Solrj1.3

<!> Solr1.3
⚠️ Solr1.2

- Setting the classpath
  - Ant
    - With Solr 1.3
    - With Solr 1.2
  - Maven
- CommonsHttpSolrServer
  - Setting XMLResponseParser
  - Changing other Connection Settings
- EmbeddedSolrServer
- Usage
  - Adding Data to Solr
    - Streaming documents for an update
    - Directly adding POJOs to Solr
  - Reading Data from Solr
    - Advanced usage
    - Highlighting

Solrj is a java client to access solr. It offers a java interface to add, update, and query the solr index. This page describes the usage of the SolrJ release included with Solr1.3, with Solr1.2 and Solr1.3 war files.

Setting the classpath

Ant

With Solr 1.3

SolrJ is a part of the 1.3 release. The jars required in the classpath for SolrJ are,

- commons-io-1.3.1.jar
- commons-httpclient-3.1.jar
- commons-codec-1.3.jar
- commons-logging-1.0.4.jar
- apache-solr-common-1.3.0.jar
- apache-solr-solrj-1.3.0.jar

The above jars can be found in the dist/solrj-lib directory of the Solr 1.3.0 download.

With Solr 1.2

If you are using Solrj 1.3 with Solr 1.2, add the stax jars in addition to the jars above for Solr 1.3. Do not forget to set the XML parser

- geronimo-stax-api_1.0_spec-1.0.1.jar
- wstx-asl-3.2.7.jar
- stax-utils.jar

Maven

Solrj is available in the official Maven repository. Add the following dependency to your pom.xml to use SolrJ

```
        <dependency>
               <artifactId>solr-solrj</artifactId>
               <groupId>org.apache.solr</groupId>
               <version>1.3.0</version>
               <type>jar</type>
               <scope>compile</scope>
        </dependency>
```

If you need to use the EmbeddedSolrServer, you need to add the solr-core dependency too.

```
        <dependency>
                <artifactId>solr-core</artifactId>
                <groupId>org.apache.solr</groupId>
                <version>1.3.0</version>
                <type>jar</type>
                <scope>compile</scope>
        </dependency>
```

CommonsHttpSolrServer

The CommonsHttpSolrServer uses the Apache Commons HTTP Client to connect to solr.

```
  String url = "http://localhost:8983/solr";
  /*
    CommonsHttpSolrServer is thread-safe and if you are using the following constructor,
    you *MUST* re-use the same instance for all requests.  If instances are created on
    the fly, it can cause a connection leak. The recommended practice is to keep a
    static instance of CommonsHttpSolrServer per solr server url and share it for all requests.
    See https://issues.apache.org/jira/browse/SOLR-861 for more details
  */
  SolrServer server = new CommonsHttpSolrServer( url );
```

Setting XMLResponseParser

SolrJ 1.3 uses a binary format as the default format. For use with Solr 1.2 you must explicitly ask SolrJ to use XML format.

```
server.setParser(new XMLResponseParser());
```

Changing other Connection Settings

CommonsHttpSolrServer allows setting connection properties.

```
  String url = "http://localhost:8983/solr"
  CommonsHttpSolrServer server = new CommonsHttpSolrServer( url );
  server.setSoTimeout(1000);  // socket read timeout
  server.setConnectionTimeout(100);
  server.setDefaultMaxConnectionsPerHost(100);
  server.setMaxTotalConnections(100);
  server.setFollowRedirects(false);  // defaults to false
  // allowCompression defaults to false.
  // Server side must support gzip or deflate for this to have any effect.
  server.setAllowCompression(true);
  server.setMaxRetries(1); // defaults to 0.  > 1 not recommended.
```

EmbeddedSolrServer

The EmbeddedSolrServer provides the same interface without requiring an HTTP connection.

```
  // Note that the following property could be set through JVM level arguments too
  System.setProperty("solr.solr.home", "/home/shalinsmangar/work/oss/branch-1.3/example/solr");
  CoreContainer.Initializer initializer = new CoreContainer.Initializer();
  CoreContainer coreContainer = initializer.initialize();
  EmbeddedSolrServer server = new EmbeddedSolrServer(coreContainer, "");
```

If you want to use MultiCore features, then you should use this:

```
    File home = new File( "/path/to/solr/home" );
    File f = new File( home, "solr.xml" );
    CoreContainer container = new CoreContainer();
    container.load( "/path/to/solr/home", f );

    EmbeddedSolrServer server = new EmbeddedSolrServer( container, "core name as defined in solr.xml" );
    ...
```

If you need to use solr in an embedded application, this is the recommended approach. It allows you to work with the same interface whether or not you have access to HTTP.

⚠️ Note – EmbeddedSolrServer works only with handlers registered in solrconfig.xml. A RequestHandler must be mapped to /update for a request to /update to function.

Usage

Solrj is designed as an extendable framework to pass SolrRequest to the SolrServer and return a SolrResponse.

For simplicity, the most common commands are modeled in the SolrServer:

Adding Data to Solr

- Get an instance of server first

```
    SolrServer server = getSolrServer();
```

The `getSolrServer()` method body can be as follows if you use a remote server,

```
public SolrServer getSolrServer(){
    //the instance can be reused
    return new CommonsHttpSolrServer();
}
```

if it is a local server use the following,

```
public SolrServer getSolrServer(){
    //the instance can be reused
    return new EmbeddedSolrServer();
}
```

- If you wish to clean up the index before adding data do this

```
    server.deleteByQuery( "*:*" );// delete everything!
```

- Construct a document

```
    SolrInputDocument doc1 = new SolrInputDocument();
    doc1.addField( "id", "id1", 1.0f );
    doc1.addField( "name", "doc1", 1.0f );
    doc1.addField( "price", 10 );
```

- Construct another document. Each document can be independently be added but it is more efficient to do a batch update. Every call to `SolrServer` is an Http Call (This is not true for EmbeddedSolrServer).

```
    SolrInputDocument doc2 = new SolrInputDocument();
    doc2.addField( "id", "id2", 1.0f );
    doc2.addField( "name", "doc2", 1.0f );
    doc2.addField( "price", 20 );
```

- Create a collection of documents

```
    Collection<SolrInputDocument> docs = new ArrayList<SolrInputDocument>();
    docs.add( doc1 );
    docs.add( doc2 );
```

- Add the documents to Solr

```
    server.add( docs );
```

- Do a commit

```
    server.commit();
```

- To immediately commit after adding documents, you could use:

```
  UpdateRequest req = new UpdateRequest();
  req.setAction( UpdateRequest.ACTION.COMMIT, false, false );
  req.add( docs );
  UpdateResponse rsp = req.process( server );
```

Streaming documents for an update

This is the most optimal way of updating all your docs in one http request.

```
CommonsHttpSolrServer server = new CommonsHttpSolrServer();
Iterator<SolrInputDocument> iter = new Iterator<SolrInputDocument>(){
     public boolean hasNext() {
        boolean result ;
        // set the result to true false to say if you have more documensts
        return result;
      }

      public SolrInputDocument next() {
        SolrInputDocument result = null;
        // construct a new document here and set it to result
        return result;
      }
};
server.add(iter);
```

you may also use the `addBeans(Iterator<?> beansIter)` method to write pojos

Directly adding POJOs to Solr

- Create a Java bean with annotations. The `@Field` annotation can be applied to a field or a setter method. If the field name is different from the bean field name give the aliased name in the annotation itself as shown in the categories field.

```
import org.apache.solr.client.solrj.beans.Field;

 public class Item {
    @Field
    String id;

    @Field("cat")
    String[] categories;

    @Field
    List<String> features;

  }
```

The `@Field` annotation can be applied on setter methods as well example:

```
    @Field("cat")
   public void setCategory(String[] c){
       this.categories = c;
   }
```

There should be a corresponding getter method (without annotation) for reading attributes

- Get an instance of server

```
    SolrServer server = getSolrServer();
```

- Create the bean instances

```
    Item item = new Item();
    item.id = "one";
    item.categories =  new String[] { "aaa", "bbb", "ccc" };
```

- Add to Solr

```
   server.addBean(item);
```

- Adding multiple beans together

```
  List<Item> beans ;
  //add Item objects to the list
  server.addBeans(beans);
```

⚠️ Note – Reuse the instance of SolrServer if you are using this feature (for performance)

Reading Data from Solr

- Get an instance of server first

```
    SolrServer server = getSolrServer();
```

- Construct a SolrQuery

```
    SolrQuery query = new SolrQuery();
    query.setQuery( "*:*" );
    query.addSortField( "price", SolrQuery.ORDER.asc );
```

- Query the server

```
    QueryResponse rsp = server.query( query );
```

- Get the results

```
SolrDocumentList docs = rsp.getResults();
```

- To read Documents as beans, the bean must be annotated as given in the example.

```
   List<Item> beans = rsp.getBeans(Item.class);
```

Advanced usage

SolrJ provides a APIs to create queries instead of hand coding the query . Following is an example of a faceted query.

```
SolrServer server = getSolrServer();
SolrQuery solrQuery = new  SolrQuery().
              setQuery("ipod").
              setFacet(true).
              setFacetMinCount(1).
              setFacetLimit(8).
              addFacetField("category").
              addFacetField("inStock");
QueryResponse rsp = server.query(solrQuery);
```

All the setter/add methods return its instance . Hence these calls can be chained

Highlighting

Highlighting parameters are set like other common parameters.

```
  SolrQuery query = new SolrQuery();
  query.setQuery("foo");

  query.setHighlight(true).setHighlightSnippets(1); //set other params as needed
  query.setParam("hl.fl", "content");

  QueryResponse queryResponse = getSolrServer().query(query);
```

Then to get back the highlight results you need something like this:

```
  Iterator<SolrDocument> iter = queryResponse.getResults().iterator();

  while (iter.hasNext()) {
    SolrDocument resultDoc = iter.next();

    String content = (String) resultDoc.getFieldValue("content");
    String id = (String) resultDoc.getFieldValue("id"); //id is the uniqueKey field

    if (queryResponse.getHighlighting().get(id) != null) {
      List<String> highlightSnippets = queryResponse.getHighlighting().get(id).get("content");
    }
  }
```