Solrj

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 use of the SolrJ releases included with Solr 1.4.x releases, with the 1.4.x war files.

For information on using SolrJ with Solr1.3 and Solr1.2, see the Solrj1.3 page.

SolrJ/Solr cross-version compatibility

SolrJ generally maintains backwards compatibility, so you can use a newer SolrJ with an older Solr, or an older SolrJ with a newer Solr. There are some minor exceptions to this general rule:

If you're mixing 1.x and a later major version, you must set the response parser to XML, because the two versions use incompatible versions of javabin.

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

Mixing 3.x with a 4.x or later version will work fine, as long as you have NOT changed the request writer to binary. ⚠️ There is discussion in SOLR-3038 about changing the default request writer to binary. If this is done, then compatibility between 3.x and later versions would required setting the request writer explicitly to the XML version. If that change is made, there is another discussion in SOLR-4820 about adding automatic transport detection.

The one exception to this is the javabin format, which has changed in incompatible ways between major releases. You cannot use javabin to communicate between 1.4.1 or older and 3.1 or later (due to a change in character encoding in the protocol), and you cannot use javabin to communicate between 4.x client and 3.x or older server due to a change in the way the service urls are mapped, as well as backwards-incompatible extensions to the protocol. In these situations, simply don't provide a SolrServer request writer, and you will get the default (XML) wire format.

If you are using an older SolrJ with a 4.x or later server, the most likely feature you might need from SolrJ 4.0 that you won't find in older versions is the new setRequestHandler method on SolrQuery objects. It's worth noting that this method is even useful if SolrJ is newer than Solr.

Setting the classpath

There are several folders containing jars used by SolrJ: /dist, /dist/solrj-lib and /lib. A minimal set of jars (you may find need of others depending on your usage scenario) to use SolrJ is as follows:

From /dist:

- apache-solr-solrj-*.jar

From /dist/solrj-lib

- commons-codec-1.3.jar
- commons-httpclient-3.1.jar
- commons-io-1.4.jar
- jcl-over-slf4j-1.5.5.jar
- slf4j-api-1.5.5.jar

From /lib

- slf4j-jdk14-1.5.5.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>1.4.0</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>1.4.0</version>
               <type>jar</type>
               <scope>compile</scope>
        </dependency>
```

Also, if using EmbeddedSolrServer, keep in mind that Solr depends on the Servlet API. This may already be present in your web based applications, but even command line will require a dependency like so:

```
        <dependency>
               <groupId>javax.servlet</groupId>
               <artifactId>servlet-api</artifactId>
               <version>2.5</version>
        </dependency>
```

If you see any exceptions saying NoClassDefFoundError, you will also need to include:

```
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-simple</artifactId>
            <version>1.5.6</version>
        </dependency>
```

If you use solr-solrj version 1.4.1 and slf4j-simple 1.5.6 you may get IllegalAccessError because of slf4j-api change. Solrj uses slf4j-api 1.5.5 so you have to use slf4j-simple 1.5.5 or other binding with appropriate version. See slf4j FAQ.

HttpSolrServer

The HttpSolrServer uses the Apache Commons HTTP Client to connect to solr. Note: CommonsHttpSolrServer was changed to HttpSolrServer and StreamingUpdateSolrServer is now ConcurrentUpdateSolrServer as of solr 4.0.

```
  String url = "http://localhost:8983/solr";
  /*
    HttpSolrServer 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 HttpSolrServer 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 HttpSolrServer( url );
```

Changing other Connection Settings

HttpSolrServer allows setting connection properties.

```
String url = "http://localhost:8983/solr"
HttpSolrServer server = new HttpSolrServer( url );
server.setMaxRetries(1); // defaults to 0.  > 1 not recommended.
server.setConnectionTimeout(5000); // 5 seconds to establish TCP
// Setting the XML response parser is only required for cross
// version compatibility and only when one side is 1.4.1 or
// earlier and the other side is 3.1 or later.
server.setParser(new XMLResponseParser()); // binary parser is used by default
// The following settings are provided here for completeness.
// They will not normally be required, and should only be used
// after consulting javadocs to know whether they are truly required.
server.setSoTimeout(1000);  // socket read timeout
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);
```

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.

⚠️ Note – *Isn't EmbeddedSolrServer not considered a "Best Practice"? Perhaps some issue with isolation of JVMs.*

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. For a local server, use HttpSolrServer:

```
   SolrServer server = new HttpSolrServer("http://HOST:8983/solr/");
```

- To use a local, embedded server instead:

```
SolrServer server = new EmbeddedSolrServer();
```

- If you wish to delete all the data from the index, do this

```
server.deleteByQuery( "*:*" );// CAUTION: deletes 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 );
```

Fields "id","name" and "price" are already included in Solr installation, you must add your new custom fields in SchemaXml.

- 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

In most cases ConcurrentUpdateSolrServer will suit your needs. Alternatively, the workaround presented below can be applied.

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

```
HttpSolrServer server = new HttpSolrServer();
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)

Solr Transactions

Solr implements transactions at the server level. This means that every commit, optimize, or rollback applies to all requests since the last commit/optimize /rollback.

The most appropriate way to update solr is with a single process in order to avoid race conditions when using commit and rollback. Also, ideally the application will use batch processing since commit and optimize can be expensive routines.

Reading data from a database

⚠️ This example class using SolrJ has not yet been tested, but hopefully it's complete enough for community comment.

It will get tested eventually and updated here if there are problems. The addResultSet method takes a JDBC ResultSet and adds the documents to Solr in batches. Managing the database connection and constructing the query are not included here.

As it's written it maps the database field names to the same fields in Solr, but there are some examples in the comments of how you could manually assign one or more fields.

```
import java.io.IOException;
import java.net.MalformedURLException;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Types;
import java.util.ArrayList;
import java.util.Collection;

import org.apache.solr.client.solrj.SolrServerException;
import org.apache.solr.client.solrj.impl.HttpSolrServer;
import org.apache.solr.common.SolrInputDocument;

public class Test
{
    private static int fetchSize = 1000;
    private static String url = "http://localhost:8983/solr/core1/";
    private static HttpSolrServer solrCore;

    public Test() throws MalformedURLException
    {
             solrCore = new HttpSolrServer(url);
    }

    /**
     * Takes an SQL ResultSet and adds the documents to solr. Does it in batches
     * of fetchSize.
     *
     * @param rs
     *            A ResultSet from the database.
     * @return The number of documents added to solr.
     * @throws SQLException
     * @throws SolrServerException
     * @throws IOException
     */
    public long addResultSet(ResultSet rs) throws SQLException,
            SolrServerException, IOException
    {
        long count = 0;
        int innerCount = 0;
        Collection<SolrInputDocument> docs = new ArrayList<SolrInputDocument>();
        ResultSetMetaData rsm = rs.getMetaData();
        int numColumns = rsm.getColumnCount();
        String[] colNames = new String[numColumns + 1];
```

```
/**
 * JDBC numbers the columns starting at 1, so the normal java convention
 * of starting at zero won't work.
 */
for (int i = 1; i < (numColumns + 1); i++)
{
    colNames[i] = rsm.getColumnName(i);
    /**
     * If there are fields that you want to handle manually, check for
     * them here and change that entry in colNames to null. This will
     * cause the loop in the next section to skip that database column.
     */
    // //Example:
    // if (rsm.getColumnName(i) == "db_id")
    // {
    // colNames[i] = null;
    // }
}

while (rs.next())
{
    count++;
    innerCount++;

    SolrInputDocument doc = new SolrInputDocument();

    /**
     * At this point, take care of manual document field assignments for
     * which you previously assigned the colNames entry to null.
     */
    // //Example:
    // doc.addField("solr_db_id", rs.getLong("db_id"));

    for (int j = 1; j < (numColumns + 1); j++)
    {
        if (colNames[j] != null)
        {
            Object f;
            switch (rsm.getColumnType(j))
            {
                case Types.BIGINT:
                {
                    f = rs.getLong(j);
                    break;
                }
                case Types.INTEGER:
                {
                    f = rs.getInt(j);
                    break;
                }
                case Types.DATE:
                {
                    f = rs.getDate(j);
                    break;
                }
                case Types.FLOAT:
                {
                    f = rs.getFloat(j);
                    break;
                }
                case Types.DOUBLE:
                {
                    f = rs.getDouble(j);
                    break;
                }
                case Types.TIME:
                {
                    f = rs.getDate(j);
                    break;
                }
                case Types.BOOLEAN:
```

```
                        {
                            f = rs.getBoolean(j);
                            break;
                        }
                        default:
                        {
                            f = rs.getString(j);
                        }
                    }
                    doc.addField(colNames[j], f);
                }
            }
            docs.add(doc);

            /**
             * When we reach fetchSize, index the documents and reset the inner
             * counter.
             */
            if (innerCount == fetchSize)
            {
                solrCore.add(docs);
                docs.clear();
                innerCount = 0;
            }
        }

        /**
         * If the outer loop ended before the inner loop reset, index the
         * remaining documents.
         */
        if (innerCount != 0)
        {
            solrCore.add(docs);
        }
        return count;
    }
}
```

Setting the RequestWriter

SolrJ lets you upload content in XML and Binary format. The default is set to be XML. Use the following to upload using Binary format. This is the same format which SolrJ uses to fetch results, and can greatly improve performance as it reduces XML marshalling overhead.

```
server.setRequestWriter(new BinaryRequestWriter());
```

⚠️ Note – be sure you have also enabled the "BinaryUpdateRequestHandler" in your solrconfig.xml for example like:

```
<requestHandler name="/update/javabin" class="solr.BinaryUpdateRequestHandler" />
```

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");
      }
    }
```

Load Balancer for Queries

Although you are certainly welcome to use an external load balancer of your choice when you have multiple Solr servers that can process queries, SolrJ does provide a simple built-in round-robin load balancer, LBHttpSolrServer.

Use Groovy and Grape

```
@Grab(group='org.apache.solr', module='solr-solrj', version='1.4.1')
@Grab(group='org.slf4j', module='slf4j-jdk14', version='1.5.5')
import org.apache.solr.client.solrj.impl.HttpSolrServer
import org.apache.solr.common.SolrInputDocument

String url = "http://localhost:8983/solr"
def server = new HttpSolrServer( url );

def doc = new SolrInputDocument()

doc.addField("id", 2)
doc.addField("word_s", "Cow")
doc.addField("desc_t", "Farm Animal")

server.add(doc)
server.commit()

println 'done'
```

Using with SolrCloud

SolrJ includes a 'smart' client for SolrCloud, which is ZooKeeper aware. This means that your Java application only needs to know about your Zookeeper instances, and not where your Solr instances are, as this can be derived from ZooKeeper.

To interact with SolrCloud, you should use an instance of CloudSolrServer, and pass it your ZooKeeper host or hosts.

Beyond the instantiation of the CloudSolrServer, the behaviour should be the same as regular SolrJ.

```
import org.apache.solr.client.solrj.impl.CloudSolrServer;
import org.apache.solr.common.SolrInputDocument;

CloudSolrServer server = new CloudSolrServer("localhost:9983");
server.setDefaultCollection("collection1");
SolrInputDocument doc = new SolrInputDocument();
doc.addField( "id", "1234");
doc.addField( "name", "A lovely summer holiday");
server.add(doc);
server.commit();
```