

Project 4 – PRIMARY CHOICE

Book Search Engine

In the following report, we will explore a containerized solution for a simple book search engine running a separate datastore in Elasticsearch. 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 the queries (even the misspelled or incomplete queries) and find the most relevant information in a short time.

Many challenges can be found, even in this small project with lesser complexity than the referenced web applications mentioned above. Most standard relational databases frameworks offer basic text searching capabilities due to limitations on their existing query and index structures, that is why this project will implement a high-quality text search datastore separately, open-sourced, and optimized to perform a flexible and fast text search.

For the containerized environment, we will rely on Docker, an 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, this project will be aligned to a baseline of best practices as well.

Finally, we'll be using data from Project Gutenberg¹ - an online project aiming to provide free, digital copies of books within the public domain. The library datastore will be populated with more than 150 books from 3 different languages (english, spanish, and french).

¹ Project Gutenberg URL: <https://www.gutenberg.org/>

1. ARCHITECTURE OF THE PROJECT

Below is the high-level architecture diagram with the main components that will be in the scope of this project.

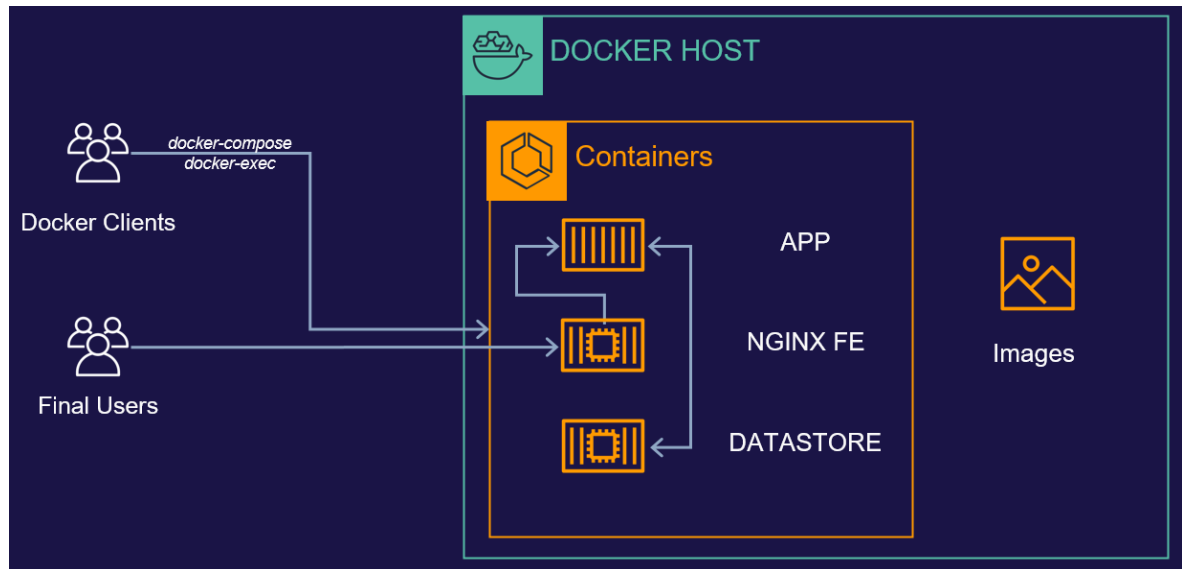


Fig. 1 - High level architecture diagram

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 (upload books with a defined data schema) and searching. Both ports TCP 9200 and 9300 will be used and forwarded to the localhost.
- 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 container will have access to a pre-downloaded directory (included in the repository of this project) with all books in order to bulk them to the Elasticsearch instance as a first step.

- 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. The port TCP 8080 will be forwarded to port TCP 80 to the localhost and will be deployed under Vue.js framework.

2. ELASTICSEARCH²

In order to provide a book search engine, we want our features to be more than basic string matching functionality. Fast, flexible, and intuitive capabilities can be met with an open-source in-memory datastore used by many popular websites since it is a heavily requested feature in modern applications.

Why? because, Elasticsearch is able to provide fast and flexible full-text search through the use of inverted indices, as a server that can process JSON requests and give you back JSON data. An Elasticsearch index is a collection of documents that are related to each other. Elasticsearch stores data as JSON documents. Each document correlates a set of keys (names of fields or properties) with their corresponding values (strings, numbers, Booleans, dates, arrays of values, geolocations, or other types of data).

Inverted indexes are data structures that work in a substantially different manner from a common self-balancing tree data structure or Binary Search Tree. An inverted index lists every unique word that appears in any document and identifies all of the documents each word occurs in. During the indexing process, Elasticsearch stores documents and builds an inverted index to make the document data searchable in near real-time.

That is why Elasticsearch enables us to perform some very powerful and customizable full-text searches on our stored data.

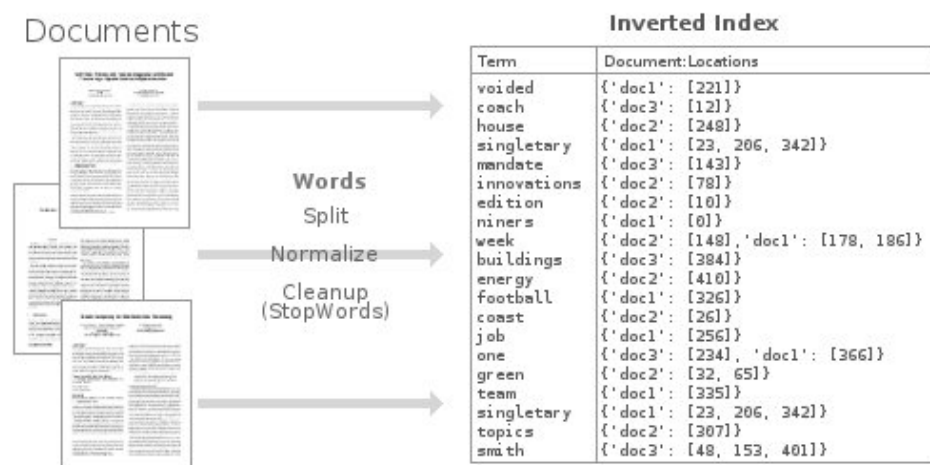


Fig. 2 - Inverted Indexes³

² What is Elasticsearch? <https://www.elastic.co/what-is/elasticsearch>

³ Inverted Indexes <http://mocalas.github.io/2015/11/18/Python-Inverted-Index-for-dummies/>

3. 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
```

Fig. 3 - API Docker container definitions

No need to install Node in our localhost, however, is not an official prebuilt image, so we need to build it ourselves by defining a Dockerfile in the root directory of the project:

```
# Use Node v8.9.0 LTS
FROM node:carbon

# Setup app working directory
WORKDIR /usr/src/app

# Copy package.json and package-lock.json
COPY package*.json ./

# Install app dependencies
RUN npm install

# Copy sourcecode
COPY . .

# Start app
CMD [ "npm", "start" ]
```

Fig. 4 - API Docker image definition

If Elasticsearch already has an HTTP interface why do we need an API in the middle? From a security perspective, exposing this interface directly to the frontend would be considered a vulnerability and a significant risk to our datastore. The API exposes administrative functionality (adding and deleting documents only by docker clients in our case), and should ideally not ever be exposed publicly. Instead, the Node.js API will simply receive requests from the client, and make the appropriate query (within our private local network) to Elasticsearch.

4. FRONT-END AND DATASTORE CONTAINER

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

- gs-frontend - An Nginx container for serving the frontend webapp files.
- gs-search - An Elasticsearch container for storing and searching data.

```
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 separate "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"
```

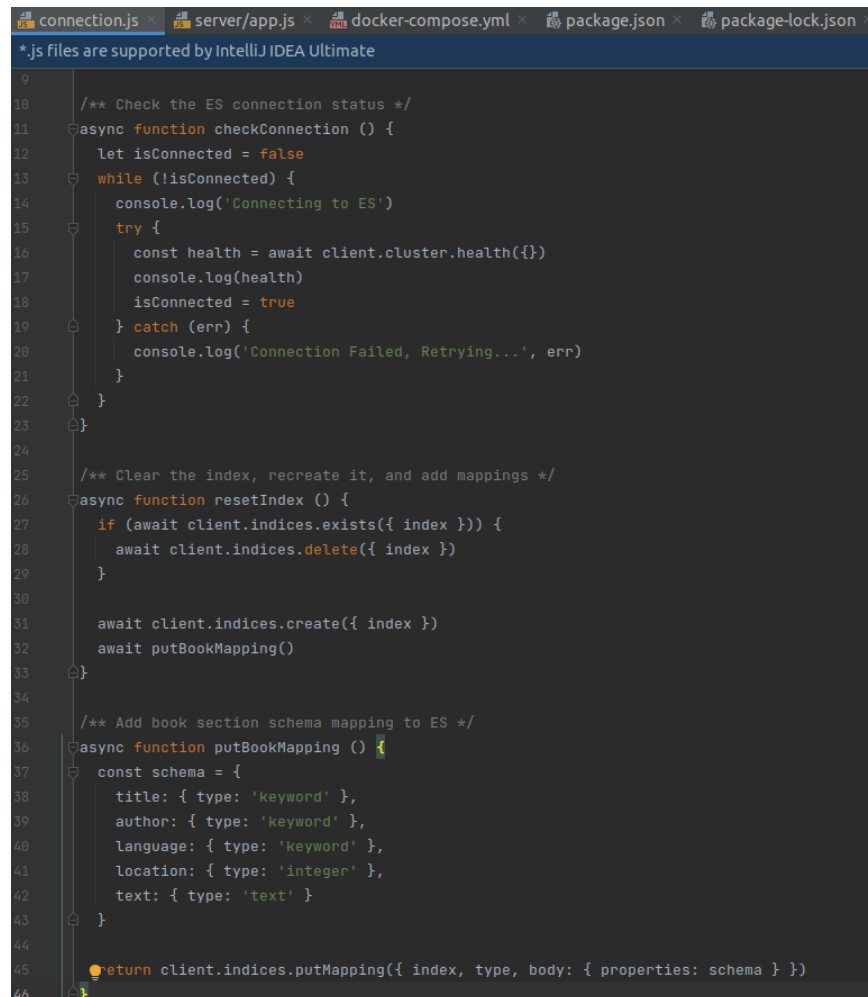
Fig. 5 - Frontend and Datastore Docker container definitions

This defines the rest of the application stack - no need to install Elasticsearch or Nginx on the local system. Each container is forwarding ports to the host system (localhost), in order for us to access (as docker clients and final users) and debug the Node API, Elasticsearch instance, and fronted web app from our host machine.

5. CONNECTION TO DATASTORE

Firstly, the application must be able to connect to the local Elasticsearch instance. For that, a connection function is defined where, not only the instance itself is defined, but the index for this project as well. Three main functions are defined so far:

- checkConnection
- resetIndex:
- putBookMapping



```
9
10 /** Check the ES connection status */
11 async function checkConnection () {
12   let isConnected = false
13   while (!isConnected) {
14     console.log('Connecting to ES')
15     try {
16       const health = await client.cluster.health({})
17       console.log(health)
18       isConnected = true
19     } catch (err) {
20       console.log('Connection Failed, Retrying...', err)
21     }
22   }
23 }
24
25 /** Clear the index, recreate it, and add mappings */
26 async function resetIndex () {
27   if (await client.indices.exists({ index })) {
28     await client.indices.delete({ index })
29   }
30
31   await client.indices.create({ index })
32   await putBookMapping()
33 }
34
35 /** Add book section schema mapping to ES */
36 async function putBookMapping () {
37   const schema = {
38     title: { type: 'keyword' },
39     author: { type: 'keyword' },
40     language: { type: 'keyword' },
41     location: { type: 'integer' },
42     text: { type: 'text' }
43   }
44
45   return client.indices.putMapping({ index, type, body: { properties: schema } })
46 }
```

Fig. 6 - Connection and functions between API and Datastore

With that function definition, docker clients will be able to run scripts to load the desired data into Elasticsearch, for instance, the books of our library. We can observe that the function defines the data schema, where the definitions of the mapping of each book are

set, specifying each field and datatype for the stored documents, giving us more control over how the data is handled within the application.

6. UPLOAD LIBRARY

Inside the repository, the directory **/books** contain more than 150 books in english, spanish, and french for the purpose of this project. All books are from Gutemberg Project.

Every book has the same structure in terms of standard information that we can use for the mapping and searching functions. For instance:

- Each .txt file starts with an open-access license, followed by some lines identifying the book title, author, release dates, language, and character encoding.
- After the following lines: ***** START OF THIS PROJECT GUTENBERG EBOOK <TITLE> *****, the book content actually starts.
- The content ends with **** END OF THIS PROJECT GUTENBERG EBOOK <TITLE> *****

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 the text file and finds title, author, and language. After finding the start and end lines from the Gutemberg metadata splits the content of the book into an array of paragraphs and cleans the text. The parsing process is done using regular expressions:

```
// 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(/^\\s*START OF (THIS|THE) PROJECT GUTENBERG EBOOK.+\\s*$/m)
const startOfBookIndex = startOfBookMatch.index + startOfBookMatch[0].length
const endOfBookIndex = book.match(/^\\s*END OF (THIS|THE) PROJECT GUTENBERG EBOOK.+\\s*$/m).index
```

Fig. 7 - Parsing process with regex

As a return value, the function forms an object containing the book's metadata as defined in the data schema.

- insertBookData: Bulk index the book data in ElasticSearch, the insertion of the paragraphs uses a bulk operation, which is much faster than indexing each paragraph individually. It uses batches of 500 paragraphs per operation, which may help in containerized environments with limited resources:

```
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}`)
}
```

Fig. 8 - The bulk process with batch formation of 500 paragraphs

- readAndInsertBooks: Clear Elastisearch index, parse, and index all files from the books directory calling insertBookData per book.

From the docker client, we can run the script to upload all the data to our instance once the docker containers are working:

```
sudo docker exec gs-api "node" "server/load_data.js"
```

```
(base) ion@ion-Nitro:~/Documents/SCELEITA_BOOK-SEARCH-ENGINE$ sudo docker exec gs-api "node" "server/load_data.js"
Connecting to ES
{ cluster_name: 'docker-cluster',
  status: 'yellow',
  timed_out: false,
  number_of_nodes: 1,
  number_of_data_nodes: 1,
  active_primary_shards: 7,
  active_shards: 7,
  relocating_shards: 0,
  initializing_shards: 0,
  unassigned_shards: 7,
  delayed_unassigned_shards: 0,
  number_of_pending_tasks: 0,
  number_of_in_flight_fetch: 0,
  task_max_waiting_in_queue_millis: 0,
  active_shards_percent_as_number: 50 }
Found 200 Files
Reading File - 10.txt
Reading Book - The King James Bible By Unknown Author written_in English
Parsed 24609 Paragraphs

Indexed Paragraphs 1 - 500
Indexed Paragraphs 501 - 1000
Indexed Paragraphs 1001 - 1500
Indexed Paragraphs 1501 - 2000
Indexed Paragraphs 2001 - 2500
```

Fig. 9 - Load data into Elasticsearch

7. SEARCH

Now that Elasticsearch has been populated with two hundred books we can try out some search queries, for instance: http://localhost:9200/library/_search?q=text:Sargon&pretty

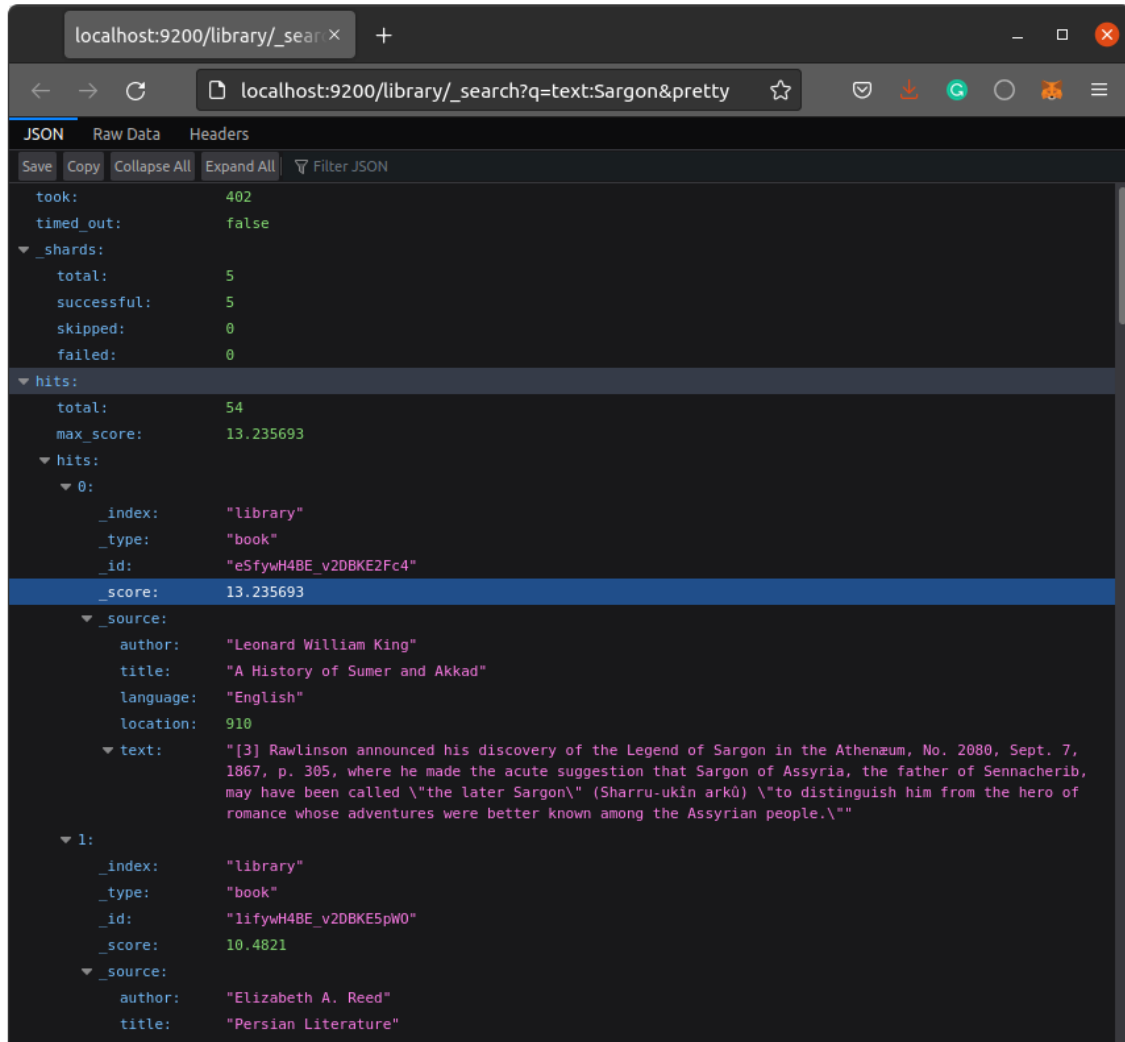


Fig. 10 - JSON search directly to the Datastore instance via Elasticsearch HTTP interface

We will use relevance scoring from Elasticsearch to show the query results. How those scores are determined? Before Elasticsearch starts scoring documents, the candidate documents are reduced by applying a boolean test - does the document match the query? Once the results that match are retrieved, the score they receive will determine how they are rank-ordered for relevancy.

The match on that boolean test is defined by the search query definition, however, just getting a match to one or more terms in a document field does not equate with relevance. In our scenario, Elasticsearch uses Lucene's Practical Scoring Function⁴. Lucene takes the Boolean model, TF/IDF, and the vector space model and combines them in a single efficient package that collects matching documents and scores them as it goes.

$$\text{score}(q,d) = \text{queryNorm}(q) \cdot \text{coord}(q,d) \cdot \sum (\text{tf}(t \text{ in } d) \cdot \text{idf}(t)^2 \cdot t.\text{getBoost}() \cdot \text{norm}(t,d)) (t \text{ in } q)$$

- 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).

Fig. 11 - Elasticsearch search scoring with Lucene's Practical Scoring Function

Score, tf and idf were covered in the course, however, some of the components are slightly new and could be complex to include them in the report. We suggest visiting the referenced link⁵ and try to understand each of them, for instance: the query normalization factor (`queryNorm`) is an attempt to normalize a query so that the results from one query may be compared with the results of another.

$$\text{queryNorm} = 1 / \sqrt{\text{sumOfSquaredWeights}}$$

- 1 The `sumOfSquaredWeights` is calculated by adding together the IDF of each term in the query, squared.

Fig. 12 - Query Normalization Factor within Lucene's Practical Scoring Function

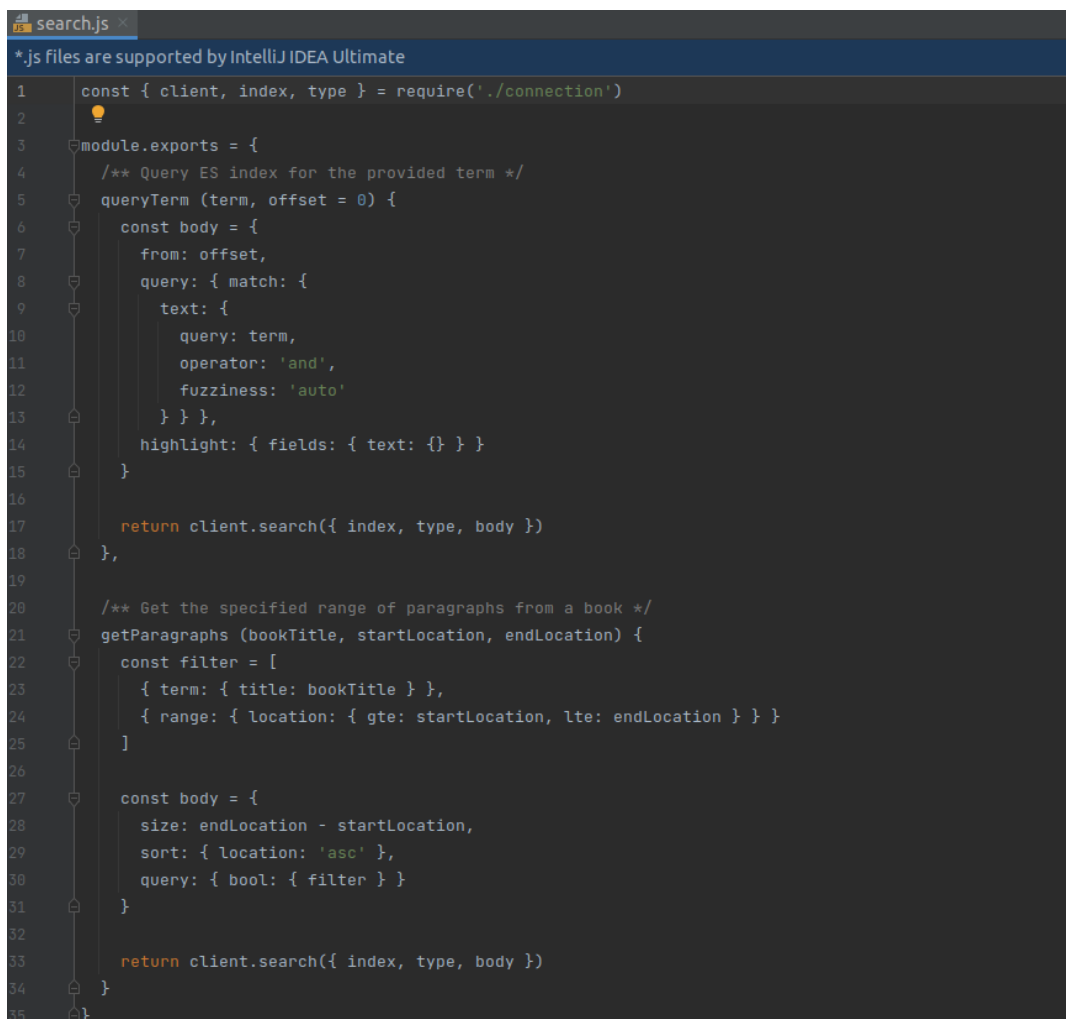
⁴ Lucene's Practical Scoring Function

<https://www.elastic.co/guide/en/elasticsearch/guide/current/practical-scoring-function.html>

⁵ Scoring Elasticsearch How it Works?

<https://www.compose.com/articles/how-scoring-works-in-elasticsearch/>

As mentioned at the beginning, for security reasons we must do the querying to Elasticsearch from the Node.js application. There is a function called search.js where we define a full-text search capability:



```
1  const { client, index, type } = require('./connection')
2
3  module.exports = {
4    /** Query ES index for the provided term */
5    queryTerm (term, offset = 0) {
6      const body = {
7        from: offset,
8        query: { match: {
9          text: {
10            query: term,
11            operator: 'and',
12            fuzziness: 'auto'
13          } } },
14        highlight: { fields: { text: {} } }
15      }
16
17      return client.search({ index, type, body })
18    },
19
20    /** Get the specified range of paragraphs from a book */
21    getParagraphs (bookTitle, startLocation, endLocation) {
22      const filter = [
23        { term: { title: bookTitle } },
24        { range: { location: { gte: startLocation, lte: endLocation } } }
25      ]
26
27      const body = {
28        size: endLocation - startLocation,
29        sort: { location: 'asc' },
30        query: { bool: { filter } }
31      }
32
33      return client.search({ index, type, body })
34    }
35  }
```

Fig. 12 - Text Search API Function

Our search module defines a simple search function, which will perform a match query using the input term. Some fields are explained below⁶:

- from - Allows us to paginate the results.
- operator - search behavior; in this case, using the "and" operator prioritize results that contain all of the tokens (words) in the query.
- fuzziness - Adjusts tolerance for spelling mistakes, auto defaults to fuzziness: 2. A higher fuzziness will allow for more corrections in result hits.

⁶ Full text Queries with Elasticsearch

<https://www.elastic.co/guide/en/elasticsearch/reference/current/full-text-queries.html>

Finally, the API defined in **server/app.js** will let us HTTP access to the search functionality from the frontend application.

Making a regex advanced search or a recommendation (more like this) function will have the same logic, changing the search field parameters to allow the user to:

Advanced RegEx Search⁷: returns documents that contain terms matching a regular expression given by the user, for example:

```
/** 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 })
},
```

Fig. 12 - Regular Expression Advanced Search API Function

More like this (recommendation)⁸: finds documents that are "like" a given set of documents. In order to do so, the function selects a set of representative terms of these input documents, forms a query using these terms, executes the query, and returns the results. We can control the input documents, how the terms should be selected and how the query is formed.

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`.

⁷ Regex query in Elasticsearch

<https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-regexp-query.html>

⁸ More like this in Elasticsearch

<https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-mlt-query.html>

8. DEMO

Please find in the repository the demo video, please find below a couple of screenshots of the frontend working for a basic text search over a library with more than 150 books in 3 different languages:

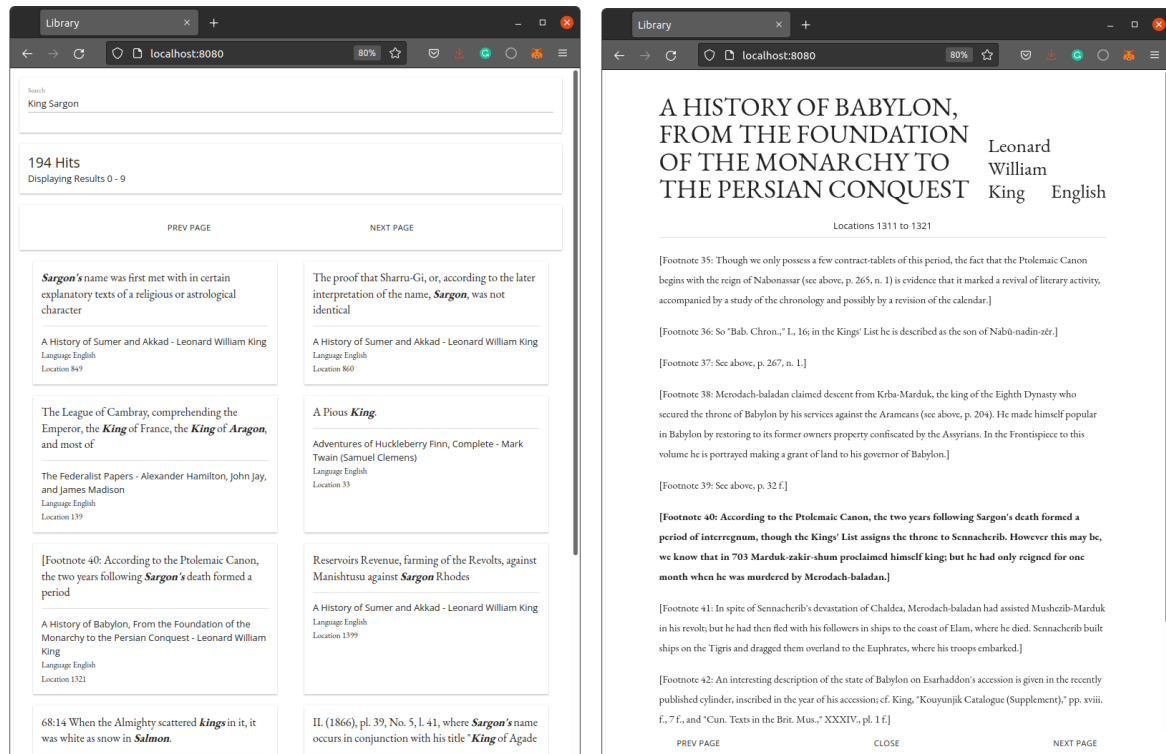


Fig. 13 - Frontend with results of a text search from the demo

9. RESULTS AND FURTHER IMPROVEMENTS

The main constraints and disadvantages are the limited local resources for the containerized environment. Elasticsearch is computationally demanding and giving almost real-time search results could turn into a memory footprint that not every business is ready to take.

However, this project could be deployed in any cloud (private or public) and will be ready for production in a matter of minutes, improving the CI/CD and integration with a team of developers geographically distant. We can even think about using machine learning

solutions such as Open Search from AWS⁹, the referenced link is an interesting showcase of a movie search engine that mimics our project but with a movie dataset. The integration of different cloud-native solutions minimizes the risk of resource limitation.

Form the other hand, we have seen text search capabilities as one of the most important features in many modern applications, and with difficult implementation as well. We highlighted Elasticsearch as the ideal option for adding fast and customizable text search to this project application. Many other capabilities and nice-to-have features could be added to this project, is indeed a baseline hands-on exercise to understand the problem, explore modern solutions and integrate them for a minimum viable product (MVP)

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.

⁹ AWS Open Search hands-on:
<https://docs.aws.amazon.com/opensearch-service/latest/developerguide/search-example.html>