[Elasticsearch Reference [6.5]](https://www.elastic.co/guide/en/elasticsearch/reference/current/index.html)

Getting Started[edit](https://github.com/elastic/elasticsearch/edit/6.5/docs/reference/getting-started.asciidoc)

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.

For the rest of this tutorial, you will be guided through the process of getting Elasticsearch up and running, taking a peek inside it, and performing basic operations like indexing, searching, and modifying your data. At the end of this tutorial, you should have a good idea of what Elasticsearch is, how it works, and hopefully be inspired to see how you can use it to either build sophisticated search applications or to mine intelligence from 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.

### 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

Warning

### Deprecated in 6.0.0.

See [Removal of mapping types](https://www.elastic.co/guide/en/elasticsearch/reference/current/removal-of-types.html)

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](https://www.elastic.co/guide/en/elasticsearch/reference/current/removal-of-types.html) 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](http://json.org/) (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](https://www.elastic.co/guide/en/elasticsearch/reference/6.5/indices-shrink-index.html) and [\_split](https://www.elastic.co/guide/en/elasticsearch/reference/6.5/indices-split-index.html) 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](https://issues.apache.org/jira/browse/LUCENE-5843), the limit is 2,147,483,519(= Integer.MAX\_VALUE - 128) documents. You can monitor shard sizes using the [\_cat/shards](https://www.elastic.co/guide/en/elasticsearch/reference/6.5/cat-shards.html) API.

With that out of the way, let’s get started with the fun part…

## Installation

Tip

You can skip installation completely by using our hosted Elasticsearch Service on [Elastic Cloud](https://www.elastic.co/cloud), which is available on AWS and GCP. You can [try out the hosted service](https://www.elastic.co/cloud/elasticsearch-service/signup) for free.

Elasticsearch requires at least Java 8. Specifically as of this writing, it is recommended that you use the Oracle JDK version 1.8.0\_131. Java installation varies from platform to platform so we won’t go into those details here. Oracle’s recommended installation documentation can be found on [Oracle’s website](http://docs.oracle.com/javase/8/docs/technotes/guides/install/install_overview.html). Suffice to say, before you install Elasticsearch, please check your Java version first by running (and then install/upgrade accordingly if needed):

java -version

echo $JAVA\_HOME

Once we have Java set up, we can then download and run Elasticsearch. The binaries are available from [www.elastic.co/downloads](http://www.elastic.co/downloads) along with all the releases that have been made in the past. For each release, you have a choice among a zip or tar archive, a DEB or RPM package, or a Windows MSI installation package.

### Installation example with tar

For simplicity, let’s use the [tar](https://www.elastic.co/guide/en/elasticsearch/reference/6.5/zip-targz.html) file.

Let’s download the Elasticsearch 6.5.4 tar as follows:

curl -L -O https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-6.5.4.tar.gz

Then extract it as follows:

tar -xvf elasticsearch-6.5.4.tar.gz

It will then create a bunch of files and folders in your current directory. We then go into the bin directory as follows:

cd elasticsearch-6.5.4/bin

And now we are ready to start our node and single cluster:

./elasticsearch

### Installation with Homebrew

On macOS, Elasticsearch can also be installed via [Homebrew](https://brew.sh/):

brew install elasticsearch

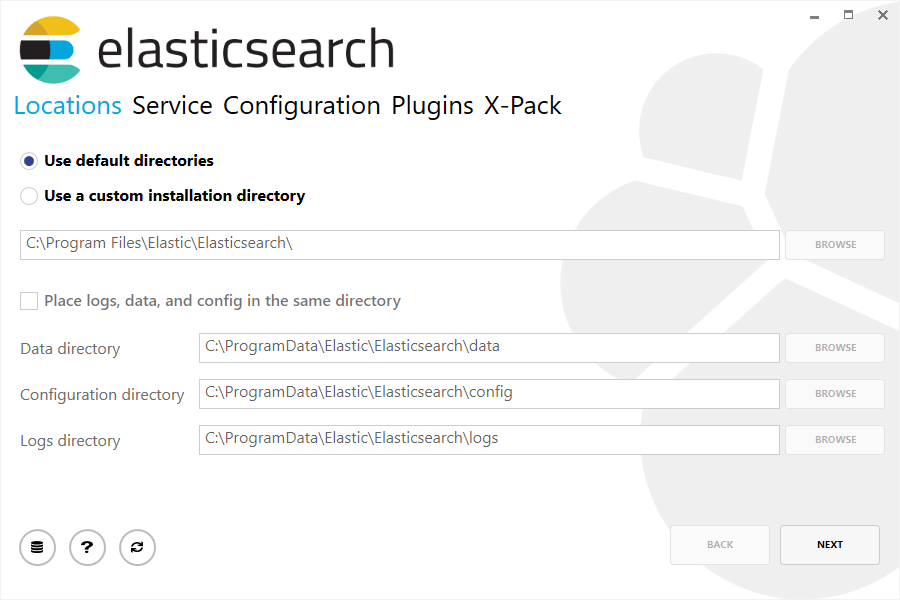
If installation succeeds, Homebrew will finish by saying that you can start Elasticsearch by enteringelasticsearch. Do that now. The expected response is described below, under [Successfully running node](https://www.elastic.co/guide/en/elasticsearch/reference/current/getting-started-install.html#successfully-running-node)[edit](https://github.com/elastic/elasticsearch/edit/6.5/docs/reference/getting-started.asciidoc).

### Installation example with MSI Windows Installer

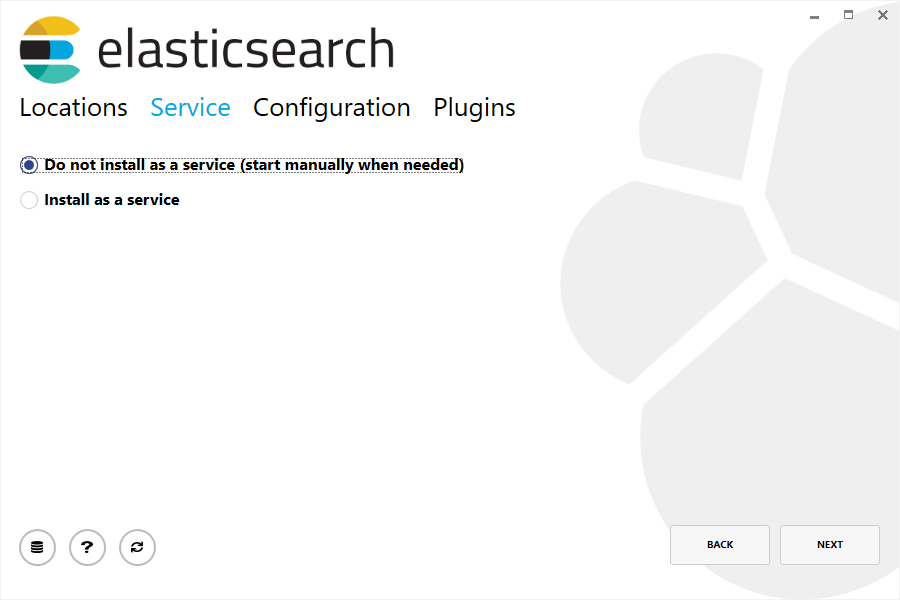
For Windows users, we recommend using the [MSI Installer package](https://www.elastic.co/guide/en/elasticsearch/reference/6.5/windows.html). The package contains a graphical user interface (GUI) that guides you through the installation process.

First, download the Elasticsearch 6.5.4 MSI from<https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-6.5.4.msi>.

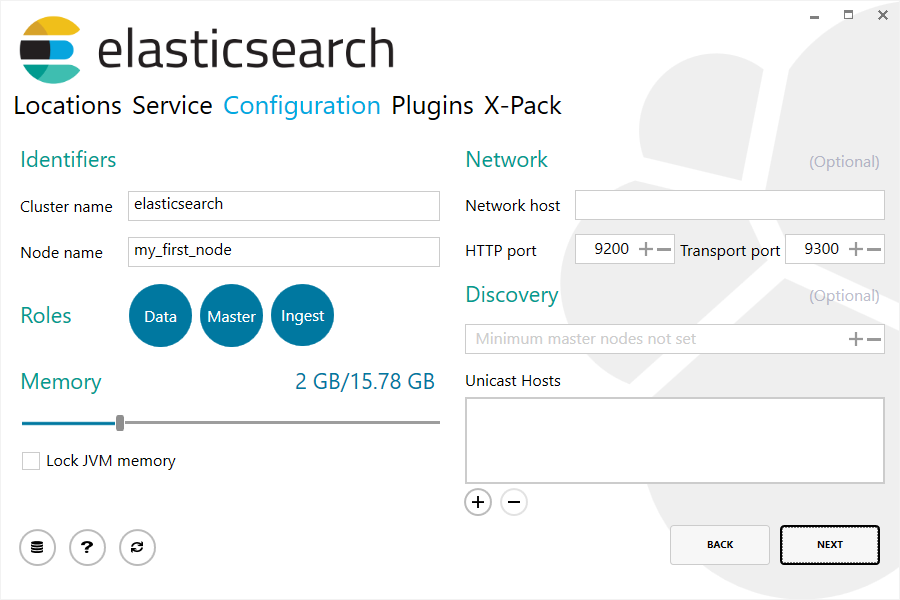
Then double-click the downloaded file to launch the GUI. Within the first screen, select the deployment directories:



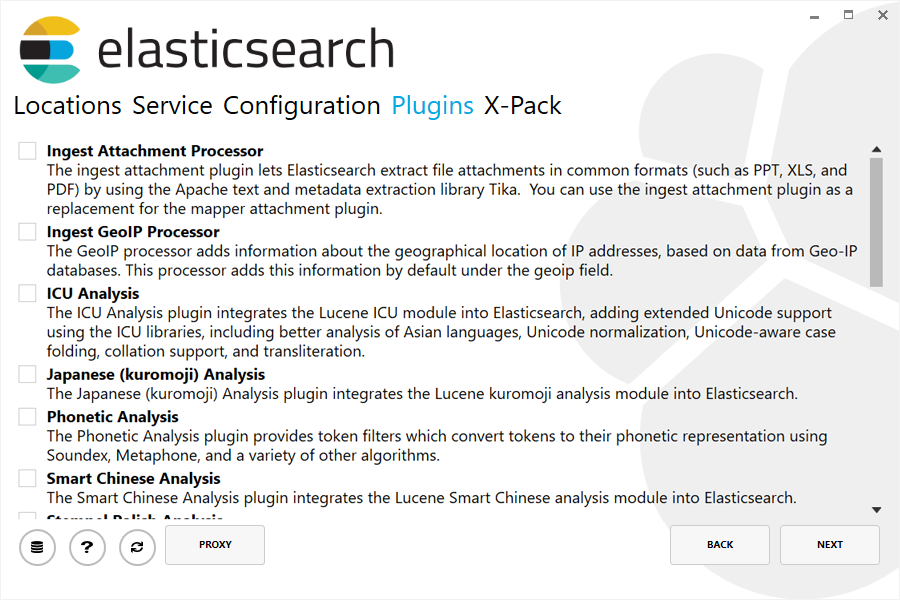
Then select whether to install as a service or start Elasticsearch manually as needed. To align with the tar example, choose not to install as a service:



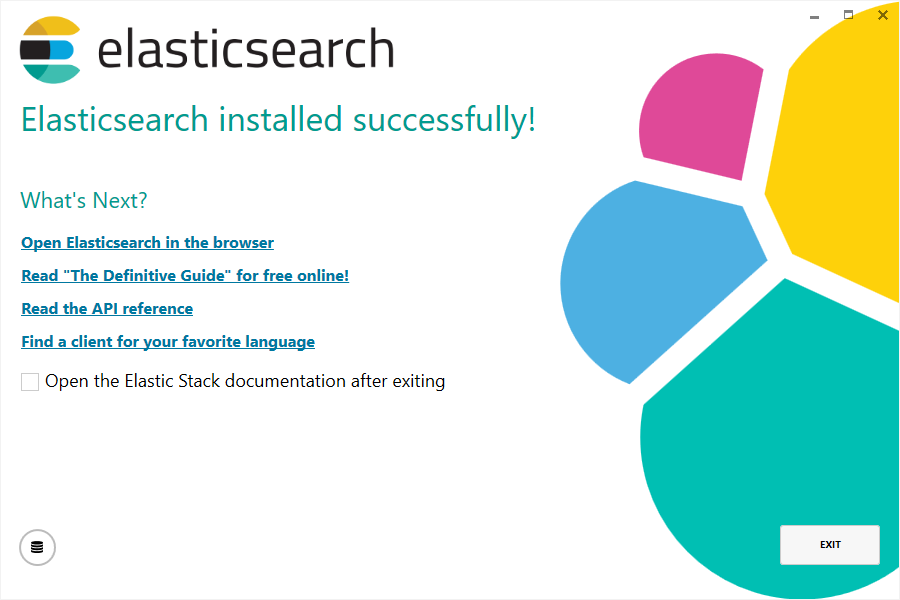
For configuration, simply leave the default values:



Again, to align with the tar example, uncheck all plugins to not install any plugins:



After clicking the install button, Elasticsearch will be installed:



By default, Elasticsearch will be installed at %PROGRAMFILES%\Elastic\Elasticsearch. Navigate here and go into the bin directory as follows:

**with Command Prompt:**

cd %PROGRAMFILES%\Elastic\Elasticsearch\bin

**with PowerShell:**

cd $env:PROGRAMFILES\Elastic\Elasticsearch\bin

And now we are ready to start our node and single cluster:

.\elasticsearch.exe

### Successfully running node

If everything goes well with installation, you should see a bunch of messages that look like below:

[2016-09-16T14:17:51,251][INFO ][o.e.n.Node ] [] initializing ...

[2016-09-16T14:17:51,329][INFO ][o.e.e.NodeEnvironment ] [6-bjhwl] using [1] data paths, mounts [[/ (/dev/sda1)]], net usable\_space [317.7gb], net total\_space [453.6gb], spins? [no], types [ext4]

[2016-09-16T14:17:51,330][INFO ][o.e.e.NodeEnvironment ] [6-bjhwl] heap size [1.9gb], compressed ordinary object pointers [true]

[2016-09-16T14:17:51,333][INFO ][o.e.n.Node ] [6-bjhwl] node name [6-bjhwl] derived from node ID; set [node.name] to override

[2016-09-16T14:17:51,334][INFO ][o.e.n.Node ] [6-bjhwl] version[6.5.4], pid[21261], build[f5daa16/2016-09-16T09:12:24.346Z], OS[Linux/4.4.0-36-generic/amd64], JVM[Oracle Corporation/Java HotSpot(TM) 64-Bit Server VM/1.8.0\_60/25.60-b23]

[2016-09-16T14:17:51,967][INFO ][o.e.p.PluginsService ] [6-bjhwl] loaded module [aggs-matrix-stats]

[2016-09-16T14:17:51,967][INFO ][o.e.p.PluginsService ] [6-bjhwl] loaded module [ingest-common]

[2016-09-16T14:17:51,967][INFO ][o.e.p.PluginsService ] [6-bjhwl] loaded module [lang-expression]

[2016-09-16T14:17:51,967][INFO ][o.e.p.PluginsService ] [6-bjhwl] loaded module [lang-mustache]

[2016-09-16T14:17:51,967][INFO ][o.e.p.PluginsService ] [6-bjhwl] loaded module [lang-painless]

[2016-09-16T14:17:51,967][INFO ][o.e.p.PluginsService ] [6-bjhwl] loaded module [percolator]

[2016-09-16T14:17:51,968][INFO ][o.e.p.PluginsService ] [6-bjhwl] loaded module [reindex]

[2016-09-16T14:17:51,968][INFO ][o.e.p.PluginsService ] [6-bjhwl] loaded module [transport-netty3]

[2016-09-16T14:17:51,968][INFO ][o.e.p.PluginsService ] [6-bjhwl] loaded module [transport-netty4]

[2016-09-16T14:17:51,968][INFO ][o.e.p.PluginsService ] [6-bjhwl] loaded plugin [mapper-murmur3]

[2016-09-16T14:17:53,521][INFO ][o.e.n.Node ] [6-bjhwl] initialized

[2016-09-16T14:17:53,521][INFO ][o.e.n.Node ] [6-bjhwl] starting ...

[2016-09-16T14:17:53,671][INFO ][o.e.t.TransportService ] [6-bjhwl] publish\_address {192.168.8.112:9300}, bound\_addresses {{192.168.8.112:9300}

[2016-09-16T14:17:53,676][WARN ][o.e.b.BootstrapCheck ] [6-bjhwl] max virtual memory areas vm.max\_map\_count [65530] likely too low, increase to at least [262144]

[2016-09-16T14:17:56,731][INFO ][o.e.h.HttpServer ] [6-bjhwl] publish\_address {192.168.8.112:9200}, bound\_addresses {[::1]:9200}, {192.168.8.112:9200}

[2016-09-16T14:17:56,732][INFO ][o.e.g.GatewayService ] [6-bjhwl] recovered [0] indices into cluster\_state

[2016-09-16T14:17:56,748][INFO ][o.e.n.Node ] [6-bjhwl] started

Without going too much into detail, we can see that our node named "6-bjhwl" (which will be a different set of characters in your case) has started and elected itself as a master in a single cluster. Don’t worry yet at the moment what master means. The main thing that is important here is that we have started one node within one cluster.

As mentioned previously, we can override either the cluster or node name. This can be done from the command line when starting Elasticsearch as follows:

./elasticsearch -Ecluster.name=my\_cluster\_name -Enode.name=my\_node\_name

Also note the line marked http with information about the HTTP address (192.168.8.112) and port (9200) that our node is reachable from. By default, Elasticsearch uses port 9200 to provide access to its REST API. This port is configurable if necessary.

## 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

Let’s start with a basic health check, which we can use to see how our cluster is doing. We’ll be using curl to do this but you can use any tool that allows you to make HTTP/REST calls. Let’s assume that we are still on the same node where we started Elasticsearch on and open another command shell window.

To check the cluster health, we will be using the [\_cat API](https://www.elastic.co/guide/en/elasticsearch/reference/6.5/cat.html). You can run the command below in [Kibana’s Console](https://www.elastic.co/guide/en/kibana/6.5/console-kibana.html) by clicking "VIEW IN CONSOLE" or with curl by clicking the "COPY AS CURL" link below and pasting it into a terminal.

GET /\_cat/health?v

And the response:

epoch timestamp cluster status node.total node.data shards pri relo init unassign pending\_tasks max\_task\_wait\_time active\_shards\_percent

1475247709 17:01:49 elasticsearch green 1 1 0 0 0 0 0 0 - 100.0%

We can see that our cluster named "elasticsearch" is up with a green status.

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.

Also from the above response, we can see a total of 1 node and that we have 0 shards since we have no data in it yet. Note that since we are using the default cluster name (elasticsearch) and since Elasticsearch uses unicast network discovery by default to find other nodes on the same machine, it is possible that you could accidentally start up more than one node on your computer and have them all join a single cluster. In this scenario, you may see more than 1 node in the above response.

We can also get a list of nodes in our cluster as follows:

GET /\_cat/nodes?v

And the response:

ip heap.percent ram.percent cpu load\_1m load\_5m load\_15m node.role master name

127.0.0.1 10 5 5 4.46 mdi \* PB2SGZY

Here, we can see our one node named "PB2SGZY", which is the single node that is currently in our cluster.

## List All Indices[edit](https://github.com/elastic/elasticsearch/edit/6.5/docs/reference/getting-started.asciidoc)

Now let’s take a peek at our indices:

GET /\_cat/indices?v

And the response:

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

Which simply means we have no indices yet in the cluster.

## Create an Index

Now let’s create an index named "customer" and then list all the indexes again:

PUT /customer?pretty

GET /\_cat/indices?v

**COPY AS CURL**[VIEW IN CONSOLE](http://localhost:5601/app/kibana#/dev_tools/console?load_from=https://www.elastic.co/guide/en/elasticsearch/reference/current/snippets/getting-started-create-index/1.json)

The first 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).

And the response:

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

yellow open customer 95SQ4TSUT7mWBT7VNHH67A 5 1 0 0 260b 260b

The results of the second command tells 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.

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,

"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

GET /\_cat/indices?v

And the response:

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

Which means that the index was deleted successfully and we are now back to where we started with nothing in our cluster.

Before we move on, let’s take a closer look again at some of the API commands that we have learned so far:

PUT /customer

PUT /customer/\_doc/1

{

"name": "John Doe"

}

GET /customer/\_doc/1

DELETE /customer

If we study the above commands carefully, we can actually see a pattern of how we access data in Elasticsearch. That pattern can be summarized as follows:

<HTTP Verb> /<Index>/<Type>/<ID>

This REST access pattern is so pervasive throughout all the API commands that if you can simply remember it, you will have a good head start at mastering Elasticsearch.

## 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

We’ve previously seen how we can index a single document. Let’s recall that command again:

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). See [docs-update-by-query API](https://www.elastic.co/guide/en/elasticsearch/reference/6.5/docs-update-by-query.html)

## 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

See the [\_delete\_by\_query API](https://www.elastic.co/guide/en/elasticsearch/reference/6.5/docs-delete-by-query.html) to delete all documents matching a specific query. It is worth noting that it is much more efficient to delete a whole index instead of deleting all documents with the Delete By Query API.

## 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](https://www.elastic.co/guide/en/elasticsearch/reference/6.5/docs-bulk.html). 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.

## Exploring Your Data

### Sample Dataset

Now that we’ve gotten a glimpse of the basics, let’s try to work on a more realistic dataset. I’ve prepared a sample of fictitious JSON documents of customer bank account information. Each document has the following schema:

{

"account\_number": 0,

"balance": 16623,

"firstname": "Bradshaw",

"lastname": "Mckenzie",

"age": 29,

"gender": "F",

"address": "244 Columbus Place",

"employer": "Euron",

"email": "bradshawmckenzie@euron.com",

"city": "Hobucken",

"state": "CO"

}

For the curious, this data was generated using [www.json-generator.com/](http://www.json-generator.com/), so please ignore the actual values and semantics of the data as these are all randomly generated.

### Loading the Sample Dataset

You can download the sample dataset (accounts.json) from [here](https://github.com/elastic/elasticsearch/blob/master/docs/src/test/resources/accounts.json?raw=true). Extract it to our current directory and let’s load it into our cluster as follows:

curl -H "Content-Type: application/json" -XPOST "localhost:9200/bank/\_doc/\_bulk?pretty&refresh" --data-binary "@accounts.json"

curl "localhost:9200/\_cat/indices?v"

And the response:

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

yellow open bank l7sSYV2cQXmu6\_4rJWVIww 5 1 1000 0 128.6kb 128.6kb

Which means that we just successfully bulk indexed 1000 documents into the bank index (under the \_doc type).

## The Search API

Now let’s start with some simple searches. There are two basic ways to run searches: one is by sending search parameters through the [REST request URI](https://www.elastic.co/guide/en/elasticsearch/reference/6.5/search-uri-request.html) and the other by sending them through the [REST request body](https://www.elastic.co/guide/en/elasticsearch/reference/6.5/search-request-body.html). The request body method allows you to be more expressive and also to define your searches in a more readable JSON format. We’ll try one example of the request URI method but for the remainder of this tutorial, we will exclusively be using the request body method.

The REST API for search is accessible from the \_search endpoint. This example returns all documents in the bank index:

GET /bank/\_search?q=\*&sort=account\_number:asc&pretty

**COPY AS CURL**[VIEW IN CONSOLE](http://localhost:5601/app/kibana#/dev_tools/console?load_from=https://www.elastic.co/guide/en/elasticsearch/reference/current/snippets/getting-started-search-API/1.json)

Let’s first dissect the search call. We are searching (\_search endpoint) in the bank index, and the q=\* parameter instructs Elasticsearch to match all documents in the index. The sort=account\_number:asc parameter indicates to sort the results using the account\_number field of each document in an ascending order. The pretty parameter, again, just tells Elasticsearch to return pretty-printed JSON results.

And the response (partially shown):

{

"took" : 63,

"timed\_out" : false,

"\_shards" : {

"total" : 5,

"successful" : 5,

"skipped" : 0,

"failed" : 0

},

"hits" : {

"total" : 1000,

"max\_score" : null,

"hits" : [ {

"\_index" : "bank",

"\_type" : "\_doc",

"\_id" : "0",

"sort": [0],

"\_score" : null,

"\_source" : {"account\_number":0,"balance":16623,"firstname":"Bradshaw","lastname":"Mckenzie","age":29,"gender":"F","address":"244 Columbus Place","employer":"Euron","email":"bradshawmckenzie@euron.com","city":"Hobucken","state":"CO"}

}, {

"\_index" : "bank",

"\_type" : "\_doc",

"\_id" : "1",

"sort": [1],

"\_score" : null,

"\_source" : {"account\_number":1,"balance":39225,"firstname":"Amber","lastname":"Duke","age":32,"gender":"M","address":"880 Holmes Lane","employer":"Pyrami","email":"amberduke@pyrami.com","city":"Brogan","state":"IL"}

}, ...

]

}

}

As for the response, we see the following parts:

* took – time in milliseconds for Elasticsearch to execute the search
* timed\_out – tells us if the search timed out or not
* \_shards – tells us how many shards were searched, as well as a count of the successful/failed searched shards
* hits – search results
* hits.total – total number of documents matching our search criteria
* hits.hits – actual array of search results (defaults to first 10 documents)
* hits.sort - sort key for results (missing if sorting by score)
* hits.\_score and max\_score - ignore these fields for now

Here is the same exact search above using the alternative request body method:

GET /bank/\_search

{

"query": { "match\_all": {} },

"sort": [

{ "account\_number": "asc" }

]

}

**COPY AS CURL**[VIEW IN CONSOLE](http://localhost:5601/app/kibana#/dev_tools/console?load_from=https://www.elastic.co/guide/en/elasticsearch/reference/current/snippets/getting-started-search-API/2.json)

The difference here is that instead of passing q=\* in the URI, we provide a JSON-style query request body to the \_search API. We’ll discuss this JSON query in the next section.

It is important to understand that once you get your search results back, Elasticsearch is completely done with the request and does not maintain any kind of server-side resources or open cursors into your results. This is in stark contrast to many other platforms such as SQL wherein you may initially get a partial subset of your query results up-front and then you have to continuously go back to the server if you want to fetch (or page through) the rest of the results using some kind of stateful server-side cursor.