MusicSurf

Rajdeep Mukherjee

NIIT University

Idea

MusicSurf is an offline music search application. You provide give the directory of your personal music collection to the application for indexing and once the app is run you can use it for the purposes of searching within the music collection using natural language queries.

Features

- Natural Language Querying
- Filters
  - singers
  - genre
  - year range
- export search results as playlist

Architecture and its choice

The app is built using a server-client architecture. The choice for this architecture was made keeping in mind the time for planning and implementation of the project and the familiarity of the team with the server-client model. All members of the team were highly familiar with all aspects of this model having built larger scale applications using it.

Contribution

On the drawing board

The project required an information retrieval system that would index all metadata of the songs and effectively and efficiently be able to return results for natural language queries. We for therefore in search of a search engine system that could do the same for us.

ElasticSearch (ES)

ElasticSearch is one of the most popular search engines out there fully open-source and ready to be integrated with software. It is also built on a server-client architecture and uses. All communication with the elastic-search server is made with RESTful API calls. It is amazingly easy to use and integrate with software of all types.

Features of ES relevant to project

- API is RESTful hence client and server sides are not tightly coupled and hence there is less rigidity

between client and server. All that the server needs are certain requests (as in HTTP requests like GET, POST and so on) in the form of API calls and the server sends a response based on these API calls.
- Extensive documentation- For the success of any application software, the developers must have an extensive documentation so that he can easiy refer to solutions to problems that he/she cannot otherwise solve.
- Default capability of handling natural language queries.

Design

My primary contribution to the project was integrating elasticsearch with the application.

Choice of Programming language: python

Python was chosen as the primary choice of programming langauge for the whole application development because of its familiarity with the whole team.

Choices for implementation styles:

- Using python elasticSearch client module
- Using http request modules like requests and httplib

I chose to use requests module for api calls. This would make the whole system less rigid as it allowed me to make suitable modifications in queries as per my need and also allowed me understand the request-response model of elasticsearch inside out.

API Constructs: The basic architecture involves storing the base structure of all api constructs in json configuration files and to read and modify the files according to our need. This is a cleaner and more robust approach of design as it avoids hard coding all the constructs in the main file itself.

Results: The results are returned after appropirate retrieval and used directly for rendering to the client template. There is no intermmediate persistence layer storing the results. This also goes to save memory as we would want the basic file structure to be polluted by unnecessary storage of files.

Configuring ElasticSearch

After installation and starting of the elastic search server, we need to configure elastic search for efficient and effective search requirements. This can be done by setting 2 basic parameters during the creation of an index for indexing all documents of the user's music collection.

```
"settings":{
        "index":{
            "number_of_shards":5,
            "number_of_replicas":1
        }
```

shards and replicas are essential in the development of an effective information retrieval. Shards defines the number of partitions the index can be divided into and the number of replicas defines the number of copies of the index to keep incase of data recovery on data loss from one or more of the indexes.

Mappings:

```
"mappings":{
        "music":{
            "_all": {"enabled":false},
            "properties":{
                "title":{"type":"text"},
                "artist":{"type":"text"},
                "album":{"type":"text"},
                "year":{
                    "type":"date",
                    "format":"yyyy"
                },
                "lyrics":"text",
                "path":"text"
            }
        }
    }
```

During index creation itself mappings are defined that tells the system which attribute of documents can carry which type(as in data type) of data. This informtion is crucial to the efficient information retrieval from the indexes.

Indexing

The very first step in the process of any information retrieval system is the indexing of documents. In our case the documents were simply ID3 tags of individual songs. Each document was indexed in the format required by the document posting API of elasticSearch.

Steps for indexing of document:

- Extract id3 tag information of all songs in the offline music collection into a json file.
- read the json file to construct index api request.
- construct index query and send the request as an API call to the elasticsearch server.

Thats all, Indexing of all documents is handled easily using simply API calls.

Search

After indexing of all documents search on all these documents can be performed using a search api. The maximum search results returned are 10. In case the number of documents is more than 10 at one time 10 documents will be returned by default. This can however be changed during the query

step by defining parameters 'from' and 'size'. The from parameter defines the starting point of the document generation or retrieval and size defines the number of documents to be returned.

The query is structured following the basic search structure according to the documentation of elasticsearch.

```
"query":
{
    "bool":
    {
    "should":[
        {
        "match":{
            "title":{
                "query":"queryString",
                "boost":1
            }
        }
        },
        {
        "match":{
            "artist":{
                "query":"queryString",
                "boost":1
            }
        }
        },
        {
            "match":{
                "album":{
                    "query":"queryString",
                    "boost":1
                }
            }
        },
        {
            "match":{
                "year":{
                    "query":"queryString",
                    "boost":1
                }
            }
        },
        {
            "match":{
                "lyrics":{
                    "query":"queryString",
                    "boost":1
                }
            }
        }
        ]
    }

    },
```

```
    "sort" : [
        {
            "_score": {
                "order": "desc"
            }
        }
    ]
```

The **bool** parameter creates a filter with the following attributes for filter application: artist, album, year and lyrics. Note that these attributes are very much parts of the documents themselves. The **should** parameter makes sure that there is an **OR** condition being applied for the various attributes of the filter.

Prioritizing Attributes (The boost parameter)

The boost parameter can be used effectively for prioritizing attributes. When an attribute is proritized for searching the boost property is incremented by one for that property. The direct impact of this increment is that during the search process, the corresponding attribute is given higher priority and search is first preformed in that attribute and the score is decided accordingly

Conclusion

Check out my contributions to MusicSurf here

Here is a snapshot for quick reference