluster. Once that replica gets allocated onto a second node, the health status for this index will turn to green.

**Index and Query a Document**

Let’s now put something into our customer index. We’ll index a simple customer document into the customer index, with an ID of 1 as follows:

|  |
| --- |
| PUT /customer/\_doc/1?pretty  {  "name": "John Doe"  } |

And the response:

|  |
| --- |
| {  "\_index" : "customer",  "\_type" : "\_doc",  "\_id" : "1",  "\_version" : 1,  "result" : "created",  "\_shards" : {  "total" : 2,  "successful" : 1,  "failed" : 0  },  "\_seq\_no" : 0,  "\_primary\_term" : 1  } |

From the above, we can see that a new customer document was successfully created inside the customer index. The document also has an internal id of 1 which we specified at index time.

**It is important to note** that Elasticsearch does not require you to explicitly create an index first before you can index documents into it. In the previous example, Elasticsearch will automatically create the customer index if it didn’t already exist beforehand.

**Retrieve document**

Let’s now retrieve that document that we just indexed:

|  |
| --- |
| GET /customer/\_doc/1?pretty |

And the response:

|  |
| --- |
| {  "\_index" : "customer",  "\_type" : "\_doc",  "\_id" : "1",  "\_version" : 1,  "\_seq\_no" : 25,  "\_primary\_term" : 1,  "found" : true,  "\_source" : { "name": "John Doe" }  } |

Nothing out of the ordinary here other than a field, found, stating that we found a document with the requested ID 1 and another field, \_source, which returns the full JSON document that we indexed from the previous step.

**Delete an Index**

Now let’s delete the index that we just created and then list all the indexes again:

|  |
| --- |
| DELETE /customer?pretty |

And the response: -

|  |
| --- |
| {  "acknowledged" : true  } |

**Modifying Your Data**

Elasticsearch provides data manipulation and search capabilities in near real time. By default, you can expect a one second delay (refresh interval) from the time you index/update/delete your data until the time that it appears in your search results. This is an important distinction from other platforms like SQL wherein data is immediately available after a transaction is completed.

**Indexing/Replacing Documents**

|  |
| --- |
| PUT /customer/\_doc/1?pretty  {  "name": "John Doe"  } |

Again, the above will index the specified document into the customer index, with the ID of 1. If we then executed the above command again with a different (or same) document, Elasticsearch will replace (i.e. reindex) a new document on top of the existing one with the ID of 1:

|  |
| --- |
| PUT /customer/\_doc/1?pretty  {  "name": "Jane Doe"  } |

The above changes the name of the document with the ID of 1 from "John Doe" to "Jane Doe". If, on the other hand, we use a different ID, a new document will be indexed and the existing document(s) already in the index remains untouched.

|  |
| --- |
| PUT /customer/\_doc/2?pretty  {  "name": "Jane Doe"  } |

The above indexes a new document with an ID of 2.

When indexing, the ID part is optional. If not specified, Elasticsearch will generate a random ID and then use it to index the document. The actual ID Elasticsearch generates (or whatever we specified explicitly in the previous examples) is returned as part of the index API call.

**This example shows how to index a document without an explicit ID:**

|  |
| --- |
| POST /customer/\_doc?pretty  {  "name": "Jane Doe"  } |

**Note** that in the above case, we are using the POST verb instead of PUT since we didn’t specify an ID.

**Updating Documents**

In addition to being able to index and replace documents, we can also update documents. Note though that Elasticsearch does not actually do in-place updates under the hood. Whenever we do an update, Elasticsearch deletes the old document and then indexes a new document with the update applied to it in one shot.

This example shows how to update our previous document (ID of 1) by changing the name field to "Jane Doe":

|  |
| --- |
| POST /customer/\_doc/1/\_update?pretty  {  "doc": { "name": "Jane Doe" }  } |

This example shows how to update our previous document (ID of 1) by changing the name field to "Jane Doe" and at the same time add an age field to it:

|  |
| --- |
| POST /customer/\_doc/1/\_update?pretty  {  "doc": { "name": "Jane Doe", "age": 20 }  } |

Updates can also be performed by using simple scripts. This example uses a script to increment the age by 5:

|  |
| --- |
| POST /customer/\_doc/1/\_update?pretty  {  "script" : "ctx.\_source.age += 5"  } |

In the above example, ctx.\_source refers to the current source document that is about to be updated.

Elasticsearch provides the ability to update multiple documents given a query condition (like an SQL UPDATE-WHERE statement).

**Deleting Documents**

Deleting a document is fairly straightforward. This example shows how to delete our previous customer with the ID of 2:

|  |
| --- |
| DELETE /customer/\_doc/2?pretty |

**Batch Processing**

In addition to being able to index, update, and delete individual documents, Elasticsearch also provides the ability to perform any of the above operations in batches using the \_bulk API. This functionality is important in that it provides a very efficient mechanism to do multiple operations as fast as possible with as few network roundtrips as possible.

As a quick example, the following call indexes two documents (ID 1 - John Doe and ID 2 - Jane Doe) in one bulk operation:

|  |
| --- |
| POST /customer/\_doc/\_bulk?pretty  {"index":{"\_id":"1"}}  {"name": "John Doe" }  {"index":{"\_id":"2"}}  {"name": "Jane Doe" } |

This example updates the first document (ID of 1) and then deletes the second document (ID of 2) in one bulk operation:

|  |
| --- |
| POST /customer/\_doc/\_bulk?pretty  {"update":{"\_id":"1"}}  {"doc": { "name": "John Doe becomes Jane Doe" } }  {"delete":{"\_id":"2"}} |

Note above that for the delete action, there is no corresponding source document after it since deletes only require the ID of the document to be deleted.

The Bulk API does not fail due to failures in one of the actions. If a single action fails for whatever reason, it will continue to process the remainder of the actions after it. When the bulk API returns, it will provide a status for each action (in the same order it was sent in) so that you can check if a specific action failed or not.