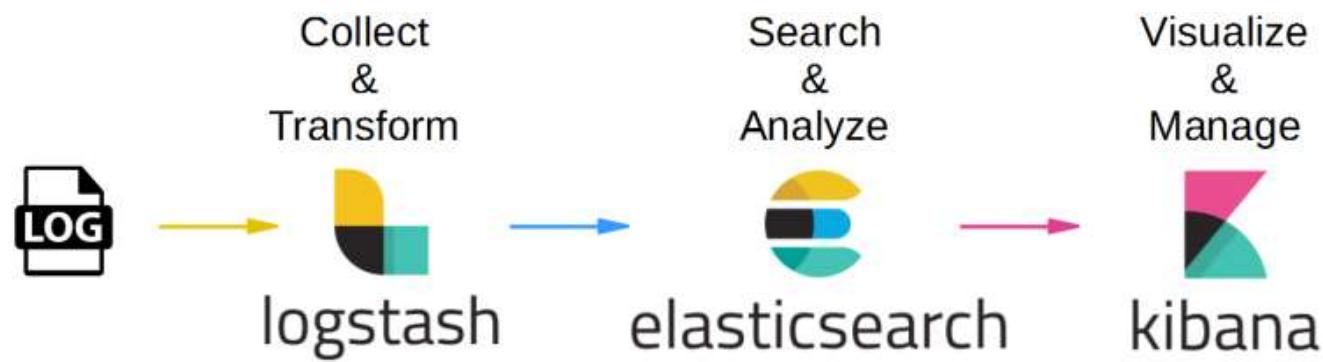
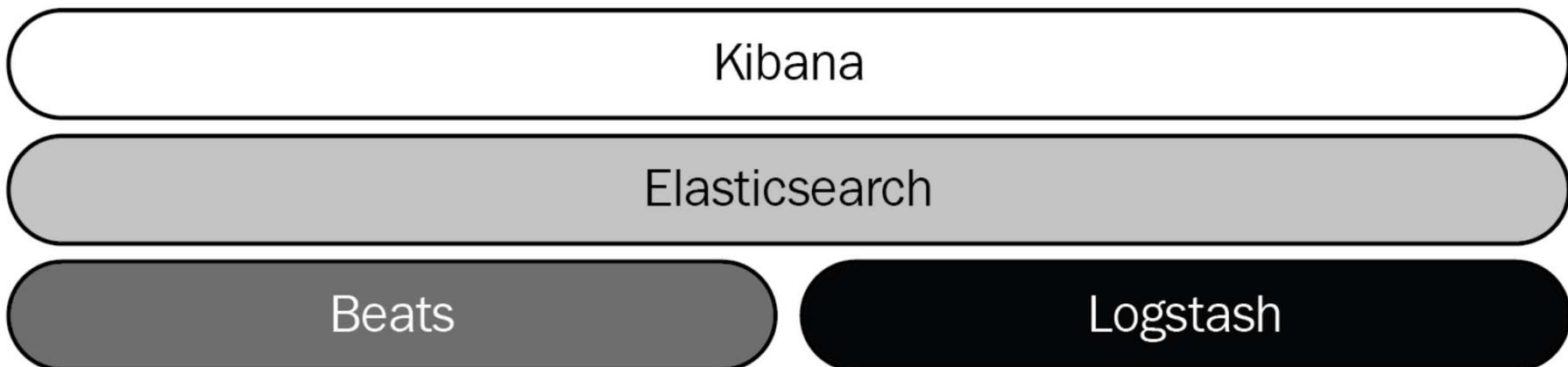


# Elasticsearch

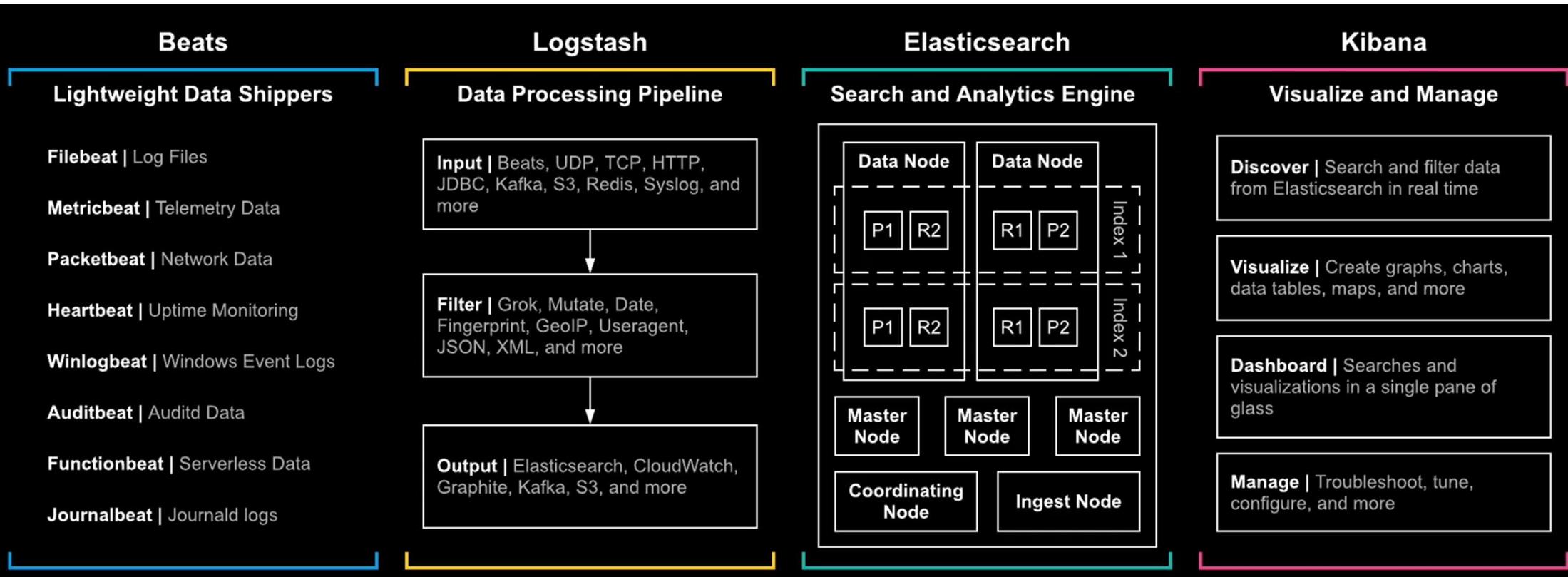


# Getting Started

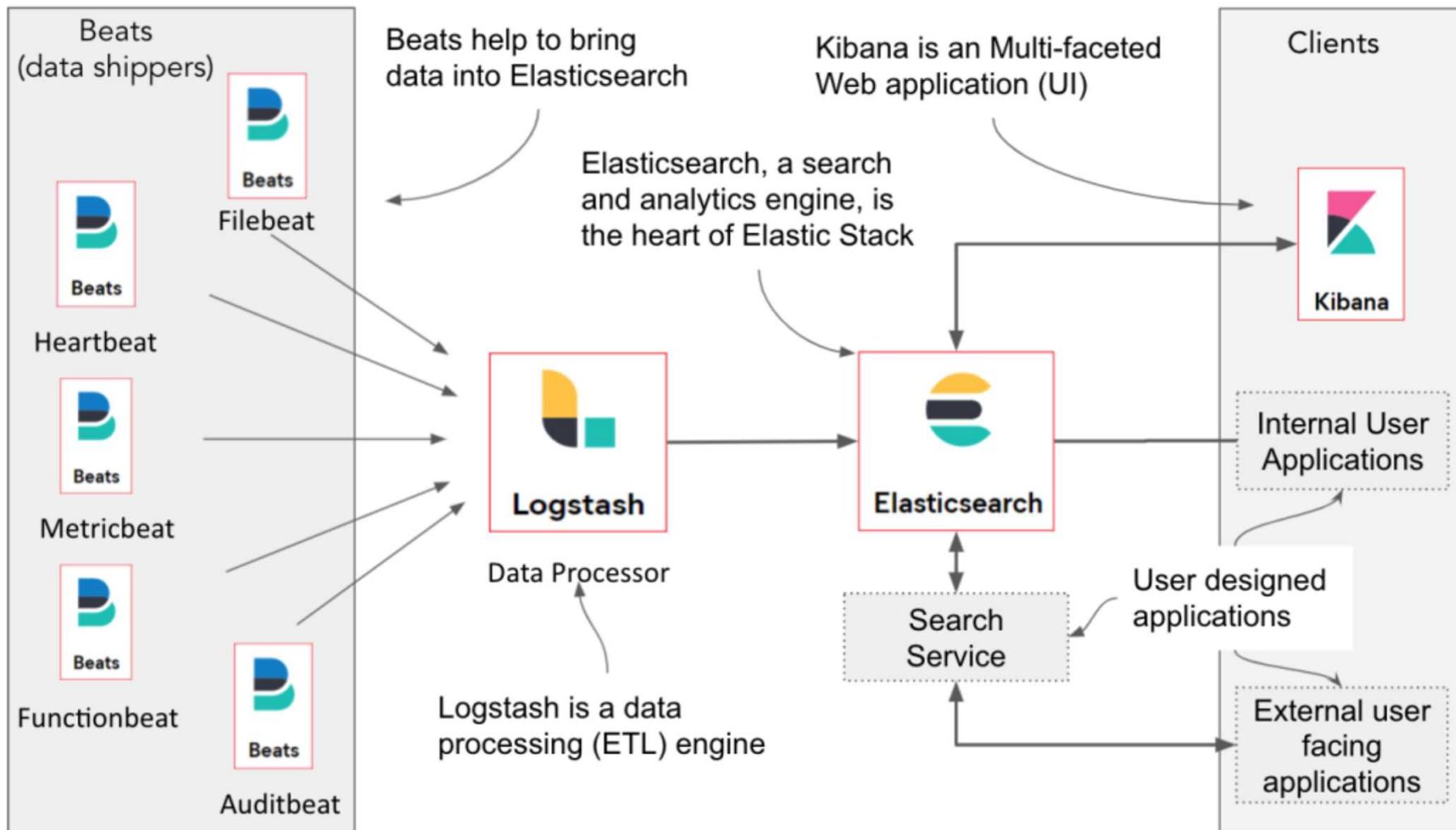
# Components of the Elastic Stack



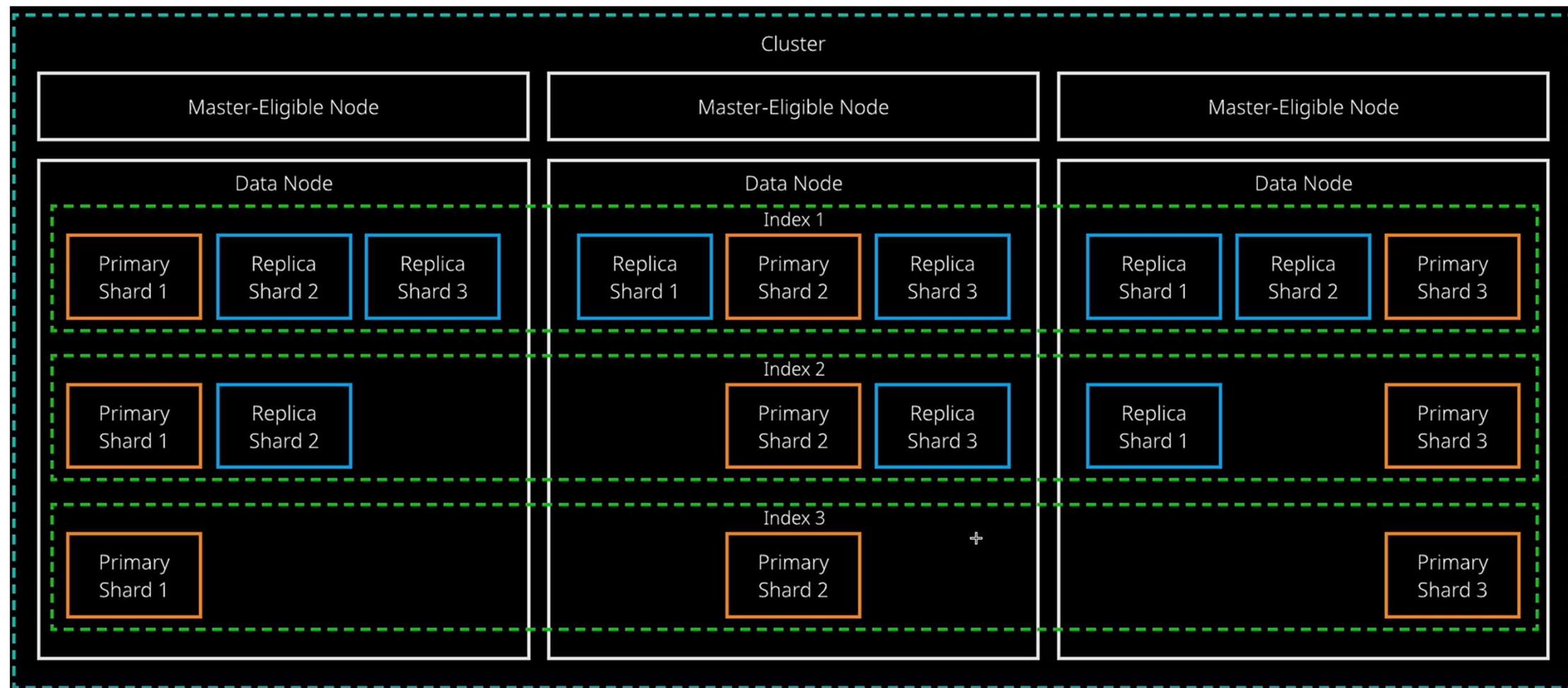
# Components of the Elastic Stack



# Elastic Stack ecosystem



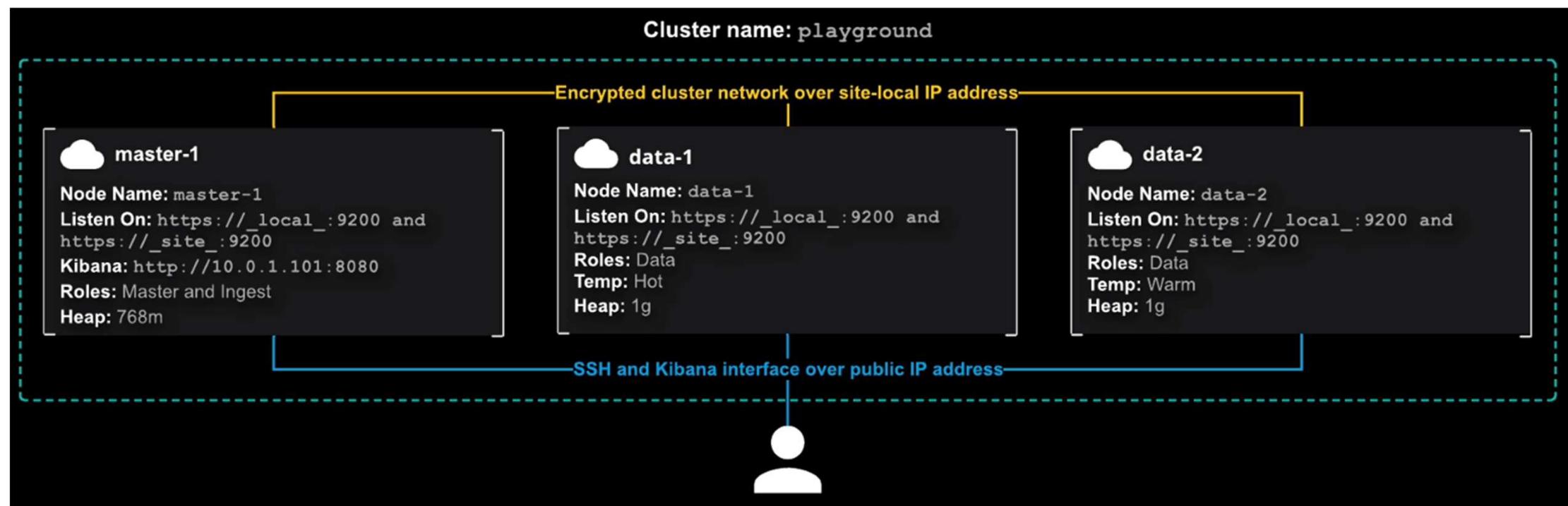
# Elasticsearch



# Hands-on

# Installation and Configuration

# Our deployment Architecture



# Hands-on

# Security

# Hands-on

# Index and Data Model

# Data model for a book represented in a tabular format

Field	Explanation	Example
title	The title of a book	"Effective Java"
author	The author of the book	"Joshua Bloch"
release_date	Data of release	01-06-2001
amazon_rating	Average rating on Amazon	4.7
best_seller	Flag that qualifies the book as a best seller	true
prices	Inner object with individual prices in three currencies	<pre>"prices": {     "usd": 9.95,     "gbp": 7.95,     "eur": 8.95 }</pre>

# A JSON representation of a book entity

```
{  
    "title": "Effective Java",  
    "author": "Joshua Bloch",  
    "release_date": "2001-06-01",  
    "amazon_rating": 4.7,  
    "best_seller": true,  
    "prices": {  
        "usd": 9.95,  
        "gbp": 7.95,  
        "eur": 8.95  
    }  
}
```

Title and author of the book, both textual information

Rating of the book in floating point data

An inner-inner object representing individual prices of the book

Release date of the book

A boolean flag to indicate the status of the book

Individual prices of the book in different currencies

# Indexing two more documents

```
PUT books/_doc/2
{
  "title": "Core Java Volume I - Fundamentals",
  "author": "Cay S. Horstmann",
  "release_date": "2018-08-27",
  "amazon_rating": 4.8,
  "best_seller": true,
  "prices": {
    "usd": 19.95,
    "gbp": 17.95,
    "eur": 18.95
  }
}
```



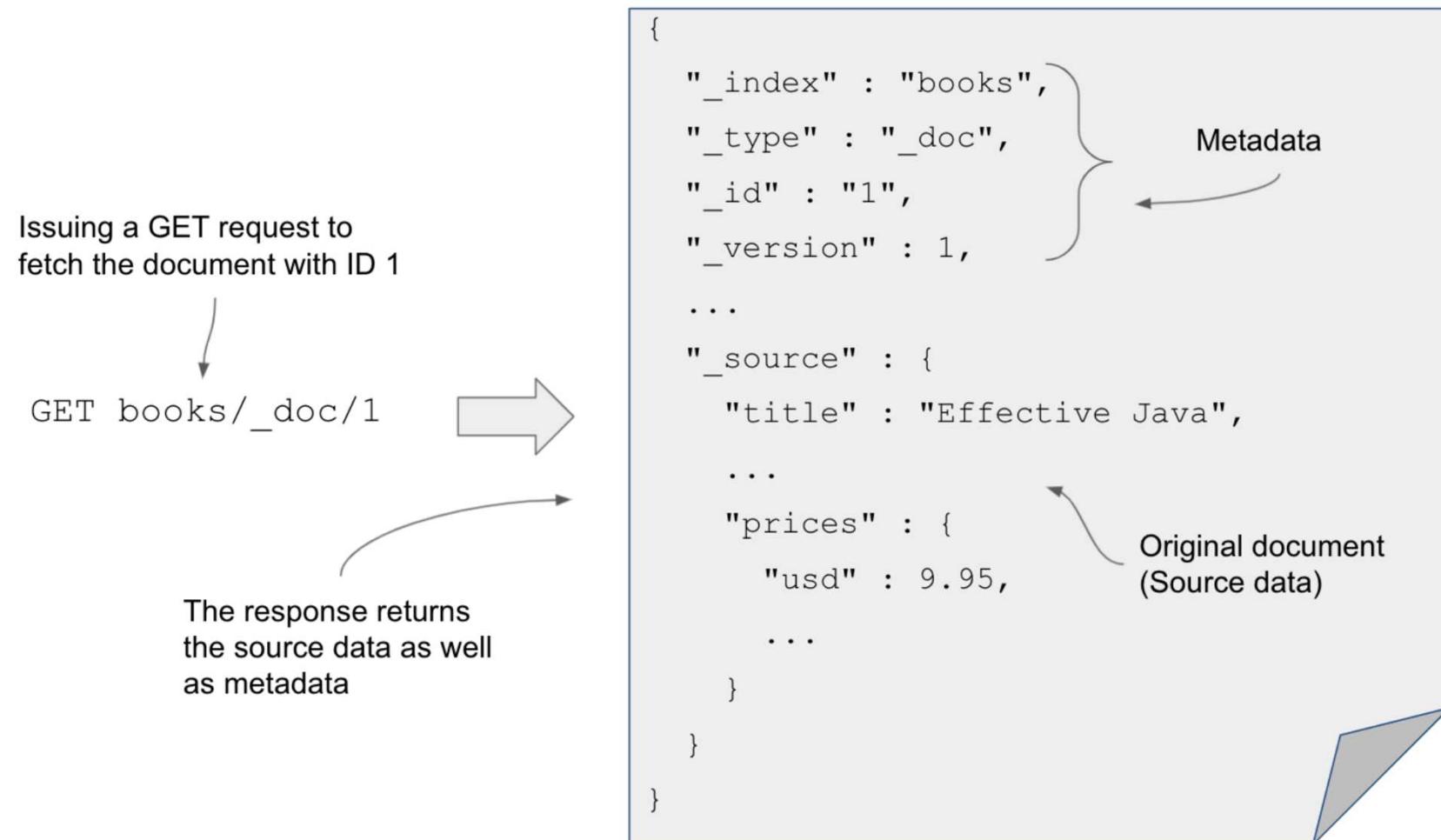
Indexing a document with ID 2

```
PUT books/_doc/3
{
  "title": "Java: A Beginner's Guide",
  "author": "Herbert Schildt",
  "release_date": "2018-11-20",
  "amazon_rating": 4.2,
  "best_seller": true,
  "prices": {
    "usd": 19.99,
    "gbp": 19.99,
    "eur": 19.99
  }
}
```



Indexing a document with ID 3

# Fetching a book document by an ID



# Retrieving documents given a set of IDs using an ids

A GET request on a `_search` endpoint with a query to fetch the multiple documents

GET books/\_**search**

```
{  
  "query": {  
    "ids": {  
      "values": [1,2,3]  
    }  
  }  
}
```

The `ids` query expects an array with document IDs

The response returns all three documents

```
"hits" : [  
  {"_index" : "books",  
   "_type" : "_doc",  
   "_id" : "1",  
   "_source" : {  
     "title" : "Effective Java",  
     ..  
   }},  
  {"_index" : "books",  
   "_id" : "2",  
   ...  
 },  
  ...  
 ]
```

# Retrieving all documents using the search API

The generic search (with no request body) will return all the books from the index

GET books/\_search



The actual book is enclosed in the \_source object

```
{  
  ...  
  "hits" : {  
    "total" : {  
      "value" : 3,  
      "relation" : "eq"  
    },  
    "max_score" : 1.0,  
    "hits" : [  
      {  
        "_index" : "books",  
        "_type" : "_doc",  
        "_id" : "1",  
        "_score" : 1.0,  
        "_source" : {  
          "title" : "Effective Java",  
          "author" : "Joshua Bloch"  
        }  
      },  
      ...  
    ]  
  }  
}
```

The hits object indicates the number of results returned

The hits array is composed of the actual returned results

# Fetching books authored by Joshua

A match query fetching all books written by Joshua

```
GET books/_search
{
  "query": {
    "match": {
      "author": "Joshua"
    }
  }
}
```

The match query with an author clause

The response returns document with a match.

```
"hits" : [
  {
    "_index" : "books",
    "_type" : "_doc",
    "_id" : "1",
    "_score" : 1.0417082,
    "_source" : {
      "title" : "Effective Java",
      "author" : "Joshua Bloch"
      ...
    }
  }
]
```

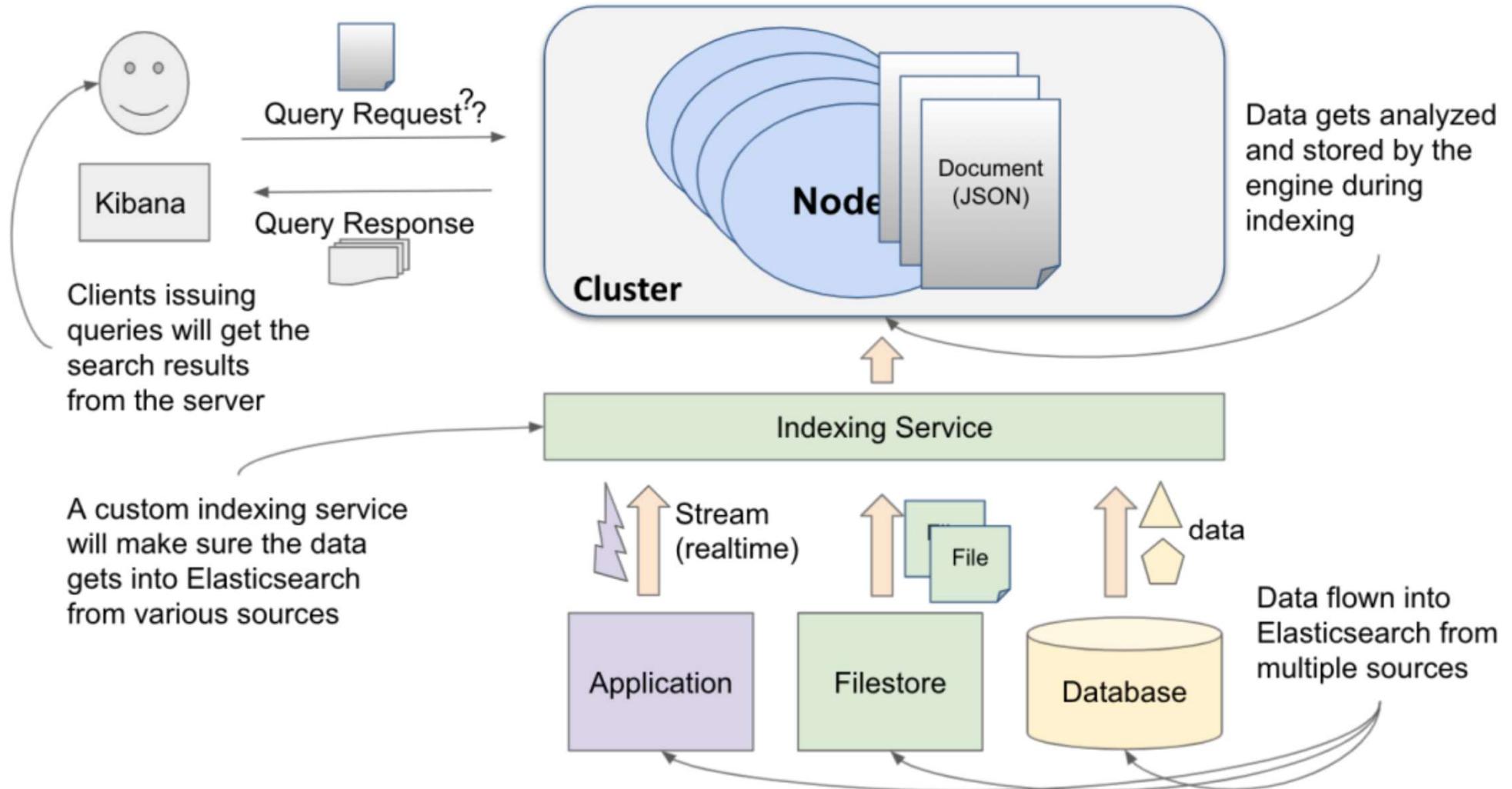
# Fetching an exact match for a title using the and operator

A match query fetching a specific title. The and operator searches for a title with both words in the title ("Effective and Java")

```
GET books/_search
{
  "query": {
    "match": {
      "title": {
        "query": "Effective Java",
        "operator": "and"
      }
    }
  }
}
```



# Elasticsearch with data



# A JSON document vs. a relational database table structure

JSON representation of a Student data in Elasticsearch (the data is denormalized)

```
{  
    "title": "John Doe",  
    "date_of_birth": "1972-14-03",  
    "age": 23,  
    "address": {  
        //..  
    }  
}
```

Records are split into two individual tables and joined up by a foreign key between these two tables

Student data represented as relational data in a database (the data is normalized)

↓

STUDENT TABLE

ID	TITLE	DATE_OF_BIRTH	AGE	ADDRESS_ID
1	John Doe	1972-14-03	23	123456

↓

ADDRESS TABLE

ADDRESS_ID	ADDRESS_LINE1	POSTCODE
123456	34, Johndoe Land, London	LD1DNN

# Removing document types from Elasticsearch

The url consists of the type (car) of the document too (which is deprecated and be removed in version 8.0)

```
PUT cars/car/1
{
  "make": "Toyota",
  "model": "Avensis"
}
```

> V7.0: The explicit type is replaced with an endpoint named `_doc` (not document type)

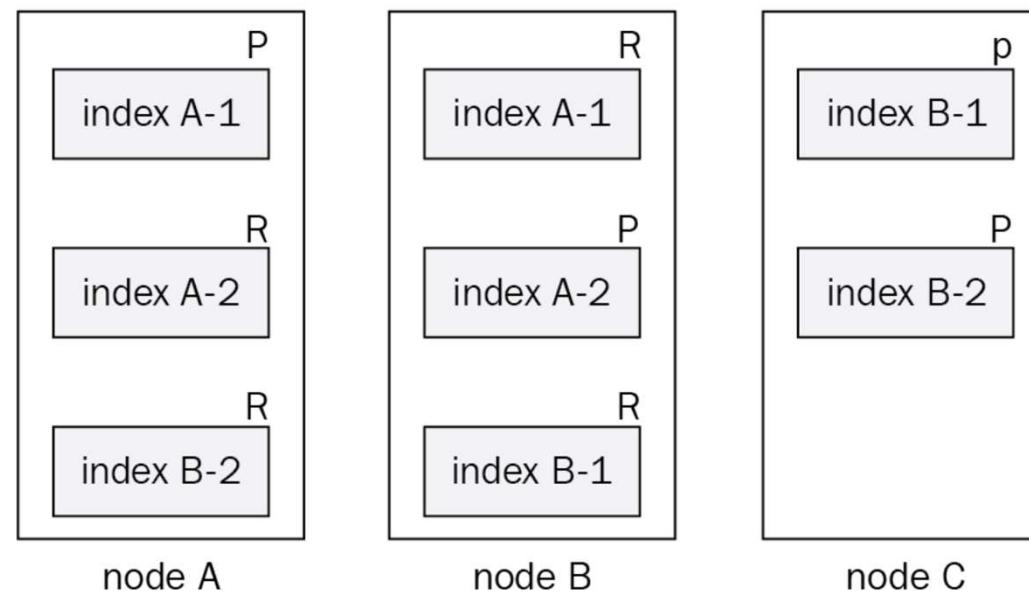
```
PUT cars/_doc/1
{
  ...
}
```

#! [types removal] Specifying types in document index requests is deprecated, use the typeless endpoints instead  
({index}/\_doc/{id}, /{index}/\_doc, or /{index}/\_create/{id}).

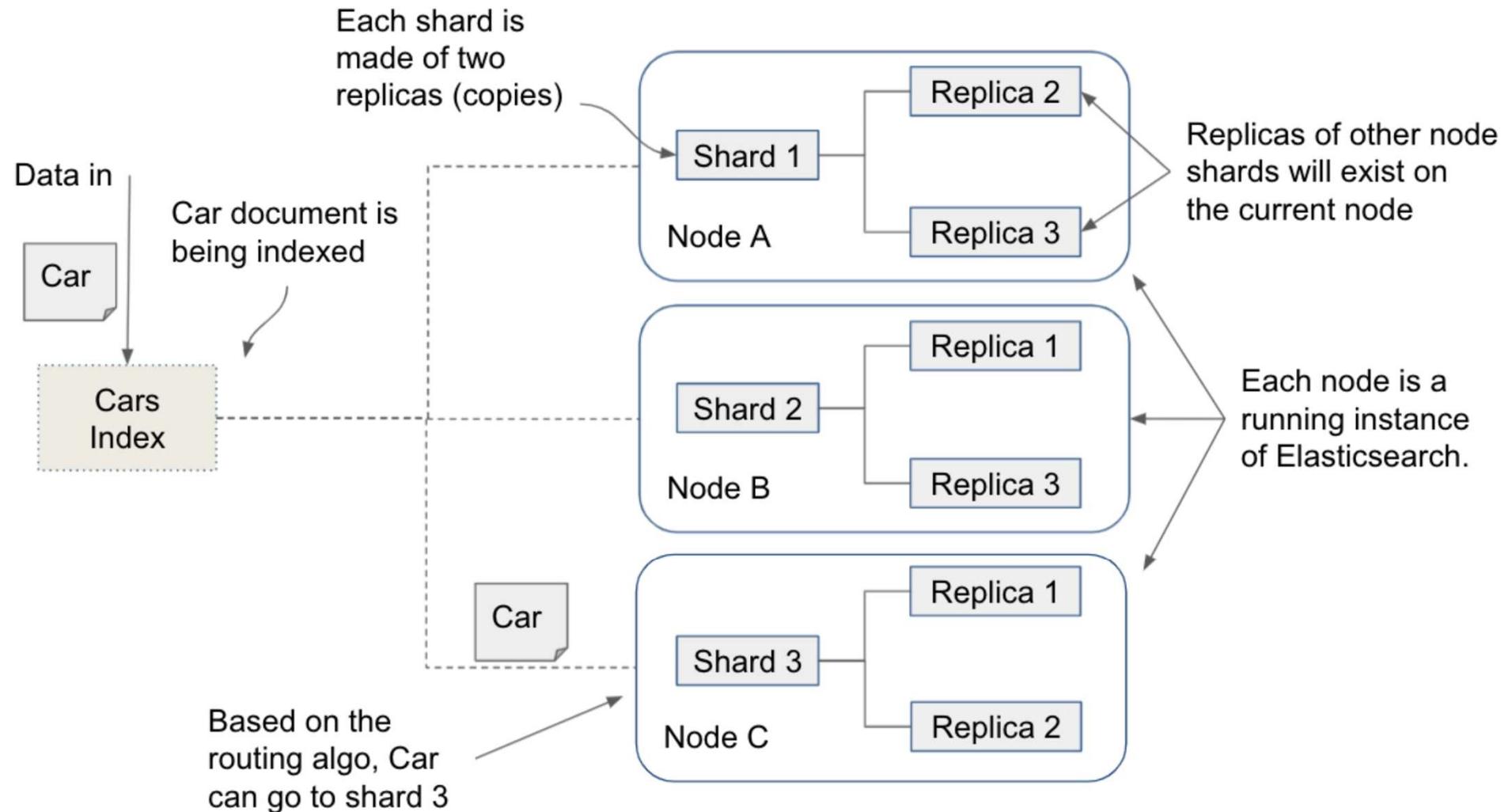
```
{
  "_index": "cars",
  "_type": "car",
  "_id": "1",
  "_version": 1,
  "result": "created",
  "_shards": {
    "total": 2,
    "successful": 1,
    "failed": 0
  },
  "_seq_no": 0,
  "_primary_term": 1
}
```

While using the type upto 7.x is allowed (you will receive a warning as shown here), it is advisable to drop the type completely

# Shards of data distributed across nodes



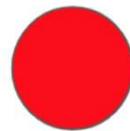
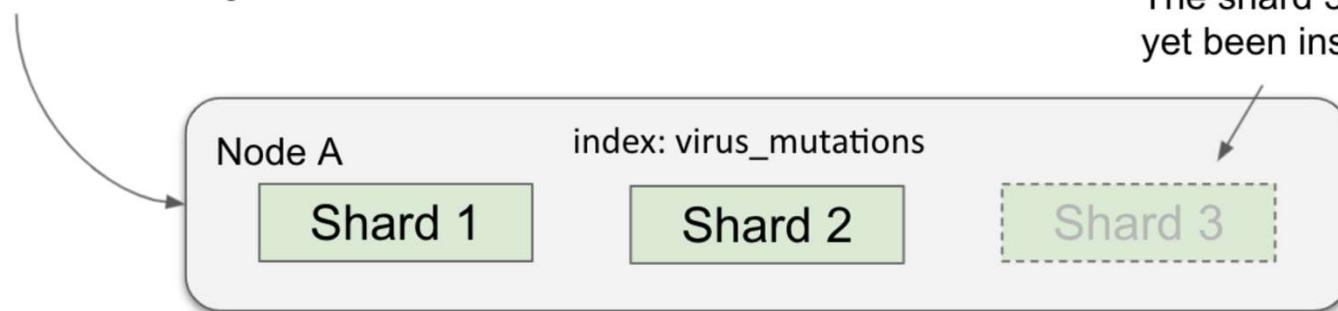
# An index designed with three shards and three replicas



# Engine is not ready

Node A has two primary shards ready  
and third shard is being instantiated

The shard 3 has not yet been instantiated

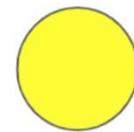
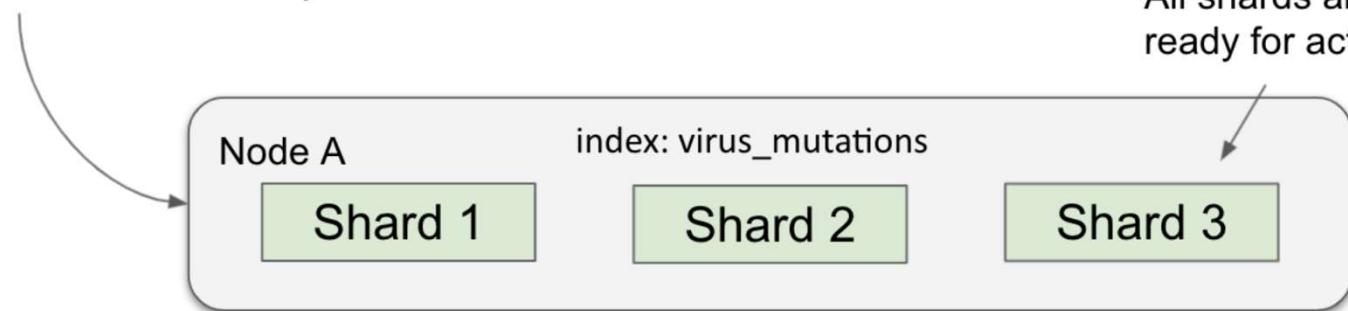


All shards are not yet assigned.  
Cluster status is RED

# Single node with three shards joining a single-node cluster

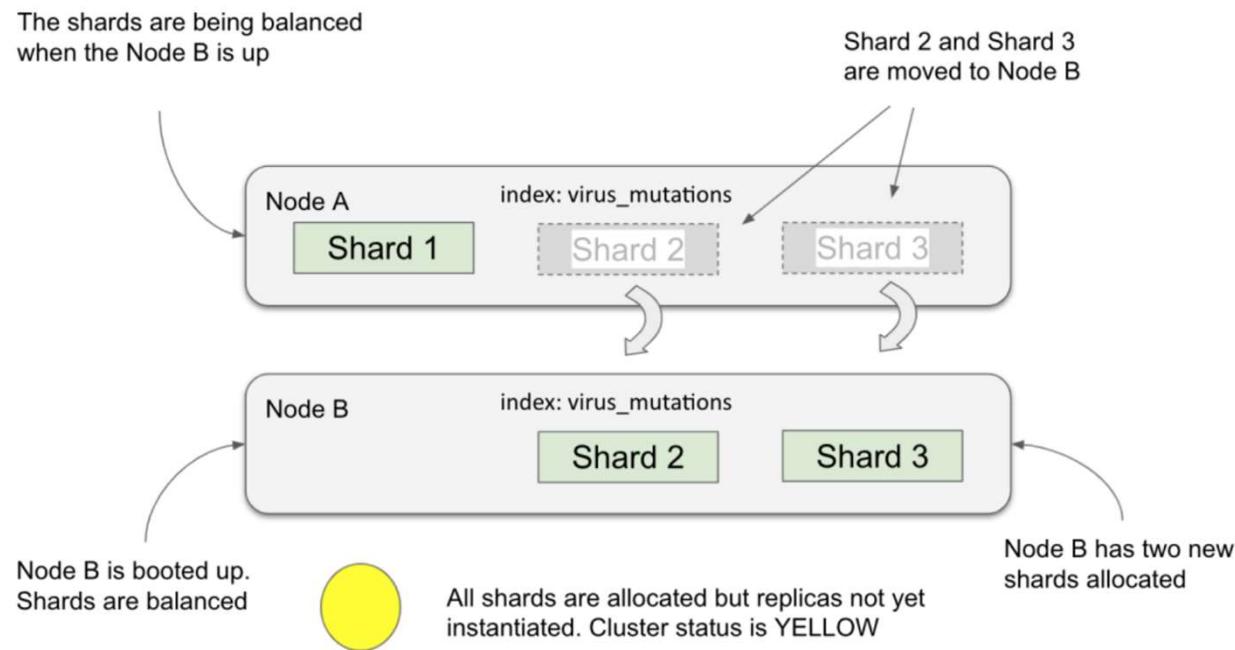
Node A has three primary shards,  
all instantiated and ready

All shards are  
ready for action

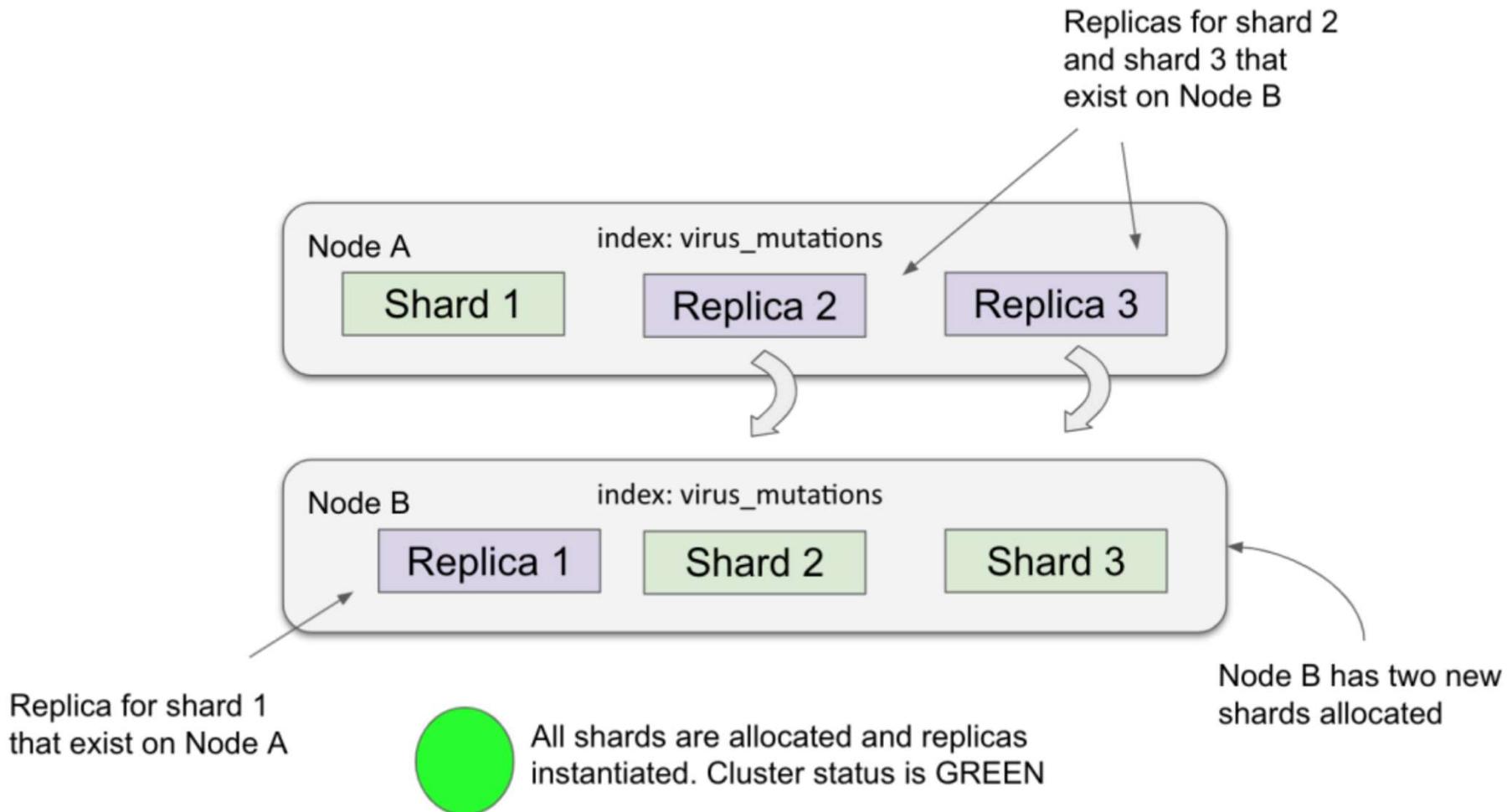


All shards are ready but replicas are not yet assigned. Cluster status is YELLOW

Shards were balanced on to the new node but replicas aren't assigned yet (yellow status).



# All shards and replicas are allocated



# Health of shards using a traffic light signal board

**RED**

Not all shards are assigned and ready  
(cluster being prepared state)

**YELLOW**

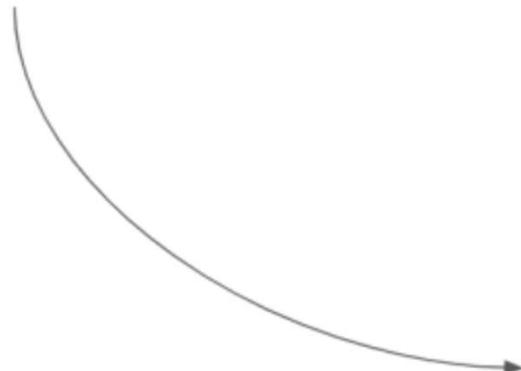
Shards are assigned and ready but  
replicas aren't assigned and ready

**GREEN**

Shards and replicas are all assigned  
and ready

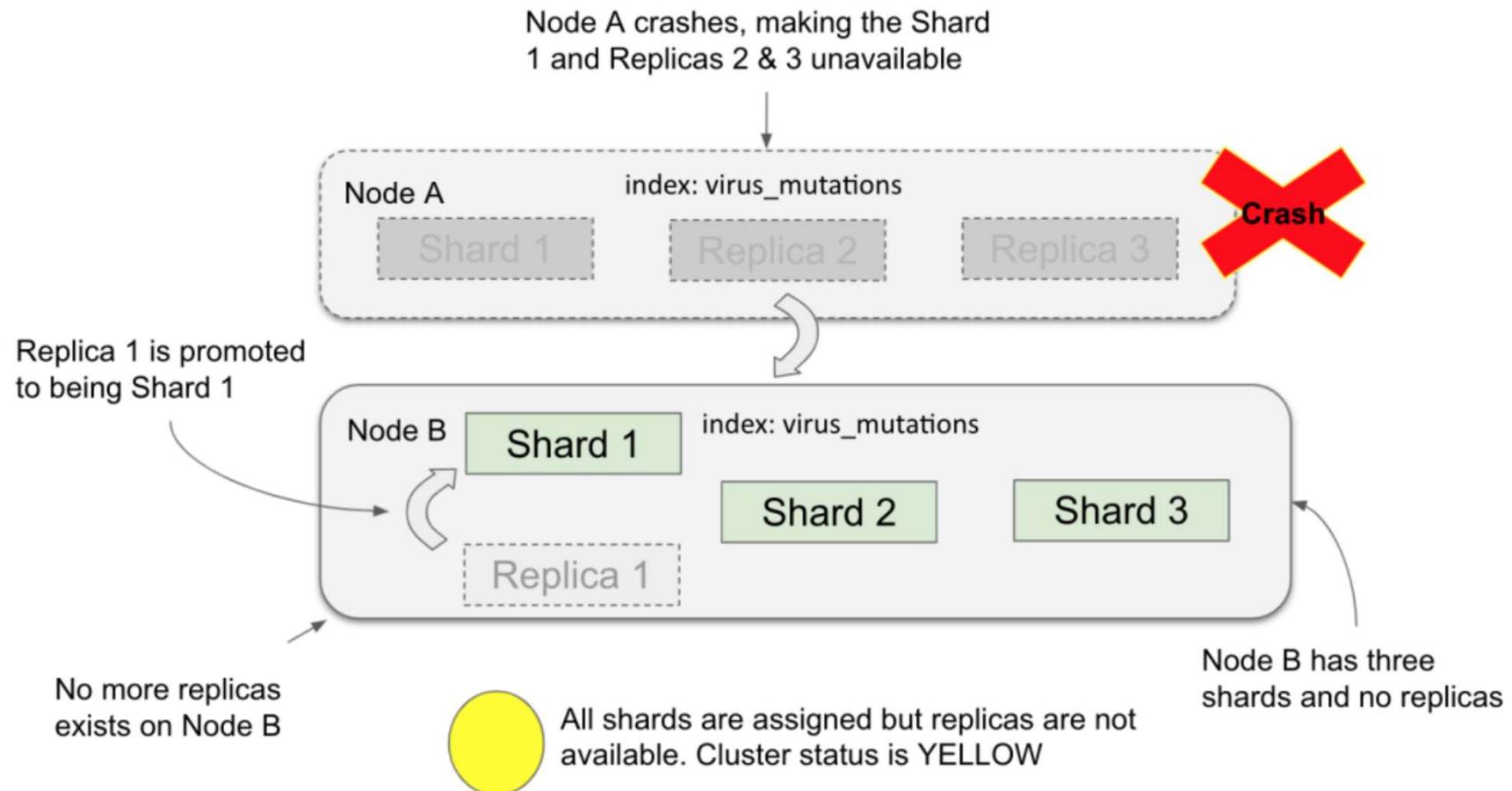
# Fetching the status of a cluster

```
GET _cluster/health
```



```
{  
    "cluster_name" : "elasticsearch",  
    "status" : "red",  
    "timed_out" : false,  
    "number_of_nodes" : 1,  
    "number_of_data_nodes" : 1,  
    "active_primary_shards" : 25,  
    "active_shards" : 25,  
    "unassigned_shards" : 24  
    ...  
}
```

# Replicas were lost (or promoted to a shard) when a node crashed.

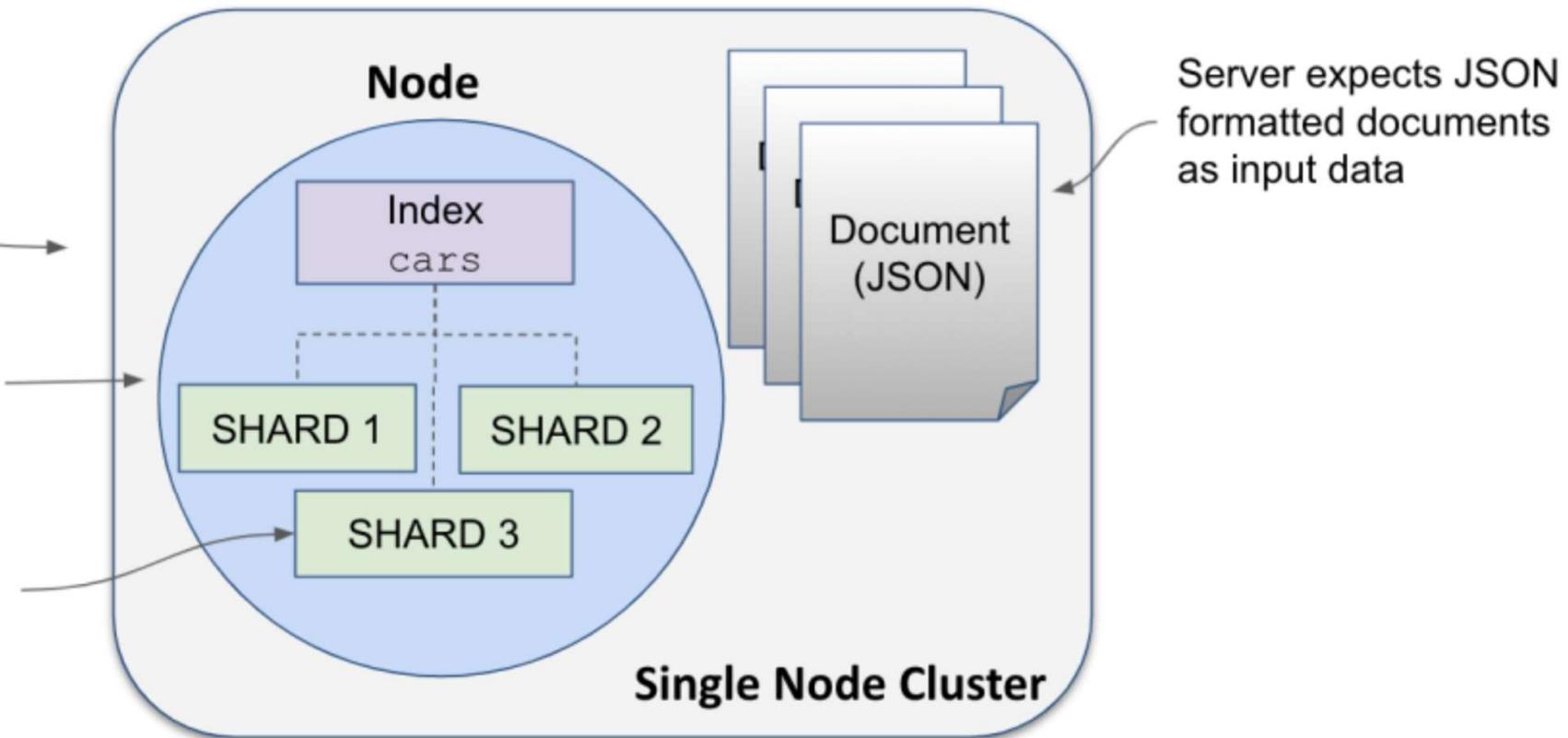


# A single node Elasticsearch cluster

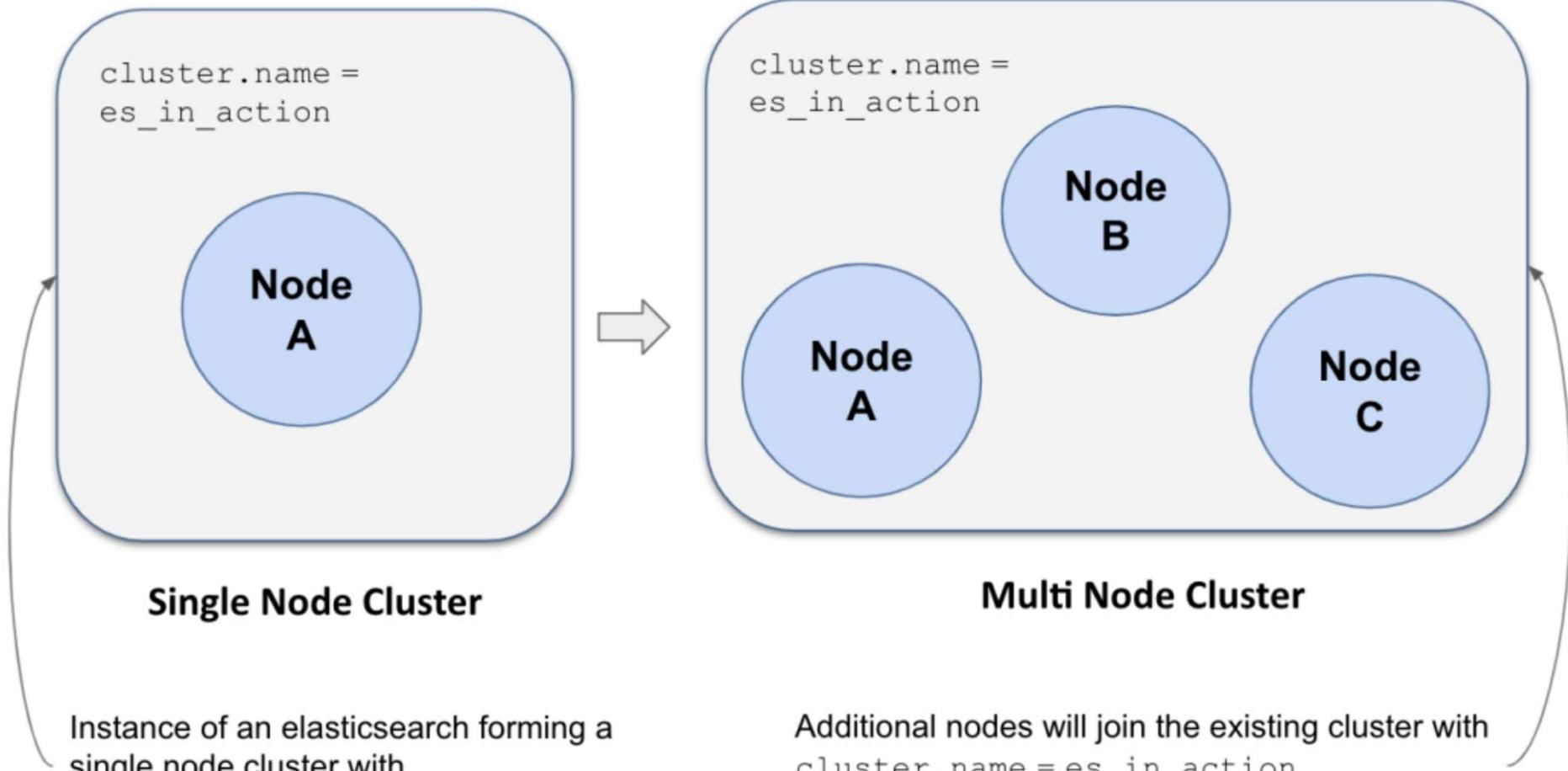
A node forming forming  
a single-node-cluster

Node is a running  
instance of Elasticsearch

The documents will be  
housed in shards



# Cluster formation from single node to multi-node cluster



# An inverted index data structure

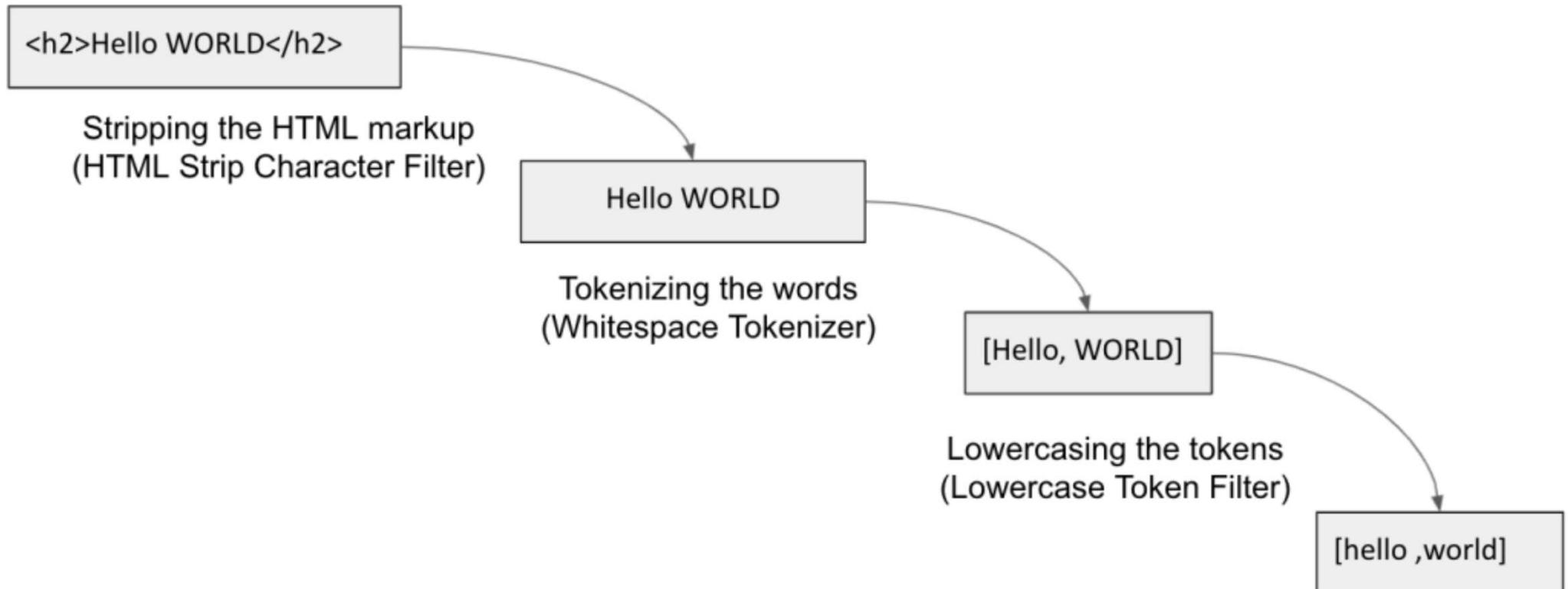
**Inverted Index**

Two full-text fields:  
"Hello, World!"  
"Hello, Mate"

Word	Doc Num
hello	1, 2
world	1
mate	2

- This inverted index is the key to faster retrieval of documents during the full-text search phase
- For each document that consists of full-text fields, the server creates the respective inverted indices.

# Text analysis procedure



The tokenized words for “Hello, World” and the documents they are found in

Word	Frequency	Document ID
hello	1	1
world	1	1

The tokenized words for “Hello, Mate” and the documents they are found in

Word	Frequency	Document ID
hello	2	1,2
world	1	1
mate	1	2

# Relevant results for Java in a title search

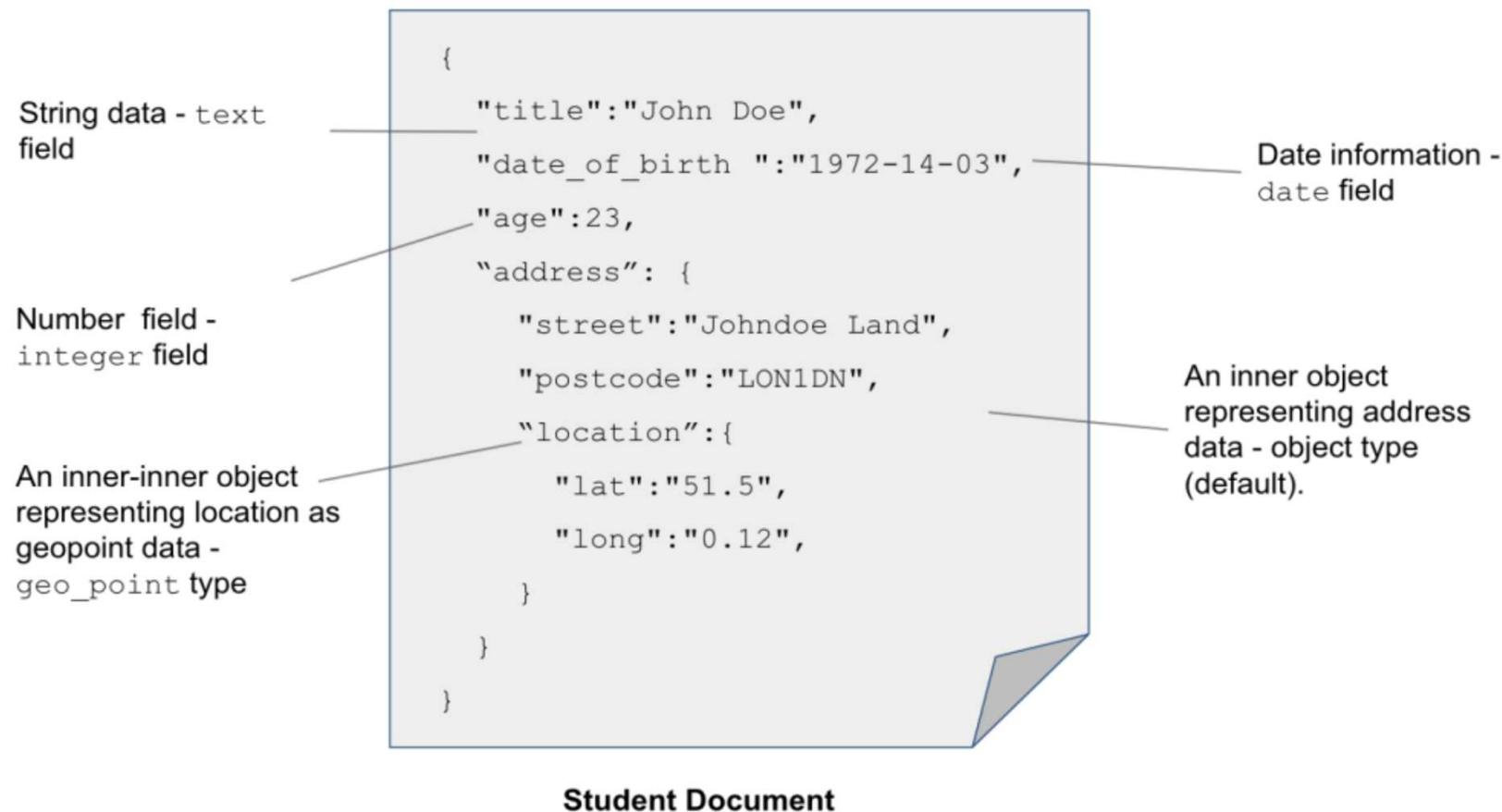
```
GET books/_search
{
  "_source": "title",
  "query": {
    "match": {
      "title": "Java"
    }
  }
}
```



```
"hits" : [
  ...
  "max_score" : 0.33537668,
  "hits" : [
    {
      "_score" : 0.33537668,
      "_source" : { "title" : "Effective Java" }
    },
    {
      "_score" : 0.30060259,
      "_source" : { "title" : "Head First Java" }
    },
    {
      "_score" : 0.18531466,
      "_source" : { "title" : "Test-Driven: TDD and Acceptance TDD for Java Developers" }
    }...
  ]
}
```

# Mapping

# An example of a student document with the data represented in JSON



# Indexing a document for the first time

- PUT movies/\_doc/1
- {
- "title": "Godfather", #A The title of the movie
- "rating": 4.9, #B The rating given to the movie
- "release\_year": "1972/08/01" #C Movie's release year(note the date format)
- }

# What happens when this document hits the engine?

- A new index (movies) is created automatically with default settings.
- A new schema is created for the movies index with the data types deduced from this document's fields
- The document gets indexed and stored in the Elasticsearch data store
- Subsequent documents get indexed without undergoing the previous steps as Elasticsearch consults the newly created schema for further indexing

# Movie index mapping derived from the document values

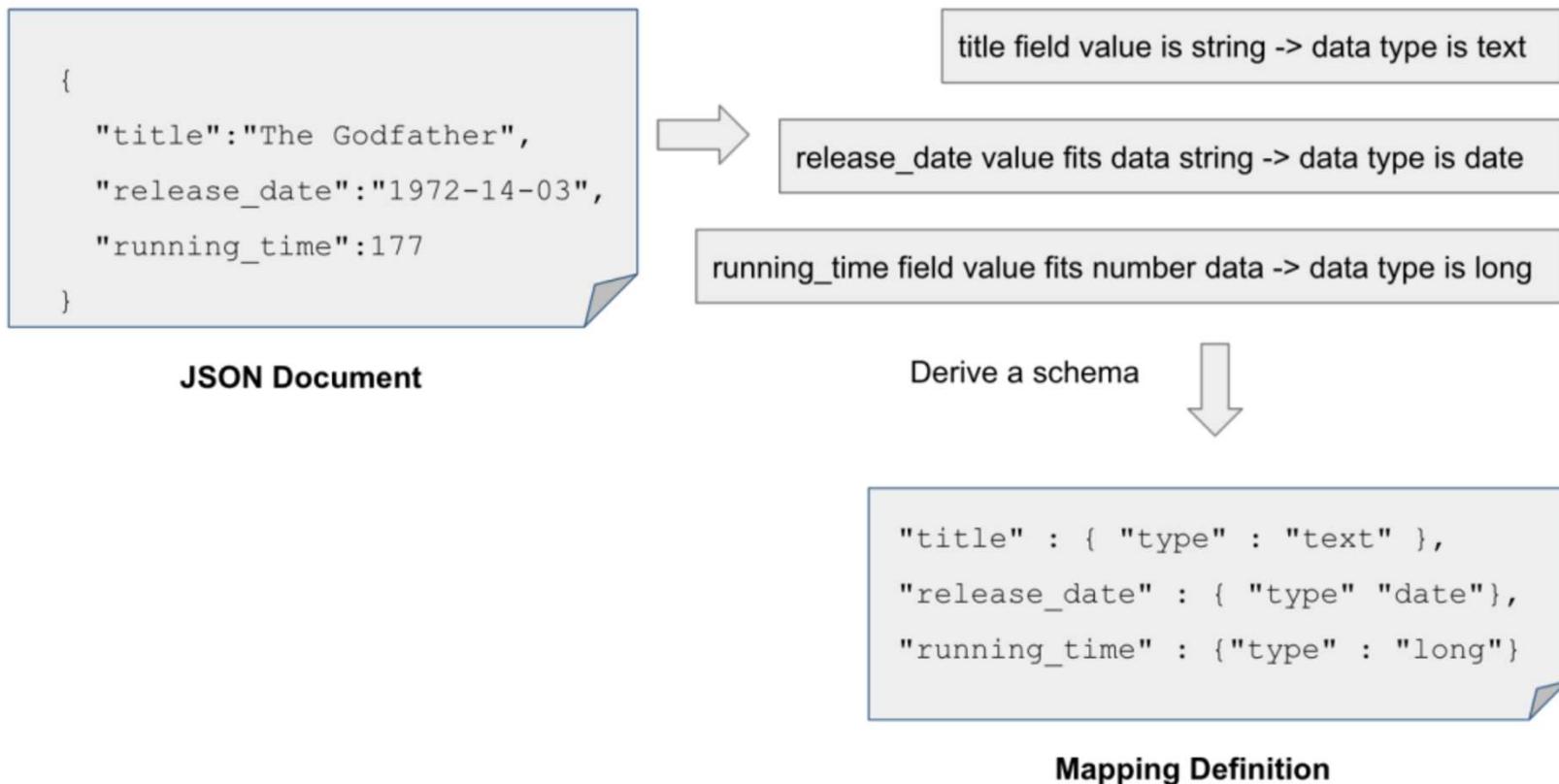
```
...  
  "properties" : {  
    "rating" : {"type" : "float"}, ← Due to the rating value of 4.9,  
    "release_year" : {  
      "type" : "date", ← a floating point number, the type  
      "format" : "yyyy/MM/dd HH:mm:ss || yyyy/MM/dd || epoch_millis"  
    }, ← is deduced as "float"  
    "title" : {  
      "type" : "text", ← As the release_year value  
      "fields" : { ← is in year's format (ISO  
        "keyword" : { ← format), the type is deduced  
          "type" : "keyword", as "date"  
          "ignore_above" : 256  
        }  
      }  
    }  
  }  
}
```

GET movies/\_mapping

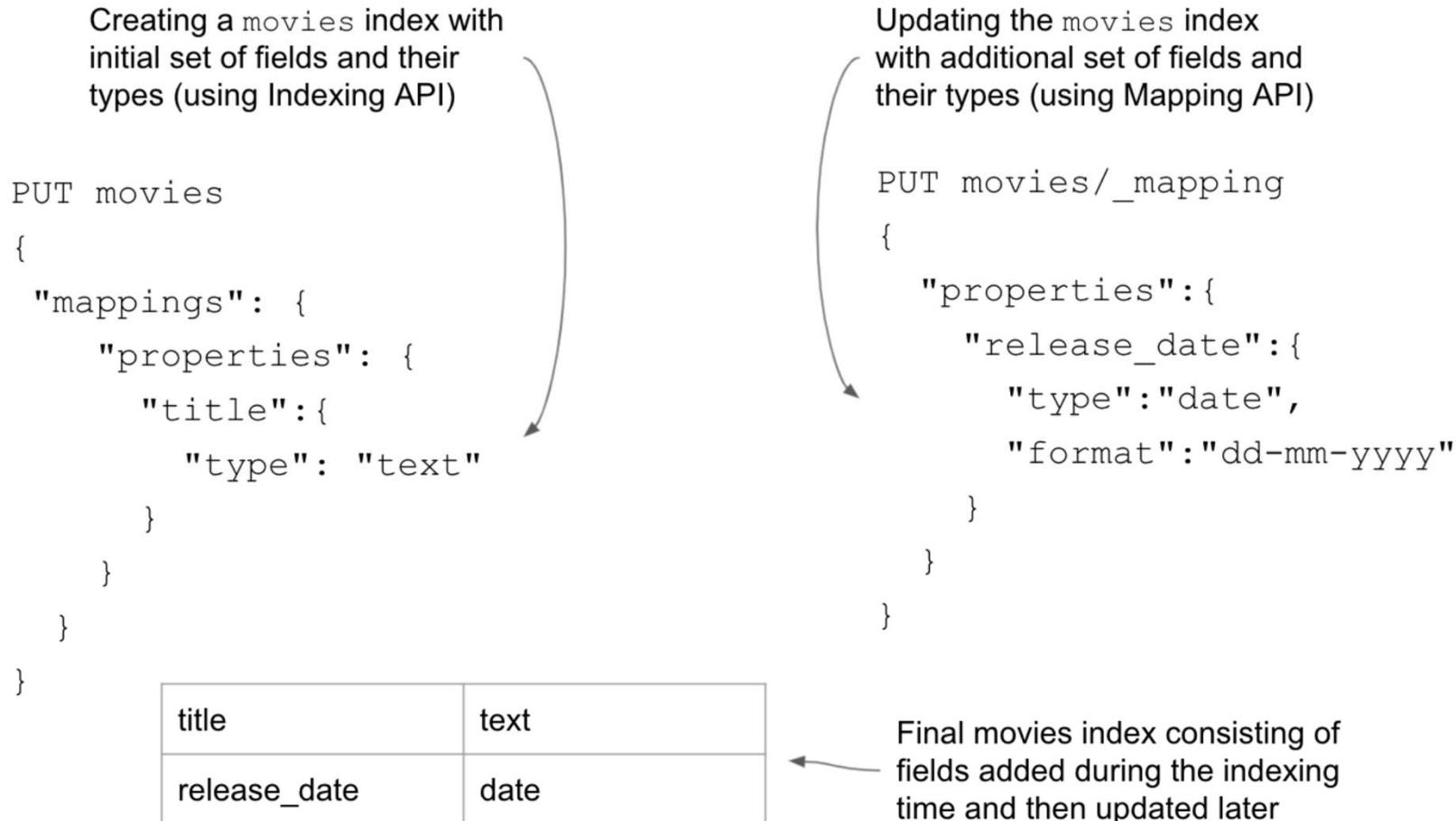
The title is textual information, so a "text" data type is assigned

The fields object indicates the second data type defined for the same title field (multi data types)

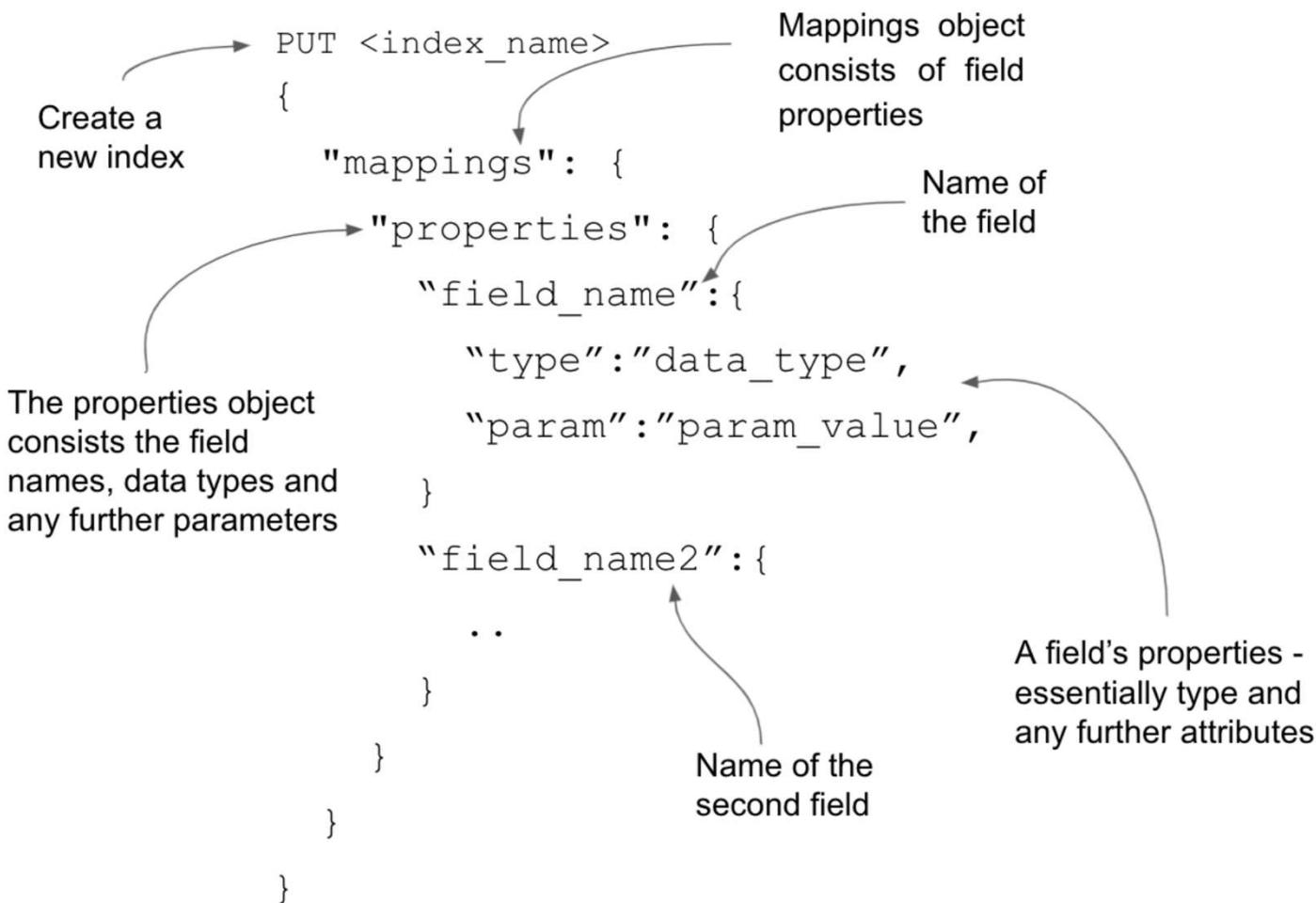
# Dynamically deriving the indexing schema



# Creating and updating schema using indexing and mapping APIs



# Creating a mapping definition during index creation



# Common data types with examples

Type	Description	Examples
text	Represents textual information (such as string values); unstructured text	Movie title, blog post, book review, log message
integer/long/short/byte	Represents a number	Number of infectious cases, flights canceled, products sold
float/double	Represents a floating-point number	Student's grade point, moving average of sales, reviewer ratings
boolean	Represents a binary choice, true or false	Is the movie a blockbuster? Are COVID cases on the rise? Has the student passed the exam?
keyword	Represents structured text, text that must not be broken down or analyzed	Error codes, email addresses, phone numbers, Social Security numbers
object	Represents a JSON object	(In JSON format) employee details, tweets, movie objects
nested	Represents an array of objects	An employee's address, email's routing data, a movie's technical crew

# Creating the mapping definition (left) and retrieving the schema (right)

```
PUT students
{
  "mappings": {
    "properties": {
      "name": {
        "type": "text"
      },
      "age": {
        "type": "byte"
      }
    }
  }
}
```

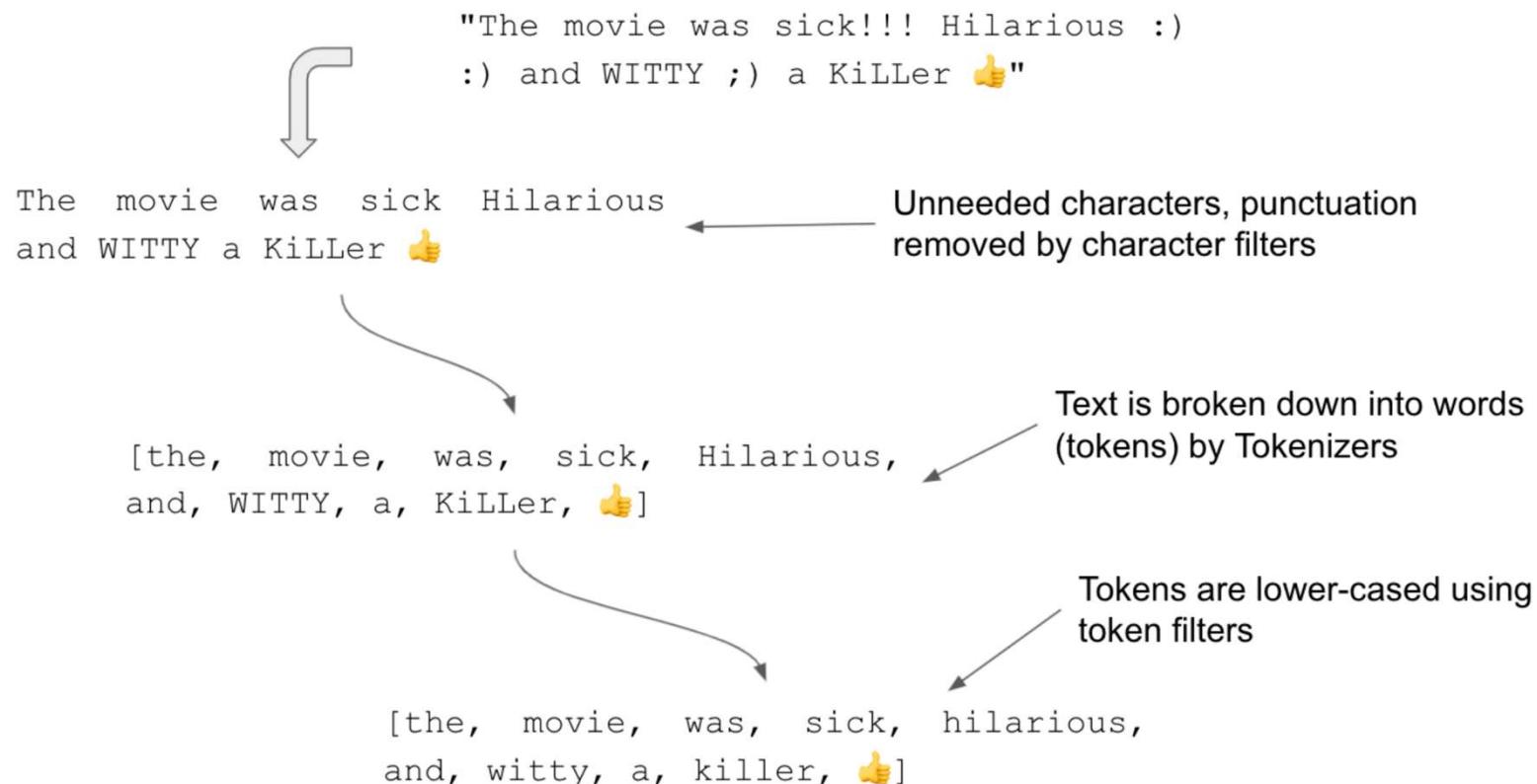
The `students` index is defined with a schema consisting of name (text) and age (byte)

```
GET students/_mapping
{
  "students" : {
    "mappings" : {
      "properties" : {
        "age" : {
          "type" : "byte"
        },
        "name" : {
          "type" : "text"
        }
      }
    }
  }
}
```

↓      ↗  
Fetching the mappings for students index

Students mapping definition

# Processing a full text field during indexing by standard analyzer module



# Numeric data types

<b>Integer types</b>	<code>byte</code>	<b>Signed 8 bit integer</b>
	<code>short</code>	<b>Signed 16 bit integer</b>
	<code>integer</code>	<b>Singed 32 bit integer</b>
	<code>long</code>	<b>Signed 64 bit integer</b>
	<code>unsigned_long</code>	<b>64 bit unsigned integer</b>
<b>Floating Point types</b>	<code>float</code>	<b>32 bit single precision floating-point number</b>
	<code>double</code>	<b>64 bit double precision floating-point number</b>
	<code>half_float</code>	<b>16 bit half precision floating-point number</b>
	<code>scaled_float</code>	<b>Floating-point number backed by long</b>

# Working with documents

# Indexing a document with an identifier

```
PUT movies/_doc/1
{
  "title": "The Godfather",
  "synopsis": "The aging patriarch
               of an organized crime ..."
}
```

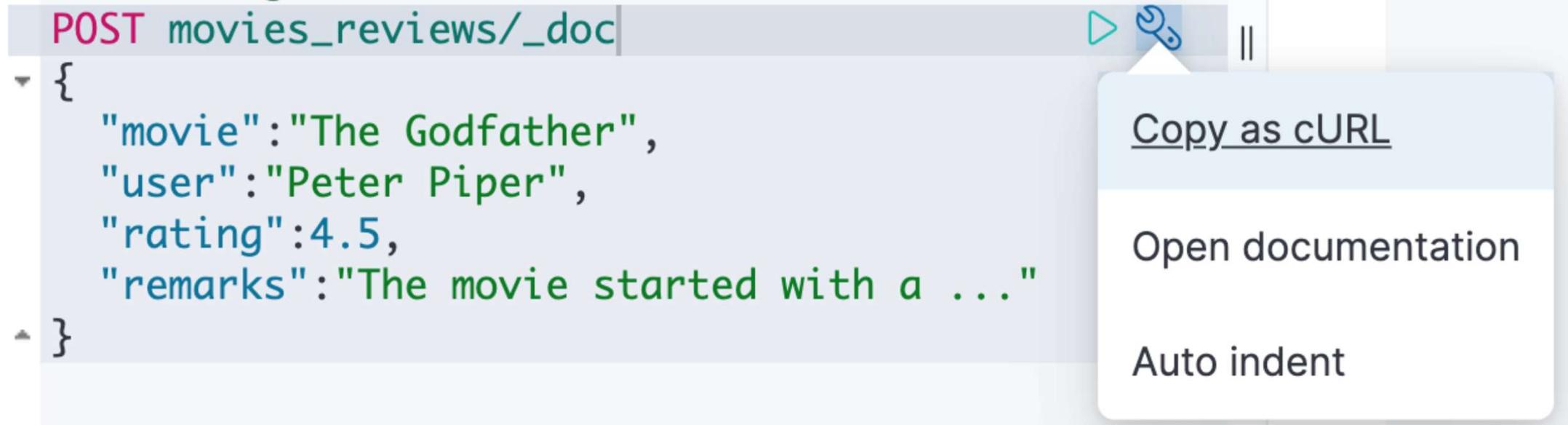
A request to the server to index a movie document with an id as 1. The body of the request is a JSON document

```
{
  "_index": "movies",
  "_type": "_doc",
  "_id": "1",
  "_version": 1,
  "result": "created",
  "_shards": {
    "total": 4,
    "successful": 1,
    "failed": 0
  },
  "_seq_no": 0,
  "_primary_term": 1
}
```

Response from the server indicating that the movie document was indexed successfully

# Exporting the query as cURL

```
POST movies_reviews/_doc
{
  "movie": "The Godfather",
  "user": "Peter Piper",
  "rating": 4.5,
  "remarks": "The movie started with a ..."
}
```



The screenshot shows a code editor or a browser developer tools interface. A context menu is open over a JSON object. The menu items are: "Copy as cURL" (highlighted in blue), "Open documentation", and "Auto indent". The JSON object itself is as follows:

```
POST movies_reviews/_doc
{
  "movie": "The Godfather",
  "user": "Peter Piper",
  "rating": 4.5,
  "remarks": "The movie started with a ..."
}
```

# Response from the server when a document is indexed successfully

```
{  
  "_index" : "movies",  
  "_type" : "_doc",  
  "_id" : "1",  
  "_version" : 1,  
  "result" : "created",  
  "_shards" : {  
    "total" : 4,  
    "successful" : 1,  
    "failed" : 0  
  },  
  "_seq_no" : 0,  
  "_primary_term" : 1  
}
```

The index name (`movies`) of the document

Type of the document - defaulted to `_doc` ( type of a document is deprecated and hence set to `_doc`)

Identifier of the document

Version of the document, 1 indicating it's the first version

The resource was created successfully, hence result = created

# The server creates and assigns a randomly auto generated ID to the document

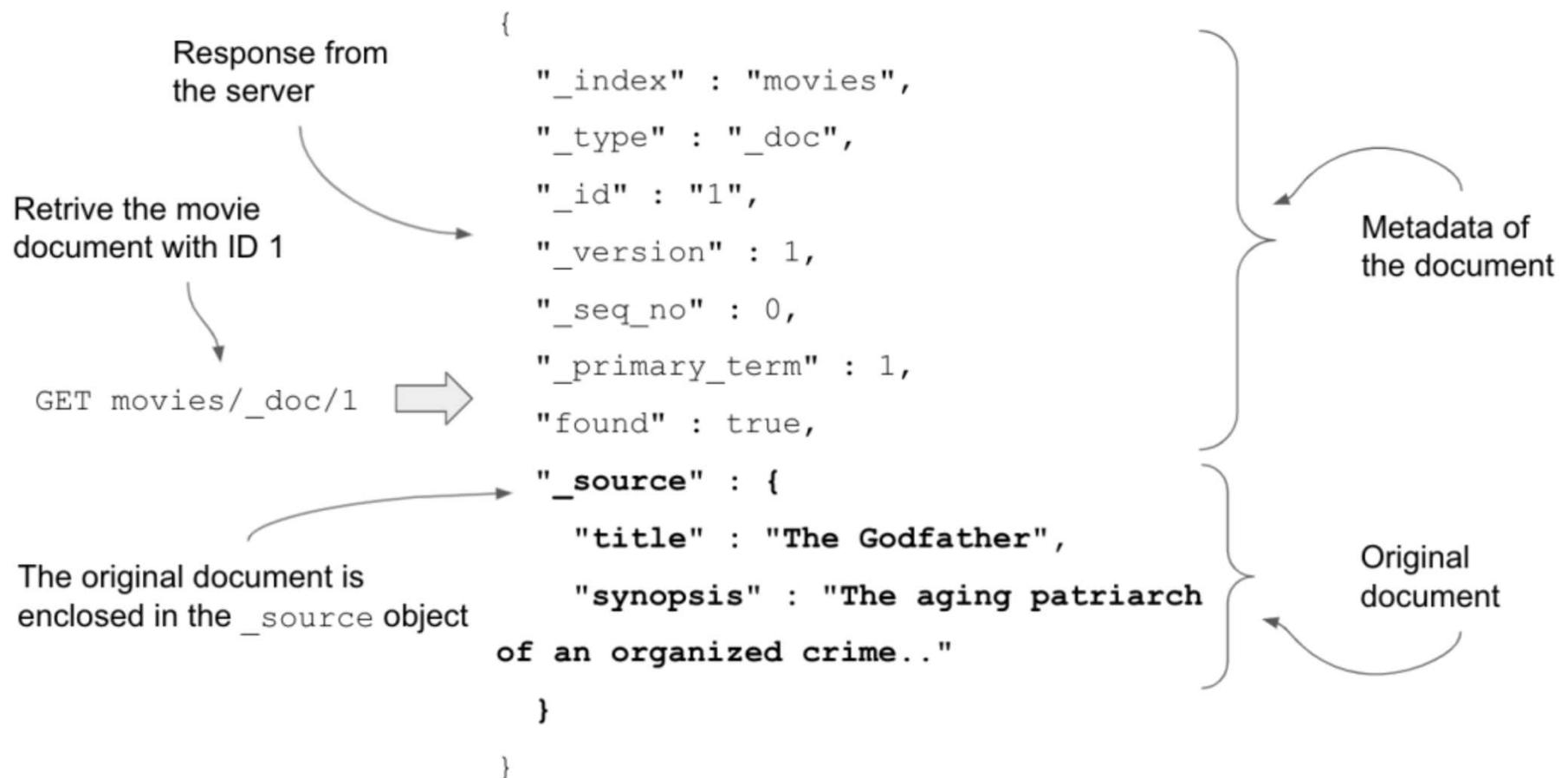
```
{  
    "_index": "movies_reviews",  
    "_type": "_doc",  
    "_id": "53NyfXoBW8A1B2amKR5j",  
    "_version": 1,  
    "result": "created",  
    "_shards": {  
        "total": 4,  
        "successful": 1,  
        "failed": 0  
    },  
    "_seq_no": 0,  
    "_primary_term": 1  
}
```

Response from the server

The ID is a auto-generated UUID generated by the server and assigned to movie review

The result indicates the document was indexed successfully

# Retrieving a document using the GET API call



# Fetching a non-existent document returns a Not Found message



# Fetching multiple documents using the \_mget API

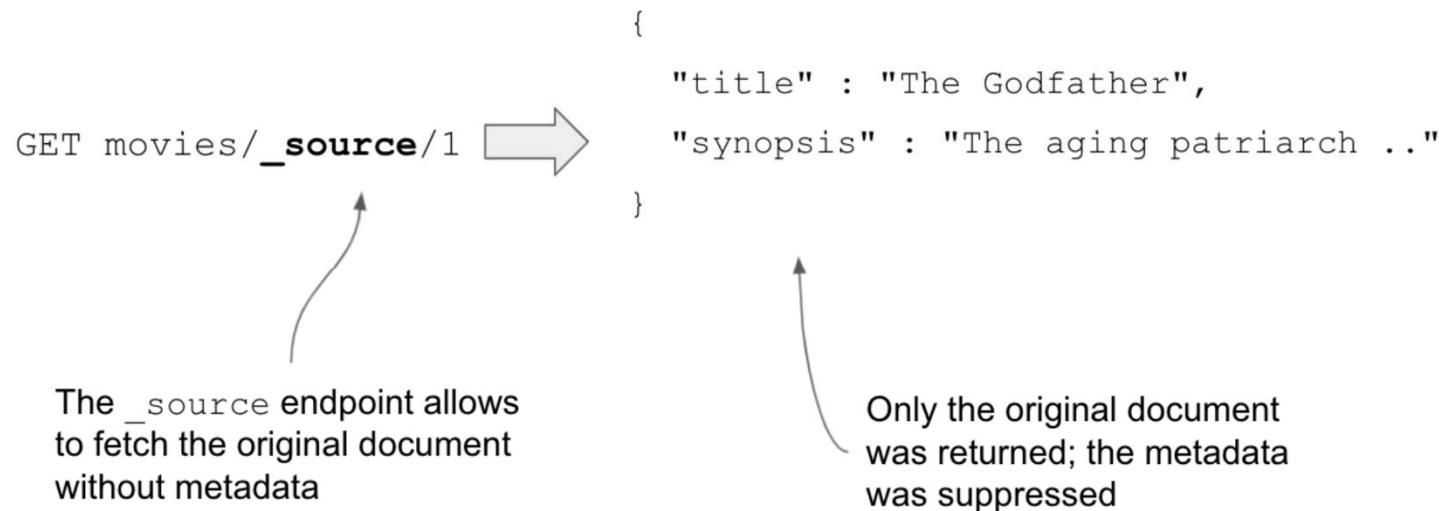
The \_mget endpoint takes in a request body with docs object.

```
GET _mget
{
  "docs": [
    {
      "_index": "index_1",
      "_id": 88
    },
    {
      "_index": "index_2",
      "_id": 99
    }
  ]
}
```

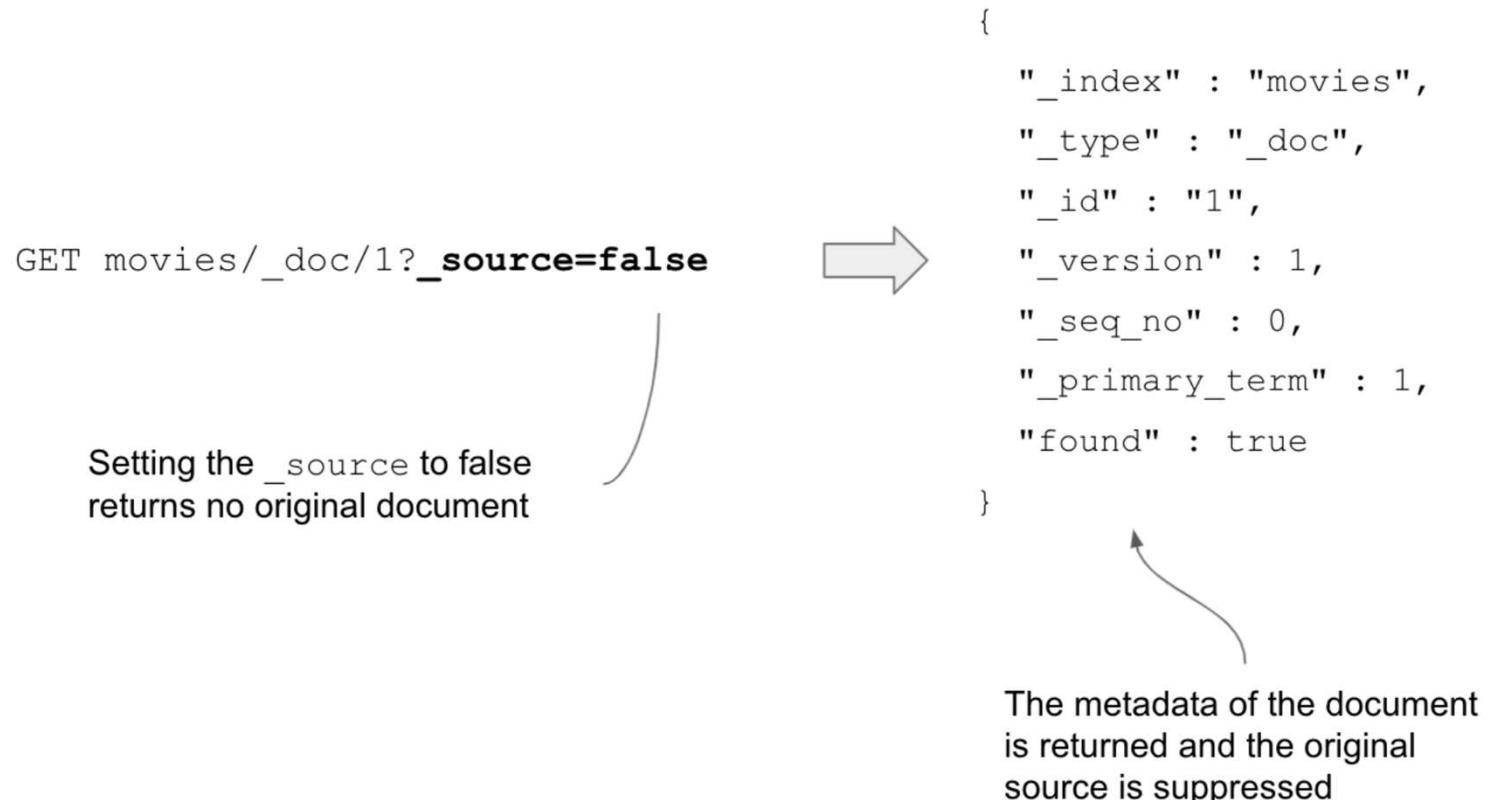
The docs object consists of an array of request objects

Each individual request object will consist of the index (\_index) and the ID (\_id) of the document

# Invoking the `_source` endpoint returns the original document with no metadata



# Setting the `_source` flag in the request returns metadata only



# Tweaking what attributes can and cannot be as part of the return result

```
GET movies/_source/3?_source_includes=rating*&_source_excludes=rating_amazon
```

The \_source\_includes parameter will allow all the ratings due to the wildcard (rating\*)

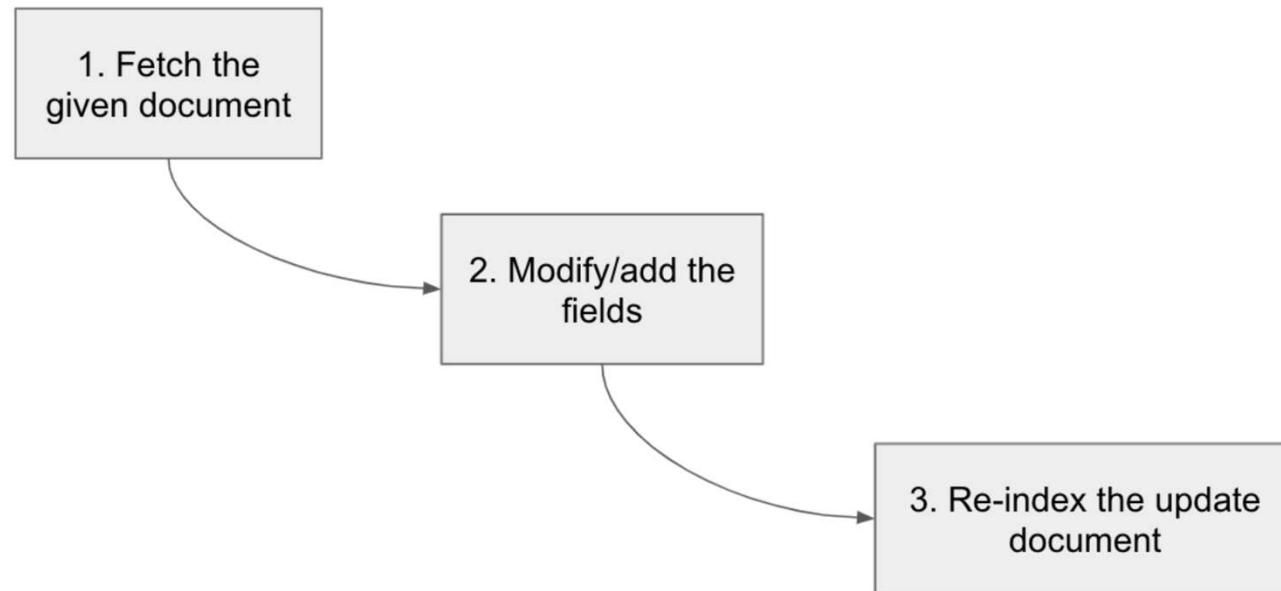
{

```
    "rating" : "9.3",
    "rating_rotten_tomatoes" : 80,
    "rating_metacritic" : 90
}
```

The \_source\_excludes parameter will disallow amazon's rating (rating\_amazon)

The output will be all ratings but no amazon's rating field

# Updating or modifying the document is a three-step process



# Indexing the new movie Mission Impossible using the \_bulk API

```
POST _bulk
{"index": {"_index": "movies", "_id": "100"}}
{"title": "Mission Impossible", "release_date": "1996-07-05"}
```

The movie will be indexed into movies index

The identity of the movie is 100

The title and release\_date fields of the movie are provided in the second line of the request

The index action(highlighted) lets the new document to be indexed by the engine

# Thanks