The Linux Academy Elastic Certification Preparation Course

Study Guide

Myles Young
myles@linuxacademy.com
Jan 22, 2020

Contents

Installation and Configuration

Deploy and Start an Elasticsearch Cluster that Satisfies a Given Set of Requirements

Preparing the Host

Because we use the archive installation method for Elasticsearch, we need to configure system values that would otherwise have been configured by an RPM installation: * Create an `elastic` user * Increase the `nofile` limit via `/etc/security/limits.conf` to `65536` * Increase the `vm.max_map_count` limit via `/etc/sysctl.conf` and reload `sysctl` with `sysctl -p`

Deploying Elasticsearch

For the exam, we use the archive installation method to `/home/elastic/elasticsearch`. Note that this is not going to be the preferred way to install Elasticsearch in a production environment.

- Download Elasticsearch 7.2.x from `https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-7.2.1.tar.gz`
- Unpack the tar from `/home/elastic`
- Rename or symlink the `elasticsearch-7.2.1` directory to `elasticsearch`

Starting Elasticsearch

For the exam, we will not configure Elasticsearch as a Linux service. Instead, we will start Elasticsearch manually by executing the binary as the `elastic` user.

- Start Elasticsearch as a background daemon from `/home/elastic/elasticsearch` with `./bin/elasticsearch -d -p pid`
- To stop Elasticsearch use `pkill -F pid`
- Always check the cluster log file during startup at `logs/cluster_name.log`
- To check cluster health after startup, use `curl localhost:9200/_cat/health?v`
- To check individual node health after startup, use `curl localhost:9200/_cat/nodes?v`

Configure the Nodes of a Cluster to Satisfy a Given set of Requirements

There will be two main configuration files that you will want to be familiar with. The `config/elasticsearch.yml` file and the `config/jvm.options` file.

Notable configurations from `elasticsearch.yml` are, but not limited to, the following:

- `cluster.name`: Unique name of the cluster
- `node.name`: Unique name of the node
- `node.master`: Boolean value for the master-eligible role
- `node.data`: Boolean value for the data role
- `node.ingest`: Boolean value for the ingest role
- `node.attr.some_attribute`: Creates a node attribute to tag/label nodes (think hot/warm architecture)
- `path.repo`: Specifies a filesystem path to be used as a snapshot repository
- `network.host`: Interfaces for Elasticsearch to bind to (typically `_local_` and/or `_site_`)

- `discovery.seed_hosts`: List of seed nodes to ping on startup (typically specify master-eligible nodes here)
- `cluster.initial_master_nodes`: List of master-eligible node names for bootstrapping the cluster and preventing split-brain
- `xpack.security.enabled`: Boolean value to enable the security plugin
- `xpack.security.transport.ssl.enabled`: Boolean value to enable transport network encryption
- `xpack.security.transport.ssl.verification_mode`: Set to `full` for node-level verification, or `certificate` for just certificate-level verification
- `xpack.security.transport.ssl.keystore.path`: Path to keystore file for transport network encryption
- `xpack.security.transport.ssl.truststore.path`: Path to truststore file for transport network encryption
- `xpack.security.http.ssl.keystore.path`: Path to keystore file for http network encryption
- `xpack.security.http.ssl.truststore.path`: Path to truststore file for http network encryption

Notable configurations from `jvm.options` are, but not limited to, the following:

- `-Xms` initial heap size
- `-Xmx` max heap size

Secure a Cluster Using Elasticsearch Security

Generating Certificates

You can generate a Certificate Authority (CA) and node certificates with the `certutil` tool.

- Generate a CA with `./bin/elasticsearch-certutil ca`
- Generate a node cert with `./bin/elasticsearch-certutil cert --ca your_ca --name node_name --dns hostname --ip ip_address`
- This generates PKCS#12 files by default which can be used as both the keystore and truststore for a node

Encrypting the Transport Network

First, you must have created or been provided with node certificates to enabled this feature. You do not need to enable this for single-node clusters unless you need to perform cross-cluster-search with another cluster that has transport network encryption enabled.

Each of the following settings can be set in `elasticsearch.yml`:

- Enable with `xpack.security.transport.ssl.enabled: true`
- Configure the certificate verification mode with `xpack.security.transport.ssl.verification_mode`
- Provide the keystore path with `xpack.security.transport.ssl.keystore.path`
- Provide the truststore path with `xpack.security.transport.ssl.truststore.path`

If you set a passphrase for a node's certificate then you must `add` it to the Elasticsearch node's keystore by setting the following with the `./bin/elasticsearch-keystore` utility:

- `xpack.security.transport.ssl.keystore.secure_password`
- `xpack.security.transport.ssl.truststore.secure_password`

Restart Elasticsearch after configuring transport network encryption.

Setting Built-In User Passwords

In Elasticsearch, we have several accounts that must have their passwords set after enabling Security. Use the `./bin/elasticsearch-setup-passwords` utility with `automatic` for randomized passwords, or `interactive` to specify your own passwords. If you're given passwords to use on the exam, then use `interactive`.

Encrypting HTTP Network

You must first create or be provided node certificates to enabled this feature. You typically want to set the built-in user passwords before enabling this feature in order to avoid potential issues with the `setup-passwords` utility.

Each of the following settings can be set in `elasticsearch.yml`:

- Enable with `xpack.security.http.ssl.enabled: true`
- Provide the keystore path with `xpack.security.http.ssl.keystore.path`
- Provide the truststore path with `xpack.security.http.ssl.truststore.path`

If you set a passphrase for a node's certificate then you must `add` it to the Elasticsearch node's keystore by setting the following with the `./bin/elasticsearch-keystore` utility:

- `xpack.security.http.ssl.keystore.secure_password`
- `xpack.security.http.ssl.truststore.secure_password`

Restart Elasticsearch after configuring transport network encryption.

Define Role-Based Access Control Using X-Pack Security

You may use the Kibana Management UI to create roles and users with a click interface, or you can use the `_security` APIs.

Creating a Role

Create a custom role with:

```
PUT _security/role/role_name_here
{
```

```
  "run_as": [ ... ],                # user impersonation
  "cluster": [ ... ],               # cluster level permissions
  "indices": [                      # index level permissions
    {
      "names": [ ... ],             # index names, patterns, or aliases
      "privileges": [ ... ],        # index permissions
      "field_security" : { ... },   # field level security
      "query": "..."                # document level security
    },
    ...
  ]
}
```

Creating a User

Create a custom user with:

```
PUT _security/user/user_name_here
{
  "password" : "",    # user's password
  "roles" : [ ... ],  # assigned roles
  "full_name" : "",   # display name
  "email" : ""        # email address
}
```

Indexing Data

Define an Index that Satisfies a Given set of Requirements

Create an index with:

```
PUT index_name_here
{
  "aliases": { ... },   # filtered and/or non-filtered aliases
  "mappings": { ... },  # explicit and/or dynamic mappings
  "settings": { ... }   # index and analysis settings
}
```

Perform Index, Create, Read, Update, and Delete Operations on the Documents of an Index

Create

Index a document with:

```
PUT index_name/_doc/doc_id
{
  "field_1": "string_value",   # string values are quoted
  "field_2": numerical_value,  # numerical values are unquoted
  "field_3": {                 # object field
    "inner_field_1": "",       # inner object field
    ...
```

```
  },
  ...
}
```

Read

Get a document with:

```
GET index_name/_doc/doc_id
```

Update

Perform a document type update with:

```
POST index_name/_update/doc_id
{
  "doc": {  # document update
    "field_1": "new_value",
    ...
  }
}
```

Perform a scripted type update with:

```
POST index_name/_update/doc_id
{
  "script": {  # scripted update
    "lang": "painless",
    "source": ""
```

```
  }
}
```

Delete

Delete a document with:

```
DELETE index_name/_doc/doc_id
```

Define and Use Index Aliases

Use the `_aliases` API to add or remove aliases with:

```
POST _aliases
{
  "actions": [
    {
      "add": {.            # add an alias
        "index": "",       # index name or pattern
        "alias": "",       # the alias to add
        "filter": { ... }  # filtered alias
      }
    },
    {
      "remove": {          # removes an alias
        "index": "",       # index name or pattern
        "alias": ""        # alias to remove
      }
    },
    ...
```

```
  ]
}
```

Define aliases in indexes or index templates like this:

```
{
  "aliases" : {
    "alias_1" : {},          # non-filtered alias
    "alias_2" : {            # filtered alias
      "filter" : {
        "term" : {
          "field_1" : "value"
        }
      }
    }
  }
}
```

Define and Use an Index Template for a Given Pattern That Satisfies a Given Set of Requirements

Create index templates with:

```
PUT _template/template_name
{
  "index_patterns": [ ... ],  # patterns to match index names on
  "aliases": { ... },         # filtered and/or non-filtered aliases
  "mappings": { ... },        # explicit and/or dynamic mappings
  "settings": { ... }         # index and analysis settings
}
```

Define and Use a Dynamic Template That Satisfies a Given set of Requirements

Create a dynamic template to control dynamic mapping behavior with:

```
PUT index_name
{
  "mappings": {
    "dynamic_templates": [
      {
        "template_name": {
          "match_mapping_type": "",  # datatype to match on
          "match": "",               # field value pattern to match
          "unmatch": "",             # field value pattern to ignore
          "mapping": {               # desired mapping
            "type": ""
          }
        }
      },
      ...
    ]
  }
}
```

You can also define dynamic templates in index templates in the same way.

Use the Reindex API and Update By Query API to Reindex and/or Update Documents

`_reindex` API

Use the `_reindex` API to copy and optionally mutate data with:

```
POST _reindex
{
  "source": {
    "index": "",          # source index
    "remote": {           # remote cluster reindexing
      "host": "",         # remote cluster host
      "username": "",     # remote cluster user
      "password": ""      # remote cluster user's password
    },
    "query": { ... }      # selective reindex via query
  },
  "script": {.            # script to modify documents with
    "lang": "painless",
    "source": ""          # script source code
  }
  "dest": {
    "index": "",          # destination index
    "pipeline": ""        # ingest pipeline to modify documents with
  },
}
```

`_update_by_query` API

Update many documents in place with a single request using the `_update_by_query` API like this:

```
POST index_name/_update_by_query
{
  "script": {.            # script to modify documents with
    "lang": "painless",
    "source": ""          # script source code
```

```
  },
  "query": { ... }       # selective update via query
}
```

If you want to simply touch all the documents in an index, to pick up a new mapping or apply a new index setting, use the `_update_by_query` API with no request body.

The `_update_by_query` can be used with an ingest pipeline with `POST index_name/_update_by_query?pipeline=pipeline_name`.

Define and Use an Ingest Pipeline That Satisfies a Given Set of Requirements, Including the Use of Painless to Modify Documents

Create an ingest pipeline with:

```
PUT _ingest/pipeline/pipeline_name
{
  "description": "",
  "processors": [ ... ]
}
```

There are a lot of processors available for use within ingest pipelines. If there is no processor for what you need to do, then you can always use the `script` processor to write your own painless source code for more advanced data mutation requirements.

Mappings and Text Analysis

Define a Mapping That Satisfies a Given Set of Requirements

Elasticsearch has dynamic mapping built in to map fields to what it thinks is the desired datatype. However, you can explicitly define your own mappings for a given index or index template. Here is the basic structure of how mappings are defined:

```
PUT index_name
{
  "mappings": {
    "properties": {
      "field_1": {   # field name
        "type": ""   # datatype
      },
      ...
    }
  }
}
```

Define and Use a Custom Analyzer That Satisfies a Given Set of Requirements

Elasticsearch ships with many analyzers depending on your full-text searching needs. In addition, it makes all the components used for its built-in analyzers available for you, so that you can build your own analyzer for specific use cases. Analyzers are comprised of one tokenizer, zero to many token filters, and zero or many character filters. You can combine each analyzer component in the analysis settings of an index like this:

```
PUT index_name
{
  "settings": {
    "analysis": {
      "analyzer": {
        "custom_analyzer_1": {     # name your analyzer
          "type": "custom",        # custom analyzers are always type custom
          "tokenizer": "",         # tokenizer
          "char_filter": [ ... ],  # character filters
          "filter": [ ... ]        # token filters
        }
      }
    }
  }
}
```

Define and Use Multi-Fields with Different Data Types and/or Analyzers

You can index a single field as multiple datatypes so that it can be used for multiple use cases through the use of multi-fields. Use the `fields` parameter in your mapping to define multi-fields like this:

```
PUT index_name
{
  "mappings": {
    "properties": {
      "field_1": {            # field name
        "type": ""            # datatype
        "fields": {           # multi-field context
          "multi_field_1": {  # multi-field name
            "type": ""        # multi-field datatype
          },
```

```
          ...
        }
      },
      ...
    }
  }
}
```

Configure an Index so That It Properly Maintains the Relationships of Nested Arrays of Objects

Arrays of objects are flatted into field arrays. This removes the relationships of field values to their originating object. The `nested` datatype maintains this relationship. The objects are still flattened, but the relationships of an object's field values are tracked and joined in the background at search time. Define a type `nested` field in your mapping like this:

```
PUT index_name
{
  "mappings": {
    "properties": {
      "field_1": {          # field name
        "type": "nested"    # nested datatype
      },
      ...
    }
  }
}
```

Configure an Index That Implements a Parent/Child Relationship

Join fields allow for parent/child relationships, but with some limitations:

- Only 1 join field per index mapping
- Parent and child documents must be indexed on the same shard (use `?routing=`)
- There can be many children to a parent but only one parent

Map a join field with:

```
PUT index_name
{
  "mappings": {
    "properties": {
      "join_field_1": {                # field name
        "type": "join",                # nested datatype
        "relations": {                 # define the parent/child relationship
          "parent_type": "child_type"  # name the parent and child types
        }
      },
      ...
    }
  }
}
```

Index a document as a parent with:

```
PUT index_name/_doc/doc_id
{
  "join_field_1": "parent_type",
```

```
  ...
}
```

Index a document as a child with:

```
PUT index_name/_doc/doc_id?routing=doc_id_of_parent_doc
{
  "join_field_1": {
    "name": "child_type",
    "parent": "doc_id_of_parent_doc"
  },
  ...
}
```

Documents with a parent/child relationship can utilize the special search queries `has_parent` and `has_child`.

Cluster Administration

Allocate the Shards of an Index to Specific Nodes Based on a Given set of Requirements

You can route indexes to nodes based on custom node attributes and also through a set of built-in node attributes like `_name`, `_host_ip`, `_publish_ip`, `_ip`, and `_host` by using the `index.routing.allocation` index setting. You have the option to `include`, `exclude`, or `require` nodes with a given attribute value.

Allow allocation to a node with `value_1` for `attribute_1` with:

```
PUT index_name/_settings
{
  "index.routing.allocation.include.attribute_1": "value_1"
}
```

Do not allow allocation to a node with `value_1` for `attribute_1` with:

```
PUT index_name/_settings
{
  "index.routing.allocation.exclude.attribute_1": "value_1"
}
```

Force allocation to a node with `value_1` for `attribute_1` with:

```
PUT index_name/_settings
{
```

```
  "index.routing.allocation.require.attribute_1": "value_1"
}
```

Configure Shard Allocation Awareness and Forced Awareness for an Index

Shard Allocation Awareness

If your cluster hardware is divided in some way (zone, rack, host, etc.), you can label the location of each node through node attributes. Then, you can configure shard allocation awareness to prefer that replicas are allocated to different locations than their primaries. That way, if you have two zones and one of them fails, you won't lose any data. All the replica data is allocated in the other zone. If you enable shard allocation awareness with two locations, but one of them is down, the cluster will try to allocate the replicas to the same location as the primaries.

Enable shard allocation awareness by adding the following to each master-eligible node:

```
cluster.routing.allocation.awareness.attributes: location_attribute
```

Forced Awareness

Forced awareness works the same way as shard allocation awareness. However, it will *never* allocate missing replicas to the last remaining zone. You must provide all the possible values of the location attribute when configuring forced awareness.

Enable forced awareness by adding the following to each master-eligible node:

```
cluster.routing.allocation.awareness.attributes: location_attribute
cluster.routing.allocation.awareness.force.zone.values: location_1,location_2,...
```

Diagnose Shard Issues and Repair a Cluster’s Health

Use the cluster allocation explanation API to get an explanation for a specific shard's allocation with:

```
GET /_cluster/allocation/explain
{
  "index": "index_name",    # provide the index name
  "shard": shard_number,    # provide the shard number
  "primary": true_or_false  # whether or not it is primary
}
```

You can also use the cluster allocation explanation API without providing a body, to return the first unassigned primary or replica shard it finds like this:

```
GET /_cluster/allocation/explain
```

When the cluster allocation explanation API is used on an assigned shard, it will provide an explanation as to its most recent allocation, and why it is currently allocated where it is. When used on an unassigned shard, it will provide an explanation as to why it is not able to assign a shard to a node.

Backup and Restore a Cluster and/or Specific Indices

Using the snapshot API, you can back up specific indices or an entire cluster to a snapshot repository. To set up a repository, you first need to add the path to the shared filesystem in each node's `elasticsearch.yml` configuration with:

```
path.repo: /path/to/shared/filesystem
```

Then create the snapshot repository with:

```
PUT /_snapshot/repo_name
{
  "type": "fs",
  "settings": {
    "location": "/path/specified/in/path.repo"
```

```
  }
}
```

Take a snapshot with:

```
PUT /_snapshot/repo_name/snapshot_name?wait_for_completion=true
{
  "indices": "index_1,index_2",  # list or regex pattern of indices
  "include_global_state": false  # include or exclude the cluster state
}
```

Restore from a snapshot with:

```
POST /_snapshot/repo_name/snapshot_name/_restore
{
  "indices": "index_1,index_2",             # list or regex pattern of indices
  "include_global_state": true,             # restore or ignore the cluster state
  "rename_pattern": "regex_patter_(group)", # match indices with a regex group
  "rename_replacement": "new_name_$1"       # rename indices using the regex group
}
```

Configure a Cluster for Use with a Hot/Warm Architecture

Configure `hot` nodes with a `temp` attribute in the `elasticsearch.yml` with:

```
node.attr.temp: hot
```

Configure `warm` nodes with a `temp` attribute in the `elasticsearch.yml` with:

```
node.attr.temp: warm
```

Configure indices to allocate to `hot` nodes with:

```
PUT index_name/_settings
{
  "index.routing.allocation.require.temp": "hot"
}
```

Configure indices to allocate to `warm` nodes with:

```
PUT index_name/_settings
{
  "index.routing.allocation.require.temp": "warm"
}
```

You typically want to allocate all new indices to your `hot` nodes and then re-allocate them to the `warm` nodes after the data becomes less relevant and is not searched for or indexed to as often.

Configure a Cluster for Cross-Cluster Search

To enabled cross-cluster search, you need to add some seed nodes for each remote cluster you want to search across to the cluster you want to search from:

```
PUT _cluster/settings
{
  "persistent": {
    "cluster": {
      "remote": {
        "cluster_1": {
          "seeds": [ "ip_address_1:9300", ... ]
        },
        "cluster_2": {
          "seeds": [ "ip_address_1:9300", ... ]
        },
```

```
          ...
        }
      }
    }
}
```

If any of the clusters are configured with X-Pack Security and transport network encryption, then they all need to be configured with X-Pack Security and transport network encryption. In addition, each cluster either needs to use the same CA for their transport network node certificates or trust the other nodes CA by adding it to their keystore.

Queries

Write and Execute a Search Query for Terms and/or Phrases in One or More Fields of an Index

Search queries can be used in API requests for `_search`, `_aliases`, `_reindex`, `_update_by_query`, `_security`, and much more. Elasticsearch is capable of dozens of queries. We will go over some of the common ones, but you should definitely read the documentation to familiarize yourself with all the query options.

Full-text Analyzed Search Queries

`match` Query

```
{
  "query": {
    "match": {
      "field_name": "matches the analyzed tokens from this string"
    }
  }
}
```

`match_phrase` Query

```
{
  "query": {
```

```
    "match_phrase": {
      "field_name": "matches the analyzed phrase in this string"
    }
  }
}
```

`multi_match` Query

```
{
  "query": {
    "multi_match": {
      "query": "matches the analyzed tokens from this string",
      "fields": [ "field_1", "field_2", ... ]
    }
  }
}
```

Term-Level Non-Analyzed Queries

`term` Query

```
{
  "query": {
    "term": {
      "field_1": "search_term"
    }
  }
}
```

`terms` Query

```
{
  "query": {
    "terms": {
      "field_1": [ "search_term_1", "search_term_2", ... ]
    }
  }
}
```

`range` Query

```
{
  "query": {
    "range": {
      "field_1": {
        "gte": "search_term",    # can also be exclusive with "gt"
        "lte": "search_term",    # can also be exclusive with "lt"
        "format": "date_format"  # format of the date string
      }
    }
  }
}
```

Write and Execute a Search Query that is a Boolean Combination of Multiple Queries and Filters

The `bool` query allows for the use of multiple queries and query types surrounded by conditional logic:

```
{
  "query": {
    "bool": {
      "must": [ ... ],           # queries that have to match
      "must_not": [ ... ],       # queries that will unmatch documents
      "should": [ ... ],         # queries that can match but don't all have to
      "minimum_should_match": 1, # how many provide "should" queries must match
      "filter": { ... }          # a query that does not effect relevancy scoring
    }
  }
}
```

Highlight the Search Terms in the Response of a Query

`highlight` shows where the query matches are by surrounding matched search terms with tags:

```
GET index_name/_search
{
  "query": { ... },
  "highlight": [
    "pre_tags": "",   # define your own starting tag
    "post_tags": "",  # define your ending tag
    "fields": {       # list the fields to highlight matches from
      "field_1": {},
      "field_2": {},
      ...
    }
  }
}
```

Sort the Results of a Query by a Given Set of Requirements

`sort` can organize the results of a query in descending or ascending order on multiple sort levels:

```
GET index_name/_search
{
  "query": { ... },
  "sort": [
    {
      "field_1": {       # first sort by field
        "order": "asc"   # ascending order
      }
    },
    {
      "field_2": {       # next field to sort on
        "order": "desc"  # descending order
      }
    },
    ...
  ]
}
```

Paginate the Results of a Search Query

Implement pagination using `size` and `from`. `size` + `from` is limited to 10,000 results:

```
GET index_name/_search
{
  "query": { ... },
  "from": 0,         # the result offset to start from
```

```
  "size": 25         # the number of results starting from the offset
}
```

Use the Scroll API to Retrieve Large Numbers of Results

`scroll` allows you to query huge result sets beyond the 10,000 limit of `size` + `from`. It can be used with either `GET` or `POST` requests. To start `scroll`, you first have to initial a `_search` query and specify how long to keep the search context open with the `?scroll=` parameter. For example, `POST index_name/_search?scroll=1m` will start a search query on the index `index_name`, and leave the search context open for one minute. Each subsequent `scroll` will reset the open search context timer.

Start scrolling with:

```
POST index_name/_search?scroll=open_search_context_time
{
  "query": { ... },
  "size": 1000,                 # how many documents to return each scroll
  "slice": {                    # slice into parallel scrolls
    "id": scroll_slice_number,  # the slice for this scroll
    "max": max_scroll_slices    # maximum number of slices
  }
}
```

This will return the search results but with a `scroll_id`. Then continue to scroll with:

```
POST _search/scroll
{
  "scroll": "",    # how long to leave the search context open for
  "scroll_id": ""  # the scroll ID from the last scroll
}
```

This will return the next batch of search results and another scroll ID. Repeat until you have the desired number of search results.

It is a good idea to delete scrolls after you are done with them, instead of leaving them open to timeout. Delete open scrolls with:

```
DELETE _search/scroll
{
  "scroll_id": [ ... ]  # array of scroll IDs or _all
}
```

Apply Fuzzy Matching to a Query

The `fuzzy` query allows for the matching of non-analyzed terms that are similar to the term being searched for.

```
{
  "query": {
    "fuzzy": {
      "field_name": {
        "value": "search_term",              # the starting search term
        "fuzziness": edit_distance,          # max char edits
        "prefix_length": starting_distance, # "fuzzify" after x chars
        "max_expansions": max_unique_terms, # maximum unique terms to match
        "transpositions": true_or_false     # char swapping counts as an edit
      }
    }
  }
}
```

The `match` query allows for the matching of an analyzed query that is similar to the query being searched for.

```
{
  "query": {
    "match": {
      "field_name": {
        "query": "search_query",      # the starting search query
        "fuzziness": edit_distance,   # max char edits
      }
    }
  }
}
```

Define and Use a Search Template

Search templates allow the use of the Mustache language to create search requests using parameters. The Mustache language makes it possible to mutate and convert the parameters in a number of ways as they are inputted into a search query. You can define a search template like this:

```
GET index_name/_search/template
{
  "source": {                              # the query specification
    "query": {
      "query_type": {
        "{{search_on}}": "{{search_for}}"  # parameter references with mustache
      }
    },
    "size": "{{page_size}}",
    "from": "{{offset}}"
  },
  "params": {                              # the input parameters
    "search_on": "field_1",
    "search_for": "search terms",
```

```
    "size": result_size,
    "from": starting_result
  }
}
```

Save a search template with the `_scripts` API to be reused later, like this:

```
POST _scripts/template_name
{
  "script": {
    "lang": "mustache",
    "source": {                                 # the query specification
      "query": {
        "query_type": {
          "{{search_on}}": "{{search_for}}"  # parameter references with mustache
        }
      },
      "size": "{{page_size}}",
      "from": "{{offset}}"
    }
  }
}
```

Use a saved search template like this:

```
GET index_name/_search/template
{
  "id": "template_name",                    # the saved search template name
  "params": {                               # the input parameters
    "search_on": "field_1",
    "search_for": "search terms",
    "size": result_size,
```

```
    "from": starting_result
  }
```

Write and Execute a Query That Searches Across Multiple Clusters

After configuring cross-cluster search, you can search across multiple clusters in the same search request. In the search URL, simply preface the remote index with the remote cluster label given during cross-cluster search configuration. Then provide multiple local or remote indices, separated with commas like this:

```
GET local_index,remote_cluster_1:remote_index,remote_cluster_2:remote_index/_search
{
  "query": { ... }
}
```

Aggregations

Write and Execute Metric and Bucket Aggregations

Metric aggregations compute a single or multi-value output from the input data. Bucket aggregations organize the input data into groups, otherwise known as buckets. All metric and bucket aggregations start out the same way as follows:

```
GET index_name/_search
{
  "size": 0,            # we typically don't want documents when aggregating
  "aggs": {             # aggregation context
    "aggregation_1": {  # your own label for this aggregation
      "": { ... }.      # the type of aggregation
    }
  },
  "query": { ... }      # use a query to reduce the dataset you aggregate on
}
```

There are many metric and bucket aggregations made available by Elasticsearch, too many to explore individually in this course. We covered most of the common ones, but you will need to read the documentation to familiarize yourself with the rest. The hardest thing about aggregation questions on the exam is not building them, but rather figuring out that you need aggregations in the first place, and which ones you specifically need to answer whatever question is being asked.

Write and Execute Aggregations That Contain Sub-Aggregations

Often times, the questions you need to answer with aggregations are not going to require a single-level aggregation. This is when you need to think about nesting aggregations in the same request. The most common form of this is using a bucket aggregation to divide your data into groups, and then performing a metric aggregation on each group:

```
GET index_name/_search
{
  "size": 0,                    # we typically don't want documents when aggregating
  "aggs": {                     # aggregation context
    "aggregation_1": {          # your own label for this aggregation
      "": { ... },              # the type of aggregation
      "aggs": {                 # sub-aggregation context
        "sub_aggregation_1": { # your own label for this aggregation
          "": { ... }           # the type of aggregation
        }
      }
    }
  },
  "query": { ... }              # use a query to reduce the dataset you aggregate on
}
```

Write and Execute Pipeline Aggregations

Pipeline aggregations are either the `sibling` or `parent` type. Sibling pipeline aggregations use the output of a sibling aggregation to provide a new aggregation output at the same level as its sibling. Parent pipeline aggregations use the output of a parent aggregation to compute new buckets to add to existing buckets:

```
GET index_name/_search
{
  "size": 0,                        # we typically don't want documents
```

```
  "aggs": {                           # aggregation context
    "aggregation_1": {                # your own label for this aggregation
      "": { ... },                    # the type of aggregation
      "aggs": {                       # sub-aggregation context
        "sub_aggregation_1": {        # your own label for this aggregation
          "": { ... }                 # the type of aggregation
        },
        "sub_aggregation_2": {        # your own label for this aggregation
          "": {                       # the type of parent pipeline aggregation
            "buckets_path": "sub_aggregation_1"
          }
        },
        ...
      }
    },
    "aggregation_2": {                # your own label for this aggregation
      "": {                           # type of sibling pipeline aggregation
        "buckets_path": "aggregation_1>sub_aggregation_1"
      }
    },
    ...
  },
  "query": { ... }                    # use a query to reduce the dataset
}
```

You can have as many parent or sibling pipeline aggregations as you need in order to transform the data into the desired result.