elasticsearch-lua Documentation

Release alpha

Dhaval Kapil

Sep 04, 2017

Basic Documentation

1 Quickstart **3**
- 1.1 Installation . . . 3
- 1.2 Setting up Client . . . 3
- 1.3 Operations . . . 4
- 1.4 Wrap up . . . 6

2 Installation **7**
- 2.1 Using luarocks . . . 7
- 2.2 Directly from source . . . 7

3 JSON Arrays and Lua Tables **9**

4 Client Configuration **11**
- 4.1 Host Configuration . . . 11
- 4.2 Additional Parameters . . . 12

5 Indexing Documents **13**
- 5.1 Indexing Single Document . . . 13
- 5.2 Indexing Bulk Documents . . . 14

6 Getting Documents **17**
- 6.1 Getting Single Document . . . 17
- 6.2 Getting Multiple Documents . . . 17

7 Searching Documents **19**
- 7.1 URI Search . . . 19
- 7.2 Request Body Search . . . 19
- 7.3 Response . . . 20
- 7.4 Scan/Scroll Search . . . 20

8 Updating Documents **23**

9 Deleting Documents **25**

10 Namespaces **27**

11 Connection **29**

12 Selector **31**
12.1 In-Built Selectors 31
12.2 Custom Selector 32

13 Connection Pool **33**
13.1 In-Built Connection Pools 33
13.2 Custom Connection Pool 33

A simple low level client for elasticsearch written in lua. elasticsearch-lua is designed to be in accordance with other official clients for elasticsearch.

All API functions are developed to closely match with the REST API of elasticsearch. The client is extensible and developers can attach their own plug-ins.

CHAPTER 1

Quickstart

A quick guide for installing and using elasticsearch-lua.

Installation

- Directly using luarocks:

```
$ luarocks install elasticsearch
```

- Using elasticsearch as a dependency in your project
  - Add elasticsearch in your 'rockspec':

```
dependencies = {
  "elasticsearch"
}
```

  - Install dependencies using luarocks:

```
$ luarocks install <your_rockspec_file>
```

Setting up Client

Requiring elasticsearch in source file:

```
local elasticsearch = require "elasticsearch"
```

Creating a client:

```
local client = elasticsearch.client{
  hosts = {
    {
      host = "localhost",
      port = "9200"
    }
  }
}
```

Note: **host** and **port** are optional. In case any parameter is not specified, **host** defaults to 'localhost' and **port** defaults to '9200'.

Operations

elasticsearch-lua uses Lua tables to pass parameters for any operation. Common keys include **index**, **type**, **id** and **body**. Each kind of operation may have additional parameters. The **body** itself is a Lua table.

Every operation returns two values

```
local value1, value2 = client:operation()
```

The result depends on whether the operation succeeded or failed

	Success	Failure
value1	Actual result	nil
value2	HTTP status code	error message

Indexing Documents

To index a document, you need to pass **index**, **type**, **id** and **body** as parameters:

```
local res, status = client:index{
  index = "my_index",
  type = "my_type",
  id = "my_id",
  body = {
    my_key = "my_value"
  }
}
```

On success, the response will be returned in **res** as a Lua table and the HTTP status code in **status**. Sample output:

```
{
  ["_index"] = "my_index",
  ["_type"] = "my_type",
  ["_id"] = "my_id",
  ["created"] = true,
  ["_version"] = 1.0,
  ["_shards"] = {
    ["successful"] = 1.0,
    ["failed"] = 0.0,
    ["total"] = 2.0,
```

```
  }
}
```

Getting Documents

To get a document, you need to pass **index**, **type** and **id** of the document as parameters:

```
local res, status = client:get{
  index = "my_index",
  type = "my_type",
  id = "my_id"
}
```

The following response is returned if the document can be retrieved:

```
{
  ["_index"] = "my_index",
  ["_type"] = "my_type",
  ["_id"] = "my_id",
  ["found"] = true,
  ["_version"] = 1.0,
  ["_source"] = {
    ["my_key"] = "my_value"
  }
}
```

Otherwise, if the document is not present or cannot be retrieved, **nil** and an **error string** is returned.

Searching Documents

For searching documents, you can either perform a URI based search(by passing a **q** parameter) or a request body search(by passing the search DSL in **body** parameter). Searches can be restricted to 'index', 'type', or even both, by optionally passing **index** and **type** parameters. A sample request body search:

```
local res, status = client:search{
  index = "my_index",
  type = "my_type",
  body = {
    query = {
      match = {
        my_key = "my_value"
      }
    }
  }
}
```

The returned response consists of some metadata(**took**, **timed_out**, etc.) and a **hits** table. **hits.total** contains the total number of matches. **hits.hits** is a lua array, each entry represents one matching document.

```
{
  ["took"] = 3.0,
  ["timed_out"] = false,
  ["_shards"] = {
    ["failed"] = 0.0,
    ["total"] = 5.0,
```

```
    ["successful"] = 5.0
  },
  ["hits"] = {
    ["total"] = 1.0,
    ["max_score"] = 7.7399282,
    ["hits"] = {
      ["1"] = {
        ["_index"] = "my_index",
        ["_type"] = "my_type",
        ["_id"] = "my_id",
        ["_score"] = 7.7399282,
        ["_source"] = {
          ["my_key"] = "my_param"
        }
      }
    }
  }
}
```

Deleting Documents

To delete a document, you need to pass **index**, **type**, **id** and **body** as parameters:

```
local res, status = client:delete{
  index = "my_index",
  type = "my_type",
  id = "my_id"
}
```

On deletion, the following response is returned back:

```
{
  ["_index"] = "my_index",
  ["_type"] = "my_type",
  ["_id"] = "my_id",
  ["found"] = true,
  ["_version"] = 2.0,
  ["_shards"] = {
    ["failed"] = 0.0,
    ["total"] = 2.0,
    ["successful"] = 1.0,
  }
}
```

Wrap up

This was just a brief overview of using elasticsearch-lua. The **client** functions, the **body** parameter and the response returned bears resemblance with the Elasticsearch REST API.

Read the rest of the documentation to know more about the client.

CHAPTER 2

Installation

elasticsearch-lua has the following requirements:

- Lua 5.1 or higher

There are two ways to install elasticsearch-lua:

Using luarocks

luarocks is the package manager for Lua modules. To install Lua and LuaRocks luaver can be used. You can directly install elasticsearch-lua using LuaRocks:

```
$ luarocks install elasticsearch
```

The client will be installed and you can *require 'elasticsearch'* anywhere in your Lua code. If elasticsearch-lua is a dependency, add it in your rockspec file:

```
dependencies = {
  "elasticsearch"
}
```

Directly from source

elasticsearch-lua can also be installed directly from the source. However this is not recommended.

- Clone the repository:

  ```
  git clone https://github.com/DhavalKapil/elasticsearch-lua.git
  ```

- Install the following dependencies:
  - luasocket

  - lua-cjson
  - lunitx

Note: lunitx is *not* needed for using the client. You will need to install it **only if** you wish to run tests.

- Add the following code to use elasticsearch-lua:

```
package.path = package.path .. ";/path/to/elasticsearch-lua/src/?.lua";

local elasticsearch = require "elasticsearch";
```

CHAPTER 3

JSON Arrays and Lua Tables

Elasticsearch uses JSON API. The request body and the response returned is in JSON format. elasticsearch-lua converts JSON to Lua table and vice versa using the lua-cjson library. Hence, the user directly works with Lua tables. The request body passed to the client and the response returned by the client is a Lua table.

Sample example conversion:

```
{
  "query": {
    "match": { "content": "quick brown fox" }
  },
  "sort": [
    {
      "time": { "order": "desc" }
    },
    {
      "popularity": { "order": "desc" }
    }
  ]
}
```

Note the presence of an array in the above JSON. While creating a corresponding Lua table, take care to handle arrays using the standard 1-indexable format:

```
{
  query = {
    match = { content = "quick brown fox" }
  },
  sort = {
    {
      time = { order = "desc" }
    },
    {
      popularity = { order = "desc" }
    }
```

```
    }
}
```

CHAPTER 4

Client Configuration

elasticsearch-lua was designed to allow users to configure almost all of the parameters. The standard way of creating and configuring the client is:

```
local client = elasticsearch.client{
  hosts = {
  -- array of elasticsearch hosts
    {
      protocol = "http",
      host = "localhost",
      port = "9200"
    }
  },
  params = {
    -- additional parameters to configure the client
    pingTimeout = 2,
    logLevel = "warn"
  }
}
```

Every configuration passed while creating a client is *optional*. Default settings are used for configurations that are not provided by the user, as detailed below.

Host Configuration

A '**host**' refers to a single node of elasticsearch server. It may or may not be part of a cluster. Hosts are specified by using the key **hosts**. It consists of an array of hosts, wherein each host has 3 parameters:

- **protocol** : The underlying protocol to be used while communicating with the host. Defaults to '**http**'. (Presently, the client only supports **http**)
- **host**: The domain name or the IP address at which the host is running. Defaults to '**localhost**'.
- **port**: The port on which the host is listening. Defaults to '**9200**'.

Additional Parameters

You can also specify some additional parameters to configure the elasticsearch server. Again, these parameters are optional and have default values.

Parameter	Description	Default
pingTimeout	The timeout (in seconds) for any ping or sniff HTTP request made by the client to the elasticsearch server	1
requestEngine	The connection request ending to be used. For more details, see *Connection*.	'LuaSocket'
selector	The selector to be used. For more details, see *Selector*.	'RoundRobinSelector'
connectionPool	The connection pool to be used. For more details, see *Connection Pool*.	'StaticConnectionPool'
maxRetryCount	The number of times to retry an HTTP request before exiting with a *TransportError*	5
logLevel	The level of the inbuilt console logger. Follows the convention of log4j: ALL, DEBUG, ERROR, FATAL, INFO, OFF, TRACE, WARN. (ignores case)	'WARN'

CHAPTER 5

Indexing Documents

Elasticsearch accepts documents in the form of JSON. In elasticsearch-lua documents are passed as Lua tables. There are several ways to index documents.

Indexing Single Document

To index a document, you need to pass **index** and **type** as parameters. The document to be indexed is passed as a Lua table using the **body** parameter. Optionally, you can also provide an **id** for the document or let Elasticsearch generate it for you.

```
local res, status = client:index{
  index = "my_index",
  type = "my_type",
  id = "my_id",             -- Optional
  body = {
    my_key = "my_value"
  }
}
```

You can specify additional parameters such as **routing**, **refresh**, etc. alongside **index**, **type**.

```
local res, status = client:index{
  index = "my_index",
  type = "my_type",
  id = "my_id",             -- Optional
  routing = "company_xyz", -- Optional
  body = {
    my_key = "my_value"
  }
}
```

Refer to the Elasticsearch documentation for a complete list of allowed parameters.

Indexing Bulk Documents

Elasticsearch also supports bulk indexing of documents (indexing more than one document in one HTTP request). It is advised to use the Bulk API if you have to index many documents together. The client accepts an array of tables. You have to specify a pair of tables for every document. The first represents an action('index') in this context and the second represents the body. So basically the array consists of action, body, action, body, etc. tables.

```
local res, status = client:bulk{
  body = {
    -- First action
    {
      index = {
        ["_index"] = "my_index1",
        ["_type"] = "my_type1"
      }
    },
    -- First body
    {
      my_key1 = "my_value1",
    },
    -- Second action
    {
      index = {
        ["_index"] = "my_index2",
        ["_type"] = "my_type2"
      }
    },
    -- Second body
    {
      my_key2 = "my_value2",
    }
  }
}
```

You can also specify a common **index** or a **type** separately that would be used as default in every action.

```
local res, status = client:bulk{
  index = "my_index",
  body = {
    -- First action
    {
      index = {
        ["_type"] = "my_type1"
      }
    },
    -- First body
    {
      my_key1 = "my_value1",
    },
    -- Second action
    {
      index = {
        ["_type"] = "my_type2"
      }
    },
    -- Second body
    {
      my_key2 = "my_value2",
```

```
    }
  }
}
```

CHAPTER 6

Getting Documents

Like indexing documents, there are several ways to 'get' document(s) from the Elasticsearch server using elasticsearch-lua.

Getting Single Document

To get a single document, provide **index**, **type** and **id** of the document.

```
local res, status = client:get{
  index = "my_index",
  type = "my_type",
  id = "my_id"
}
```

Getting Multiple Documents

Elasticsearch supports getting multiple documents using a single request. It is advised to use this method in case you want to retrieve multiple documents. You need to pass an array of Lua tables to the MGET API. Each table represents details about one document and consists of three fields **_index**, **_type** and **_id**.

```
local res, status = client:mget{
  body = {
    docs = {
      -- First document
      {
        ["_index"] = "my_index1",
        ["_type"] = "my_type1",
        ["_id"] = "my_id1"
      },
      -- Second document
      {
```

```
        ["_index"] = "my_index2",
        ["_type"] = "my_type2",
        ["_id"] = "my_id2"
      }
    }
  }
}
```

In case every document has the same **index** or **type**, they can be specified separately instead of passing them in every document table.

```
local res, status = client:mget{
  index = "my_index",
  type = "my_type",
  body = {
    docs = {
      -- First document
      {
        ["_id"] = "my_id1"
      },
      -- Second document
      {
        ["_id"] = "my_id2"
      }
    }
  }
}
```

CHAPTER 7

Searching Documents

Search is the primary operation of Elasticsearch. The client supports all kinds of searches supported by the Elasticsearch REST API. There are two different ways to search.

URI Search

This kind of search uses a query string, which internally translates to a URI Search. Only a limited number of search options are available in this kind of search. However, it can be used for quick and handy searches. The optional **index** and **type** are passed along with **q**, the search query.

```
local res, status = client:search{
  index = "my_index",           -- Optional
  type = "my_type",             -- Optional
  q = "my_key:my_value"
}
```

This internally transforms to the following 'curl' request:

```
curl -XGET 'http://localhost:9200/my_index/my_type/_search?q=my_key:my_value'
```

Request Body Search

This kind of search involves a search DSL to be passed as **body**. All kinds of searches are possible using mode. Searches can be restricted to 'index', 'type', or even both, by optionally passing **index** and **type** parameters.

```
local res, status = client:search{
  index = "my_index",
  type = "my_type",
  body = {
    query = {
      match = {
```

```
        my_key = "my_value"
      }
    }
  }
}
```

The client allows all kinds of searches supported by Elasticsearch. Refer to the official documentation of Elasticsearch for details.

Response

The JSON response returned from Elasticsearch is parsed to a Lua table and returned directly. It consists of some metadata(**took**, **timed_out**, etc.) and a **hits** table. **hits.total** contains the total number of matches. **hits.hits** is a lua array and each entry represents one matching document.

```
{
  ["took"] = 3.0,
  ["timed_out"] = false,
  ["_shards"] = {
    ["failed"] = 0.0,
    ["total"] = 5.0,
    ["successful"] = 5.0
  },
  ["hits"] = {
    ["total"] = 1.0,
    ["max_score"] = 7.7399282,
    ["hits"] = {
      ["1"] = {
        ["_index"] = "my_index",
        ["_type"] = "my_type",
        ["_id"] = "my_id",
        ["_score"] = 7.7399282,
        ["_source"] = {
          ["my_key"] = "my_param"
        }
      }
    }
  }
}
```

Scan/Scroll Search

Elasticsearch provides a scan and scroll search functionality for retrieving a large number of documents efficiently, without paying the penalty of deep pagination. It works by first executing a 'scan' search. This initiates a 'scan window' which will remain open for the duration of the scan. This allows proper, consistent pagination. After a 'scan' search, the 'scroll' search is used to fetch paginated results over that window.

Scan Search

A scan search is just a search request with an additional **search_type** of **scan**, **scroll** and **size** parameters.

```
local res, status = client:search{
  index = "my_index",
  type = "my_type",
  search_type = "scan",
  scroll = "30s",          -- How long between scroll requests
  size = 50,               -- How many results *per shard* you want back
  body = {
    query = {
      match_all = {}
    }
  }
}
```

The scroll id is returned in the response, which is later used while 'scrolling'.

```
local scroll_id = res["_scroll_id"]
```

Scroll Search

Using the above generated **scroll_id**, scroll search can be performed repeatedly till no more results are found. The client exposes a separate **scroll** function for this purpose.

```
while true do
  -- Scroll request
  res, status = client:scroll{
    scroll = "30s",
    scroll_id = scroll_id
  }

  -- If no results obtained, break
  if #res["hits"]["hits"] == 0 then
    break
  end

  --
  -- Handle results
  --

  -- Update scroll_id
  scroll_id = res["_scroll_id"]
end
```

Note: On every request, a new **scroll_id** is generated. Always remember to update it.

Note: The behavior has changed a lot in Elasticsearch 2.1, we don't have *search_type* any more.

CHAPTER 8

Updating Documents

Elasticsearch supports updating documents. **index**, **type** and **id** parameters are required to be passed. The fields needed to be updated in the document are passed inside the **doc** parameter which is inside the **body**.

```
local res, status = client:update{
  index = "my_index",
  type = "my_type",
  id = "my_id",
  body = {
    doc = {
      my_key = "new_value",
      my_new_key = "another_value"
    }
  }
}
```

CHAPTER 9

Deleting Documents

Documents can be deleted by specifying **index**, **type** and the **id** of the document.

```
local res, status = client:delete{
  index = "my_index",
  type = "my_type",
  id = "my_id"
}
```

CHAPTER 10

Namespaces

The client exposes administrative functionalities through 'namespaces'.

Namespaces	Functionality
indices	Index related functions such as create, delete, etc.
nodes	Nodes related functions such as stats, info, etc.
cluster	Cluster related functions such as get and update settings, stats, etc.

These namespaces are accessible as *client.indices*, *client.nodes* and *client.cluster*. Sample code for using the namespaces:

```
-- Creating an index
local res, status = client.indices:create{
  index = "my_index"
}

-- Getting Nodes Stats
local res, status = client.nodes:stats()

-- Getting Cluster Stats
local res, status = client.cluster:stats()
```

Refer to the API documentation for a complete listing.

CHAPTER 11

Connection

A connection represents the lowest level of bare interaction with the Elasticsearch server in the form of HTTP requests. The client presently supports HTTP requests using the LuaSocket library.

Each time a request is to be made, the *request* function is called. However, support is provided to overload this function. While creating the client, *requestEngine* can be passed in *params*.

```
local client = elasticsearch.client{
  hosts = {
    {
      host = "localhost",
      port = "9200"
    }
  },
  params = {
    requestEngine = customRequestEngine
  }
}
```

customRequestEngine should be a Lua function which takes as arguments the http *method*, *uri*, *body* and *timeout*. It should return a table *response* with keys *code*, *statusCode* and *body*.

```
-------------------------------------------------------------------------
-- Makes a request to target server
--
-- @param   method  The HTTP method to be used
-- @param   uri     The HTTP URI for the request
-- @param   body    The body to passed if any
-- @param   timeout The timeout(if any) in seconds
--
-- @return  table   The response returned
-------------------------------------------------------------------------
function customRequestEngine(method, uri, body, timeout)
  -- Make an HTTP 'method' request to 'uri' with 'body' and 'timeout'
  response.code       = -- non nil for a successful request
  response.statusCode = -- HTTP status code returned
```

```
  response.body       = -- Response body
  return response
end
```

CHAPTER 12

Selector

The selector is an internal structure used in the client. Given an array of connections, it chooses a single connection. Some selectors don't even worry much about the internals of the connection. There are some in-built selectors that you can use or you can even write and use your own custom selector.

Note: A selector is called every time a request to the Elasticsearch server is to be made. The list of all available connections are passed to the selector.

In-Built Selectors

These selectors are defined inside *elasticsearch.selectors* module. There are three of them:

- **RoundRobinSelector (Default)**: The connections are selected in a round robin fashion. i.e. #1 connection will be chosen on the first request, #2 connection will be chosen on the second request and so on. This ensures a nearly even load across each node in the cluster.
- **StickyRoundRobinSelector**: This selector will always return('stick') the same connection each time, unless a request fails. In that case, it will move on to the next connection in a round robin fashion. This case is ideal for persistent connections where a considerable time is spent in opening and closing connections.
- **RandomSelector**: This selector returns a random connection from the list irrespective of whether it is *alive* or *not*. It internally uses *math.random* function.

To use any particular selector you have to specify it in the parameters while creating a client. By default, the RoundRobinSelector will be used.

```
local client = elasticsearch.client{
  hosts = {
    {
      host = "localhost",
      port = "9200"
    }
  },
```

```
  params = {
    selector = "StickyRoundRobinSelector"
    -- selector = "RoundRobinSelector"
    -- selector = "RandomSelector"
  }
}
```

Custom Selector

You can also implement your own custom selector and pass it to the client. To create a custom selector, extend *elasticsearch.selector.Selector* and implement the *selectNext* function.

```
-- Requiring the Base Class
local Selector = require "elasticsearch.selector.Selector"

-- Create a custom selector
local CustomSelector = Selector:new()

-- Implement the constructor function
function CustomSelector:new(o)
  o = o or {}
  -- Custom initialization code related to your algorithm
  -- End custom code
  setmetatable(o, self)
  self.__index = self
  return o
end

--------------------------------------------------------------------------------
-- Implement the logic to select and return a single connection from
-- an array of connections
--
-- @param   connections   A table of connections
-- @return  Connection    The connection selected
--------------------------------------------------------------------------------
function CustomSelector:selectNext(connections)
  local connection = -- Select a connection
  return connection
end
```

After creating a custom selector, it needs to be passed as a parameter while creating a client:

```
local client = elasticsearch.client{
  params = {
    selector = CustomSelector
  }
}
```

Note: A string is passed in **selector** when setting an in-built selector. Otherwise, an object is passed while setting a custom selector.

CHAPTER 13

Connection Pool

Connection Pool is an internal construct that maintains a list('pool') of connections to nodes that may be alive or dead. The job of a Connection Pool is to handle these dead and alive connections and return back an alive connection to provide the best behavior for the client. In case no alive connection can be found, a *nil* is returned. There are some in-built connection pools that you can use or you can even write and use your own custom connection pool.

Note: A connection pool is called every time a request to the Elasticsearch server is to be made. It internally uses the selector to choose a connection.

In-Built Connection Pools

These connection pools are defined inside *elasticsearch.connectionpool* module. There are two of them:

- **StaticConnectionPool (Default)**: The StaticConnectionPool selects a connection using a selector. It returns the connection if it is alive. If the connection is dead and a certain time interval has passed, it is tested again. If it is still dead, another connection is selected using the selector and the process is repeated. If no alive connection is found, the remaining dead connections are tested one by one.
- **SniffConnectionPool**: The SniffConnectionPool iterates the list of connections and returns the first alive connection found. For dead connections, it pings again to update its status. Also, after a certain time interval, it sniffs the existing connections to discover new nodes in the cluster and update its list of connections.

To use any particular connection pool you have to specify it in the parameters while creating a client. By default, the StaticConnectionPool will be used.

Custom Connection Pool

You can also implement your own custom connection pool and pass it to the client. To create a custom connection pool, extend *elasticsearch.connectionpool.ConnectionPool* and implement the *nextConnection* function.

```
-- Requiring the Base Class
local ConnectionPool = require "elasticsearch.connectionpool.ConnectionPool"

-- Create a custom connection pool
local CustomConnectionPool = ConnectionPool:new()

-- Implement the constructor function
function CustomConnectionPool:new(o)
  o = o or {}
  -- Custom initialization code related to your algorithm
  -- End custom code
  setmetatable(o, self)
  self.__index = self
  return o
end

--------------------------------------------------------------------------------
-- Implement the logic to return a single connection
--
-- @return  Connection    The connection selected
--------------------------------------------------------------------------------
function CustomConnectionPool:nextConnection()
  local connection = -- Select a connection
  return connection
end
```

After creating a custom ConnectionPool, it needs to be passed as a parameter while creating a client:

```
local client = elasticsearch.client{
  params = {
    connectionPool = CustomConnectionPool
  }
}
```

Note: A string is passed in **connectionPool** when setting an in-built Connection Pool. Otherwise, an object is passed while setting a custom Connection Pool.