

Web Document Index and Search

Code and Documentation Written By
Steven Fan

Overview of System

Both the indexer and the searcher files are written in Java and mainly uses the Lucene architecture for building indexes and searching through them. The indexer utilizes the jsoup library to extract information from HTML files. The searcher utilizes the Spring program to allow configuration with HTML to build an interactive web searcher. The following documentation will document the source code itself. For convenience, Java files *Index.java* and *Search.java* are stored in the *files/* directory separate of the Java package hierarchy of the program.

Index.java

Initialize Code Block

```
System.out.print("Input index directory: ");
Scanner scanner = new Scanner(System.in);
String path = scanner.nextLine();
Analyzer analyzer = new StandardAnalyzer();
Directory directory = FSDirectory.open(Paths.get(path));
IndexWriterConfig config = new IndexWriterConfig(analyzer);
IndexWriter indexWriter = new IndexWriter(directory, config);
```

This code section initializes the Lucene class variables. The analyzer is selected as the *StandardAnalyzer* which parses the input based on the standard, such as taking into account capitalization, stop words, etc. The index will be written to the index directory folder provided by the user.

Extraction and Indexing

Lines 31 through 115 of the code consists mainly of extraction of HTML information using *jsoup* libraries. The program extracts data based on the UTF-8 code. The code block extracts an HTML's title, metadata, header, URL, and content. The program searches for HTML files in the data directory folder provided by the user.

```
Document doc = new Document();
try
{
    doc.add(new TextField("title", title, Field.Store.YES));
    doc.add(new TextField("metadata", metadata, Field.Store.YES));
    doc.add(new TextField("header", header, Field.Store.YES));
```

```

        doc.add(new StringField("url", url, Field.Store.YES));
        doc.add(new TextField("content", content, Field.Store.YES));
        indexWriter.addDocument(doc);
    }

```

This code block adds the extracted information from the HTML file into a document where the *indexWriter* can then create an index based on this document.

The extraction and indexing of HTML files is done through a loop, where the loop terminates when the expected file is not found. The program searches for HTML files in the data directory folder and expects the HTML files to be named exactly like the data directory folder followed by a number. For example, if the user states the data directory folder is named *data/*, then the first HTML file to parse must be *data/data0.html*. The program iterates through each number. Thus, after extracting and parsing *data/data0.html*, the program attempts to extract and parse *data/data1.html*. The syntax of these HTML file names are based on the HTML files generated by the web crawler in Part A.

Search.java

Class and Function Start

```

@RestController
@CrossOrigin("*")
public class Search
{
    @RequestMapping(value = "/search", method = RequestMethod.GET)
    public @ResponseBody String search(@RequestParam(required = false,
        defaultValue = "") String query) throws IOException, \
        ParseException
    {

```

The *@RestController* and *@CrossOrigin* designates types as controllers and allows sending data back to the origin connection (HTML search engine interface). The *@RequestMapping* allow execution of the program's *search()* function by appending "/search" after the URL of the server the program is running on. The program expects an HTTP GET signal from the client program. The search program also accepts parameters which will become the query for the searching of the index as denoted by *@RequestParam*.

Initialize Code Block

```

Analyzer analyzer = new StandardAnalyzer();
Directory directory = FSDirectory.open(Paths.get("index/"));
DirectoryReader directoryReader = DirectoryReader.open(directory);

```

```

IndexSearcher indexSearcher = new IndexSearcher(directoryReader);
String[] fields = {"title", "metadata", "header", "url", "content"};
Map<String, Float> boosts = new HashMap<>();
boosts.put(fields[0], 1.0f);
boosts.put(fields[1], 0.5f);
boosts.put(fields[2], 0.5f);
boosts.put(fields[3], 0.25f);
boosts.put(fields[4], 0.25f);
MultiFieldQueryParser parser = new MultiFieldQueryParser(fields, analyzer,\
    boosts);

```

This code block initializes the Lucene class variables. Once again, the *StandardAnalyzer* is selected as the default analyzer to use. The directory opens the index directory folder, which is expected to be *index/* in this version of the code. The code also initializes score weights for the features of the index documents. The program thus reads the query and utilizes the analyzer and score weights in order to determine the document scoring and fetches the documents that best fit the query.

Scoring and Fetching

```

Query parsed = parser.parse(query);
ScoreDoc[] score = indexSearcher.search(parsed, 10).scoreDocs;

```

This code segment simply scores the documents based on the query and feature weights and then fetches the top ten documents based on their score.

Lines 53 through 95 of the code block simply extracts the document data from the list of top ten documents and generates snippets for each document. The data is then passed into a *String* variable and returned, thus sending the data back to the origin connection.

engine.html

The purpose of the *engine.html* file is to create an interactive interface to send queries and receive documents. The HTML file displays snippets of the returned documents and also provides a hyperlink to the original web page that was fetched from the web crawler. The HTML file allows a cleaner and simpler method in sending and receiving data as a web search program. Otherwise, the user will have to utilize the console in order to execute searches. The back end of the web search interface is built utilizing Spring and Maven. Spring and Maven aid in setting up a local host server to which the *Search.java* file is run. An external browser, such as *engine.html*, which acts as the front end, can then communicate with *Search.java* and search documents. The *engine.html* file is built using HTML, CSS, Javascript, and AngularJS.

Limitations

The index and search programs do contain limitations. One limitation is the rigidity of the scoring, in which once the boosting of the score has been set, the architecture cannot be changed without reindexing the entirety of the collection. Another limitation is the inability of the search algorithm to establish semantic meaning on the language of the query. For example, the query “white house” cannot connect the semantic meaning of the two words together to indicate the meaning of the actual White House in Washington D.C. In extension, the program cannot derive definitions of the word. It simply fetches strings that match the terms of the query. For example, it cannot distinguish that “money” and “capital” are synonyms. Furthermore, it cannot distinguish whether the word “bank” refers to an establishment where money is stored or a slope next to a body of water. Another limitation of the program is its inability to query snippets of URLs, especially with the incorporation of non-alphanumeric symbols.

Instructions

The program is built using Eclipse. It is recommended to run the indexer on Eclipse due to the hierarchy nature of Java. Since *Index.java* uses Lucene and jsoup, the necessary JAR files where the classes are stored in are necessary. The JAR files are stored in the *class/* directory. Before running *Index.java*, the data collection of the HTML documents must be present in the root directory (directory that contains *bin/*, *class/*, *files/*, *src/*, *mvnw.cmd*, etc.) in its designated data directory folder. The data directory folder and the HTML files within it must share the same name format mentioned above. Due to the size of the collection, I have not included the collection in my upload. Running the *Index.java* file will build the indexes within the index directory folder designated by the user.

To run the search program, the user can simply run *Search.java* and communicate through the console. However, that is not as neat as a web interface. In order to run Maven, your environment variable `JAVA_HOME` must match the location of your Java installation. Once all initial setups are finished, run the local host server by executing the command in the root directory:

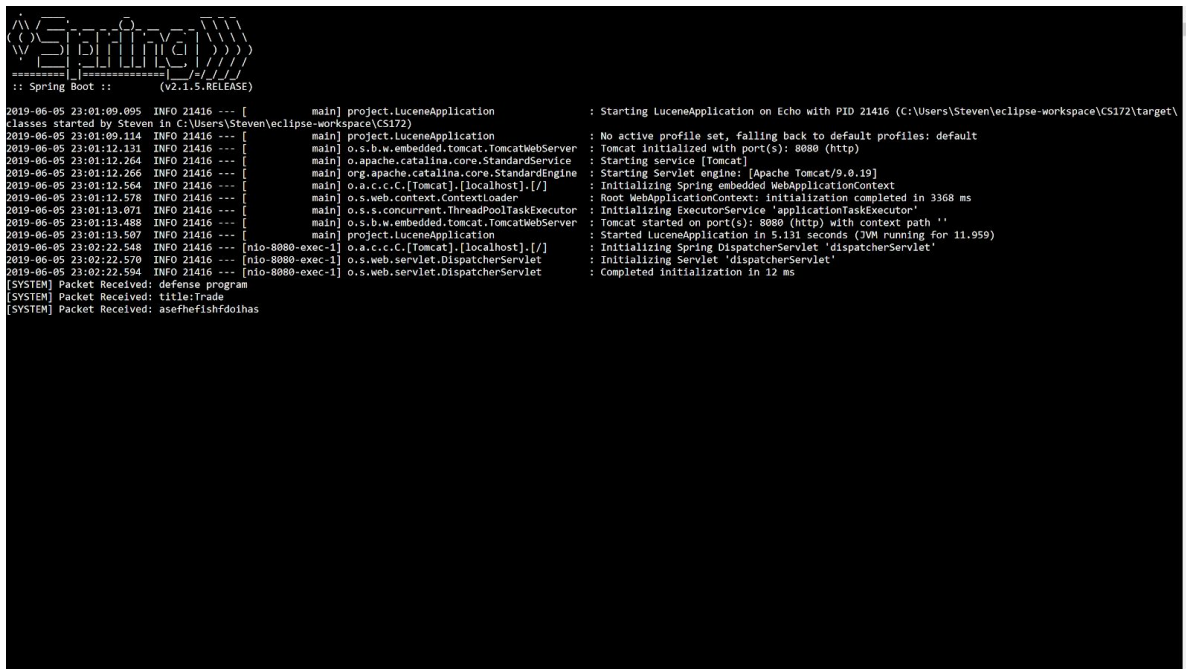
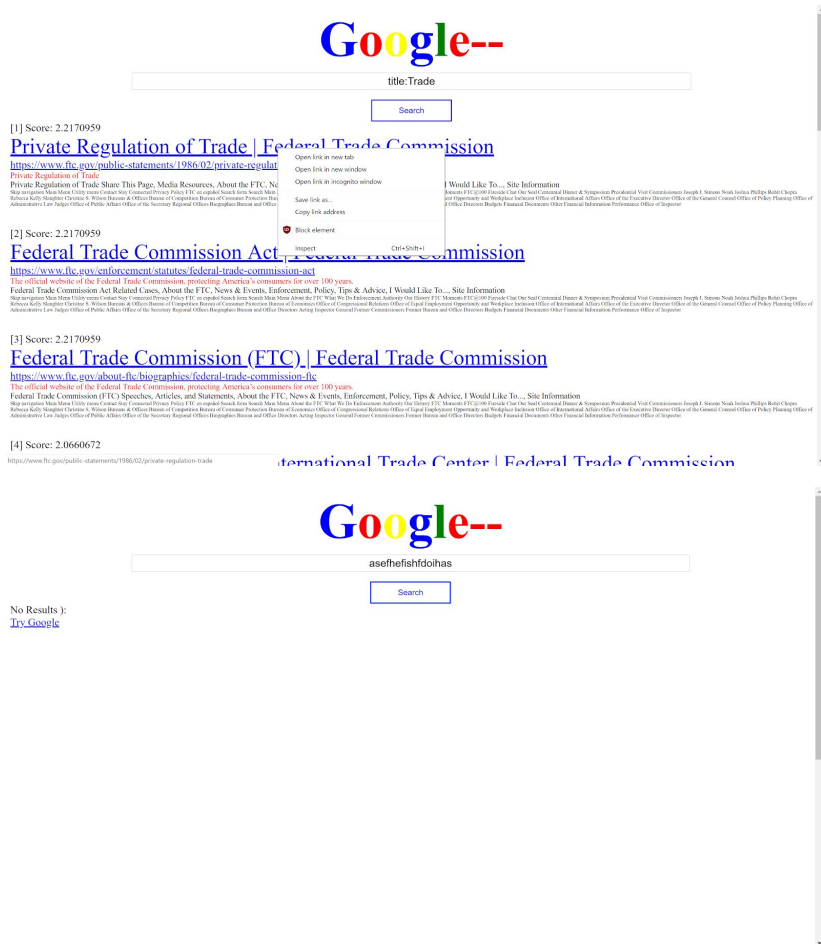
```
mvnw.cmd spring-boot:run
```

Once the server is running, simply open the *engine.html* located in *files/* to open a web browser. The user can then write a query in the search box and press search in order to execute a search through the index. The user can simply write a normal query, or search through a specific document feature as shown in this example:

```
title:Trade
```

Note: The web document index and search program is meant to be compatible with the Web Crawler program: <https://github.com/steventfan/Web-Crawler>

[illegible]



Note: Google retains all rights to the namesake Google. The name Google-- is used to portray this program's inferiority to the real deal.