ElasticSearch :

**#create an index**

**curl -XPUT http://127.0.0.1:9200/test**

**#{acknowledged":true}**

**#put a document**

**curl -XPUT http://127.0.0.1:9200/test/mytype/1 -d '{"name":"Paul", "age":35}'**

**# {"ok":true,"\_index":"test","\_type":"mytype","\_id":"1","\_ version":1}**

**#get the mapping and pretty print it**

**curl –XGET http://127.0.0.1:9200/test/mytype/\_ mapping?pretty=true**

**O/p**

{

"mytype" : {

"properties" : {

"age" : {

"type" : "long"

},

"name" : {

"type" : "string"

}

}

}

}

During the document's indexing phase, ElasticSearch checks whether the mytype type exists; if not, it creates the type dynamically.

ElasticSearch reads all the default properties for the field of the mapping and starts processing them:

1. If the field is already present in the mapping, and the value of the field is valid (that is, if it matches the correct type), then ElasticSearch does not need to change the current mapping.
2. If the field is already present in the mapping but the value of the field is of a different type, the type inference engine tries to upgrade the field type (such as from an integer to a long value). If the types are not compatible, then it throws an exception and the index process will fail.
3. If the field is not present, it will try to auto detect the type of field; it will also update the mapping to a new field mapping.

In ElasticSearch, the separation of documents in types is logical: the ElasticSearch core engine transparently manages it. Physically, all the document types go in the same Lucene index, so they are not fully separated .The user is not bothered about this internal management, but in some cases, with a huge amount of records, this has an impact on performance. This affects the reading and writing of records because all the records are stored in the same index file.

Every document has a unique identifier, called UID, for an index; it's stored in the special **\_uid** field of the document. It's automatically calculated by adding the type of the document to the \_**id** value. (In our example, the **\_uid** value will be **mytype#1**.) The \_**id** value can be provided at the time of indexing, or it can be assigned automatically by ElasticSearch if it's missing.

When a mapping type is created or changed, ElasticSearch automatically propagates mapping changes to all the nodes in the cluster so that all the shards are aligned.

Using explicit mapping allows you to be faster when you start inserting data using a schema-less approach, without being concerned about the field types. Therefore, in order to achieve better results and performance when indexing, it's necessary to manually define a mapping.

Fine-tuning the mapping has some advantages, as follows:

1. Reduces the size of the index on disk (disabling functionalities for custom fields)
2. Indexes only *interesting* fields (a general boost to performance)
3. *Precooks* data for a fast search or real-time analytics (such as aggregations)
4. Correctly defines whether a field must be analyzed in multiple tokens or whether it should be considered as a single token.

Example:

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| id | Identifier | Order identifier |
| date | Date (time) | Date of order |
| customer\_id | Id reference | Customer ID reference |
| name | String | Name of the item |
| quantity | Integer | Number of items |
| vat | Double | VAT for the item |
| sent | Boolean | Status, if the order was sent |

{

"order" : {

"properties" : {

"id" : {"type" : "string", "store" : "yes" , "index":"not\_analyzed"},

"date" : {"type" : "date", "store" : "no" , "index":"not\_analyzed"},

"customer\_id" : {"type" : "string", "store" : "yes" , "index":"not\_analyzed"},

"sent" : {"type" : "boolean", "index":"not\_analyzed"},

"name" : {"type" : "string", "index":"analyzed"},

"quantity" : {"type" : "integer", "index":"not\_analyzed"},

"vat" : {"type" : "double", "index":"no"}

}

}

}

Properties:

**store**: This marks the field to be stored in a separate index fragment for fast retrieval. Storing a field consumes disk space, but it reduces computation if you need to extract the field from a document (that is, in scripting and aggregations). The possible values for this option are no and yes (the default value is no). Stored fields are faster than others at faceting.

**index**: This configures the field to be indexed (the default value is analyzed). The following are the possible values for this parameter:

* + **no**: This field is not indexed at all. It is useful to hold data that must not be searchable.
  + **analyzed**: This field is analyzed with the configured analyzer. It is generally lowercased and tokenized, using the default ElasticSearch configuration (StandardAnalyzer).
  + **not**\_**analyzed**: This field is processed and indexed, but without being changed by an analyzer. The default ElasticSearch configuration uses the KeywordAnalyzer field, which processes the field as a single token.

**null\_value**: This defines a default value if the field is missing.

**boost**: This is used to change the *importance* of a field (the default value is 1.0).

**index\_analyzer**: This defines an analyzer to be used in order to process a field. If it is not defined, the analyzer of the parent object is used (the default value is null).

**search\_analyzer**: This defines an analyzer to be used during the search. If it is not defined, the analyzer of the parent object is used (the default value is null).

**analyzer**: This sets both the index\_analyzer and search\_analyzer field to the defined value (the default value is null).

**include\_in\_all**: This marks the current field to be indexed in the special \_all field (a field that contains the concatenated text of all the fields). The default value is true.

**index\_name:** This is the name of the field to be stored in the Index. This property allows you to rename the field at the time of indexing. It can be used to manage data migration in time without breaking the application layer due to changes.

**norms**: This controls the Lucene norms. This parameter is used to better score queries, if the field is used only for filtering. Its best practice to disable it in order to reduce the resource usage (the default value is true for analyzed fields and false for the not\_analyzed ones).

An important parameter, available only for string mapping, is the **term\_vector** field (the vector of the terms that compose a string; check out the Lucene documentation for further details at http://lucene.apache.org/core/4\_4\_0/core/org/apache/lucene/ index/Terms.html) to define the details:

* no: This is the default value, which skips term\_vector field
* yes: This stores the term\_vector field
* with\_offsets: This stores term\_vector with a token offset (the start or end position in a block of characters)
* with\_positions: This stores the position of the token in the term\_vector field
* with\_positions\_offsets: This stores all the term\_vector data

Term vectors allow fast highlighting but consume a lot of disk space due to the storage of additional text information. It's best practice to activate them only in the fields that require highlighting, such as title or document content

**Multi value attribute:**

ElasticSearch transparently manages the array; there is no difference whether you declare a single value or multiple values, due to its Lucene core nature.

eg.

{"name": "document2", "tag": ["cool", "awesome", "amazing"]}

mapping:

**"tag" : {"type" : "string", "store" : "yes" , "index":"not\_ analyzed"}**,…

For people with a SQL background, this behavior might be strange, but this is a key point in the NoSQL world as it reduces the need for a join query and the need to create different tables to manage multiple values. An array of embedded objects has the same behavior as that of simple fields.

**Mapping Objects:**

ElasticSearch extends the traditional use of objects, allowing the use of recursive embedded objects.

{

"order" : {

"properties" : {

"id" : {"type" : "string", "store" : "yes", "index":"not\_analyzed"},

"date" : {"type" : "date", "store" : "no", "index":"not\_analyzed"},

"customer\_id" : {"type" : "string", "store" : "yes", "index":"not\_analyzed"},

"sent" : {"type" : "boolean", "store" : "no", "index":"not\_analyzed"},

**"item" : {**

**"type" : "object",**

**"properties" : {**

**"name" : {"type" : "string", "store" : "no", "index":"analyzed"},**

**"quantity" : {"type" : "integer", "store" : "no", "index":"not\_analyzed"},**

**"vat" : {"type" : "double", "store" : "no", "index":"not\_analyzed"}**

**}**

**}**

}

}

}

The following are the most important attributes for an object:

**properties**: This is a collection of fields or objects (we consider them as columns in the SQL world).

**enabled**: This is enabled if the object needs to be processed. If it's set to false, the data contained in the object is not indexed as it cannot be searched (the default value is true).

**dynamic**: This allows ElasticSearch to add new field names to the object using reflection on the values of inserted data (the default value is true). If it's set to false, when you try to index an object containing a new field type, it'll be rejected silently. If it's set to strict, when a new field type is present in the object, an error is raised and the index process is skipped. Controlling the dynamic parameter allows you to be safe about changes in the document structure.

**include\_in\_all**: This adds the object values (the default value is true) to the special \_all field (used to aggregate the text of all the document fields).

Java to ES mapping:

|  |  |  |
| --- | --- | --- |
| Type | ES type | Description |
| String, VarChar, Text | string | A text field: such as a nice text and CODE0011 |
| Integer | integer | An integer (32 bit): such as 1, 2, 3, 4 |
| Long | long | A long value (64 bit) |
| Float | float | A floating-point number (32 bit): such as 1, 2, 4, 5 |
| Double | double | A floating-point number (64 bit) |
| Boolean | boolean | A Boolean value: such as true, false |
| Date/Datetime | date | A date or datetime value: such as 2013-12-25, 2013-12-25T22:21:20 |
| Bytes/Binary | binary | This is used for binary data such as a file or stream of bytes. |