Project 4 – PRIMARY CHOICE Book Search Engine

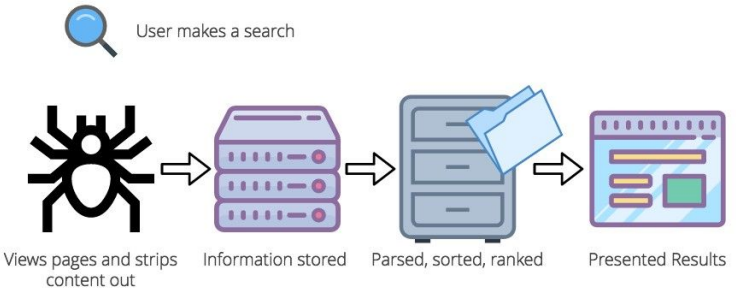
Sebastian Celeita - Student ID 28716405

Algorithm Development for Network Applications
Sorbonne Université 2021

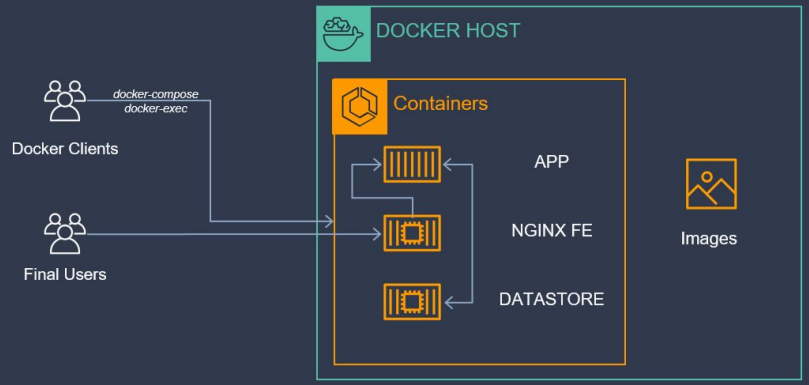
PROBLEM

Before beginning, we will ask ourselves about the complexity of scalable data embed in Wikipedia, Social Networks, the whole Internet, and how is managed by their own providers or popular search engines, in order to handle any query.

This project explores one of many alternatives to face the challenge of an online library with a basic search engine, based on the initiative of Gutenberg Project.



ARCHITECTURE



In short words, our environment will be a stack of 3 containers:

- The **datastore** itself deployed over **Elasticsearch** and exposed to the application container for different functions such as loading the data.

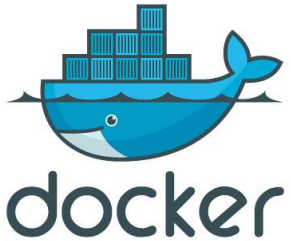
- The **application container** is the middleframe between our UI and the datastore with all the basic methods behind this book search engine. Built in Node.js and Koa framework that allows exposing an HTTP API in order to access the search functionality from the frontend.

- The **frontend container** built as a simple nginx server with .html and .css resources. It will be linked to the application container in order to interact with the functions to search and view the results from the UI.

WHY DOCKER?

Is engine used by many remarkable services such as Spotify and Uber. One of the main advantages of building a containerized application is that the project setup is virtually the same no matter what operating system is being used on the host side.

This means that for further improvements, any developer can work in a separate environment and then integrate each new capability for test and production environment. Continuous integration and deployment with Docker containers is being part of the workflow for any business with digital environments.

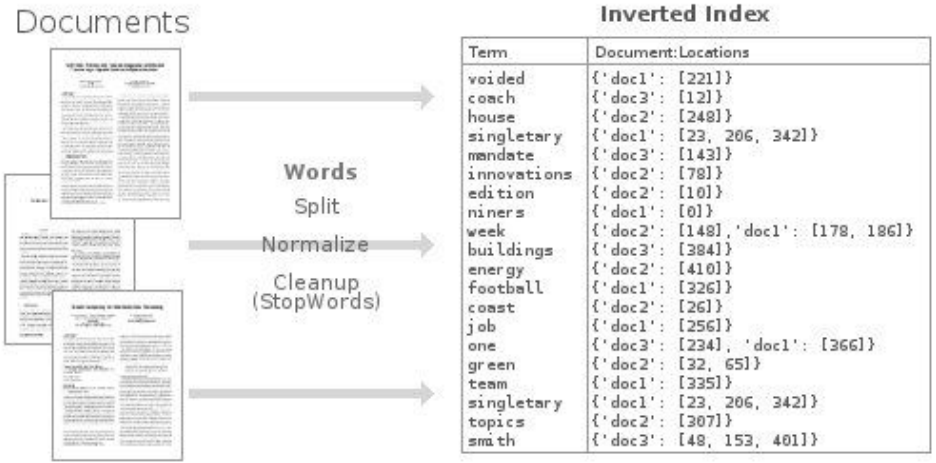


WHY ELASTICSEARCH?

What we want for this project to be:

- Fast and responsive user experience.
- Flexible, to be able to modify how the search is performed.
- Intuitive, what could the user might have been trying to search for?
- Ambitious: search everything in our datastore!

The data structure that fits right into our problem is the inverted index! That is why Elasticsearch enables us to perform some very powerful and customizable full-text searches on our stored data.



APPLICATION CONTAINER

Defined in the docker-compose.yml file the stack has one container for the application: gs-api - The Node.js container for the backend application logic.

```
api: # Node.js App
  container_name: gs-api
  build: .
  ports:
    - "3000:3000" # Expose API port
    - "9229:9229" # Expose Node process debug port (disable in production)
  environment: # Set ENV vars
   - NODE_ENV=local
   - ES_HOST=elasticsearch
   - PORT=3000
  volumes: # Attach local book data directory
    - ./books:/usr/src/app/books
```

FRONTEND AND DATASTORE CONTAINERS

```
frontend: # Nginx Server For Frontend App
  container_name: gs-frontend
  image: nginx
  volumes: # Serve local "public" dir
    - ./public:/usr/share/nginx/html
  ports:
    - "8080:80" # Forward site to localhost:8080

elasticsearch: # Elasticsearch Instance
  container_name: gs-search
  image: docker.elastic.co/elasticsearch/elasticsearch:6.1.1
  volumes: # Persist ES data in seperate "esdata" volume
    - esdata:/usr/share/elasticsearch/data
  environment:
    - bootstrap.memory_lock=true
    - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
    - discovery.type=single-node
  ports: # Expose ElasticSearch ports
    - "9300:9300"
    - "9200:9200"
```

Defined in the docker-compose.yml file the stack has 2 more containers:

- gs-frontend - An Ngnix container for serving the frontend webapp files.

- gs-search - An Elasticsearch container for storing and searching data.

CONNECTION TO DATASTORE

Three main functions are defined:

- checkConnection
- resetIndex:
- putBookMapping

READ FILES AND UPLOAD LIBRARY

In order to read the metadata and content for each book, a function **load_data.js**. is defined taking into account the above observations. This function will have 3 sub-functions:

- **parseBookFile**: Reads and parse the text file
- **insertBookData**: Bulk index the book data in ElasticSearch
- **readAndInsertBooks**: Clear Elastisearch index, parse, and index all files from the books directory calling insertBookData per file.

```
// Find book title, author and language
const title = book.match(/^Title:\s(.+)$/m)[1]
const language = book.match(/^Language:\s(.+)$/m)[1]
const authorMatch = book.match(/^Author:\s(.+)$/m)
const author = (!authorMatch || authorMatch[1].trim() === '') ? 'Unknown Author' : authorMatch[1]

console.log(`Reading Book - ${title} By ${author} written_in ${language}`)

// Find Guttenberg metadata header and footer
const startOfBookMatch = book.match(/^\*{3}\s*START OF (THIS|THE) PROJECT GUTENBERG EBOOK.+\*{3}$/m)
const startOfBookIndex = startOfBookMatch.index + startOfBookMatch[0].length
const endOfBookIndex = book.match(/^\*{3}\s*END OF (THIS|THE) PROJECT GUTENBERG EBOOK.+\*{3}$/m).index
```

```
if (i > 0 && i % 500 === 0) { // Do bulk insert after every 500 paragraphs
  await esConnection.client.bulk({ body: bulkOps })
  bulkOps = []
  console.log(`Indexed Paragraphs ${i - 499} - ${i}`)
}
```

SEARCH 1/2

Elasticsearch uses Lucene's Practical Scoring Function. Lucene takes the Boolean model, TF/IDF, and the vector space model and combines them in a single efficient package

```
score(q,d)  =  (1)
            queryNorm(q)  (2)
          · coord(q,d)    (3)
          · ∑ (           (4)
              tf(t in d)  (5)
            · idf(t)²     (6)
            · t.getBoost() (7)
            · norm(t,d)   (8)
          ) (t in q)      (4)
```

(1) `score(q,d)` is the relevance score of document `d` for query `q`.

(2) `queryNorm(q)` is the *query normalization factor* (new).

(3) `coord(q,d)` is the *coordination factor* (new).

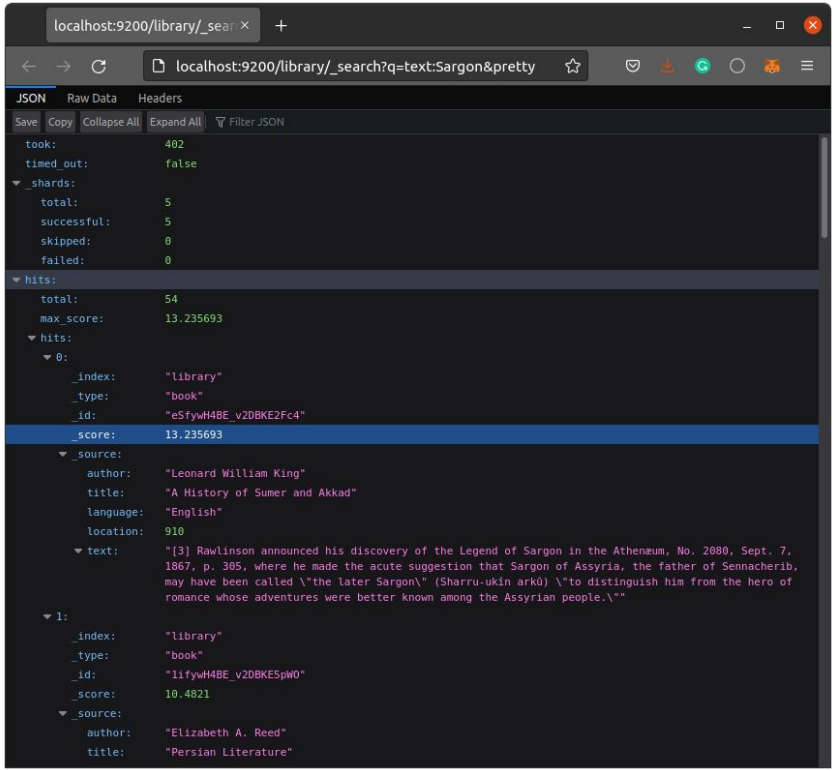
(4) The sum of the weights for each term `t` in the query `q` for document `d`.

(5) `tf(t in d)` is the term frequency for term `t` in document `d`.

(6) `idf(t)` is the inverse document frequency for term `t`.

(7) `t.getBoost()` is the *boost* that has been applied to the query (new).

(8) `norm(t,d)` is the field-length norm, combined with the index-time field-level boost, if any. (new).



SEARCH 2/2

Search querying is done to Elasticsearch from the Node.js application. There is a function called **search.js** where we define a full-text search capability

- from - Allows us to paginate the results.
- operator - search behavior
- fuzziness - Adjusts tolerance for spelling mistakes

search.js

*.js files are supported by IntelliJ IDEA Ultimate

```
const { client, index, type } = require('./connection')

module.exports = {
  /** Query ES index for the provided term */
  queryTerm (term, offset = 0) {
    const body = {
      from: offset,
      query: { match: {
        text: {
          query: term,
          operator: 'and',
          fuzziness: 'auto'
        } } },
      highlight: { fields: { text: {} } }
    }

    return client.search({ index, type, body })
  },

  /** Get the specified range of paragraphs from a book */
  getParagraphs (bookTitle, startLocation, endLocation) {
    const filter = [
      { term: { title: bookTitle } },
      { range: { location: { gte: startLocation, lte: endLocation } } }
    ]

    const body = {
      size: endLocation - startLocation,
      sort: { location: 'asc' },
      query: { bool: { filter } }
    }

    return client.search({ index, type, body })
  }
}
```

ADDITIONAL FEATURES

RegEx Advanced Search

```
/** Query ES index for the provided RegEx term */
queryTerm (term, offset = 0) {
  const body = {
    from: offset,
    query: { regexp: {
      text: {
        query: term,
      } } },
    highlight: { fields: { text: {} } }
  }

  return client.search({ index, type, body })
},
```

More like this or recommendation capabilities:

An even more interesting alternative could be using Terms Vector API inside Elasticsearch, it would be possible to present users with a selection of topical keywords found in a document's text, allowing them to select words of interest to drill down on, rather than using the more "black-box" approach of matching used by more_like_this.

DEMO SCREENSHOTS!

Other possible features that can be handled by this MVP in further stages are:

- User account to make goodreads look-a-like service, where they can not only download, but make a review, favorite list and bucket list to read.
- Improve recommendation with collaborative recommendation algorithm between different users.
- Continue reading capabilities and library history of searches.