# Elasticsearch Quick Start

Some commands:

|  |  |
| --- | --- |
| Action/Description | Rest URL |
| Using Kibana/Sense | <http://localhost:5601/app/sense> |
| List of all indexes (databases) | GET /\_cat/indices |
| All documents | GET /\_search |
| All documents in imdb index | GET /imdb/\_search |
| All docs in imdb with type = movies | GET /imdb/movies/\_search |
| Document with id = 4 | GET /imdb/movies/4 |
| This is the basic command that searches your database. Might get an error because there are currently no indexes. | GET \_search  {  "query": {  "match\_all": {}  }  } |
| Adding (indexing) three new documents (books in a bookshelf) | POST bookshelf/book/1  {  "title" : "Soft Skills",  "author" : "John Sonmez",  "pages" : 504  }  [[1]](#footnote-1)  POST bookshelf/book/2  {  "title" : "Mazes for Programmers",  "author" : "Jamis Buck",  "tags" : ["programming", "ruby", "algorithms" ]  }  POST bookshelf/book/3  {  "title" : "Clean Code",  "author" : "Bob Martin",  "pages" : 464 "tags" : ["programming", "software design", "architecture" ]  } |
| Get schema of type = book | GET bookshelf/book/\_mapping |
| Update book with id = 3 (fails because mapping is incorrect for pages…need to just make it a long value. | PUT bookshelf/book/3  {  "title" : “Clean Code 2",  "author" : "Bob Martin",  "pages" : ["464”] "tags" : ["programming", "software design", "architecture" ]  } |

Couple things:

Format is the <VERB> <index name>/<index type>/{id}

If a document with the same type and ID already exists it's overwritten.

Index and type are required while the id part is optional. If we don't specify an ID ElasticSearch will generate one for us.

However, if we don't specify an id we should use POST instead of PUT.

The index name is arbitrary. If there isn't an index with that name on the server already one will be created using default configuration. As for the type name it too is arbitrary. It serves several purposes, including:

Each type has its own ID space. Different types can have different mappings ("schema" that defines how properties/fields should be indexed).

Although it's possible, and common, to search over multiple types, it's easy to search only for one or more specific type(s).

When doing a PUT notice:

1) you get created:false

2) version is 2. The version number can be used to track how many times a document has been indexed.

It's primary purpose however is to allow for optimistic concurrency control as we can supply

a version in indexing requests as well and ElasticSearch will then only overwrite the document if the

supplied version is higher than what's in the index.

Optimistic concurrency control

Used by Elasticsearch, this approach assumes that conflicts are unlikely to happen and doesn’t block operations from being attempted. However, if the underlying data has been modified between reading and writing, the update will fail. It is then up to the application to decide how it should resolve the conflict. For instance, it could reattempt the update, using the fresh data, or it could report the situation to the user.

===> GET

GET movies/movie/8

//DO NOT SPECIFY A BODY OR YOU WILL DO A POST AND WIPE OUT THE RECORD!!!!!!!!!!!!!!!!!!!!

You can also use curl...but you'll still get the response without the body :(

http://www.confusedbycode.com/curl/

curl "http://localhost:9200/movies/movie/1"

Fidder, just do http://localhost:9200/movies/movie/1"

===> DELETE

curl -XDELETE "http://localhost:9200/movies/movie/1" -d''

===> Search (Advanced)

http://localhost:9200/\_search - Search across all indexes and all types.

http://localhost:9200/movies/\_search - Search across all types in the movies index.

http://localhost:9200/movies/movie/\_search - Search explicitly for documents of type movie within the movies index.

Now if you type the following in sense you get everything back:

GET \_search

One may wonder what the query DSL is. It's ElasticSearch's own domain specific language...

...based on JSON in which queries and filters can be expressed.

Think of it like ElasticSearch's equivalent of SQL for a relational database

===>QUERIES

The query DSL features a long list of different types of queries that we can use.

For "ordinary" free text search we'll most likely want to use one called "query string query"

1) ====> query string query

POST \_search

{

"query": {

"query\_string": {

"query": "keanu"

}

}

}

This will find all Keanu's in in the database on any field. Things to note:

- Notice, it's a POST not a GET because we have a body

- Not case sensitive

- Could have done POST

\_score and max\_score have to do with..... ranking

The following query is by field

{

"query": {

"query\_string": {

"query": "ford",

"fields": ["title"]

}

}

}

Notes:

- "fields" is case-sensitive

- To search in multiple fields, just keep adding elements to the array e.g. "fields" : ["title","Director"]

================================================================================================

# PART III => Working with Elasticsearch as a developer

1) first thru REST

2) second, thru NEST

1) REST==> fire up winforms app.

Nothing really to see here, but say that this isn't the way you'd do it. Maybe if you were writing your own "explorer". Hmmm…makes a nice win8 app ☺

Implementing basic search

1) Switch over to Visual Studio 2012

2) Load up Music Store MVC

2.5) Step thru the data model

3) Show the Reindex code in StoreManagerController + execute

4) Show the new Search action in the StoreController + execute

Implementing the Browse by genre

1) go to public ActionResult Browse(string genre) and notice the one that is there and comment in the elastic search one (comment out the original)

2) The point is that we might not want to do this, but rather than using the database, we can use elastic search to do the queries

3) Execute

Implementing the facets on the LHS (with counts!)

1) Go to public ActionResult GenreMenu(). Comment out the original, comment in the elasticsearch version

2) Talk through the code, maybe show the data in Sense

3) Execute

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Advanced

1) Run the code w/o highlighting using "chocolatez"

2) Now look at code. Notice that we are pulling from MongoDB (although we didn't need to). Use debug

3) Run the code w highlighting using "chocolatez"

4) Explain what highlighting is from the code

1. You should get back:

   {

   "\_index": "bookshelf",

   "\_type": "book",

   "\_id": "1",

   "\_version": 1,

   "\_shards": {

   "total": 2,

   "successful": 1,

   "failed": 0

   },

   "created": true

   } [↑](#footnote-ref-1)