Lab 9. Client APIs -- An Overview

In the previous lab, we saw the various steps to be configured for production readiness. We also explored the fine-tuning configuration needs to be considered during production setup for better performance. We learned how to secure a Solr, take backups, and configure logs. Then we got an overview of SolrCloud. Now we have a complete configured Solr with all the required configurations to meet any search request. In this lab, we will learn how Solr can be used with a web application and how to call APIs in different languages. We'll have an overview of various Client APIs supported by Solr. Client API overview

---

Solr comes with a bunch of REST APIs, which exposes its features such as query, index, delete, commit, and optimize; it also allows a web application to connect with Solr and perform any operation by calling these APIs. Solr has taken care of these REST APIs such that any web application developed in any programming language can connect to them. A REST API is developed based on the HTTP protocol; so a web application developed in any programming language, such as Java, .NET, Python, and Ruby, can easily connect to and call this API to perform various Solr operations. Using this API, a web application asks Solr to perform some operations, such as querying and indexing. Solr performs those operations and provides a response to the application. Solr also supports various response formats based on programming languages such as Java, JavaScript/JSON, Python, Ruby, PHP, and many more. So it becomes very easy for any programming languages to deal with Solr and to parse a response in expected format. Now Solr is the first choice for web applications developed in any language.

Queries are executed by creating a URL that contains all the query parameters. Solr examines the request URL, performs the query, and returns the response. The default response format is JSON, but we can configure the response format by the wt parameter. The other operations are similar, although in certain cases the HTTP request is a POST operation and contains information beyond whatever is included in the request URL. For example, an index operation may contain a document in the body of the request. JavaScript Client API

JavaScript client is very easy, simple, and straightforward. We don't need to create any client to connect to Solr. Also, no packages need to be installed for the JavaScript client. JavaScript sends the request to Solr using `XMLHttpRequest`. Solr processes the request and returns the response in JSON format, which can easily be parsed in JavaScript. We don't need to configure the wt response parameter as Solr, by default, returns the response in JSON format.

**Example:* *Configure `hostURL= http://localhost:8983/solr/techproducts/select` in JavaScript as follows:

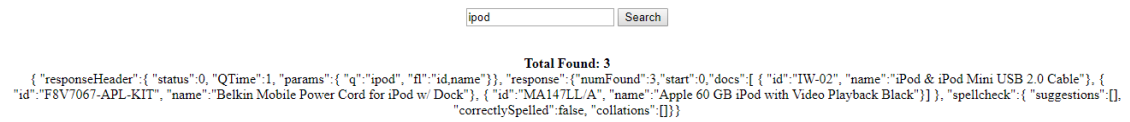
```
<html>
<head>
<title>Solr Javascript API Example</title>
<script language="Javascript">
//main function called when clicking on search button
function search() {
  //Solr search url
  var hostURL='http://localhost:8983/solr/techproducts/select';
    var xmlHttpReq = false;
    var xmlHttpClient = this;

  // Mozilla/Safari
    if (window.XMLHttpRequest) {
        xmlHttpClient.xmlHttpReq = new XMLHttpRequest();
    }
  // IE
```

```
    else if (window.ActiveXObject) {
        xmlHttpClient.xmlHttpReq = new ActiveXObject("Microsoft.XMLHTTP");
    }

  xmlHttpClient.xmlHttpReq.open('POST', hostURL, true);
    xmlHttpClient.xmlHttpReq.setRequestHeader('Content-Type', 'application/x-www-form-
urlencoded');
    xmlHttpClient.xmlHttpReq.onreadystatechange = function() {
        if (xmlHttpClient.xmlHttpReq.readyState == 4) {
            showResponse(xmlHttpClient.xmlHttpReq.responseText);
        }
    }

    var queryString = appendParams();
    xmlHttpClient.xmlHttpReq.send(queryString);
}

// get entered text in query parameter
function appendParams() {
  var querystring = document.getElementById("querystring").value;
  qstr = 'q=' + escape(querystring)+"&fl=id,name";
  return qstr;
}

//paring and displaying the response
function showResponse(str){
  document.getElementById("responsestring").innerHTML = str;
  var rsp = eval("("+str+")");
  var html = '<strong>Response</strong>';
  html= "</br><strong>Total Found: "+ rsp.response.numFound+"</strong>";
  document.getElementById("result").innerHTML = html;
}
</script>
</head>

<body>
  <div align='center'>
    <p>
      <input id="querystring" name="querystring" type="text" placeholder='Search
Here'>
      <input value="Search" type="button" onClick='search();'>
    </p>
    <div id="result"></div>
    <div id="responsestring"></div>
  </div>
</body>
</html>
```

This is a simple implementation to call the Solr API using JavaScript's `XMLHttpRequest`. Now if we want to run this code, we create a `.html` file and paste the preceding code in this file. The created HTML file should reside in the same environment in which Solr is running because modern browsers don't allow cross-site access in JavaScript for

security reasons. It may be possible that searching will not work due to an `Access-Control-Allow-Origin` error. There are various solutions available for this error; it's up to us how we can deal with it.

Now open the HTML file in your browser (supporting `XMLHttpRequest`); a search input box and a button will be displayed. Enter whatever text you want to search and click on the **`Search button. If everything goes well, the Solr API will be called and the response will be displayed as follows. Here we are searching for ipod:`**



Response:


```
{
   "responseHeader":{
      "status":0,
      "QTime":1,
      "params":{
         "q":"ipod",
         "fl":"id,name"
      }
   },
   "response":{
      "numFound":3,
      "start":0,
      "docs":[
         {
            "id":"IW-02",
            "name":"iPod & iPod Mini USB 2.0 Cable"
         },
         {
            "id":"F8V7067-APL-KIT",
            "name":"Belkin Mobile Power Cord for iPod w/ Dock"
         },
         {
            "id":"MA147LL/A",
            "name":"Apple 60 GB iPod with Video Playback Black"
         }
      ]
   },
   "spellcheck":{
      "suggestions":[

      ],
      "correctlySpelled":false,
      "collations":[

      ]
   }
}
```

We've got a JSON response; now we can apply our JavaScript skills to parse and display responses as per the application requirement. Likewise, we can test more capabilities of JavaScript towards the Solr API.

SolrJ Client API

SolrJ is built in Java technologies to connect with Solr from a Java application over HTTP.

Solr and SolrJ both are built-in Java technologies, so communication between them is easy and straightforward. While uploading a document, Solr needs all documents in XML or JSON format. SolrJ uses an internal binary protocol by default, called JavaBin. Normally, the client application sends an update request using HTTP POST with JSON or XML format, but the SolrJ client can send the update request as JSON, XML, or Solr's internal binary JavaBin format. The JavaBin protocol is more efficient than XML or JSON.

Apart from normal communication to Solr, SolrJ also supports load balancing across Solr nodes, automatically discovers locations of Solr servers in a SolrCloud mode, and easily handles bulk indexing for large amounts of data. It is also possible to embed Solr within a Java application and connect to it directly without establishing an HTTP connection to the server.

To create a SolrJ client, we do not need to worry much about the libraries; SolrJ libraries are already available in the Solr structure. Just navigate to `%SOLR_HOME%/dist`. You will find `solr-solrj.jar` (with a specific number); copy that jar and add it to your Java application build path. That's the only library required for your SolrJ implementation! Additionally needed libraries are available inside the `%SOLR_HOME%/dist/solrj-lib` directory; add all those to the Java application class path. Once the configuration is done, we can communicate to Solr using SolrJ.

Here is the simple SolrJ client that connects to the Solr API to run a search query:

```
package com.demo.solr.solrj;

import java.io.IOException;
import org.apache.solr.client.solrj.SolrQuery;
import org.apache.solr.client.solrj.SolrServerException;
import org.apache.solr.client.solrj.impl.HttpSolrClient;
import org.apache.solr.client.solrj.response.QueryResponse;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class SolrJSearchClientAPI {
  public static Logger _log = LoggerFactory.getLogger(SolrJSearchClientAPI.class);

  public static void main(String[] args){

    String hostURL = "http://localhost:8983/solr/techproducts";
    HttpSolrClient solr = new HttpSolrClient.Builder(hostURL).build();

    //set response parser
    //solr.setParser(new XMLResponseParser());

    //query configurations
    SolrQuery query = new SolrQuery();
    query.set("q", "ipod");

    query.set("fl", "id,name");
```

```
    /*alternate way to configure fl parameter
     * query.setFields("id","name");*/

    /*select different request handler
     query.setRequestHandler("/spell");*/

    try {
      QueryResponse response = solr.query(query);
      _log.info(response.toString());
    } catch (IOException e) {
      _log.error(e.getMessage());
    } catch (SolrServerException e) {
      _log.error(e.getMessage());
    }
  }
}
```

Response:

```
{responseHeader={status=0,QTime=1,params=
{q=ipod,fl=id,name,wt=javabin,version=2}},response={numFound=3,start=0,docs=
[SolrDocument{id=IW-02, name=iPod & iPod Mini USB 2.0 Cable}, SolrDocument{id=F8V7067-
APL-KIT, name=Belkin Mobile Power Cord for iPod w/ Dock}, SolrDocument{id=MA147LL/A,
name=Apple 60 GB iPod with Video Playback Black}]},spellcheck={suggestions=
{},correctlySpelled=false,collations={}}}
```

To query from the Solr instances running on the cloud using a `zkHostString` key:

```
String zkHostString = "zkServerA:2181,zkServerB:2181,zkServerC:2181/solr";
CloudSolrClient solr = new CloudSolrClient.Builder().withZkHost(zkHostString).build();
```

Let's run a client that adds a document (builds an index). In our selected core, `techproducts`, Let's add one more product:

```
package com.demo.solr.solrj;

import java.io.IOException;
import org.apache.solr.client.solrj.SolrServerException;
import org.apache.solr.client.solrj.impl.HttpSolrClient;
import org.apache.solr.client.solrj.response.UpdateResponse;
import org.apache.solr.common.SolrInputDocument;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class SolrJAddDocumentClientAPI {
  public static Logger _log =
LoggerFactory.getLogger(SolrJAddDocumentClientAPI.class);

  public static void main(String[] args){

    String hostURL = "http://localhost:8983/solr/techproducts";
    HttpSolrClient solr = new HttpSolrClient.Builder(hostURL).build();
```

```
    SolrInputDocument document = new SolrInputDocument();

    document.addField("id","HPPRO445");
    document.addField("name","HP Probook 445");
    document.addField("manu","Hewlett Packard");
    document.addField("features", "8GB DDR3LSD RAM");
    document.addField("weight","1.2");
    document.addField("price","800");

    try {
      UpdateResponse response = solr.add(document);
      solr.commit();
      _log.info(response.toString());
    } catch (SolrServerException e) {
      _log.error(e.getMessage());
    } catch (IOException e) {
      _log.error(e.getMessage());
    }
  }
}
```

Through this client, a new product, `HP Probook 445`, has been added to `techproducts`. Now if we search for the query `q=HP Probook 445`, we will get the following response if the product was added successfully:

```
{
  "responseHeader":{
    "status":0,
    "QTime":8,
    "params":{
      "q":"HP Probook 445"}},
  "response":{"numFound":1,"start":0,"docs":[
      {
        "id":"HPPRO445",
        "name":"HP Probook 445",
        "manu":"Hewlett Packard",
        "features":["8GB DDR3LSD RAM"],
        "weight":1.2,
        "price":800.0,
        "price_c":"800,USD",
        "_version_":1592089897249800192,
        "price_c____l_ns":80000}]
  },
  "spellcheck":{
    "suggestions":[],
    "correctlySpelled":false,
    "collations":[]}
}
```

Here we have added a single product, but if we have a product list (bulk) to add, we can do it like this:

```
package com.demo.solr.solrj;
```

```
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

import org.apache.solr.client.solrj.SolrServerException;
import org.apache.solr.client.solrj.impl.HttpSolrClient;
import org.apache.solr.client.solrj.response.UpdateResponse;
import org.apache.solr.common.SolrInputDocument;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class SolrJAddDocumentsClientAPI {

  public static Logger _log =
LoggerFactory.getLogger(SolrJAddDocumentsClientAPI.class);

  public static void main(String[] args) {

    String hostURL = "http://localhost:8983/solr/techproducts";
    HttpSolrClient solr = new HttpSolrClient.Builder(hostURL).build();

    List<SolrInputDocument> documentList = new ArrayList<SolrInputDocument>();

      SolrInputDocument document1 = new SolrInputDocument();
      document1.addField("id","id1");
      document1.addField("name","product1");
      documentList.add(document1);

      SolrInputDocument document2 = new SolrInputDocument();
      document2.addField("id","id2");
      document2.addField("name","product2");
      documentList.add(document2);

      //...
      //...

      SolrInputDocument documentn = new SolrInputDocument();
      documentn.addField("id","idn");
      documentn.addField("name","productn");
      documentList.add(documentn);

      try {
        UpdateResponse response = solr.add(documentList);
          solr.commit();
          _log.info(response.toString());
      } catch (SolrServerException e) {
        _log.error(e.getMessage());
      } catch (IOException e) {
        _log.error(e.getMessage());
      }
  }
}
```

For bulk processes, Solr provides a thread-safe class called `ConcurrentUpdateSolrClient`, which first holds all the documents in buffers and then writes to the HTTP connections.

Using SolrJ, we can delete a document (index) as follows:

```
package com.demo.solr.solrj;

import java.io.IOException;

import org.apache.solr.client.solrj.SolrServerException;
import org.apache.solr.client.solrj.impl.HttpSolrClient;
import org.apache.solr.client.solrj.response.UpdateResponse;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class SolrJDeleteDocumentClientAPI {
  public static Logger _log =
LoggerFactory.getLogger(SolrJDeleteDocumentClientAPI.class);

  public static void main(String[] args){

    String hostURL = "http://localhost:8983/solr/techproducts";
    HttpSolrClient solr = new HttpSolrClient.Builder(hostURL).build();

    try {
        UpdateResponse response = solr.deleteById("HPPRO445");
        solr.commit();
        _log.info(response.toString());
    } catch (SolrServerException e) {
      _log.error(e.getMessage());
    } catch (IOException e) {
      _log.error(e.getMessage());
    }
  }
}
```

Now if we search for the query `q=HP Probook 445`, it should not return any results if it was deleted successfully. In the same way, we delete documents in bulk:

```
solr.deleteById(List<String> ids);
```

Note

After adding/updating/deleting, don't forget to commit the transaction using `solr.commit()`; otherwise, the indexes will not be affected. Ruby Client API

Like JavaScript and SolrJ, Ruby can also connect with Solr using the Solr API and performs various operations such as search, indexing, and removal over the HTTP protocol. Solr also provides sufficient API support for the Ruby programming language. So, the application that builds on a Ruby platform can also take advantage of Solr. Solr reverts a query response in Ruby format, which can be interpreted easily in Ruby programming.

For Ruby applications, to communicate with Solr, `rsolr` is an extensive library. It provides all the functionalities needed to meet your expectations. To install the `rsolr` dependency, please refer

to http://www.rubydoc.info/gems/rsolr. To install `rsolr`, use this command:

```
gem install rsolr
```

Now Let's search using `rsolr` for the query `q=ipod`:

```
require 'rsolr'

solr = RSolr.connect :url => 'http://localhost:8983/solr/techproducts'

response = solr.get 'select', :params => {:q => 'ipod',:fl=>'id,name',:wt=>:ruby}

puts response
```

Response:

```
{"responseHeader"=>{"status"=>0, "QTime"=>2, "params"=>{"q"=>"ipod", "fl"=>"id,name",
"wt"=>"ruby"}}, "response"=>{"numFound"=>3, "start"=>0, "docs"=>[{"id"=>"IW-02",
"name"=>"iPod & iPod Mini USB 2.0 Cable"}, {"id"=>"F8V7067-APL-KIT", "name"=>"Belkin
Mobile Power Cord for iPod w/ Dock"}, {"id"=>"MA147LL/A", "name"=>"Apple 60 GB iPod
with Video Playback Black"}]}, "spellcheck"=>{"suggestions"=>[],
"correctlySpelled"=>false, "collations"=>[]}}
```

Solr treats the response format based on the `:wt` parameter value. For `:wt=>:ruby`, the response format will be a hash (a similar object returned by Solr but evaluated as Ruby), and for `:wt=>"ruby"`, the response will be a string. For other formats, Solr returns the response as expected.

Now Let's try to add a document (index) to Solr:

```
require 'rsolr'

solr = RSolr.connect :url => 'http://localhost:8983/solr/techproducts'

response = solr.add(:id=>'HPPRO445', :name=>'HP Probook 445', :manu=>'Hewlett
Packard', :features=>'8GB DDR3LSD RAM', :weight=>1.2, :price=>800)

solr.commit

puts response
```

After adding, if we search for the query `q=HP Probook 445`, it will give the following response, provided it was added successfully:

```
{"responseHeader"=>{"status"=>0, "QTime"=>4, "params"=>{"q"=>"HP Probook 445",
"wt"=>"ruby"}}, "response"=>{"numFound"=>1, "start"=>0, "docs"=>[{"id"=>"HPPRO445",
"name"=>"HP Probook 445", "manu"=>"Hewlett Packard", "features"=>["8GB DDR3LSD RAM"],
"weight"=>1.2, "price"=>800.0, "price_c"=>"800,USD", "_version_"=>1591126555425243136,
"price_c____l_ns"=>80000}]}, "spellcheck"=>{"suggestions"=>[],
"correctlySpelled"=>false, "collations"=>[]}}
```

After adding/updating/deleting, don't forgot to commit the transaction using `solr.commit()`, or else the indexes will not be affected.

We can add multiple documents at a time:

```
require 'rsolr'

solr = RSolr.connect :url => 'http://localhost:8983/solr/techproducts'

response = solr.add([
  { :id => 'id1', :name => "product1"},
  { :id => 'id2', :name => "product2"},
  { :id => 'idn', :name => "productn"}
  ])

solr.commit

puts response
```

Now delete a document using `rsolr`, like this:

```
require 'rsolr'

solr = RSolr.connect :url => 'http://localhost:8983/solr/techproducts'

response = solr.delete_by_id 'HPPRO445'

solr.commit

puts response
```

Now if we search for the query `q=HP Probook 445`, it should not give any response for the preceding ID if it was deleted successfully. In the same way, we can delete multiple documents in a single transaction:

```
require 'rsolr'

solr = RSolr.connect :url => 'http://localhost:8983/solr/techproducts'

response = solr.delete_by_id ['id1','id2']

solr.commit

puts response
```

Here we have explored various Solr APIs using basic Ruby programming steps. Now, referring to this, we can take a deep dive and add more configurations to Ruby and Solr API connectivity to achieve the expected results. Python Client API

Solr also comes with APIs that can be connected through applications developed in the Python programming language. Solr supports responses in Python format, which can be easily interpreted in Python programming. As we have seen before for JavaScript, SolrJ, and Ruby, Python also provides all the required packages to perform search, add, delete and so on.

Let's start with search. Here is the simple search configuration for searching a query `q=ipod` in `techproducts`:

```
import urllib.request

connection = urllib.request.urlopen('http://localhost:8983/solr/techproducts/select?
q=ipod&fl=id,name&wt=python')

response = eval(connection.read())

print(response)
```

Response:

```
{'responseHeader': {'status': 0, 'QTime': 51, 'params': {'q': 'ipod', 'fl': 'id,name',
'wt': 'python'}}, 'response': {'numFound': 3, 'start': 0, 'docs': [{'id': 'IW-02',
'name': 'iPod & iPod Mini USB 2.0 Cable'}, {'id': 'F8V7067-APL-KIT', 'name': 'Belkin
Mobile Power Cord for iPod w/ Dock'}, {'id': 'MA147LL/A', 'name': 'Apple 60 GB iPod
with Video Playback Black'}]}, 'spellcheck': {'suggestions': [], 'correctlySpelled':
False, 'collations': []}}
```

The response format is Python; here, even if we do not set `wt=python`, Solr will give the response in Python format by default. The `eval()` function is not recommended in Python because executing arbitrary code inside eval is easy to attack on the server. As a solution, Python provides a package called `json` or `simplejson`, which is more secure that `eval()`. This is the syntax for `json`:

```
import json
...
...
response = json.load(connection)
```

Now Let's add documents to Solr using Python programming. Python provides many libraries for adding documents to Solr through APIs. `pysolr` is one of the rich libraries provided, which gives us a good and simple way to add documents to Solr. We need to add `pysolr` packages like this from the command line:

```
pip install pysolr
```

This is a simple implementation to add a single document to Solr:

```
import pysolr

solr = pysolr.Solr('http://localhost:8983/solr/techproducts', timeout=5)

solr.add([
    {
        'id':'HPPRO445',
        'name':'HP Probook 445',
        'manu':'Hewlett Packard',
        'features':'8GB DDR3LSD RAM',
        'weight':1.2,
        'price':800
    }
])
```

After adding the document through the preceding implementation, searching for `q=HP Probook 445` will provide the same response as we have added if the document got added successfully to `techproducts`. Using `pysolr`, we do not need to perform any commit operation as it will be handled internally for any transactions.

In the same way, to add multiple documents at a time, here is a simple implementation:

```
import pysolr

solr = pysolr.Solr('http://localhost:8983/solr/techproducts', timeout=5)

solr.add([
    {
        'id':'id1',
        'name':'product1'
    },
    {
        'id':'id2',
        'name':'product2'
    },
    {
        'id':'idn',
        'name':'productn'
    }
])
```

All the documents will be added successfully inside `techproducts`.

We can delete a document using `pysolr` in Python. This a sample to delete a single document by providing the ID:

```
import pysolr

solr = pysolr.Solr("http://localhost:8983/solr/techproducts", timeout=5)

solr.delete(id='HPPRO445')
```

After deletion, if we search for `q=HP Probook 445`, it should not return the previously added document. Similar is the way to delete multiple documents at a time; here is a sample:

```
import pysolr

solr = pysolr.Solr("http://localhost:8983/solr/techproducts", timeout=5)

solr.delete(id=['id1','id2'])
```

All the documents will be deleted successfully for these IDs.

So we have seen basic Python configurations for operations such as search/add/delete, dealing with the Solr API. The Python programming language supports more detailed configurations to meet the all expectations from any web application. Now, taking this as a reference, we can dive deep and explore Python and Solr bounding in more detail.
Summary

In this lab, we got a basic overview of various JavaScript, SolrJ, Ruby, and Python supports for Solr API. We explored the basic configurations required to connect with the Solr API for each one of them. Then we designed basic

programs for searching, adding, adding in bulk, deleting, and deleting in bulk. We executed them and analyzed the response for each of them.