SOLR in OLE Technical Document

Table of Contents

Last Published October 14, 2015

Purpose

Apache Solr is an enterprise search server with REST like APIs to retrieve documents put in through the process of indexing. OLE uses Solr extensively to index, store and retrieve Bibliographic data for faster search capabilities.

Detailed documentation on Apache Solr on how it works and other real technical details are better explained by the Solr project team available here. This document would concentrate more on how OLE uses Solr and the configurations and other files associated with it.

Dependencies

Though the bibliographic data resides in Database tables, Solr doesn't make use of data from the database. It neither reads nor writes data to database tables. The indexing is done and the search queries are constructed as Solr queries.

Data Model

The Solr data model is unique and does not resemble the conventional RDBMS (Relational DataBase Management System) data models used throughout the rest of the system. However, they share some similarities with conventional data models.

Document

A Document represents the basic unit of Information in Solr. It contains both fields and values that belong to a given entity. A Document can be thought of as a record in RDBMS. It could have a primary key, which becomes the logical identity of the data it represents. It also has a structure consisting of one or more attributes. Each attribute has a name, type and value. However, unlike database records, attributes are not restricted to hold one value. Also attributes cannot have a null value, they either exist or they don't.

A simple way to represent a Solr document is as a map. Map is a general data structure that maps a unique key to values. Each key can have one or more values. If represented as a JSON (JavaScript Object Notation), the documents would look as shown below.

```
{
    {
        "id":1,
        "title":"Master of the game",
        "author":"Sheldon, Sidney",
    }
    {
        "id":2,
        "title":"Mechanics of materials",
        "author":["Beer","Johnston","DeWolf","Mazurek"],
    }
    {
        "id":3,
        "title":"The other side of midnight",
        "author:"Sheldon, Sidney",
    }
    {
        "id":4,
        "title":"Bloodline",
        "author":"Sheldon, Sidney",
    }
}
```

Inverted Index

Solr uses an underlying, persistent structure called Inverted Index which is designed and optimized to deliver quick results during searches at retrieval time. An inverted index consists of an ordered list of all the terms that appear in a set of documents. Besides each term, the index also includes a list of documents where the term appears. In Solr, the index files are housed in the Solr data directory. This directory is configured in the *solrconfig.xml* file, the main configuration file.

Considering the above documents as example, the Inverted Index of title would look as follows.

Terms	Document Id			
	1	2	3	4
master	X			
of	X	X	X	
the	X		X	
game	X			
mechanics		X		
materials		X		
other			X	
side			X	
midnight			X	
bloodline				X

Solr Core

The index configuration of a Solr instance resides in a Solr core. The core is a container for a specific inverted index. On the disk, Solr cores are directories, each with some configuration files that defines its features and characteristics. The typical content found are a *core.properties* file that describes the core, a *conf* folder that contains configuration files - *schema.xml, solrconfig.xml* and other files, depending on components in use such as *stopwords.txt* and *synonyms.txt,* a *lib* folder where JAR files that are used by the core are placed.

Solr Schema

The Solr Schema declares what kinds of fields are there, which field should be used as the primary key, which fields are required and how to index and search the fields. It describes the structure and constraints of the document. The Solr Schema is defined in a file called the *schema.xml*. It contains several levels as defined below.

Field Types

Field Types are one of the top-level entities declared in Solr Schemas. It is declared using the <fieldType> element. The element allows multiple attributes such as name (Mandatory), class (Mandatory), sortMissingFirst, sortMissingLast, indexed, stored, multiValued, omitNorms, omitTermsAndFrequencyPositions, omitPositions, positionsIncrementGap, autogeneratePhraseQueries, compressed and compressThreshold.

Analyzer

An Analyzer pre-processes the input text fields and generates a stream of tokens at index or search time. The word token is interchangeably used with words and terms. An analyzer thus helps in converting an incoming text into simpler text using Tokenizers and Filters. It can be defined separately for index and query by using attribute *type*. If just one analyzer is defined with no attribute then that is valid for both phases.

A simple analyzer can be configured as shown below.

```
<fieldType name="text" class="solr.TextField">
    <analyzer class="org.apache.lucene.analysis.WhitespaceAnalyzer"/>
</fieldType>
```

The WhitespaceAnalyzer analyzes the content of the text field and emit the corresponding tokens.

For example, INPUT: “To be, or not to be”
OUTPUT: “To”, “be,”, “or”, “not”, “to”, “be”

More complex analyzers are defined as shown in the schema entry below.

Char Filters

Char Filters are optional components. They can manipulate a character by adding, removing, or replacing characters while preserving the original character position.

A simple Char Filter can be configured as shown below.

```
<fieldType name="text" class="solr.TextField">
    <analyzer>
        <charFilter class="solr.HTMLStripCharFilterFactory"/>
    </analyzer>
</fieldType>
```

The HTMLStripCharFilterFactory, as the name suggests, strips the HTML from the input and passes the result.

For example, INPUT: my <a href=”www.myblog.com”>page</a>

OUTPUT: my page

Tokenizer

A Tokenizer breaks an incoming character stream into one or more tokens based on a specific criteria. The resulting set of tokens is usually referred to as the token stream.

A simple Tokenizer can be configured as shown below.

```
<fieldType name="text" class="solr.TextField">
    <analyzer>
        <tokenizer class="solr.LowerCaseTokenizerFactory" />
    </analyzer>
</fieldType>
```

The LowerCaseTokenizerFactory creates tokens by lowering all letters and dropping non-letters.

For example, INPUT: I live in London

OUTPUT: “i”, “live”, “in”, “london”

Token Filters

The Token Filters work on an input token stream from the Tokenizer. A filter can apply its logic to tokens in order to add, remove or replace tokens and produce new output token stream. They can be chained together with other filters to effect complex manipulations.

A simple Token Filter configuration looks as shown below.

```
<fieldType name="text" class="solr.TextField">
    <analyzer>
      <tokenizer class="solr.LowerCaseTokenizerFactory" />
      <filter class="solr.StopFilterFactory" words="stopwords.txt" ignoreCase="true" />
    </analyzer>
</fieldType>
```

The StopFilterFactory discards words specified in the stopwords.txt file during index or search.

For example, If ‘the’ is present in the stopwords.txt file then,

A search for ‘The Godfather” would return results of only ‘Godfather’.

A Solr Schema entry looks as shown below.

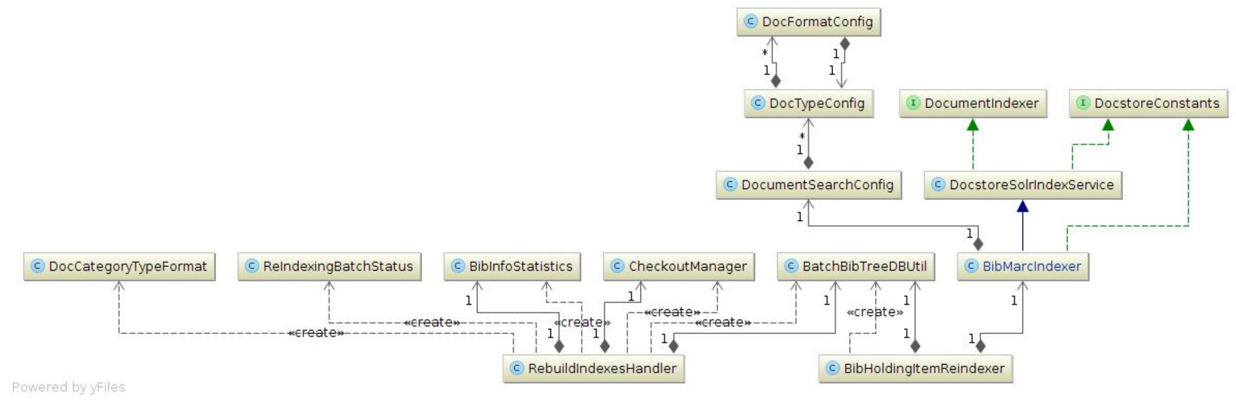
```
<fieldType name="bucketFirstLetter" class="solr.TextField"
                   sortMissingLast="true" omitNorms="true">
       <analyzer type="index">
                <charFilter class="solr.HTMLStripCharFilterFactory"/>
                <tokenizer class="solr.PatternTokenizerFactory"
                           pattern="^([a-zA-Z]).*" group="1" />
                <filter class="solr.SynonymFilterFactory"
                        synonyms="mb_letterBuckets.txt" ignoreCase="true"
                        expand="false"/>
       </analyzer>
       <analyzer type="query">
                <tokenizer class="solr.KeywordTokenizerFactory"/>
       </analyzer>
</fieldType>
```

Service Interface Design

OLE primarily uses Apache Solr to quicken the process of storing and retrieving Bibliographic data of institutions which typically run into volumes ranging from few hundred thousands to millions of records. The retrieving part involves Solr queries and is discussed here. This part would detail data on the Indexing part of the process.

Solr Indexing happens in two different scenarios in OLE. One is the reindexing triggered externally from the oledocstore interface (<OLE Link>/oledocstore/admin.jsp) after importing Bibliographic records. This method usually is done when the institution is importing records into the system for the first time. The other is the reindexing that happens as soon as a new record is added through the user interface.

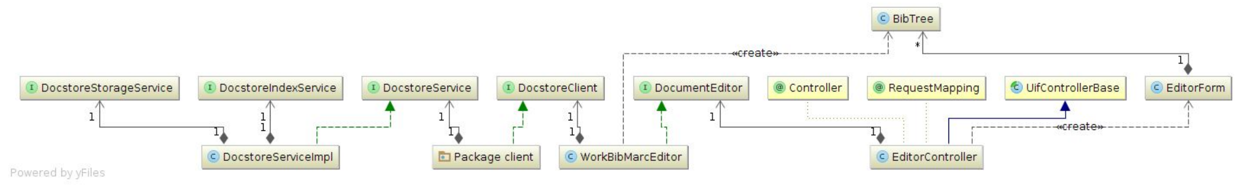
Reindexing triggered externally



The *run* method of the RebuildIndexesHandler class is where the indexing process is initiated. The *docCategory, docType* and *docFormat* parameters are retrieved from the properties file, docstore-category.properties, docstore-type.properties and docstore-format.properties file respectively. The different fields and their mapping are configured in the *DocumentConfig.xml* file. The *reIndex* method of the RebuildIndexesHandler class further calls the *index* method of the BibHoldingItemReindexer class.

The *init* method of BatchBibTreeDBUtil class is being called. This method is where queries are run to retrieve the list of Bibliographic data from the table *OLE_DS_BIB_T, OLE_DS_HOLDINGS_T* and *OLE_DS_ITEM_T*. The result set returns bibliographic data which are converted to objects and finally to Solr Documents in the *buildSolrDocsForBibTree* method of the BibMarcIndexer class. Finally the *indexSolrDocuments* method indexed the Solr Documents and commits them. The commit used here is not soft commit and hence searches immediately don't reflect the data.

Reindexing triggered internally



The Editor form is loaded by the *load* method of the EditorController class. Once the form is filled and submitted, the *save* method of the EditorController class takes over. The DocType is ascertained and a flag is set depending on which further method calls are made. The *save* method further calls the *saveDocument* method of the DocumentEditor interface class which is implemented by various classes based on the document type.

In case of Bib record in Marc format, the *saveDocument* method of the WorkBibMarcEditor class is called. The bib object is created from the data present in the editor form. The bib object is passed as a parameter to the *createBib* method of the DocStore Client. The *createBib* method of DocstoreServiceImpl class uses the Bib object to get the record into both the database and Solr index. The *create* method of RdbmsBibDocumentManager is called where the data is validated and modified and saved to the database. The *create* method of the DocstoreSolrIndexService class is where the data is indexed into Solr. The *buildSolrInputDocument* method builds the Solr document. This is followed by the *indexSolrDocuments* method where the Solr document generated by the *buildSolrInputDocument* method is passed and indexed. A soft commit is used here so that the record reflects immediately in the search.

Service Interface Design (REST/SOAP)

OLE supports REST services for Solr functions. This is documented here.

User Interface Design

The search workbench uses KRAD's UIF (User Interface Framework). A very good guide on this can be found here.

Data Importing

Not Applicable.

Data Exporting

Not Applicable

Workflow

Not applicable.

System Parameters

No System Parameters are currently used.

Roles and Permissions

Not applicable.