Notes on "Overview of the Elastic Stack":

* The Elastic Stack is a collection of technologies developed and maintained by the company behind Elasticsearch. It consists of several components that work together for various purposes.
* The central component of the Elastic Stack is Elasticsearch, which serves as the heart of the stack. It is a distributed search and analytics engine that stores and indexes data, allowing for fast and efficient searching and retrieval.
* Kibana is another key component of the Elastic Stack. It is an analytics and visualization platform that provides a web interface for Elasticsearch. Kibana allows users to easily visualize data from Elasticsearch, create visualizations like pie charts and line charts, and build dashboards to monitor performance and track key metrics.
* Kibana also provides an interface for configuring change detection, forecasting, and managing certain aspects of Elasticsearch, such as authentication and authorization. It acts as a web interface to the data stored within Elasticsearch and utilizes the same REST API for querying and displaying results.
* Logstash is a versatile data processing pipeline that plays a crucial role in the Elastic Stack. It is responsible for ingesting, processing, and shipping data from various sources to Elasticsearch or other destinations. Logstash consists of three main stages: inputs, filters, and outputs. Inputs receive data, filters process and transform the data, and outputs send the processed data to specified destinations.
* X-Pack is a comprehensive set of features that enhances the functionality of Elasticsearch and Kibana. It includes capabilities such as security, monitoring, alerting, reporting, machine learning, graph exploration, and SQL querying. X-Pack enables authentication and authorization for both Elasticsearch and Kibana, allowing for secure access control. It also provides monitoring capabilities to track the performance and health of the Elastic Stack, and it allows users to set up alerts for specific events or anomalies. X-Pack enables machine learning functionalities for anomaly detection and forecasting, as well as graph exploration to uncover relationships within the data, while Kibana provides the interface for it. Additionally, it offers SQL querying capabilities for Elasticsearch, allowing users familiar with SQL to leverage their existing skills.
* Beats are lightweight data shippers that simplify the process of collecting and sending data to Elasticsearch or Logstash. There are various types of Beats, such as Filebeat, which collects log files, and Metricbeat, which collects system and service metrics. Beats can be installed on servers or other data sources, and they efficiently transmit data to the desired destination in the Elastic Stack.
* The Elastic Stack, formerly known as the ELK stack, refers to Elasticsearch, Logstash, and Kibana, which were commonly used together. With the introduction of X-Pack and Beats, the term "Elastic Stack" became more encompassing, representing the full suite of technologies provided by the company.

Walkthrough of common architectures:

* The text discusses the integration and usage of Elasticsearch in different architectural scenarios.
* In a simple architecture, Elasticsearch is added to an existing ecommerce application to improve search functionality. Search queries are sent from the web application to Elasticsearch, which returns the results to the browser.
* Data is imported into Elasticsearch by duplicating relevant data from the existing database. A script can be written to import the data, and subsequent updates are made in both the database and Elasticsearch to keep the data synchronized.
* In a more advanced architecture, Kibana is used to create a dashboard for monitoring key metrics such as orders and revenue. Kibana interacts with Elasticsearch and requires no additional data storage. Metricbeat is added to monitor system-level metrics on the web server, and the collected data is sent directly to Elasticsearch.
* Filebeat is introduced to collect and process access and error logs from the web server. Filebeat simplifies log processing and integrates seamlessly with Elasticsearch and Kibana. The logs are stored in Elasticsearch for analysis.
* As the business grows, additional web servers are added, and more data is stored in Elasticsearch, including events. To centralize event processing, Logstash is added to the architecture. Events are sent from web servers to Logstash, which processes and sends them to Elasticsearch. Metricbeat and Filebeat data can also be sent to Logstash for custom processing if needed.
* It is recommended to separate data modification and querying, with the web application primarily querying Elasticsearch rather than directly modifying data.
* The presented architecture demonstrates a typical usage of Elasticsearch and the Elastic Stack, although variations and other use cases exist.
* This summary provides an overview of the different architectural scenarios and the integration of Elasticsearch in each case.

Understanding the basic architecture:

* Elasticsearch operates on a cluster-based architecture where each node represents an instance of Elasticsearch that stores data.
* Nodes can be run on the same machine for development purposes, but in a production environment, it is recommended to run each node on a dedicated machine, virtual machine, or container.
* Nodes belong to clusters, which are collections of related nodes that contain all the data. Multiple clusters can be used for different purposes, but one cluster is typically sufficient.
* When a node starts up, it either joins an existing cluster or forms its own cluster if no other nodes are present. A cluster is automatically created when a node starts.
* Documents are the units of data stored in Elasticsearch and are represented as JSON objects. Each document is stored in an index, which groups similar documents together and provides configuration options for scalability and availability.
* Indices are collections of logically related documents. There is no hard limit on the number of documents an index can contain.
* Search queries are performed against indices to retrieve relevant documents.

Inspecting the cluster:

* The Console tool in Kibana allows interaction with the Elasticsearch cluster via the REST API.
* HTTP verbs (GET, POST, PUT, DELETE) are used to communicate with the Elasticsearch cluster.
* The cluster's health can be retrieved using the GET verb and the \_cluster/health request path.
* The results of the cluster health query display information such as the cluster's name and status.
* The CAT API provides a human-readable format for displaying cluster information.
* The nodes command within the CAT API lists the nodes in the cluster.
* Adding ?v at the end of a command provides a descriptive header row
* Cloud deployments typically have multiple nodes for high availability.
* Nodes in Elasticsearch have roles associated with them, and their responsibilities within the cluster will be discussed in future lectures.
* Query parameters can be used to retrieve more detailed information about nodes.
* The Indices API can be used to check the existing indices within the cluster.
* System indices, used by Elasticsearch and Kibana for storing various data, are initially present.
* System indices are hidden by default, but they can be viewed by adding the "expand\_wildcards" query parameter.

Sending queries with cURL:

* The text explains the process of running queries using cURL as an alternative to Kibana's Console tool.
* It highlights the advantages of using the Console tool, such as the convenience of auto-completion and the formatting of responses.
* The steps for running cURL commands are provided, starting with a basic command that specifies the Elasticsearch endpoint.
* The need to use the TLS endpoint instead of plaintext for Elasticsearch version 8 and onwards is mentioned, and the process of handling the certificate error is explained.
* The text presents two approaches to handle the certificate error: using the --insecure flag to ignore the error or providing the CA certificate using the --cacert argument.
* Authentication with the Elasticsearch cluster is covered, with instructions on using the -u argument and entering the password when prompted.
* An alternative approach is mentioned, where the password can be supplied in the -u argument itself, but it is cautioned that this exposes the password in the terminal.
* An example of sending a basic cURL request is provided, which retrieves basic information about the cluster.
* The addition of data to the request, specifically for the Search API on a non-existent products index, is demonstrated using the -d argument.
* The need to specify the Content-Type header as application/json is highlighted using the -H argument.
* Errors that may occur, such as unsupported Content-Type header or non-existent indices, are mentioned and explained.
* The sensitivity of cURL to the order of arguments is noted, and the need for careful attention in argument placement is emphasized.
* Other HTTP clients, such as Postman, are suggested as alternatives to cURL for sending requests to Elasticsearch.

Rules about communicating with elastic search:

* TLS (Transport Layer Security) is required for secure communication with Elasticsearch.
* Always use https:// instead of http:// when specifying the Elasticsearch endpoint.
* For local deployments, there are two options to handle the certificate error:
  + Use the --insecure flag with cURL to ignore the certificate error.
  + Provide the CA certificate using the --cacert argument to establish trust.
* All requests to Elasticsearch must be authenticated.
* Use the elastic user and the auto-generated password for authentication.
* When sending a request with a body, it is necessary to specify the Content-Type header as application/json to indicate the data format.
* Detailed commands and examples can be found within the GitHub repository associated with the course.

Command sent:

curl --cacert Elastic\ search/elasticsearch-8.8.2/config/certs/http\_ca.crt -u elastic:BZqgTjTAFbXuqsD-nn-2 -X GET -H "Content-Type:application/json" https://localhost:9200/products/\_search -d '{"query": {"match\_all": {}}}'

Sharding and scalability:

* A cluster in Elasticsearch consists of one or more nodes, allowing for scalability in terms of data storage and disk space.
* Sharding is a mechanism used by Elasticsearch to divide an index into separate pieces called shards.
* Sharding occurs at the index level, allowing different indices to have different numbers of shards based on their size and requirements.
* Sharding an index enables horizontal scaling of data volume, allowing it to be distributed across multiple nodes.
* Shards can be placed on any node within the cluster, and a shard is considered a fully functional index on its own.
* Each shard is essentially a Lucene index, as Elasticsearch is built on top of Apache Lucene.
* Sharding is useful for storing billions of documents within a single index and improving query performance by distributing and parallelizing queries across shards.
* Elasticsearch limits the number of documents a shard can store to just over two billion.
* Sharding also facilitates the utilization of multiple nodes' hardware resources.
* Indices are created with a default of one shard since Elasticsearch version 7, to avoid over-sharding small indices unnecessarily.
* The number of shards can be configured when creating an index, and it can be increased or decreased using the Split API and Shrink API, respectively.
* There is no definitive answer to how many shards to choose, as it depends on various factors like cluster size, node capacity, index size, and query patterns.
* As a general rule, starting with one shard is sufficient for small to medium-sized indices, but for larger indices with millions of anticipated documents, choosing five shards is a reasonable starting point.
* It's recommended to anticipate the future size of an index and consider adding shards at index creation to avoid potential bottlenecks.
* Ultimately, the default settings and gradual adjustments based on real-world usage and performance evaluation can guide shard configuration decisions.

Understanding replication:

* Replication is an important mechanism in Elasticsearch that provides fault tolerance and failover in case of node failures.
* By default, Elasticsearch enables replication for indices, ensuring data redundancy and preventing data loss.
* Replication works by creating copies of each shard in an index, known as replica shards.
* Replica shards are stored on different nodes than their primary shard to protect against disk failures.
* The number of replica shards can be configured when creating an index, with one being the default value.
* Replica shards are fully functional and can serve search requests just like primary shards.
* Replication groups consist of a primary shard and its replica shards.
* Replication is only enabled in clusters with multiple nodes, and additional replica shards are allocated as more nodes are added.
* Replication provides both data availability and increased query throughput.
* In case of node failures, replica shards ensure that data remains accessible.
* Snapshots are separate from replication and are used for data backups and restoring the state of the cluster or indices to a specific point in time.
* Replication and snapshots serve different purposes: replication protects against data loss in real-time, while snapshots provide backups and restore points.
* The number of replica shards and nodes depends on the level of fault tolerance and performance requirements for the specific use case.
* Increasing the number of replica shards can improve query throughput by parallelizing search requests across the primary and replica shards.
* Elasticsearch automatically coordinates query execution among shards and takes advantage of multiple cores in nodes to run search queries in parallel.
* Replication can increase the throughput of an index but requires additional disk space for storing replica shards.
* The optimal configuration for nodes, shards, and replica shards depends on various factors and should be determined based on specific requirements and performance testing.
* The cluster's health status can indicate if replica shards are unassigned due to a single-node cluster setup.
* The "\_cat" API with the "shards" command can be used to view shard allocation information, including primary and replica shards.
* Kibana indices are configured with dynamic replica settings, automatically adjusting the number of replicas based on the number of nodes in the cluster.
* Replica shards provide data redundancy and availability, while auto\_expand\_replicas ensures at least one replica shard for Kibana indices in multi-node clusters.
* Understanding replication is essential for ensuring data reliability, availability, and performance in Elasticsearch.

Adding more nodes to the cluster:

* Sharding enables Elasticsearch to scale the storage capacity of an index by dividing it into smaller pieces.
* However, sharding alone cannot solve the problem of limited disk space in the long term.
* Replication requires a cluster with multiple nodes to provide fault tolerance and data redundancy.
* Adding more nodes to the cluster is necessary to benefit from replication.
* The process of adding nodes is explained for a local setup, but cloud-based deployments have their own mechanisms.
* The "Cluster Health" API shows the current state of the cluster, and a "yellow" status indicates unassigned replica shards.
* An enrollment token is required to start a new node, and the script "elasticsearch-create-enrollment-token" generates it.
* Each new node should be extracted from the Elasticsearch archive separately to ensure a clean start.
* The "elasticsearch.yml" file in the "config" directory is used to configure the name of the node for easy identification.
* The "elasticsearch" script within the "bin" directory, along with the enrollment token, starts the new node.
* After starting the new node, the cluster health transitions to "green" as the replica shards are assigned.
* Checking shard distribution confirms that replica shards are stored on different nodes than their primary shards.
* The process of adding additional nodes is primarily for development purposes, and production environments require additional configuration.
* A third node cannot be run as a single node cluster, as Elasticsearch requires at least two nodes for a three-node cluster.
* The process of starting a third node follows the same steps as adding the second node.
* The cluster automatically redistributes shards evenly across available nodes, even if they are not necessary for data storage.
* Simulating a node failure, either gracefully or forcefully, triggers shard reallocation and cluster rebalancing.
* Delayed shard reallocation occurs to avoid immediate costly movements in the case of temporary network failures.
* The cluster health transitions back to "green" once the shard reallocation is complete.
* Shards that were on the leaving node become unassigned and are subsequently allocated to other nodes.
* Elasticsearch handles nodes joining and leaving the cluster automatically, ensuring data availability and replication.
* Restoring a shut-down node results in the shards being balanced across the nodes as if it had never left the cluster.
* Elasticsearch simplifies the process of adding and managing nodes, providing automatic shard allocation and rebalancing.

To get a new enrolment token for a node, run bin/elasticsearch-create-enrollment-token –scope node --url “https://localhost:9200”

To get a new enrolment token for kibana, run bin/elasticsearch-create-enrollment-token –

scope kibana

To enroll a node to an existing node, take the enrolment token and write: bin/elasticsearch --enrollment-token INSERTTOKENHERE

Overview of node roles:

* An Elasticsearch cluster consists of one or more nodes, and data is stored on shards, which are stored on nodes.
* Nodes can have different roles, such as "master," "data," "ingest," "coordination," and "voting-only."
* The "master" role makes a node eligible to be the cluster's master node, responsible for cluster-wide actions.
* Having dedicated master nodes is recommended for stability and to separate master and data nodes.
* The "data" role enables a node to store and serve queries related to cluster data. Disabling this role results in the node not storing any shards.
* Dedicated data nodes are useful for large and busy clusters to separate master and data nodes.
* The "ingest" role allows a node to run ingest pipelines, which manipulate documents before they are added to an index.
* Ingest pipelines are useful for simple data transformations, but for complex functionality, Logstash is recommended.
* Enabling or disabling ingest pipelines can be done at the node level, and dedicated ingest nodes can be used for high-volume ingestion.
* The "node.ml" and "xpack.ml.enabled" settings are related to machine learning capabilities, enabling dedicated machine learning nodes.
* "Coordination" nodes handle query routing and coordination, and having dedicated coordination nodes is useful for load balancing in large clusters.
* The "voting-only" role participates in the voting process for electing a new master node but cannot be elected as the master itself.
* Checking the "node.role" and "master" columns in Kibana shows the roles of each node, and "dim" indicates a node with the "data," "ingest," and "master" roles.
* Changing node roles is typically done for large clusters and can be useful for optimizing hardware resources and cluster performance.
* Modifying roles should be done with caution and only when necessary, and it is recommended to stick with the default roles until a specific need arises.
* Understanding the available roles helps in managing and optimizing Elasticsearch clusters.

Wrap up:

* Covered the setup of Elasticsearch and Kibana, whether it was done locally or using Elastic Cloud.
* Learned about the basic architecture of Elasticsearch, including clusters, nodes, indices, and documents.
* Explored the concepts of sharding and replication, which enable scalability and high availability in Elasticsearch.
* Introduced the concept of snapshots as a means of backing up data in Elasticsearch.
* Learned how to add additional nodes to a cluster for development purposes and discussed the various roles that nodes can have.
* Prepared to move on to the next section, which will focus on performing basic operations on documents in Elasticsearch.

Creating and deleting indices:

* Learn how to create and delete indices in Elasticsearch using the REST API.
* Deleting an index is as simple as sending a "DELETE" request with the index name.
* For creating an index, use the "PUT" request with the index name and provide the index settings in a JSON request body.
* Index settings can be specified within a nested "settings" object.
* Specify the number of shards using the "number\_of\_shards" setting.
* Specify the number of replica shards using the "number\_of\_replica" setting.
* The response includes an "acknowledged" key indicating the success of index creation and a "shards\_acknowledged" key indicating the successful startup of required primary shards.
* Moving on to adding documents to the newly created index.

To remove an index:

DELETE /INDEXNAME

To add an index with configuration:

PUT /products

{

"settings": {

"number\_of\_shards": 2,

"number\_of\_replicas": 2

}

}

Indexing documents:

* Indexing documents means adding them to an index in Elasticsearch.
* Indexing is performed by sending a POST request to the endpoint consisting of the index name followed by "/\_doc".
* The document itself is defined within the request body as a JSON object.
* An example of indexing a document in the "products" index with fields like "name," "price," and "in\_stock" is shown.
* The "\_shards" object in the response indicates the success and failure of storing the document on shards.
* The number of shards storing the document depends on the index configuration, including primary shards and replica shards.
* The document is added to the primary shard and its replica shards, forming a replication group.
* The "\_id" key in the response contains the automatically generated identifier for the document.
* To specify a custom ID for a document, change the HTTP verb to "PUT" and append the ID to the endpoint.
* The response shows that the specified ID is used for the document.
* The "action.auto\_create\_index" setting allows automatic index creation when adding documents to a non-existing index.
* It's best practice to explicitly create indices, but automatic index creation is convenient in development mode.
* Next, we'll explore retrieving documents from an index.

How to add a document with automatic id:

POST /product/\_doc

{

"name": "Coffee Maker",

"price": "1000",

"in\_stock": 80

}

How to add a document with id set manually:

PUT /product/\_doc/100

{

"name": "Toaster",

"price": "80",

"in\_stock": 10

}

Retrieving documents:

* Retrieving a document from an index requires knowing its ID.
* To retrieve a document, use the "GET" HTTP verb and specify the endpoint as the index name followed by "/\_doc" and the document ID.
* In the response, the document is returned under the "\_source" key, containing the JSON object specified when adding the document.
* If the document is not found, the "found" key in the response will be set to "false," and the "\_source" key will not be present.
* Retrieving a specific document by ID is straightforward and allows easy access to the document's data.
* More advanced searching for documents based on specific criteria will be covered later in the course.

Updating documents:

* Updating documents in Elasticsearch is done using the Update API.
* To update a document, send a POST request to the Update API endpoint followed by the document ID.
* In the request body, specify the updated fields within a "doc" object.
* The "result" key in the response indicates the outcome of the update operation, such as "updated" or "noop" (no change).
* After updating a document, you can retrieve it to verify the changes.
* When updating a document, Elasticsearch actually replaces the existing document with the updated version.
* Elasticsearch documents are immutable, meaning they cannot be changed directly.
* The Update API retrieves the document, modifies the specified fields, and reindexes the document with the same ID.
* The Update API saves time and reduces network round trips compared to manually retrieving and replacing the document.
* The Update API operates efficiently within a single request on the shard where the document is stored.
* Understanding how updates work internally helps clarify the concept of immutability in Elasticsearch.

Example of updating:

POST /product/\_update/100

{

"doc": {

"in\_stock": 3

}

}

“doc” must be included

How to add a new field:

Type in the new field and it will automatically add the field if it doesn’t exist already:

POST /product/\_update/100

{

"doc": {

"tags": ["nice product"]

}

}

Scripted updates:

* Elasticsearch supports scripted updates, which allow you to modify document fields without retrieving and replacing the entire document.
* Scripted updates use the Update API, where a "script" object is added to the request body.
* The script can access the existing document using the "\_source" property and perform custom logic on its fields.
* In the example, we decreased the value of the "in\_stock" field by one.
* Scripts can be written on a single line or as multiline strings using triple quotes.
* The result of a scripted update is indicated by the "result" key in the response, usually "updated" unless an explicit "noop" or "delete" operation is specified.
* Parameters can be defined in the request to make scripts dynamic, useful when sending queries from an application.
* The "noop" value is used when no changes are made to the document, even in scripted updates. Explicitly setting the operation to "noop" ignores the document.
* Setting the operation to "delete" deletes the document, and the "result" key will be "deleted" in the response.
* Scripted updates offer flexibility and control over document modifications, enabling conditional updates and deletions.
* Examples of scripted update queries and delete queries can be found in the accompanying GitHub repository.

To access the field and update it, you would use the context variable to access the source document through its \_source property, giving an object containing its fields. Then to access a field, use ctx.\_source.FIELDNAME.

Example:

POST /product/\_update/100

{

"script": {

"source": "ctx.\_source.in-stock--"

}

}

The above subtracts one from the in stock field for the product with id 100

POST /product/\_update/100

{

"script": {

"source": "ctx.\_source.in\_stock=10"

}

}

The above sets the value to 10 manually

How to access parameters to update values?

POST /product/\_update/100

{

"script": {

"source": "ctx.\_source.in\_stock"

}

}

Using conditional to modify values:

POST /product/\_update/100

{

"script":

{

"source": """

if (ctx.\_source.in\_stock == 0){

ctx.op = 'noop';

}

ctx.\_source.in\_stock--;

"""

}

}

Upserts:

* Upserts provide a way to conditionally update or insert documents in Elasticsearch. Let's explore how they work:
* Upserts are a way of performing operations that update documents if they already exist or insert new documents if they don't.
* When performing an upsert, you use the Update API, which allows you to specify a script that will be executed if the document already exists.
* Example:
* POST /my-index/\_update/1
* {
* "script": {
* "source": "ctx.\_source.in\_stock--;",
* "lang": "painless"
* }
* }
* To save time, I have prepared the queries for you. One query is familiar to you, and the other is an upsert query that we will now examine.
* Example:
* POST /my-index/\_update/101
* {
* "script": {
* "source": "ctx.\_source.in\_stock--;",
* "lang": "painless"
* },
* "upsert": {
* "name": "New Product",
* "price": 10,
* "in\_stock": 1
* }
* }
* The request path remains the same since we are still using the Update API. In this case, we will be indexing a new document with an ID of 101.
* In an upsert operation, if the document with the specified ID already exists, the provided script is executed. Otherwise, the contents of the "upsert" option are added as a new document.
* Example:
* POST /my-index/\_update/101
* {
* "script": {
* "source": "ctx.\_source.in\_stock--;",
* "lang": "painless"
* },
* "upsert": {
* "name": "New Product",
* "price": 10,
* "in\_stock": 1
* }
* }
* Now let's execute the query and see what happens.
* Since no document with an ID of 101 existed before, the contents of the "upsert" option should have been indexed as a new document.
* Example:
* {
* "\_index": "my-index",
* "\_type": "\_doc",
* "\_id": "101",
* "\_version": 1,
* "result": "created",
* "\_shards": {
* "total": 2,
* "successful": 1,
* "failed": 0
* },
* "\_seq\_no": 0,
* "\_primary\_term": 1
* }
* To verify this, we can inspect the query results, particularly the "result" key.
* The value of the "result" key is "created," indicating that the query successfully created a new document.
* Example:
* {
* "result": "created"
* }
* We can retrieve the newly created document to ensure that it exists and matches our expectations.
* Example:
* GET /my-index/\_doc/101
* As expected, the document exists and contains the expected fields.
* Now, let's rerun the first query, which is the upsert query.
* This time, the value of the "result" key within the query results is "updated." This indicates that the document already existed and was updated.
* Example:
* {
* "result": "updated"
* }
* Our script was executed, resulting in the increase of the "in\_stock" field to six.
* We can retrieve the document once again to confirm that the "in\_stock" field has indeed increased.
* Example:
* GET /my-index/\_doc/101
* As we can see, the value of the "in\_stock" field is now six, as expected.
* This demonstrates how upserts can be performed in Elasticsearch.
* Please note that the examples provided assume the existence of an index named "my-index" and the use of the default document type "\_doc".

Replacing documents:

* In Elasticsearch, you have the ability to replace existing documents with updated versions. Let's explore how to perform document replacement:
* Document replacement is a straightforward process and can be accomplished using a familiar query.
* Example:
* PUT /my-index/\_doc/100
* {
* "name": "Replaced Product",
* "price": 20,
* "in\_stock": 5
* }
* The query we used for indexing a new document with a specific ID can also be used for replacing existing documents.
* Example:
* PUT /my-index/\_doc/100
* {
* "name": "Replaced Product",
* "price": 20,
* "in\_stock": 5
* }
* To demonstrate document replacement, let's retrieve the current document with the ID of 100.
* Example:
* GET /my-index/\_doc/100
* The retrieved document shows that the "in\_stock" field has been modified, and a new "tags" field has been added.
* Example:
* {
* "\_index": "my-index",
* "\_type": "\_doc",
* "\_id": "100",
* "\_version": 2,
* "\_seq\_no": 1,
* "\_primary\_term": 1,
* "found": true,
* "\_source": {
* "name": "Original Product",
* "price": 10,
* "in\_stock": 3,
* "tags": ["tag1", "tag2"]
* }
* }
* Now, let's modify the "price" field and replace the entire document.
* Example:
* PUT /my-index/\_doc/100
* {
* "name": "Replaced Product",
* "price": 20,
* "in\_stock": 5
* }
* After replacing the document, we can retrieve it to ensure that the replacement was successful.
* Example:
* GET /my-index/\_doc/100
* The retrieved document now reflects the changes we made. It contains exactly the fields specified in the replacement query.
* Example:
* {
* "\_index": "my-index",
* "\_type": "\_doc",
* "\_id": "100",
* "\_version": 3,
* "\_seq\_no": 2,
* "\_primary\_term": 1,
* "found": true,
* "\_source": {
* "name": "Replaced Product",
* "price": 20,
* "in\_stock": 5
* }
* }
* It's important to note that during document replacement, the entire document is replaced, which means any fields not included in the replacement query are removed.
* This demonstrates how straightforward it is to replace documents with Elasticsearch.
* Please note that the examples provided assume the existence of an index named "my-index" and the use of the default document type "\_doc".