**Implementation of Full text Elastic search**

Full text based Elastic search is implemented for the search function in this REST API web application to make the application return relevant movie results based on the frequency of access and the number of words that match the search. Full-text search queries and performs lexicographical searches against movie data. It includes single or multiple words or phrases and returns document that matches user search object.

Configuration of elastic search is very crucial as it runs on port 9200 and it takes several steps to process the request object into an array list in turn pass the object as a key value mapping response object to localhost:3000 as the application runs locally, it is important to configure ES to listen to both port and get required response object passed back to 9200 to request object.

Importing type of elastic search includes query, it aggregates the search query and a variable to hold the search response object after assigning index values to each movie in the TMDB movie database.

Configuration setting :

**import** {Client as ElasticClient} from "elasticsearch";

**var** elasticsearch = require('elasticsearch');

**var** elasticData;

**const** request = require('request-promise');

**var** elasticSearchConfig = functions.config().elasticsearch;

**var** path = require('path');

**var** client = **new** elasticsearch.Client({

hosts: ['http://localhost:9200/', 'http://localhost:3000/']});

**import** type {

ElasticQuery,

aggsQuery,

searchResp,

searchIndex,

searchSubTrees,

searchAllSorted

} from "./types";

let searchParam={

index: 'search',

type: 'TMDBmovies',

body: {

query: {

term: {"Moviename": $(movieTitle)} }, }

};

*// Class meant to provide internal endpoints able to query Elasticsearch*

**class** Search {

constructor(movieArrayList) {

**this**.movieArrayList = movieArrayList;

**this**.elasticSearch = options.elasticSearch; }

Once the host,port,path and schema is identified in the database connection pulled from MongoDB client, The movie data is put into buildRequest by creating a http request for elastic search to get user input. It returns a promise once the query is built and passed as an object to the search.

buildRequest(index,query) {

*// Generate a http request to Elasticsearch*

**return** {

uri: `${Search.requestURI(**this**.elasticSearch)}/${index}/\_search`,

headers: { "Content-Type": "application/json" },

body: query

};

}

}

Promise<ElasticSearchResp> => elastic.search({index, body, ...opts})

The below method takes searchurl which is our localhost:3000 and takes the input username and password of the login to be able to access data for the sort.

method: elasticSearchMethod,

url: elasticSearchUrl,

auth:{

username: elasticSearchConfig.username,

password:elasticSearchConfig.password,

},

body: postData,

json: **true**

};

The terms are aggregated for the search and is assigned index values for each term before passing to input sort module.

Elastic search uses Lucene scoring algorithm that computes below functional variables to identify the most relevant results of the search

* a relevant score of documents
* query normalization factor
* query coordination factor
* term frequency of term given in the search
* inverse document frequency for the search term
* the boost applied to the query
* the field-length given for the search

Once the query is built it is passed back to bulkinputsort.js where the bulk input of movie data is sorted, indexed and scored.

Bulkinputsort is a helper class function that helps to process search results after the configuration of elastic search.

optional\_params: ['bulk\_limit', 'bulk\_timeout', 'index\_prefix', 'data\_type', 'index\_value']

The above javascript takes the params along with the index assigned in configES. Bulk input assigns input values, input limit along with the timeout error for each load of input data. Prefix is assigned to each input value as index and data\_type is assigned which here in the movie search is mainly of type “String” passed as arraylist to mapping function.

exports.indexPostsToElastic = functions.database.ref('/posts/{post\_id}')

.onWrite(event => {

let postData = event.data.val();

let post\_id = event.params.post\_id;

console.log('Indexing post:', postData);

let elasticSearchConfig = functions.config().elasticsearch;

let elasticSearchUrl = elasticSearchConfig.url + 'posts/post/' + post\_id;

let elasticSearchMethod = postData ? 'POST' : 'DELETE';

OutputElasticSearch.prototype.getPath = **function**() {

**return** **this**.index\_value ? '/' + **this**.index\_value + '/' + **this**.data\_type + '/' : elastic\_search\_helper.computePath(**this**.index\_prefix, **this**.data\_type);

};

The above code snippet posts the data back to the url provided for further sorting of relevant search data after indexing. The reference index is then passed to datamapscoreindex.js

Definition of variables is as below: source and score to be assigned to each movie search

**var** ID\_PROPERTY = '\_id',

SOURCE\_PROPERTY = '\_source',

SCORE\_PROPERTY = '\_score';

A search service which searches through domain objects given in the search input using ElasticSearch. A constructor is defined in order to set the http URL to localhost:3000 and Root which points to elastic search configuration setting.

Elasticsearchprovider module takes search input for domain objects using elasticsearch as a search provider. Each searchTerm holds the search input object i.e request object and it holds

[maxResults] the max number of results to return. And finally returns a {Promise} promise for a modelResults object. Number of hits:[] defines the frequency of the search terms that are constantly searched.

The function eliminates and cleans the search terms by removing extraneous. Fuzzy logic is also added in order to remove the quotes and evaluate the search input. Finally it parses and returns a resultant object to localhost:3000

ElasticSearchProvider.prototype.fuzzyMatchUnquotedTerms = **function** (query) {

**var** matchUnquotedSpaces = '\\s+(?=([^"]\*"[^"]\*")\*[^"]\*$)',

matcher = **new** RegExp(matchUnquotedSpaces, 'g');

**return** query

.replace(matcher, '~ ')

.replace(/$/, '~')

.replace(/"~+/, '"');

};

We finally have a search.js router file which consists all objects passed from the above elasticsearch processed domain values. We finally have configuration setup, index value assigned to search results, sorted based on the data score map values assigned for the cleaned-up search input.

We read the index value assigned and sort if the datamapindexscore.js reorganized the index values after assignment.

**var** searchIndex = (**function**(){

**var** b = {};

b.itemEachRecursive = **function** r(nodeArray, callback){

**var** len = nodeArray.length;

**var** i;

**for**(i = 0; i < len; i++){

**var** n = nodeArray[i];

callback(n);

**if**('children' **in** n){

r(n.children, callback);

}

}

};

We identify the search results and sort the list again based on the relevance of the input

b.searchAllSorted = **function**(query, callback){

query = query.toLowerCase();

**var** queryLen = query.length;

b.searchAll(query, **function**(rs){

callback(rs.sort(**function**(a, b){

**var** x = 0, y = 0;

**function** rate(n){

*//*

*// Level 0: Nothing special*

*// Level 1: Starts with*

*// Level 2: Exact match*

*//*

**var** t = n.title.toLowerCase();

**return** t == query ? 2 : (t.substr(0, queryLen) == query ? 1 : 0); }

x = rate(a);

y = rate(b);

**return** y - x;

})); }); };

Finally after the sort, all results sorted and matches the user input and outputs the resultant object with the content(movie name) and description which retrieves values from datamapscoreindex.js where the movies are sorted and indexed. If the values match the movie name the content is posted to the user.

**if**(jsMatch.test(v.url)){

setDefault({'description': searchIndex(v.title) + "<dim> (only match) - $(movieTitle)</dim>"});

}

**else** {

setDefault({'description': searchIndex(v.title) + "<dim> (only match) - </dim><url>" + searchIndex(v.url) + "</url>" });

}