



# PART 2 ELASTIC MAPPING & INDEXING



Provided by DevOpShift & Authored by Yaniv Cohen  
DevopShift CEO and Head of Devops JB

**- RUNNING THE STACK -**

# RUNNING THE STACK


## **ONLINE SESSION:**

**Connect to your remote server provided by your lecturer**

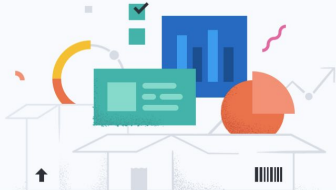
**`http://REMOTEIP:5601`**

# RUNNING THE STACK

34.217.82.1:5601/app/home#/



## Welcome to Elastic



### Start by adding integrations

Add data to your cluster from any source, then analyze and visualize it in real time. Use our solutions to add search anywhere, observe your ecosystem, and protect against security threats.

[Add integrations](#) [Explore on my own](#)

To learn about how usage data helps us manage and improve our products and services, see our [Privacy Statement](#). To stop collection, [disable usage data here](#).

**- MAPPING -**



# MAPPING

**Mapping is the process of defining how a document and its fields are stored and indexed in Elasticsearch.**

**To** create different types in an index, we need mapping types (or simply mapping) to be specified during index creation. Mapping is the process of defining how a document, and the fields it contains, are stored and indexed.

There are two types of mapping capabilities supported by Elasticsearch

- **Dynamic Mapping / Default Mapping:**
  - When we don't specify a mapping, Elasticsearch assigns a default based on the data's perceived type.
  - Example: If a field contains text, it's treated as a text type.
- **Explicit Mapping**
  - Define custom rules by explicitly setting the mapping.

**- DYNAMIC MAPPING -**



## Dynamic Mapping

One of the most important features of Elasticsearch is that it tries to get out of your way and let you start exploring your data as quickly as possible. To index a document, you don't have to first create an index, define a mapping type, and define your fields — you can just index a document and the index, type, and fields will display automatically

```
PUT data/_doc/1
{ "count": 5 }
```

This creates the **data** index, the **\_doc** mapping type, and a field called **count** with data type **long** (identified automatically)



The screenshot shows the Kibana interface for the `data` index. The left sidebar has a search bar with `GET data` and a search icon. The main panel displays the response for `GET data/_doc/1`, which is a JSON object:

```
1 {
2   "data": {
3     "aliases": {},
4     "mappings": {
5       "properties": {
6         "count": {
7           "type": "long"
8         }
9       }
10    }
```



## Dynamic Mapping

The roles govern the dynamic fields are:

<https://www.elastic.co/guide/en/elasticsearch/reference/current/dynamic-field-mapping.html>

## Dynamic Mapping using Templates

Elasticsearch can also support custom rules to configure the mapping for the dynamically added fields. We will learn about this as we progress with our session.

<https://www.elastic.co/guide/en/elasticsearch/reference/current/dynamic-templates.html>

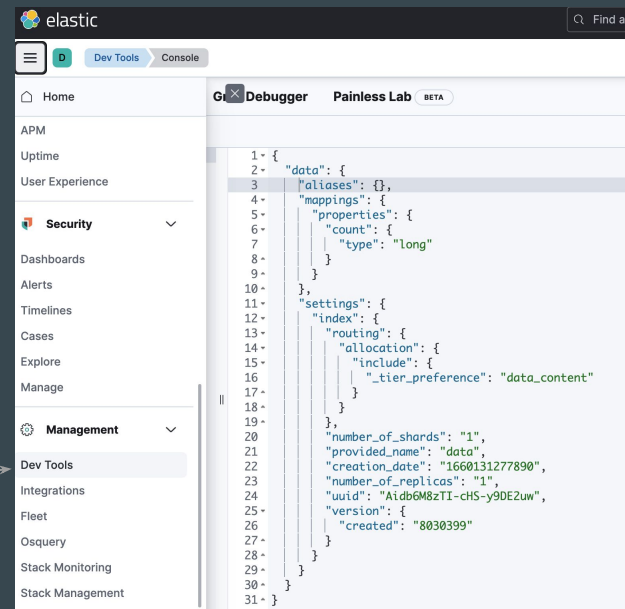
**- EXPLICIT MAPPING -**

## Explicit mapping

We know more about our data than Elasticsearch can guess, so while it's very and useful to use Dynamic mapping to get started with ES, at some point we will want to specify our own explicit mappings. For example when running our data in Production.

## USING DEVTOOLS

Make sure you know how to reach Kibana and open “Dev Tools”.  
There we will run all of our code examples.



# Explicit mapping

Let us start with the creation of our first index - **users**.

## Follow Through:

**PUT /users**

```
{  
  "mappings": {  
    "properties": {  
      "age": { "type": "integer" },  
      "email": { "type": "keyword" },  
      "name": { "type": "text" }  
    }  
  }  
}
```

**GET** \_cat/indices/users

- Get Specific index users information

**GET** users

- Get the configuration and schema of our index

Creates age, an integer field

Creates email, a keyword field

Creates name, a text field

We will deep dive into field types as we progress



## UPDATING MAPPING

**We** can't change the mapping or field type of an existing field.  
Changing an existing field could invalidate data that's already indexed !

**So how do we solve it ?**

**If we need to change the mapping** of a field in other indices, we will need to create a new index with the correct mapping and reindex your data into that index - not very recommended due to performance degradation and downtime

**Renaming** a field would invalidate data already indexed under the old field name.  
Instead, we add an alias field to create an alternate field name.

## ALIAS FIELD TYPE

An alias mapping defines an alternate name for a field in the index. The alias can be used in place of the target field in search requests, and selected other APIs like field capabilities.

**GET** /trips/\_mapping

- View the mapping of our index

**GET** /trips/\_mapping/field/route\_length\_miles

- View the mapping of specific fields

**PUT** trips

```
{
  "mappings": {
    "properties": {
      "distance": {
        "type": "long"
      },
      "route_length_miles": {
        "type": "alias",
        "path": "distance"
      },
      "transit_mode": {
        "type": "keyword"
      }
    }
  }
}
```

**- Built In Analyzers in depth -**





## Analysis intro:

Analysis is the process of converting text, like a paragraph or a page of it, into tokens or terms which are then added to the Elasticsearch index for searching.

### Key Components:

**Tokenizer:** Breaks text into terms (often words).

**Filters:** Modify, add, or remove terms. Examples include lowercase filter, stop word filter, etc.

## Usage in Elasticsearch:

- Full-Text Search: Improve search relevancy by breaking down text fields.
- Aggregations: Grouping by text fields.
- Highlighting: Highlighting keywords in search results.
- Autocompletion & Suggestions: To aid user queries by providing relevant suggestions.

## Why is it Important?

Ensures that the search is accurate and efficient by preparing and optimizing the data for the search engine.

**Remember**, analysis in Elasticsearch is crucial for handling and searching text effectively. Proper analysis ensures that users get relevant search results, even with potential textual inconsistencies in the raw data.

- Analyzer -



## **Anatomy of an Analyzer in Elasticsearch**

An analyzer in Elasticsearch is a combination of three primary building blocks that handle and process text.

## Key Components of an Analyzer

- **Tokenizer:**
  - Purpose: Splits the input text into a series of tokens.
  - **Example:** The sentence "ChatGPT is helpful" becomes ["ChatGPT", "is", "helpful"].
- **Token Filters:**
  - Purpose: Modify, add, or delete tokens.
  - Examples:
    - Lowercase filter: Converts all characters to lowercase.
    - Stop filter: Removes common words like "and", "the".
- **Character Filters:**
  - Purpose: Transform the input text before tokenization.
  - Examples:
    - HTML strip filter: Removes HTML elements.
    - Mapping filter: Replaces certain characters, e.g., "\$" to "USD".

**- Built in / Custom Analyzers -**

## Choices of Analyzers in Elasticsearch

**Built-in** Analyzers: Elasticsearch comes with several **pre-defined** analyzers which cater to a variety of needs.

- **Standard Analyzer:**
  - Uses: General-purpose analyzer.
  - Components: Unicode text segmentation, removes most punctuation, and lowercases.
- **Simple Analyzer:**
  - Uses: Divides text at non-letter characters.
  - Components: Lowercases all terms.
- **Whitespace Analyzer:**
  - Uses: Splits text on whitespace.
  - Components: No other transformations.
- **Stop Analyzer:**
  - Uses: Like the simple analyzer but also removes stop words.
  - Components: Lowercasing, stop words removal.

**Stop Analyzer:**

- Uses: Like the simple analyzer but also removes stop words.
- Components: Lowercasing, stop words removal.

**Keyword Analyzer:**

- Uses: Doesn't break the input into tokens.
- Components: Entire input as a single token.

**Pattern Analyzer:**

- Uses: Uses a regular expression to split the text into terms.
- Components: Lowercasing.



## Custom Analyzers

While the built-in analyzers cater to most general needs, Elasticsearch allows the creation of custom analyzers. This is useful to combine different tokenizers and filters to address specific textual challenges or domain-specific requirements.

### **Recommendation:**

Always evaluate the nature of your data and search requirements before selecting or designing an analyzer. Testing its effectiveness on sample data ensures optimal search performance.

**- Built in / Custom Analyzers -**

## Character filters

**Objective:** It takes the original text and can change, add, or remove characters.

### Examples:

- Converts numerals like ०१२३४५६७८९ to 0123456789.
- Removes HTML tags like <b>.

### Usage in Analyzers:

- An analyzer can use multiple character filters.
- They work in the sequence they are added.

In simple terms, character filters help in preprocessing the text before it's broken down for indexing.

Reference <https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-charfilters.html>

# Tokenizer

**Objective:** It breaks down the text into smaller pieces, often words or terms, making it ready for indexing

## Examples:

- The sentence "Elasticsearch is great!" becomes ["Elasticsearch", "is", "great"].

## Usage in Analyzers:

- Every analyzer must have one and only one tokenizer.
- It's the first step after any character filters have been applied.

Simply put, tokenizers dissect text into manageable chunks, enabling efficient searching in Elasticsearch.

Reference <https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-tokenizers.html>

## Token filters

**Objective:** After tokenization, these filters further refine or modify the tokens. They can change, add, or remove tokens.

### Examples:

- Lowercase filter: Converts "Elasticsearch" to "elasticsearch".
- Stop filter: Removes common words like "and", "the".

### Usage in Analyzers:

- An analyzer can use multiple token filters.
- They work sequentially after the tokenizer has processed the text.

In essence, token filters polish the tokens to optimize them for the index and search process.

Reference <https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-tokenfilters.html>

**- Testing Analyzers in Kibana -**



## Why should we test?

Before finalizing an analyzer, it's crucial to test its output to ensure it processes text as intended.

How do we use it in Kibana DEV:

**POST** /\_analyze

```
{  
  "analyzer": "custom_analyzer_name",  
  "text": "Your test text here"  
}
```

## Example:

Testing the english Analyzer - The english analyzer is tailored for English text, applying stemming, removing stop words, and lowercasing among other transformations.

How do we use it in Kibana DEV:

```
POST /_analyze
{
  "analyzer": "english",
  "text": "Running is beneficial for health"
}
```

## Expected Output:

["run", "is", "benefici", "for", "health"] - **Notice** how the word "Running" is stemmed to "run" and "beneficial" is stemmed to "benefici". This shows the power of the english analyzer in handling English text for more effective search results.





# Building a Custom Analyzer:

Using Kibana Dev

**In this example**, we built a custom analyzer named `my_custom_analyzer`. It uses the standard tokenizer, then applies two filters: the lowercase filter and a custom stop filter named `english_stop` which removes common English stop words.

```
PUT /custom_index
{
  "settings": {
    "analysis": {
      "analyzer": {
        "my_custom_analyzer": {
          "type": "custom",
          "tokenizer": "standard",
          "filter": ["lowercase", "english_stop"]
        }
      },
      "filter": {
        "english_stop": {
          "type": "stop",
          "stopwords": "_english_"
        }
      }
    }
  }
}
```



## Testing Custom Analyzer:

How do we use it in Kibana DEV:

```
POST /custom_index/_analyze
{
  "analyzer": "my_custom_analyzer",
  "text": "Elasticsearch is Amazing"
}
```

### Expected Output:

["elasticsearch", "amazing"]

**- Normalizers -**



## Normalizers in Elasticsearch

Normalizers are similar to analyzers, but they are used for keyword fields where full-text searching is not required. They transform text into a canonical form, ensuring consistent querying.

## Why Use Normalizers?

- **Consistency:** Ensure that variations in text, such as different cases or diacritic marks, don't affect search results.
- **Optimized for Keyword Fields:** Useful for fields where exact matching is more important than full-text search.

# **- CRUD Operations in Elasticsearch -**



## **Introduction:**

**CRUD stands for Create, Read, Update, and Delete. These operations are fundamental to manage data in Elasticsearch.**

**- Create -**





## Command: CREATE

```
PUT /index-name/_doc/document-id
{
  "field1": "value1",
  "field2": "value2"
}
```

### Pros:

- Intuitive and easy to use.
- Can specify a custom document ID.

### Cons:

- Overwrites if the document ID already exists.

- Read -



## Command: READ

```
GET /index-name/_doc/document-id
```

### Pros:

- Fast data retrieval.
- Supports complex search queries.

### Cons:

- Needs proper indexing for optimized performance.

- Update -



## Command: UPDATE

```
POST /index-name/_update/document-id
{
  "doc": {
    "field1": "new-value"
  }
}
```

### Pros:

- Partial document update, reducing index operation overhead.
- Supports scripted updates (A script that can update, delete, or skip modifying the document)

### Cons:

- Potential version conflicts if the document has changed between reading and updating (TBD)

**- Delete -**



## Command: DELETE

```
DELETE /index-name/_doc/document-id
```

### Pros:

- Simple to use.
- Frees up storage space.

### Cons:

- Permanent data removal – no retrieval possible.
- Might need a cluster reindexing for optimal space usage.

**- Update & Delete vs.  
Create-Only Approach in  
Elasticsearch -**





## Introduction:

In certain use-cases, instead of updating or deleting records, systems might choose to only create new records and rely on the most recent one. Let's examine the pros and cons of both strategies.



# Update & Delete Operations Approach

## Pros:

- **Data Freshness:** Ensure the most current and relevant data is available for search.
- **Storage Efficiency:** By updating or removing obsolete data, you can optimize storage.
- **Simplicity in Query:** Retrieve data without needing logic to fetch the most recent version.

## Cons:

- **Overhead:** Every update or delete operation requires reindexing, which can be resource-intensive.
- **Version Conflicts:** Possibility of conflicts if multiple processes try to update the same record simultaneously.
- **Data Loss Risk:** Accidental deletion can result in permanent data loss.

## Create-Only Approach

### Pros:

- **High Throughput:** Writing is often faster than updating, especially in distributed systems like Elasticsearch.
- **Immutable Records:** A historical record of all changes is maintained, which can be beneficial for auditing or time-based analysis.
- **Avoids Version Conflicts:** Since you're only adding new data, you sidestep potential versioning issues.

### Cons:

- **Increased Storage:** Over time, multiple versions of a record can consume significant storage.
- **Complex Queries:** Fetching the latest record can become complex, requiring sorting or filtering.
- **Potential Staleness:** Old, outdated records might appear in search results if not handled correctly.

- Q&A -

## What happen if we don't know the specific document ID for an update or delta operation ?

**In Elasticsearch we typically employ a two-step process (As we are doing in a RDBMS)**

- Search for the Document: Use a query to identify the document(s) you wish to update or delete based on some criteria.
- Perform the Update/Delete: Once you have identified the document ID(s), you can then proceed with the update or delete operation.

**However, Elasticsearch also provides the Update By Query and Delete By Query API to streamline this process.**

**- Streamlining Operations -**



## Introduction:

Elasticsearch's Update By Query and Delete By Query APIs allow for bulk updates and deletions based on query criteria, eliminating the need to know specific document IDs.

- Update By Query API -



## Purpose: Modify multiple documents that match specific criteria.

### Advantages:

Bulk update without ID lookup.

Flexibility with complex query criteria.

Can be combined with painless scripting for dynamic updates.

```
POST /index-name/_update_by_query
{
  "query": {
    "term": {
      "field-name": "value"
    }
  },
  "script": {
    "source": "ctx._source.field-name =
'new-value'"
  }
}
```

**- Delete By Query API -**

## Purpose: Remove multiple documents based on certain query criteria.

### Advantages:

Efficient bulk deletions.

Useful for data pruning or removing outdated data.

Can target specific subsets of data with precision.

```
POST /index-name/_delete_by_query
{
  "query": {
    "term": {
      "field-name": "value"
    }
  }
}
```

**- Considerations & Best  
Practices -**

## Considerations & Best Practices

- **Concurrency:** Be cautious about simultaneous operations, which might lead to version conflicts.
- **Task Management:** Both APIs create tasks that run in the background. Monitor the tasks and ensure they complete.
- **Rate Limiting:** Use the `requests_per_second` parameter to control the speed of operations and reduce cluster load.
- **Safety:** Especially for delete operations, consider a "dry run" first, using the search API to preview matched documents.

**In summary,** Update By Query and Delete By Query APIs in Elasticsearch provide powerful means to handle bulk data operations. However, they should be used judiciously and with a clear understanding of their implications.

- Update document and dealing with concurrency -



## Introduction:

In a distributed system like Elasticsearch, multiple clients can try to update the same document simultaneously, leading to potential conflicts and data inconsistency. Properly handling concurrency is crucial to maintain data integrity.

**- The Problem -**



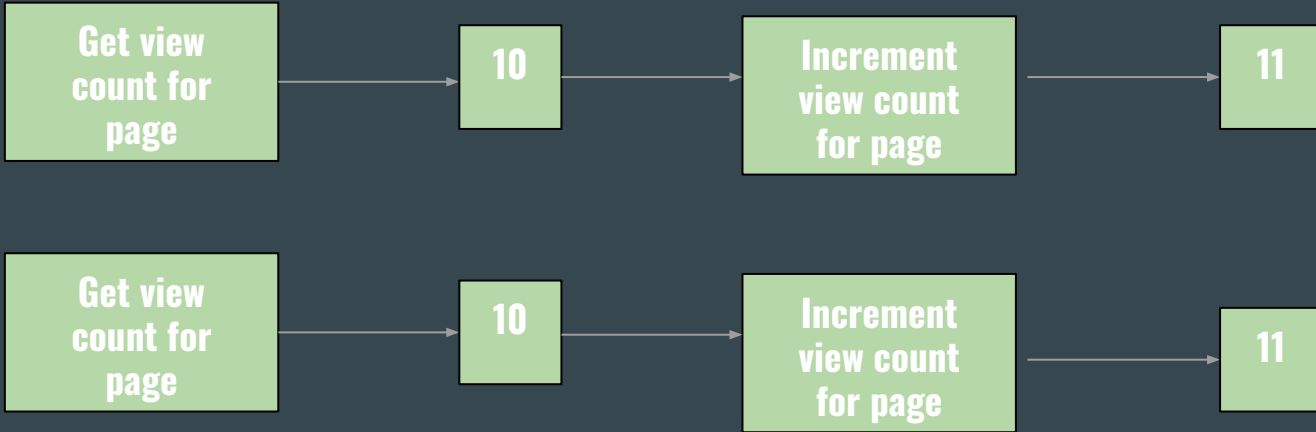


## The problem

Every document has a **\_version** field and we know that Elasticsearch documents are immutable. When we update an existing document a new document is created with incremented **\_version** while the old document is marked for deletion.

**Scenario:** Two clients read the same version of a document and make changes. Whichever client saves last will overwrite changes made by the first client.

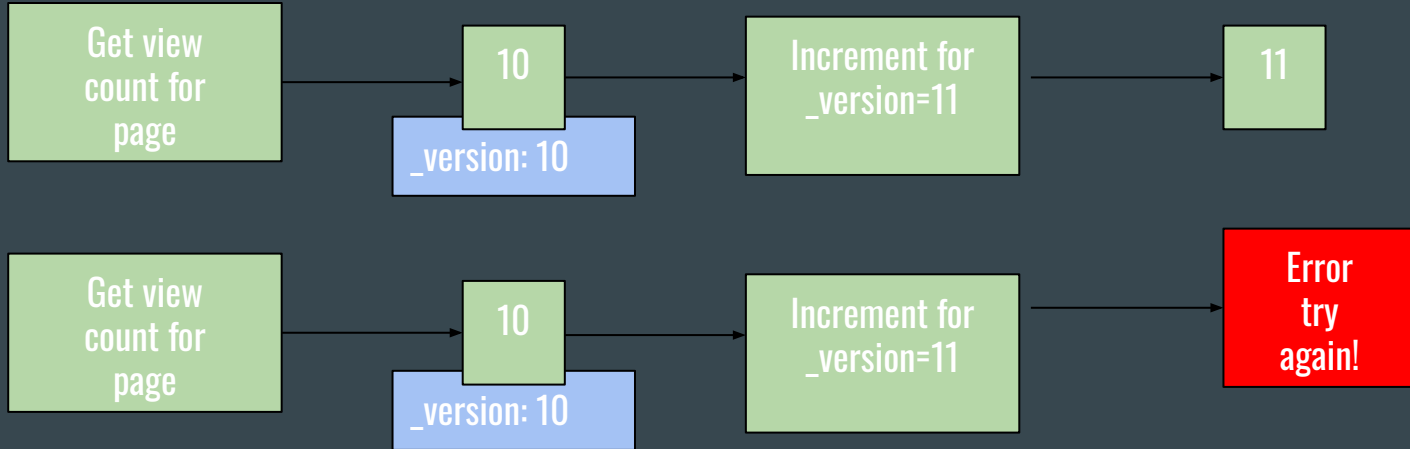
**Outcome:** Data loss for the first update, leading to inconsistent data states.



**But it should be 12!**

**- The Solution -**

- **\_version Field:** Elasticsearch assigns a version number to each document. When updating, ES checks the version to ensure no newer version exists.
- 
- **Optimistic Concurrency Control:** You can use the `if_seq_no` and `if_primary_term` parameters to ensure you're updating the correct version.
- 
- **Result:** If there's a version conflict, Elasticsearch will throw an exception, preventing accidental overwrites.



Use `retry_on_conflicts=N` to automatically retry.

**- PYTHON CODE EXAMPLE -**



**<https://github.com/yanivomc/devopshift-welcome/blob/master/welcome/elk/optimistic-concurrency/example.py>**

**- END PART II -**