<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table Of Contents</td>
<td>1</td>
</tr>
<tr>
<td>Quick-Start</td>
<td>7</td>
</tr>
<tr>
<td>Overview</td>
<td>8</td>
</tr>
<tr>
<td>What Is FlexSearch?</td>
<td>8</td>
</tr>
<tr>
<td>Basic Concepts</td>
<td>8</td>
</tr>
<tr>
<td>How FlexSearch Sees The World?</td>
<td>9</td>
</tr>
<tr>
<td>Text Analysis</td>
<td>10</td>
</tr>
<tr>
<td>Some More Jargon</td>
<td>11</td>
</tr>
<tr>
<td>Getting Started</td>
<td>12</td>
</tr>
<tr>
<td>Install FlexSearch</td>
<td>12</td>
</tr>
<tr>
<td>5 Minute Install</td>
<td>13</td>
</tr>
<tr>
<td>Creating An Index</td>
<td>13</td>
</tr>
<tr>
<td>Retrieving Index Information</td>
<td>15</td>
</tr>
<tr>
<td>Indexing Data</td>
<td>18</td>
</tr>
<tr>
<td>Retrieving Documents By Id</td>
<td>18</td>
</tr>
<tr>
<td>Modifying Data</td>
<td>19</td>
</tr>
<tr>
<td>Setting Up Demo Index</td>
<td>20</td>
</tr>
<tr>
<td>Searching Data</td>
<td>20</td>
</tr>
<tr>
<td>FAQs</td>
<td>25</td>
</tr>
<tr>
<td>Server</td>
<td>26</td>
</tr>
<tr>
<td>Configuration</td>
<td>27</td>
</tr>
<tr>
<td>Directory Structure</td>
<td>27</td>
</tr>
<tr>
<td>Configure Global Settings</td>
<td>28</td>
</tr>
<tr>
<td>Index</td>
<td>29</td>
</tr>
<tr>
<td>Indexing Overview</td>
<td>30</td>
</tr>
<tr>
<td>How FlexSearch Sees The World?</td>
<td>30</td>
</tr>
<tr>
<td>Field Analysis</td>
<td>31</td>
</tr>
<tr>
<td>FlexSearch's Schema File</td>
<td>31</td>
</tr>
<tr>
<td>Field Types</td>
<td>32</td>
</tr>
<tr>
<td>Field Types</td>
<td>32</td>
</tr>
<tr>
<td>Analysis</td>
<td>34</td>
</tr>
<tr>
<td>Using Analyzers, Tokenizers, And Filters</td>
<td>35</td>
</tr>
<tr>
<td>Analyzer</td>
<td>35</td>
</tr>
<tr>
<td>Tokenizer</td>
<td>36</td>
</tr>
<tr>
<td>Tokenizer Types</td>
<td>36</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Filter</td>
<td>38</td>
</tr>
<tr>
<td>Custom Analyzer</td>
<td>44</td>
</tr>
<tr>
<td>Strip White Space Analyzer</td>
<td>44</td>
</tr>
<tr>
<td>Strip To Numbers Analyzer</td>
<td>44</td>
</tr>
<tr>
<td>Address Analyzer</td>
<td>45</td>
</tr>
<tr>
<td>Index Model</td>
<td>48</td>
</tr>
<tr>
<td>Index Properties</td>
<td>48</td>
</tr>
<tr>
<td>Document Model</td>
<td>52</td>
</tr>
<tr>
<td>Scripting</td>
<td>53</td>
</tr>
<tr>
<td>Pre-Index Script</td>
<td>53</td>
</tr>
<tr>
<td>Search Execution Pipeline</td>
<td>56</td>
</tr>
<tr>
<td>Pre-Search Script</td>
<td>57</td>
</tr>
<tr>
<td>Post-Search Script</td>
<td>59</td>
</tr>
<tr>
<td>Search</td>
<td>60</td>
</tr>
<tr>
<td>Search Model</td>
<td>61</td>
</tr>
<tr>
<td>Model Properties Description</td>
<td>62</td>
</tr>
<tr>
<td>Paging Properties</td>
<td>63</td>
</tr>
<tr>
<td>Filtering Properties</td>
<td>63</td>
</tr>
<tr>
<td>Control Properties</td>
<td>65</td>
</tr>
<tr>
<td>Query Format</td>
<td>66</td>
</tr>
<tr>
<td>Identifier</td>
<td>66</td>
</tr>
<tr>
<td>Constant</td>
<td>66</td>
</tr>
<tr>
<td>Variable</td>
<td>67</td>
</tr>
<tr>
<td>Switch</td>
<td>67</td>
</tr>
<tr>
<td>Global Switch</td>
<td>68</td>
</tr>
<tr>
<td>Query Specific Switch</td>
<td>70</td>
</tr>
<tr>
<td>Condition</td>
<td>70</td>
</tr>
<tr>
<td>Query</td>
<td>70</td>
</tr>
<tr>
<td>EBNF Format</td>
<td>71</td>
</tr>
<tr>
<td>Query Functions</td>
<td>72</td>
</tr>
<tr>
<td>How Query Works?</td>
<td>72</td>
</tr>
<tr>
<td>General Query Properties</td>
<td>74</td>
</tr>
<tr>
<td>AllOf</td>
<td>75</td>
</tr>
<tr>
<td>AnyOf</td>
<td>76</td>
</tr>
<tr>
<td>Phrase Match</td>
<td>77</td>
</tr>
<tr>
<td>Fuzzy</td>
<td>80</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Like</td>
<td>81</td>
</tr>
<tr>
<td>Regex</td>
<td>82</td>
</tr>
<tr>
<td>Numeric Range Operator</td>
<td>83</td>
</tr>
<tr>
<td>Match All</td>
<td>83</td>
</tr>
<tr>
<td>Match None</td>
<td>84</td>
</tr>
<tr>
<td>Predefined Query</td>
<td>85</td>
</tr>
<tr>
<td>ADVANTAGES</td>
<td>85</td>
</tr>
<tr>
<td>Example</td>
<td>85</td>
</tr>
<tr>
<td>Working With Predefined Queries</td>
<td>87</td>
</tr>
<tr>
<td>Text Highlighting</td>
<td>88</td>
</tr>
<tr>
<td>REST</td>
<td>89</td>
</tr>
<tr>
<td>API Basics</td>
<td>90</td>
</tr>
<tr>
<td>Formats</td>
<td>90</td>
</tr>
<tr>
<td>JSONP Support</td>
<td>90</td>
</tr>
<tr>
<td>API Reference</td>
<td>91</td>
</tr>
<tr>
<td>FlexSearch REST Services</td>
<td>91</td>
</tr>
<tr>
<td>Methods</td>
<td>91</td>
</tr>
<tr>
<td>List Of Operations</td>
<td>91</td>
</tr>
<tr>
<td>Models</td>
<td>107</td>
</tr>
<tr>
<td>List Of Models</td>
<td>107</td>
</tr>
<tr>
<td>DataTypes</td>
<td>134</td>
</tr>
<tr>
<td>Examples</td>
<td>135</td>
</tr>
<tr>
<td>Index API Examples</td>
<td>135</td>
</tr>
<tr>
<td>Document API Examples</td>
<td>136</td>
</tr>
<tr>
<td>Search API Examples</td>
<td>137</td>
</tr>
<tr>
<td>SDK</td>
<td>144</td>
</tr>
<tr>
<td>CSharp Client</td>
<td>145</td>
</tr>
<tr>
<td>Namespaces</td>
<td>145</td>
</tr>
<tr>
<td>Request Model</td>
<td>145</td>
</tr>
<tr>
<td>Guidelines On Implementing Your Own FlexSearch Request</td>
<td>146</td>
</tr>
<tr>
<td>Response Model</td>
<td>146</td>
</tr>
<tr>
<td>Guidelines On Implementing Your Own Response</td>
<td>147</td>
</tr>
<tr>
<td>Usage Example</td>
<td>147</td>
</tr>
<tr>
<td>TypeScript Client</td>
<td>148</td>
</tr>
<tr>
<td>Summary</td>
<td>148</td>
</tr>
<tr>
<td>Initializing The APIs</td>
<td>148</td>
</tr>
</tbody>
</table>
# Table Of Contents

Accessing The API Methods .................................................. 149
Complex Example ................................................................. 150
Custom Plugins ..................................................................... 151
Definition ............................................................................. 151
Writing Your Own FlexSearch Plugin ...................................... 151
Installing A FlexSearch Plugin ............................................... 154
CSV Connector ....................................................................... 156
Summary .............................................................................. 156
Request Object ....................................................................... 156
Endpoint ............................................................................... 156
Example ................................................................................. 156
SQL Connector ....................................................................... 157
Summary .............................................................................. 157
Request Object ....................................................................... 157
Endpoint ............................................................................... 157
Example ................................................................................. 157
Guides .................................................................................. 158
Checking Index Status ............................................................. 159
Checking The Status Of An Index Using The C# Client .......... 159
Checking The Status Of An Index By Writing The HTTP Request Ourselves ........ 160
Closing An Index .................................................................... 161
Closing An Index Using The C# Client ..................................... 161
Closing An Index By Writing The HTTP Request Ourselves .... 161
Creating An Index ................................................................... 162
Creating An Index Using The C# Client ................................... 162
Creating An Index By Writing The HTTP Request Ourselves .... 162
Deleting Documents ................................................................ 164
Deleting Documents Using The C# Client ................................ 164
Deleting Documents By Writing The HTTP Request Ourselves ... 164
Indexing Documents ............................................................... 165
Indexing Documents Using The C# Client ............................... 165
Indexing Documents By Writing The HTTP Request Ourselves ... 166
Modifying Documents ............................................................ 167
Modifying Documents Using The C# Client ............................ 167
Modifying Documents By Writing The HTTP Request Ourselves ... 168
Removing An Index ............................................................... 169
# Table Of Contents

- **Removing An Index Using The C# Client** 169
- **Removing An Index By Writing The HTTP Request Ourselves** 169
- **Searching Data** 170
  - **Search Using The C# Client** 170
  - **Search By Writing The HTTP Request Ourselves** 171
  - **Query Examples** 171
- **Search With Predefined Query** 172
  - **Get The Search API** 172
  - **Create Your SearchQuery That Calls The Predefined Query** 173
  - **Calling The Predefined Query Without Any Variables Using The C# Client** 173
  - **Calling The Predefined Query While Passing A Variable Using The C# Client** 174
  - **Searching Using A Predefined Query By Writing The HTTP Request Ourselves** 174
- **Working With Predefined Query** 176
  - **Creating Or Updating A Predefined Query Using The C# Client** 176
  - **Creating Or Updating A Predefined Query By Writing The HTTP Request Ourselves** 177
Quick-Start
Overview

Contents

• What is FlexSearch?
• Basic Concepts
• How FlexSearch Sees the World?
• Text analysis
• Some more jargon
• Near Realtime (NRT)
• Shards
• Maximum size of Shard

What Is FlexSearch?

FlexSearch is a high performance REST based full-text searching platform built on top of the popular Lucene search library. At its core it is about extensibility and maintainability with minimum overhead. FlexSearch is written in F# & C# 5.0 (.net framework 4.6.1).

It has an extensive plug-in architecture with ability to customize most of the functionality with minimum amount of efforts. It also supports scripting which can be used at both search and index time to fine tune the data. One area where FlexSearch particularly excel, is providing easy extensible connector model which allows a developer to tap directly into core's indexing engine, thus avoiding the reliance on web services. This results in a greatly improved indexing performance when indexing over millions of records.

Some ideal use cases for the engine would be:

• Searching across unstructured text data on intranet and websites.
• Searching across structured data coming from SQL, CSV and other sources.
• Duplicate detection over large volume of structured data like customer information de-duplication, address matching etc.

Basic Concepts

Portions of this page are modifications based on work created and shared by the Solr Project (Apache Solr Reference Guide) and used according to terms described in the Apache 2.0 License.

Link back to the original article

The fundamental premise of FlexSearch is simple. You give it a lot of information, then later you can ask it questions and find the piece of information you want. The part where you feed in all the information is called indexing or updating. When you ask a question, it’s called a query.

One way to understand how FlexSearch works is to think of a loose-leaf book of recipes. Every time you add a recipe to the book, you update the index at the back. You list each ingredient and the page number of the recipe you just added. Suppose you add one hundred recipes. Using the index, you can very quickly find all the
recipes that use garbanzo beans, or artichokes, or coffee, as an ingredient. Using the index is much faster than looking through each recipe one by one. Imagine a book of one thousand recipes, or one million.

FlexSearch allows you to build an index with many different fields, or types of entries. The example above shows how to build an index with just one field, ingredients. You could have other fields in the index for the recipe's cooking style, like Asian, Cajun, or vegan, and you could have an index field for preparation times. FlexSearch can answer questions like “What Cajun-style recipes that have blood oranges as an ingredient can be prepared in fewer than 30 minutes?”

Index hierarchy

The schema is the place where you tell FlexSearch how it should build indexes from input documents.

In case of database analogy think of Index as a Database table which has fixed schema and requires the schema definition before creation of the table.

How FlexSearch Sees The World?

FlexSearch's basic unit of information is a document, which is a set of data that describes something. A recipe document would contain the ingredients, the instructions, the preparation time, the cooking time, the tools needed, and so on. A document about a person, for example, might contain the person's name, biography, favorite color, and shoe size. A document about a book could contain the title, author, year of publication, number of pages, and so on.

In the FlexSearch universe, documents are composed of fields, which are more specific pieces of information. Shoe size could be a field. First name and last name could be fields. Fields can contain different kinds of data. A name field, for example, is text (character data). A shoe size field might be a floating point number so that it could contain values like 6 and 9.5. Obviously, the definition of fields is flexible (you could define a shoe size field as a text field rather than a floating point number, for example), but if you define your fields correctly, FlexSearch will be able to interpret them correctly and your users will get better results when they perform a query. You can tell FlexSearch about the kind of data a field contains by specifying its field type. The field type tells FlexSearch how to interpret the field and how it can be queried. When you add a document, FlexSearch takes the information in the document's fields and adds that information to an index. When you perform a query, FlexSearch can quickly consult the index and return the matching documents.
In case of database analogy think of Document as the row of table.

Text Analysis

In FlexSearch every field has an Analyzer consisting of:

- **Preprocessing Script** – A F# based script which can be used to clean the input data. This only runs at index time and not at the search time.

- **Tokenizer** – Splits incoming text in multiple tokens based on tokenization rule like white space.

- **Zero or more Filters** – Applies custom logic on each token like case conversion, synonym & stop word processing.

Let's consider a simple example, we have an input text "The quick brown fox jumps over the lazy dog". Let's process the text using a simple analyzer consisting of a white space tokenizer and few filters. The white space tokenizer breaks the input at whitespace boundaries resulting in creation of multiple tokens from the given input. There can only be a single tokenizer per analyzer.

And the end of tokenisation we have a series of tokens which can be fed into multiple filters. In our example the tokens first go through lowercase filter which converts all the tokens to lowercase. Now the resulting output is fed into a stop word filter which removes commonly used stop words. In our case the word *the* is removed. The resulting output then goes through a synonym filter which adds synonym for words quick and lazy.
Some More Jargon

NEAR REALTIME (NRT)

Near Real Time (NRT) search means that documents are available for search almost immediately (usually 1 second) after being indexed: additions and updates to documents are seen in ‘near’ real time. FlexSearch does not block updates while a commit is in progress. Nor does it wait for background merges to complete before opening a new search of indexes and returning.

SHARDS

When your data is too large for a single index, you can break it up and store it in smaller indices by creating one or more shards. Each shard is a self contained Lucene index capable of functioning alone. When you create an index, you can define the number of shards that you want.

MAXIMUM SIZE OF SHARD

The maximum number of documents that a single shard can hold is 2,147,483,519. Refer: LUCENE-5843 for more information.
Getting Started

Contents
- Install FlexSearch
- Requirements
- Viewing Logs
- Accessing Portal
- Creating an index
- Retrieving Index information
- Indexing data
- Retrieving documents by Id
- Modifying data
- Setting up demo index
- Searching data
- Making a search request
- AllOf Query function
- AnyOf query function

The below guide will walk you through the steps involved in installing FlexSearch and utilising basic capabilities like indexing and searching.

Install FlexSearch

REQUIREMENTS
- .net Runtime Environment version 4.6 or higher
- Windows 7, Windows 2008 R2 or higher
- Ram : 4GB minimum / 8GB recommended
- CPU : Dual Core CPU (minimum) / Quad Core CPU (recommended)
5 Minute Install

Download FlexSearch Package

Extract [FlexSearch.<version_number>.zip] into a folder (root of drive recommended, assume C:\flexsearch).

Start the server by executing FlexSearch-Server.exe

INSTALL AS A SERVICE (OPTIONAL)

From an elevated command prompt (administrator mode) @ C:\FlexSearch run:

> FlexSearch-Server.exe --install

> FlexSearch-Server.exe --start

This installs FlexSearch as a service, then starts the service.

VIEWING LOGS

Once configured successfully you can access the logs from Event Viewer.

ACCESSING PORTAL

Navigate to http://localhost:9800/.

Creating An Index

Follow along the below tutorial by using your favourite HTTP request command tool like Fiddler, Curl, Postman etc.
Let’s start with a simple example of creating an index with the below information. Index field name represents the name that will be used to represent the field in the index. Field name should be lowercase and should not contain any special characters. The type of field represents the type of value we intend to save in the field to save in the field. Text field is a general-purpose field which can be used to save any kind of data which needs to be tokenized. By tokenization we refer to the process of breaking the input text into smaller tokens. For example a whitespace tokenizer will break the input text at white spaces.

Keyword field is used to represent a piece of information which should not be tokenized, fields like identifiers, constants etc where tokenization doesn't make any sense.

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Index Field Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee ID</td>
<td>employeeId</td>
<td>Keyword</td>
</tr>
<tr>
<td>First Name</td>
<td>firstname</td>
<td>Text</td>
</tr>
<tr>
<td>Last Name</td>
<td>lastname</td>
<td>Text</td>
</tr>
<tr>
<td>Date of Birth</td>
<td>dob</td>
<td>Date</td>
</tr>
</tbody>
</table>

Fire up your favourite HTTP request generation tool and make the below request. FlexSearch follows a simple mapping convention to define the rest endpoints.

Refer to the API basics guide to know more about the convention.

Here we are trying to create an index which is hosted on `indices` endpoint. According to a convention creation maps to `POST` web request. There are few other properties that can be defined for a specific field. The below only represents the bare minimum information that is required to create an index. `Text` is the default file type so we don't have to specify it for first name and last name fields.
Executing the above request will give you the below response. There are a few interesting things about this response:

- The response returned with a HTTP code of 201 which stands for created. This is one of the many cases where FlexSearch tries to follow the correct HTTP codes.

- The returned response object has two parts data and error. These two parts are always returned as part of FlexSearch response. In this case there is no error so the server returned now for the error property.

HTTP/1.1 201 Created
Transfer-Encoding: chunked
Content-Type: application/json
Server: Kestrel

{
    "data": true,
    "error": null
}

Retrieving Index Information

In order to retrieve information about an existing index, execute the below HTTP request:
GET http://localhost:9800/indices/contact HTTP/1.1

HTTP/1.1 200 OK
Date: Sun, 08 May 2016 14:35:08 GMT
Transfer-Encoding: chunked
Content-Type: application/json
Server: Kestrel

{  
  "data":{
    "indexName":"contact",
    "fields":[
      {
        "allowSort":false,
        "fieldName":"employeeid",
        "fieldType":"Keyword",
        "indexAnalyzer":"standard",
        "searchAnalyzer":"standard",
        "similarity":"TFIDF"
      },
      {
        "allowSort":false,
        "fieldName":"firstname",
        "fieldType":"Text",
        "indexAnalyzer":"standard",
        "searchAnalyzer":"standard",
        "similarity":"TFIDF"
      },
      {
        "allowSort":false,
        "fieldName":"lastname",
        "fieldType":"Text",
        "indexAnalyzer":"standard",
        "searchAnalyzer":"standard",
        "similarity":"TFIDF"
      },
      {
        "allowSort":false,
        "fieldName":"dob",
        "fieldType":"DateTime",
        "indexAnalyzer":"standard",
        "searchAnalyzer":"standard",
        "similarity":"TFIDF"
      }
    ],
    "predefinedQueries":[]
  },
  "shardConfiguration":{
    "shardCount":1
  }
}
Here the response contains a lot more information compared to what we defined in the previous section. What has happened is that FlexSearch used default values for the missing properties. You might wish to specify some of these for more specific cases, but for a simple example the defaults will do.

in order to know more about index object refer to: index object.

Again the response consists of two parts data and error. As the request was successful the returned object has null value for error property.

Have you noticed that active property in the response? The active property is used to signify if the index should be active or not. Only an active index can be searched. In order to check if an index is active or inactive we can execute the below HTTP request:

GET http://localhost:9800/indices/contact/status HTTP/1.1

HTTP/1.1 200 OK
Date: Sun, 08 May 2016 14:44:24 GMT
Transfer-Encoding: chunked
Content-Type: application/json
Server: Kestrel

{"data":{"indexStatus":"Online"},"error":null}
Indexing Data

Let's try to add a document to our newly created index. Each document in the index should have an identifier associated with it. Unlike a database FlexSearch doesn't enforce uniqueness of the identifier. So you could have multiple documents each associated with the same identifier. But in most of the cases it does make sense to have a unique identifier otherwise there will be no way to retrieve or update the document.

Refer to concurrency control to understand more about the ID requirement.

If you remember we never specified an ID field when we were creating a index. This is because FlexSearch uses an internal field called `id` to represent ID of the document. The name of this field cannot be changed.

All internal fields start with `_` character.

Since we are trying to create a document, according to our convention we have to use a POST request.

```json
POST http://localhost:9800/indices/contact/documents HTTP/1.1
{
  "fields": {
    "firstname": "John",
    "lastname": "Doe"
  },
  "id": "1",
  indexName: "contact"
}
```

```
HTTP/1.1 201 Created
Date: Sun, 08 May 2016 15:00:55 GMT
Transfer-Encoding: chunked
Content-Type: application/json
Server: Kestrel

{"data":{"id":"1"},"error":null}
```

Retrieving Documents By Id

Any document in an index can be retrieved by using its ID.

```text
GET http://localhost:9800/indices/contact/documents/1 HTTP/1.1
```
HTTP/1.1 200 OK
Date: Sun, 08 May 2016 15:10:16 GMT
Transfer-Encoding: chunked
Content-Type: application/json
Server: Kestrel

{
    "data": {
        "fields": {
            "employeeid": "",
            "firstname": "John",
            "lastname": "Doe",
            "dob": "1010100000"
        },
        "id": "1",
        "indexName": "contact",
        "timeStamp": "20160508160115647",
        "modifyIndex": 2,
        "highlights": [
            {
                "score": 0.0
            },
            {
                "error": null
            }
        ],
        "error": null
    }
}

Modifying Data

An existing record can be updated using the ID of the document. The service endpoint can be used for both creating new documents and for updating existing documents. The behaviour whether the document will be updated on a new document gets created depends on the value of `modifyIndex` field. The default behaviour is to check if a document exist with a given ID, if it does then it updates the document otherwise it creates a new one.

Refer to concurrency control to understand more about the modify index field.

PUT http://localhost:9800/indices/contact/documents/1 HTTP/1.1

{
    "fields": {
        "firstname": "John",
        "lastname": "Smith"
    },
    "id": "1",
    "indexName": "contact"
}
Setting Up Demo Index

FlexSearch ships sample data extracted from [World Fact book](http://worldfactbook.com) which is used to create a dummy index. This will be used throughout the documentation to explain major concepts. You can download the data from the following link to cross check the results: World Fact Book.

This data is extremely useful for demonstrating a number of search concepts. There are various kinds of fields which can be used for text and numeric searching.

> The data presented here is for demonstration purpose only and may not be relevant in real world.

In order to set up demo index, go to the homepage of the portal and select the demo index card from the installed tools section.

Alternatively, you could use a `PUT` request.

```
PUT http://localhost:9800/setupdemo HTTP/1.1
```

Searching Data

FlexSearch uses its own query search language to provide advanced search capabilities. At the lowest level a search expression consists of a search condition. A search condition is written as:

```
queryFunction(Field Name to be searched, 'Field Value to be searched')
```
This is similar to writing a function call in a programming language. Here query function is the name of the function to be called, the `[ ]` mark the beginning and ending of the query function. There are many query functions available with FlexSearch and new query functions can be easily written to suit individual needs. Let’s explore some basic query functions.

**MAKING A SEARCH REQUEST**

FlexSearch exposes a search endpoint which supports both get and post requests. The reason for supporting searching on both endpoint is to allow easy calling from a web browser using JSONP or CORS.

The only field which is mandatory is the query string. Let’s search the country index for country name equal to France. Here we are using allOf query and we have selected four columns from the index. In case you want to return all the columns just set `c=*`. In case of post request example we are passing the query string in the post request body. Sometimes it is easier to pass complex query strings in the body of the post request as you don’t have to escape the characters. Also browsers have a maximum character limit for the URLs.

FlexSearch uses single quotes to the present a search value. This was chosen so that a user won’t have to escape the field when used with JSON.

```
GET http://localhost:9800/indices/country/search?c=countryname,countrycode,area,population&q=allOf(countryname,'france') HTTP/1.1

POST http://localhost:9800/indices/country/search?c=countryname,countrycode,area,population HTTP/1.1
{
  "QueryString" : "allOf(countryname, 'france')"
}
```

Response
HTTP/1.1 200 OK
Date: Sun, 08 May 2016 18:02:49 GMT
Transfer-Encoding: chunked
Content-Type: application/json
Server: Kestrel

{
    "data":{
        "documents":[
            {
                "fields":{
                    "countryname":"France",
                    "countrycode":"fr",
                    "area":"643427",
                    "population":"64057792"
                },
                "id":"84",
                "indexName":"country",
                "timeStamp":20160508190041077,
                "modifyIndex":84,
                "highlights":[
                    {
                        "score":5.8636808395385742
                    }
                ],
                "recordsReturned":1,
                "bestScore":5.863681,
                "totalAvailable":1
            },
            "error":null
        }
    }
}

In order to understand more about the search object and the search response object, please refer to search section of the guide.

ALLOF QUERY FUNCTION

AllOf is a very basic query function which matches all the tokens from the given input.

The following query returns all documents containing [Wheat] and [Rice] both, in the [agriproducts] field.
AllOf single clause with a single token

allOf(agrproducts, 'rice wheat')

The above query is semantically similar to the below queries:

allOf(agrproducts, 'rice') and allOf(agrproducts, 'wheat')

allOf(agrproducts, 'rice', 'wheat')

allOf(agrproducts, 'wheat rice')

Here we have introduced logical operations. The syntax is similar to writing a logical expression in any programming language.

ANYOF QUERY FUNCTION

AnyOf function is opposite of allOf function in the sense that the function succeeds if any of the passed token is present in the document.

The following search query returns all documents containing wheat or Rice or both, in the agrproducts field.

AnyOf single clause with a single token

anyOf(agrproducts, 'rice wheat')

The above query is semantically similar to the below queries:

anyOf(agrproducts, 'rice') and anyOf(agrproducts, 'wheat')
anyOf(['agriproducts', 'rice', 'wheat'])

anyOf(['agriproducts', 'wheat rice'])

FlexSearch comes with many other query functions. Please refer to the query functions section to know more about advance query functionality.
Server
Configuration

Contents

- Directory Structure
- Configuration folder
- Data folder
- Library folder
- Logs folder
- Plug-ins folder
- Web folder
- Configure global Settings

Directory Structure

FlexSearch follow a specific directory structure, understanding this will help you in troubleshooting issues with FlexSearch. This will also help you in understanding the folders which needs to be backed up when taking an on-site or offsite backup.

```
<table>
<thead>
<tr>
<th>conf</th>
</tr>
</thead>
<tbody>
<tr>
<td>analyzers</td>
</tr>
<tr>
<td>indices</td>
</tr>
<tr>
<td>resources</td>
</tr>
<tr>
<td>config.ini</td>
</tr>
<tr>
<td>data</td>
</tr>
<tr>
<td>lib</td>
</tr>
<tr>
<td>licenses</td>
</tr>
<tr>
<td>plugins</td>
</tr>
<tr>
<td>web</td>
</tr>
</tbody>
</table>
```

CONFIGURATION FOLDER

This folder contains all the configuration related data used by FlexSearch. Backing this folder will allow you to the restore indexes, analyzers and all related settings to a new server. `config.ini` file present in the `conf` folder is the global configuration file.

DATA FOLDER

This folder contains all indexing related data used by FlexSearch. Each index present in the system has its own subfolder below this folder. Each subfolder can be independently backed up and re-stored. Backing this folder along with configuration folder will give you everything that is needed to migrate a server.
Each index folder contains a folder called shards which in turn can contain multiple subfolders each representing a shard present in the index. The shards are numbered from 0 and up. Each shard folder contains one subfolder `index`. Each FlexSearch shard is a valid Lucene index and can be opened using tools like Luke. FlexSearch uses write ahead logging to store the information which is not committed to the physical medium. This gives FlexSearch the ability to recover data which is not saved to the physical index yet. The transaction log folder is used to save transaction log per shard.

LIBRARY FOLDER

This folder contains all the third-party libraries used by FlexSearch.

LOGS FOLDER

This folder contains all the server logs. FlexSearch only writes physical files for logging when it cannot access Windows ETW logger. This folder also contains a special file called startup-log.txt, this file is always written by FlexSearch irrespective of the logger setting and can be used to identify server start-up related issues.

PLUG-INS FOLDER

This folder contains all the custom/third-party plug-ins written for FlexSearch. These plug-ins are loaded during the start-up.

WEB FOLDER

This folder contains portal related files.

Configure Global Settings

Global configuration can be accessed from `Config.ini` file under `Conf` folder present in the root directory. `HttpPort` key and the server section can be used to configure the port number used by FlexSearch to start the server. This file can also be used by custom plug-ins to allow user to configure the plug-in.

```
[Server]
HttpPort = 9800
DataFolder = ./data
NodeName = FlexNode
```
Index
Indexing Overview

Contents

- How FlexSearch Sees the World?
- Field Analysis
- FlexSearch's Schema File

The fundamental premise of FlexSearch is simple. You give it a lot of information, then later you can ask it questions and find the piece of information you want. The part where you feed in all the information is called indexing or updating. When you ask a question, it’s called a query.

One way to understand how FlexSearch works is to think of a loose-leaf book of recipes. Every time you add a recipe to the book, you update the index at the back. You list each ingredient and the page number of the recipe you just added. Suppose you add one hundred recipes. Using the index, you can very quickly find all the recipes that use garbanzo beans, or artichokes, or coffee, as an ingredient. Using the index is much faster than looking through each recipe one by one. Imagine a book of one thousand recipes, or one million.

FlexSearch allows you to build an index with many different fields, or types of entries. The example above shows how to build an index with just one field, ingredients. You could have other fields in the index for the recipe’s cooking style, like Asian, Cajun, or vegan, and you could have an index field for preparation times. FlexSearch can answer questions like “What Cajun-style recipes that have blood oranges as an ingredient can be prepared in fewer than 30 minutes?”

The schema is the place where you tell FlexSearch how it should build indexes from input documents.

How FlexSearch Sees The World?

FlexSearch’s basic unit of information is a document, which is a set of data that describes something. A recipe document would contain the ingredients, the instructions, the preparation time, the cooking time, the tools needed, and so on. A document about a person, for example, might contain the person’s name, biography, favorite color, and shoe size. A document about a book could contain the title, author, year of publication, number of pages, and so on.

In the FlexSearch universe, documents are composed of fields, which are more specific pieces of information. Shoe size could be a field. First name and last name could be fields. Fields can contain different kinds of data. A name field, for example, is text (character data). A shoe size field might be a floating point number so that it could contain values like 6 and 9.5. Obviously, the definition of fields is flexible (you could define a shoe size field as a text field rather than a floating point number, for example), but if you define your fields correctly, FlexSearch will be able to interpret them correctly and your users will get better results when they perform a query. You can tell FlexSearch about the kind of data a field contains by specifying its field type. The field type
Field Analysis

Field analysis tells FlexSearch what to do with incoming data when building an index. A more accurate name for this process would be processing or even digestion, but the official name is *analysis*.

Consider, for example, a biography field in a person document. Every word of the biography must be indexed so that you can quickly find people whose lives have had anything to do with ketchup, or dragonflies, or cryptography. However, a biography will likely contains lots of words you don't care about and don't want clogging up your index—words like "the", "a", "to", and so forth. Furthermore, suppose the biography contains the word "Ketchup", capitalized at the beginning of a sentence. If a user makes a query for "ketchup", you want FlexSearch to tell you about the person even though the biography contains the capitalized word.

The solution to both these problems is field analysis. For the biography field, you can tell FlexSearch how to break apart the biography into words. You can tell FlexSearch that you want to make all the words lower case, and you can tell FlexSearch to remove accents marks.

Field analysis is an important part of a field type. Understanding Analyzers, Tokenizers and Filters is a detailed description of field analysis.

FlexSearch's Schema File

FlexSearch stores details about the field types and fields it is expected to understand in a schema file. The name of the file is `index.json` and location of this file under the `conf/indices/{indexName}` folder in the root of your FlexSearch directory. You can either modify this file directly in case you are feeling adventurous or rely on the REST webservice exposed by FlexSearch.

Every part of the schema file can be modified using the webservices.
Field Types

A field definition can include four types of information:

- The name of the field type (mandatory)
- Type of the field (defaults to `Text`)
- Information about the associated search and index time analyzers (defaults to `StandardAnalyzer`)
- Field type properties - depends on the `FieldType`

Below is a sample Field definition from the `index.json` file.

```
{
   "allowSort": false,
   "fieldName": "b1",
   "fieldType": "Bool",
   "indexAnalyzer": "standard",
   "searchAnalyzer": "standard",
   "similarity": "TFIDF"
}
```

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AllowSort</td>
<td>Enables a field for sorting. Behind the scene a new docvalues field is created with the same name which is used for sorting.</td>
</tr>
<tr>
<td>IndexAnalyzer</td>
<td>Analyzer to be used during index time</td>
</tr>
<tr>
<td>SearchAnalyzer</td>
<td>Analyzer to be used during search time</td>
</tr>
<tr>
<td>Similarity</td>
<td>A field may optionally specify a <code>similarity</code> that will be used when scoring documents.</td>
</tr>
</tbody>
</table>

As Lucene index back-compatibility is only supported for the default codec. We took a conscious decision to hide codec level settings at the field level. Technically it is possible to have per field postings format and docvalues format. Currently FlexSearch only allows configuring Bloom Filter on the id field.

Field Types

The field type defines how FlexSearch should interpret data in a field and how the field can be queried. There are many field types included with FlexSearch by default which should cover most of the cases. The below table list the various field types supported by FlexSearch.
<table>
<thead>
<tr>
<th>Field Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>Integer</td>
</tr>
<tr>
<td>Double</td>
<td>Double</td>
</tr>
<tr>
<td>Float</td>
<td>Float</td>
</tr>
<tr>
<td>Long</td>
<td>Long</td>
</tr>
<tr>
<td>Keyword</td>
<td>Field to store keywords. The entire input will be treated as a single word. This is useful for fields like <code>customerId</code>, <code>referenceId</code> etc. These fields only support complete text matching while searching and no partial word match is available.</td>
</tr>
<tr>
<td>Text</td>
<td>General purpose field to store normal textual data. This also supports text highlighting.</td>
</tr>
<tr>
<td>Bool</td>
<td>Boolean (Internally the values are saved as 'T' and 'F' to save space)</td>
</tr>
<tr>
<td>Date</td>
<td>Fixed format date field (Supported format: <code>YYYYmmdd</code>)</td>
</tr>
<tr>
<td>DateTime</td>
<td>Fixed format datetime field (Supported format: <code>YYYYMMDDhhmmss</code>)</td>
</tr>
<tr>
<td>Stored</td>
<td>Non-indexed field. Only used for retrieving stored text. Searching is not possible over these fields.</td>
</tr>
</tbody>
</table>
The following sections describe how FlexSearch breaks down and works with textual data. There are three main concepts to understand: analyzers, tokenizers, and filters.

- **Field analyzers** are used both during ingestion, when a document is indexed, and at query time. An analyzer examines the text of fields and generates a token stream. Analyzers may be a self contained or they may be composed of a series of tokenizer and filter.
- **Tokenizers** break field data into lexical units, or tokens.

- **Filters** examine a stream of tokens and keep them, transform or discard them, or create new ones. Tokens and filters may be combined to form pipelines, or chains, where the output of one is input to the next. Such a sequence of tokenizers and filters is called an analyzer and the resulting output of an analyzer is used to match query results or build indices.

### Using Analyzers, Tokenizers, And Filters

Although the analysis process is used for both indexing and querying, the same analysis process need not be used for both operations. For indexing, you often want to simplify, or normalize, words. For example, setting all letters to lowercase, eliminating punctuation and accents, mapping words to their stems, and so on. Doing so can increase recall because, for example, `ram`, `Ram` and `RAM` would all match a query for `ram`. To increase query-time precision, a filter could be employed to narrow the matches by, for example, ignoring all-cap acronyms if you're interested in male sheep, but not Random Access Memory.

The tokens output by the analysis process define the values, or terms, of that field and are used either to build an index of those terms when a new document is added, or to identify which documents contain the terms you are querying for.

### Analyzer

An analyzer examines the text of fields and generates a token stream. Analyzers are specified as part of the `Field Properties` element in the `Fields` section of index configuration.

**StandardAnalyzer** is used when no analyzer is specified.

For simple cases, such as plain English prose, a single analyzer class like this may be sufficient. But it’s often necessary to do more complex analysis of the field content. Even the most complex analysis requirements can usually be decomposed into a series of discrete, relatively simple processing steps. As you will soon discover, the FlexSearch distribution comes with a large selection of tokenizers and filters that covers most scenarios you are likely to encounter.

### ANALYSIS PHASES

Analysis takes place in two contexts. At index time, when a field is being created, the token stream that results from analysis is added to an index and defines the set of terms (including positions, sizes, and so on) for the field. At query time, the values being searched for are analyzed and the terms that result are matched against those that are stored in the field’s index.

In many cases, the same analysis should be applied to both phases. This is desirable when you want to query for exact string matches, possibly with case-insensitivity, for example. In other cases, you may want to apply slightly different analysis steps during indexing than those used at query time.

### OUT OF BOX ANALYZERS

FlexSearch comes pre-configured with the following analyzers.
Tokenizer

The job of a tokenizer is to break up a stream of text into tokens, where each token is (usually) a sub-sequence of the characters in the text. An analyzer is aware of the field it is configured for, but a tokenizer is not. Tokenizers read from a character stream (a Reader) and produce a Characters in the input stream may be discarded, such as whitespace or other delimiters. They may also be added to or replaced, such as mapping aliases or abbreviations to normalized forms. A token contains various metadata in addition to its text value, such as the location at which the token occurs in the field. Because a tokenizer may produce tokens that diverge from the input text, you should not assume that the text of the token is the same text that occurs in the field, or that its length is the same as the original text. It's also possible for more than one token to have the same position or refer to the same offset in the original text. Keep this in mind if you use token metadata for things like highlighting search results in the field text.

Tokenizer Types

STANDARD TOKENIZER

StandardTokenizer is a good general purpose tokenizer that strips many extraneous characters and sets token types to meaningful values. This tokenizer splits the text field into tokens, treating whitespace and punctuation as delimiters. Delimiter characters are discarded, with the following exceptions:

- Periods (dots) that are not followed by whitespace are kept as part of the token.
- Words are split at hyphens.

The Standard Tokenizer supports Unicode standard annex UAX#29 word boundaries with the following token types: `<ALPHANUM>`, `<NUM>`, `<SOUTHEAST ASIAN>`, `<IDEOGRAPHIC>` and `<HIRAGANA>`.

| In: "Please, email john.doe@foo.com by 03-09, re: m37-xq." |
| Out: "Please", "email", "john.doe", "foo.com", "by", "03", "09", "re", "m37", "xq" |

CLASSIC TOKENIZER

The Classic Tokenizer preserves the same behaviour as the Standard Tokenizer of Lucene versions 3.1 and previous. It does not use the Unicode standard annex UAX#29 word boundary rules that the Standard Tokenizer uses. This tokenizer splits the text field into tokens, treating whitespace and punctuation as delimiters. Delimiter characters are discarded, with the following exceptions:

- Periods (dots) that are not followed by whitespace are kept as part of the token.
- Words are split at hyphens, unless there is a number in the word, in which case the token is not split and the numbers and hyphen(s) are preserved.
- Recognizes Internet domain names and email addresses and preserves them as a single token.
In: "Please, email john.doe@foo.com by 03-09, re: m37-xq." Out: "Please", "email", "john.doe@foo.com", "by", "03-09", "re", "m37-xq"

KEYWORD TOKENIZER

This tokenizer treats the entire text field as a single token.

In: "Please, email john.doe@foo.com by 03-09, re: m37-xq."
Out: "Please, email john.doe@foo.com by 03-09, re: m37-xq."

LETTER TOKENIZER

This tokenizer creates tokens from strings of contiguous letters, discarding all non-letter characters.

In: "Please, email john.doe@foo.com by 03-09, re: m37-xq."
Out: "Please", "email", "john", "doe", "foo", "com", "by", "re", "m", "xq"

LOWER CASE TOKENIZER

It tokenizes the input stream by delimiting at non-letters and then converting all letters to lowercase. Whitespace and non-letters are discarded.

In: "Please, email john.doe@foo.com by 03-09, re: m37-xq."
Out: "please", "email", "john", "doe", "foo", "com", "by", "re", "m", "xq"

UAX29 URL EMAIL TOKENIZER

This tokenizer splits the text field into tokens, treating whitespace and punctuation as delimiters. Delimiter characters are discarded, with the following exceptions:

- Periods (dots) that are not followed by whitespace are kept as part of the token.

- Words are split at hyphens, unless there is a number in the word, in which case the token is not split and the numbers and hyphen(s) are preserved.

- Recognizes top-level (.com) Internet domain names; email addresses; file://, http(s)://, and ftp:// addresses; IPv4 and IPv6 addresses; and preserves them as a single token.

- The UAX29 URL Email Tokenizer supports Unicode standard annex UAX#29 word boundaries with the following token types: <ALPHANUM>, <NUM>, URL, EMAIL, <SOUTHEAST_ASIAN>, <IDEOGRAPHIC> and <HIRAGANA>
WHITE SPACE TOKENIZER

It is a simple tokenizer that splits the text stream on whitespace and returns sequences of non-whitespace characters as tokens. Note that any punctuation will be included in the tokenization.

Filter

Like tokenizers, filters consume input and produce a stream of tokens. The job of a filter is usually easier than that of a tokenizer since in most cases a filter looks at each token in the stream sequentially and decides whether to pass it along, replace it or discard it.

A filter may also do more complex analysis by looking ahead to consider multiple tokens at once, although this is less common. One hypothetical use for such a filter might be to normalize state names that would be tokenized as two words. For example, the single token "california" would be replaced with "CA", while the token pair "rhode island" followed by "CA" would become the single token "RI".

Because filters consume one Token Stream and produce a new Token Stream, they can be chained one after another indefinitely. Each filter in the chain in turn processes the tokens produced by its predecessor. The order in which you specify the filters is therefore significant. Typically, the most general filtering is done first, and later filtering stages are more specialized.

```json
{
  "analyzerName": "firstnameanalyzer",
  "tokenizer": {
    "TokenizerName": "standardtokenizer"
  },
  "Filters": [
    {
      "FilterName": "standardfilter"
    },
    {
      "FilterName": "lowercasefilter"
    }
  ]
}
```
This example starts with FlexSearch’s standard tokenizer, which breaks the field’s text into tokens. Those tokens then pass through FlexSearch’s standard filter, which removes dots from acronyms, and performs a few other common operations. All the tokens are then set to lowercase, which will facilitate case-insensitive matching at query time.

ASCII FOLDING FILTER

This filter converts alphabetic, numeric, and symbolic Unicode characters which are not in the Basic Latin Unicode block (the first 127 ASCII characters) to their ASCII equivalents if one exists. This filter converts characters from the following Unicode blocks:

- C1 Controls and Latin-1 Supplement (PDF)
- Latin Extended-A (PDF)
- Latin Extended-B (PDF)
- Latin Extended-Additional (PDF)
- Latin Extended-C (PDF)
- Latin Extended-D (PDF)
- IPA Extensions (PDF)
- Phonetic Extensions (PDF)
- Phonetic Extensions Supplement (PDF)
- General Punctuation (PDF)
- Superscripts and Subscripts (PDF)
- Enclosed Alphanumerics (PDF)
- Dingbats (PDF)
- Supplemental Punctuation (PDF)
- Alphabetic Presentation Forms (PDF)
- Halfwidth and Fullwidth Forms (PDF)

In: (Unicode character 00E1) Out: (ASCII character 160)

STANDARD FILTER

This filter removes dots from acronyms and the substring ‘s’ from the end of tokens. This filter depends on the tokens being tagged with the appropriate term-type to recognize acronyms and words with apostrophes.
Even though this is the expected behaviour, we are unable to reproduce it through the unit test. Probably there has been a change in Lucene's behaviour. Refer: Standard Analyzer functionality change

BEIDER MORSE FILTER

Implements the Beider-Morse Phonetic Matching (BMPM) algorithm, which allows identification of similar names, even if they are spelled differently or in different languages.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nametype</td>
<td>GENERIC</td>
<td>Types of names. Valid values are GENERIC, ASHKENAZI, or SEPHARDIC. If not processing Ashkenazi or Sephardic names, use GENERIC.</td>
</tr>
</tbody>
</table>

DOUBLE METAPHONE FILTER

This filter creates tokens using DOUBLE METAPHONE phonetic encoding algorithm.

In: "four score and twenty"
Tokenizer to Filter: "four"(1), "score"(2), "and"(3), "twenty"(4)
Out: "four"(1), "FR"(1), "score"(2), "SKR"(2), "and"(3), "ANT"(3), "twenty"(4), "TNT"(4)

The phonetic tokens have a position increment of 0, which indicates that they are at the same position as the token they were derived from (immediately preceding).

CAVERPHONE2 FILTER

This filter creates tokens using caverphone2 phonetic encoding algorithm.

METAPHONE FILTER

This filter creates tokens using metaphone phonetic encoding algorithm.

REFINED SOUNDEX FILTER

This filter creates tokens using refined soundex phonetic encoding algorithm.

SOUNDEX FILTER
This filter creates tokens using soundex phonetic encoding algorithm.

**KEEP WORDS FILTER**

This filter discards all tokens except those that are listed in the given word list. This is the inverse of the Stop Words Filter. This filter can be useful for building specialized indices for a constrained set of terms.

```
{
    "analyzerName": "testanalyzer",
    "filters": [
        {
            "filterName": "keepwordsfilter",
            "parameters": {"filename": "keepwords.txt"}
        }
    ],
    "tokenizer": { "tokenizerName": "standardtokenizer" }
}
```

Where **keepwords.txt** contains:

```
happy
funny
silly
```

In: "Happy, sad or funny"
Tokenizer to Filter: "Happy", "sad", "or", "funny"
Out: "Happy", "funny"

**LENGTH FILTER**

This filter passes tokens whose length falls within the min/max limit specified. All other tokens are discarded.

```
{
    "FilterName": "lengthfilter",
    "Parameters": {
        "min": "3",
        "max": "7"
    }
}
```

In: "turn right at Albuquerque"
Tokenizer to Filter: "turn", "right", "at", "Albuquerque"
Out: "turn", "right"
LOWER CASE FILTER

Converts any uppercase letters in a token to the equivalent lowercase token. All other characters are left unchanged.

In: "Down With CamelCase"
Tokenizer to Filter: "Down", "With", "CamelCase"
Out: "down", "with", "camelcase"

PATTERN REPLACE FILTER

This filter applies a regular expression to each token and, for those that match, substitutes the given replacement string in place of the matched pattern. Tokens which do not match are passed though unchanged.

```
{
    "FilterName": "patternreplacefilter",
    "Parameters":
    {
        "pattern":"cat",
        "replacementtext":"dog"
    }
}
```

In: "cat concatenate catycat"
Tokenizer to Filter: "cat", "concatenate", "catycat"
Out: "dog", "condogenate", "dogycat"

STOP FILTER

This filter discards, or stops analysis of, tokens that are on the given stop words list.

```
{
    "FilterName": "stopwordsfilter",
    "Parameters":{"filename":"stopwords.txt"}
}
```

Where [stopwords.txt](#) contains:

- happy
- funny
- silly
SYNONYM FILTER

This filter does synonym mapping. Each token is looked up in the list of synonyms and if a match is found, then the synonym is emitted in place of the token. The position value of the new tokens are set such they all occur at the same position as the original token.

```
{
    "FilterName": "synonymfilter",
    "Parameters": {"filename": "synonym.txt"}
}
```

Where `synonym.txt` contains:

```
easy: simple, clear
```

In: "easy"
Tokenizer to Filter: "easy"
Out: "easy", "simple", "clear"

REVERESTRINGFILTER

This filter reverses the tokens.

In: "hello how are you"
Tokenizer to Filter: "hello", "how", "are", "you"
Out: "olleh", "woh", "era", "uoy"

TRIMFILTER

This filter trims leading and/or trailing whitespace from tokens. Most tokenizers break tokens at whitespace, so this filter is most often used for special situations.
Custom Analyzer

Contents

- Strip white space analyzer
- Strip to numbers analyzer
- Address Analyzer
- Stop words file
- Synonyms file

Refer to Text analysis to understand the basics of text analysis. In this article we will deconstruct few analyzers.

Strip White Space Analyzer

Let's start with a simple analyzer which removes all the white spaces from the input. This analyzer is useful for cleaning up ID fields which should not contain any spaces. Not is that we are using keyword tokenizer as we want the input to be treated as a single token rather than being broken down into individual tokens. This is also important as filters are applied to individual tokens coming out of the tokenizer. So if you use a filter like pattern replace it is important to understand the role of the tokenizer as you may not get expected results. In our case we want the regex expression to treat the whole input as single token.

So, in below example we are chaining the output of the keyword tokenizer to a series of lowercase and pattern replace filter.

```json
{
  "AnalyzerName": "stripwhitespaceanalyzer",
  "Tokenizer": {
    "TokenizerName": "keyword"
  },
  "Filters": [
    {
      "FilterName": "lowercase"
    },
    {
      "FilterName": "patternreplace",
      "Parameters": {
        "pattern": "\s+",
        "replacement": ""
      }
    }
  ]
}
```

Strip To Numbers Analyzer
This example is similar to the previous one. Here we are using a pattern to replace all non numeric characters from the input. This type of analyzer is useful for cleaning up telephone numbers where you don't want any non-numeric character to be present in the input. This can be easily extended to mean more complex requirements like remove all the pneumatic characters, reverse the result and just take the first six significant digits. When this logic is applied to a telephone number field, you can easily cater for a lot of bad data in the input. For example telephone numbers starting with area codes or telephone numbers with spaces in them.

```json
{
    "AnalyzerName": "striptonumbersanalyzer",
    "Tokenizer": {
        "TokenizerName": "keyword"
    },
    "Filters": [
        {
            "FilterName": "standard"
        },
        {
            "FilterName": "patternreplace",
            "Parameters": {
                "pattern": "[a-z$ ]",
                "replacement": ""
            }
        }
    ]
}
```

**Address Analyzer**

Let's look at a slightly more complex example, here we want ensure that the input does not contain any common stop words and we also want the search to return results when a synonym is used. For example, we want road to be matched to its common short form rd.

Again the setup is similar, we define a tokenizer followed by a set of filters. Here we are using standard tokenizer which is a general-purpose tokenizer. In our filter chain we first pass the tokens through a lowercase filter which converts the tokens into lowercase. This ensures that our searches are case insensitive. The output from the lowercase filter is passed to a stop words filter which removes common stop words like a, an etc. the list of stop words is coming from a text file called stopwords.txt. This file should be placed under resources folder inside conf folder.

Refer to Configuration to know more about FlexSearch's folder structure.

The output tokens from the stop word filter passed into synonym filter which adds additional tokens against a token which is defined in the addresssynonyms.txt. For example if the filter encounters a token containing road, it will replace the token with two tokens road and rd. List of the synonyms can be defined using a text file which should again be placed under the resources folder.
Refer to Analysis to know more about various filters and supported file formats.

STOP WORDS FILE
SYNONYMS FILE

road, rd
street, st
lane, ln
avenue, av, ave
close, cl, cls
drive, drv, dr
court, ct, crt
crescent, cres
place, pl
terrace, terr
gardens, gdns
Index Model

The below table represents all the fields that are present in the FlexSearch search query object. This object is used whenever you wish to execute a search against the engine.

<table>
<thead>
<tr>
<th>Model Definition</th>
<th>Required Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IndexName</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Fields</strong></td>
<td>array of [string]</td>
</tr>
<tr>
<td><strong>predefinedQueries</strong></td>
<td>array of [string]</td>
</tr>
<tr>
<td><strong>shardConfiguration</strong></td>
<td>Complex Object, Please refer: API Reference</td>
</tr>
<tr>
<td><strong>IndexConfiguration</strong></td>
<td>Complex Object, Please refer: API Reference</td>
</tr>
<tr>
<td><strong>active</strong></td>
<td>boolean</td>
</tr>
</tbody>
</table>

For more indepth description of properties please refer to the API reference.

All the functionality related to indexes is exposed on the `indices` endpoint. Some of the important endpoints are shown in the below table. For a complete list of the endpoints please refer to the API reference guide.

<table>
<thead>
<tr>
<th>Endpoint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GET \indices</code></td>
<td>Get all indices</td>
</tr>
<tr>
<td><code>GET \indices\{indexName\}</code></td>
<td>Get an index by ID</td>
</tr>
<tr>
<td><code>POST \indices</code></td>
<td>Create an index</td>
</tr>
<tr>
<td><code>DELETE \indices\{indexName\}</code></td>
<td>Delete an index by ID</td>
</tr>
</tbody>
</table>

These not beat index endpoint as the index object cannot be updated completely but certain properties can be updated. For example you can add a new feel to the index but you can't remove an existing field from an index. This is due to design as the data is not saved in a manner which allows removal of a particular field. We would advise you to re-index in case you want to remove a field from the index.

Index Properties
FIELDs

Field represents the fields to be added to the index. Refer to: Field Types to understand more about adding fields to an index.

PREDEFINED QUERIES

Predefined Query is an extension of normal searching capability of FlexSearch which allows central management of queries. It is also used by background duplicate matching. Think of it as a way to define a search criteria which is managed at the server and can be called from various systems without the need to specify the criteria as the as a part of the query. Refer to: Predefined Query to understand more about predefined queries.

SHARD CONFIGURATION

Shard configuration object contains all the shard related settings. Currently it only supports one property that is `shardCount` which signifies the total number of shards that an index should be split into.

INDEX CONFIGURATION

Index configuration object contains all the index related settings.

These are advance settings which should only be modified if are sure about the changes you are making. Changes to Index settings can have unintended effects.
<table>
<thead>
<tr>
<th>Required Properties</th>
<th>Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>commitTimeSeconds</strong></td>
<td>integer</td>
<td>60</td>
</tr>
<tr>
<td><strong>deleteLogsOnClose</strong></td>
<td>boolean</td>
<td>true</td>
</tr>
<tr>
<td><strong>commitOnClose</strong></td>
<td>boolean</td>
<td>true</td>
</tr>
<tr>
<td><strong>autoCommit</strong></td>
<td>boolean</td>
<td>true</td>
</tr>
<tr>
<td><strong>DirectoryType</strong></td>
<td>string (Enum: FileSystem MemoryMapped Ram)</td>
<td>MemoryMapped</td>
</tr>
<tr>
<td><strong>DefaultWriteLockTimeout</strong></td>
<td>integer</td>
<td>1000</td>
</tr>
<tr>
<td><strong>RamBufferSizeMb</strong></td>
<td>integer</td>
<td>100</td>
</tr>
<tr>
<td><strong>maxBufferedDocs</strong></td>
<td>integer</td>
<td>3</td>
</tr>
<tr>
<td><strong>refreshTimeMilliseconds</strong></td>
<td>integer</td>
<td>500</td>
</tr>
<tr>
<td><strong>autoRefresh</strong></td>
<td>boolean</td>
<td>true</td>
</tr>
<tr>
<td><strong>IndexVersion</strong></td>
<td>string (Enum: FlexSearch_1A FlexSearch_1B)</td>
<td>FlexSearch_1B</td>
</tr>
<tr>
<td><strong>allowReads</strong></td>
<td>boolean</td>
<td>true</td>
</tr>
<tr>
<td><strong>allowWrites</strong></td>
<td>boolean</td>
<td>true</td>
</tr>
</tbody>
</table>

For more indepth description of properties please refer to the API reference.

FlexSearch index differentiates between updating the data on the persistent medium that is disk and refreshing the data for returning the search results. What this really means is when you index a document, the document might not be saved to the physical medium immediately but will be saved at the later point of time. This is done to improve the performance of the system.

All the settings related to the persistence of the data contains `commit` in their names.

All the settings related to refreshing/nrt of the data for searches contains `refresh` in their names.

**Commit related setting**

**Commit Time Seconds** (`commitTimeSeconds`)
The amount of time in seconds that FlexSearch should wait before committing changes to the disk. This setting can have massive effect on the throughput as committing too often would reduce the throughput. There is a minor chance of losing data between commits, but this is greatly minimised by using a write ahead transaction log.

**Commit on close** (`commitOnClose`)
Determines whether to commit first before closing an index. This setting should not be changed under normal circumstances. This is only exposed for testing purposes.

**Auto commit** (`autoCommit`)
Under very specific circumstances a user may want to take control of commit. In such cases it is the responsibility of the user to perform periodic commit. Failing to do so well result in large memory usage.
Flush related settings

Flush of documents happens when a threshold is reached. Flushing merely rights the documents to the physical medium but does not commit them. Flushing is done to conserve memory. A commit still needed to make the things permanent.

**Maximum buffered documents** (*maxBufferedDocs*)
The number of buffered added documents that will trigger a flush if enabled.

**RAM buffer size** (*ramBufferSizeMb*)
Determines the amount of RAM that may be used for buffering added documents and deletions before they are flushed to the Directory.

**Delete logs on close** (*deleteLogsOnClose*)
Determines whether to clear all transaction logs before closing an index. This setting Should not be changed under normal circumstances. Changing the setting may result in increased disk usage. This is only used for investigative purposes to see the data which was indexed.

NRT related setting

**Auto refresh** (*autoRefresh*)
This setting is used to enable automatic refreshing of the index reader. What this really means is that the data will be available for searching after the refresh time even though the data hasn't been committed to the physical medium. This setting should not be changed unless you’re working with an index with static data.

**Refresh time** (*refreshTimeMilliseconds*)
The amount of time in milliseconds that FlexSearch should wait before reopening index reader. This helps in keeping writing and real time aspects of the engine separate.
Document Model
Scripting

Contents

- Pre-index script
- Usage
- Example
- Search execution pipeline
- Pre-search script
- Usage
- Example
- Post-search script

FlexSearch scripts are snippets of code that execute a certain logic at a specific moment in the FlexSearch event pipeline. Scripts are written in F# programming language. There are two separate pipelines where a script can be added.

- **Pre-index scripts** - Just before a document is added or modified in an index. This script lets you manipulate the data that is being indexed.

- **Pre-search scripts** - just before a search request is sent to Lucene. This script helps you control the data that is being sent for searching.

Pre-Index Script

Pre-index scripts are F# methods that execute a piece of code just before indexing occurs. They are perfect when you want to populate or update some fields in a document and persist that change. Compared to pre-search scripts, in pre-index scripts the new values of the document fields will be stored in the index and visible each time you search for it.
Pre-index scripts are declared in the same file as the pre-search scripts are, namely `scripts.fsx` in the index configuration folder.

Just like any other script, pre-index scripts are loaded when FlexSearch server starts or when an index is reloaded.

**USAGE**

Pre-index scripts are used when you want to modify the data that will end up in the index during indexing time (when adding or updating documents).

The name of the F# method that holds the pre-index script logic is `preIndex`.

The signature of the method is:

```fsharp
val preIndex : Document -> unit
```

e.g.

```fsharp
let preIndex (document : Document) = ()
```

This translates to a method that takes a `Document` as a parameter and returns nothing (`void`).

The `preIndex` method needs to be placed in the `script.fsx` file located in the index configuration folder. For example, if you have an index named `contact`, the `script.fsx` file would need to be in:
The way the pre-index scripts are designed to work is:

- A `Document` is received for indexing. Its fields are either empty or prepopulated with some data.
- The `preIndex` function runs against the received `Document`. This function will either modify, populate or empty some fields in the document.
- The modified `Document` is then passed to Lucene to index, store the data and make it available for searching.

**EXAMPLE**

Let's then imagine we want to populate the `gender` field from the `employee` index according to the `title` field - if it's *Mr.*, then it's *Male*, otherwise *Female*. Here is part of the definition of the `employee` index:

```json
{
    "indexName": "employee",
    "fields": [
        {
            "allowSort": false,
            "fieldName": "gender",
            "fieldType": "Text",
            "indexAnalyzer": "standard",
            "searchAnalyzer": "standard",
            "similarity": "TFIDF"
        },
        {
            "allowSort": false,
            "fieldName": "title",
            "fieldType": "Text",
            "indexAnalyzer": "standard",
            "searchAnalyzer": "standard",
            "similarity": "TFIDF"
        },
        ...
    ]
}
```

We would then write the following code in the `scripts.fsx` file.
module Script

open FlexSearch.Api.Model
open Helpers
open System

let preIndex (document : Document) =
    // Get the title field value from the document and make it lowercase
    match document.Get("title").ToLower() with
    // If it's 'Mr.' then set the gender to Male
    | "mr." -> document.Set("gender", "Male")
    // If it's empty then don't set the gender
    | "" -> ()
    // Anything else is Female
    | _ -> document.Set("gender", "Female")

The new piece of functionality will be loaded the next time FlexSearch is restarted or when the index is reloaded (closed then opened back).

Search Execution Pipeline

Let’s examine the search execution pipeline from scripting perspective. There are various places where a user can insert custom logic to modify the search behaviour. The first logical places Pre-search script. This can be used to perform operations like:

- Modify the input search data. This is helpful in cases where a user wants to enforce certain results. For example when a user searches for a specific input you may only want to return a specific result. In these cases you can investigate the incoming data and set the right parameters so that the platform always returns the right result. This stage can also be used to set default values for fields in case no data is provided. Another example would be to normalize an incoming telephone number to a certain format before submitting the search...

- Modify the search query parameters.

- Update the search profile name based on some dynamic condition.
Pre-Search Script

A FlexSearch pre-search script is a snippet of F# code that gets executed before the search is run.

The source code of the script is loaded from the `script.fsx` file from the index configuration folder. A PreSearch script is a method within the `script.fsx` file that has a name starting with `preSearch` (e.g. `preSearchTest`).

Search scripts are loaded automatically on startup or when an index is opened. To add a new script you need to either restart the system or reload the index (`close`, then `open` it back again).

**USAGE**

Pre-search scripts are stored in a `script.fsx` file located in the index configuration folder. For example, if you have an index named `contact`, you would write the `script.fsx` file at:

```
/conf/indices/contact/script.fsx
```

Pre-search script methods have the following naming convention:
preSearch<\texttt{script\_name}>  
\texttt{e.g. preSearchTest}

And the following signature:

\begin{verbatim}
val preSearch<\texttt{script\_name}> : SearchQuery -> unit 
\end{verbatim}

\texttt{e.g. let preSearchTest (query: SearchQuery) = ()}

The above signature means that we should have a function that takes a \texttt{SearchQuery} and returns nothing (\texttt{void}).

Having a \texttt{FlexSearch.Api.Model.SearchQuery} as a parameter gives you access to get or even set any of its properties. For example you can modify the columns to retrieve, or execute some code conditionally based on the query name, or modify the query string, etc.

Probably the most powerful feature is the access to the \texttt{Variables} (the \texttt{@variable\_name} pieces from the query string) property from the \texttt{SearchQuery}. This means you can modify the values that get passed into the query string.

\textbf{EXAMPLE}

Let’s say you want to bring all the employees that have been in the company for more than 10 years. We assume that we have an \texttt{employee} index with the following fields:

- \texttt{year\_joined}
- \texttt{name}

You can initially write the following query on the \texttt{employee} index:

\begin{verbatim}
gt(year\_joined, ’2006’)
\end{verbatim}

And it will work just fine. But next year you’ll realize you have to change the query from \texttt{2006} to \texttt{2007}. So here comes the variable to the rescue:

\begin{verbatim}
gt(year\_joined, @tenYearsAgo)
\end{verbatim}

In this case you would add an entry in the \texttt{SearchQuery.Variables} dictionary for:

\begin{verbatim}
searchQuery.Variables.Add("tenyearsago", DateTime.Now.AddYears(-10).Year)
\end{verbatim}

And then you would submit this search query to FlexSearch. This would work as well, but maybe it would be
easier if you would just pass the current year and then subtract 10 from it before submitting the search. You could reuse this piece of functionality for other queries as well. You can do this using a pre-search script!

You modify the search query string like so:

```plaintext
gt(yearJoined, @year)
```

And you create a new file in the `Conf/Indices/employee` folder called `script.fsx` in which you add the following code:

```fsharp
module Script

open FlexSearch.Api.Model
open Helpers
open System

let preSearchTenYearsAgo (query : SearchQuery) =
    // Get the variable called "year". Take care of upper vs lower case.
    let kvp = query.Variables |> Seq.find (fun kv -> kv.Key.ToLower() = "year")
    // Modify / populate its value
    kvp.Value <- DateTime.Now.AddYears(-10).Year
```

Lastly, before submitting the `SearchQuery` you just need to specify that you want to use the newly created pre-search query:

```fsharp
searchQuery.PreSearchScript = "TenYearsAgo";
```

### Post-Search Script

Post-search script is still in development stage and we will add relevant documentation once the feature is finalized.
Search
The below table represents all the fields that are present in the FlexSearch search query object. This object is used whenever you wish to execute a search against the engine.
Model Definition

Required Properties

- `IndexName`
- `queryString`

<table>
<thead>
<tr>
<th>All Properties</th>
<th>Type</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>queryName</code></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td><code>columns</code></td>
<td>array of string</td>
<td></td>
</tr>
<tr>
<td><code>count</code></td>
<td>integer</td>
<td>10</td>
</tr>
<tr>
<td><code>Highlights</code></td>
<td>Complex Object, Please refer: API Reference</td>
<td></td>
</tr>
<tr>
<td><code>IndexName</code></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td><code>orderBy</code></td>
<td>string</td>
<td>score</td>
</tr>
<tr>
<td><code>orderByDirection</code></td>
<td>string (Enum: Ascending, Descending)</td>
<td>Ascending</td>
</tr>
<tr>
<td><code>cutOff</code></td>
<td>number</td>
<td></td>
</tr>
<tr>
<td><code>distinctBy</code></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td><code>skip</code></td>
<td>integer</td>
<td></td>
</tr>
<tr>
<td><code>queryString</code></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td><code>ReturnScore</code></td>
<td>boolean</td>
<td>true</td>
</tr>
<tr>
<td><code>preSearchScript</code></td>
<td>string</td>
<td></td>
</tr>
<tr>
<td><code>overridePredefinedQueryOptions</code></td>
<td>boolean</td>
<td>false</td>
</tr>
<tr>
<td><code>returnEmptyStringForNull</code></td>
<td>boolean</td>
<td>true</td>
</tr>
<tr>
<td><code>variables</code></td>
<td>object of map&lt;string, string&gt;</td>
<td></td>
</tr>
</tbody>
</table>

For more indepth description of properties please refer to the API reference.

The search functionality is exposed over both `get` and `post` request types at the below endpoints:

GET `/indices/{indexName}/search?q={queryParameters}` HTTP/1.1

POST `/indices/{indexName}/search?q={queryParameters}` HTTP/1.1

```json
{
  "QueryString" : ""
}
```

Model Properties Description

In this section we will go through some of the more important properties of the search query model. Index name and query string are the only two fields which are mandatory.
COLUMNS

This property is used to define the columns which should be returned as part of search results.

- If this property is omitted then no columns will be returned as we would like users to be explicit about the requirement.

- `*` can be passed if you need all the available columns.

QUERY STRING PARAMETER

Columns can also be requested using `columns` in the URI parameters. This parameter is a string and not an array.

```
GET /indices/search?q=col1,col2 HTTP/1.1
```

```
GET /indices/search?q=* HTTP/1.1
```

Paging Properties

ORDERBY & ORDERBYDIRECTION

`OrderBy` and `OrderByDirection` parameters can be used to sort the results coming from the engine. By default the sort order is relevance. This means the records which are more relevant to the search query will be at the top followed by less relevant results. The `_score` property in the returned document is the quantification of the relevance.

SKIP & COUNT

Skip and count provides basic paging capability to the engine. Count is used to define the number of results that should be returned by a query. When count is used in conjunction with skip, it acts as page count property. Thus defining the number of records but page.

Filtering Properties

Filtering properties are a special set of properties which are used to filter out the results returned by the engine. This filtering happens after the main search logic has been executed and before returning the results to the user. Think of it as a way of looking at a table of result and identifying the ones which don’t meet the specific criteria. So this is excluding the documents from the result was search rather then excluding them through the search criteria.

The below diagram explains the concept. In the platform box once the platform logic is finished the sequence of documents are fed into the filtering criteria engine.
CUTOFF

Cut off criteria is used to filter out all the documents which score below a certain percentage. Due to the nature of Lucene it is not possible to have deterministic scores. So, here the percentages are calculated in respect of the highest scoring document. Thus keeping the cut-off relevant to the particular search query.

This feature is especially useful for duplicate detection. For example you want to identify duplicates of a given document. You already know that this particular document exist in the index. You can execute a search where this document should come out at the top of the search results. Then you can simply define that any other document which is within 10% range of this document can be deemed as duplicate. The formula used for cut-off is:

\[ \text{Cut-off value} = \frac{\text{Current Document Score}}{\text{Max score}} \times 100 \]

Cut-off value should be defined between 1 to 100.

DISTINCTBY

Think of distinct by filter as the distinct by clause of SQL. Like other filters it runs post search and removes all the documents where the value of a particular field is not unique.

For example, you want to get a list of customers who purchase from you more than once but, you only want to get one customer per postcode. In such a situation distinct by will filter out all records where the value of postcode is not unique. The order by clause will dictate which records are filtered out.
Control Properties

RETURNSCORE

This property signifies if the scored associated with the document should be returned as part of the search result.

PRESEARCHSCRIPT

This property can be used to define the name of the script which should be executed before executing the core logic. The script should be present on the server in order to be picked up. Refer to predefined query section to know more about this feature.

VARIABLES

Variables are a way to provide dynamic values to the search query. You can simply define variables in your input query using @ symbol. For example the below query has a variable called fname.

```
allOf(firstname, @fname)
```

The advantages of this approach is that you can provide dynamic values to a query.

RETURNEMPTYSTRINGFORNULL

This property is useful when you don't want to return null objects as part of your JSON response. In case of null values the engine will simply return blank strings.

OVERRIDEPREDEFINEDQUERYOPTIONS

Predefined query's are essentially a search query which has an associated name. Just like a normal query a predefined query can be configured using all the properties available on a normal search query object. There are times when you may want to override the values defined in the predefined query with the query object that you are passed to the server to execute the search.

For example, let's say in a predefined query called findcustomer you have set the number of results to be returned to be 5. In a particular case you want to get more than 5 results back, in such a case you can set overridePredefinedQuery option to true and set the count property of the query to 10.
Query Format

Contents

- Identifier
- Examples
- Constant
- Examples
- Variable
- Examples
- Switch
- Examples
- Global Switch
- MatchAll switch
- MatchNone switch
- MatchFieldDefault switch
- UseDefault switch
- Boost switch
- ConstantScore switch
- NoScore switch
- Query specific switch
- Condition
- Examples
- Query
- EBNF Format

FlexSearch utilizes custom query format which enables advance customization with minimal effort. Score manipulation, short circuiting of clauses etc can be achieved by using simple switches.

Let’s start understanding Query syntax by going through some basic stuff.

Identifier

An identifier is any set of alphanumeric characters without (, ), and space characters.

Any ALPHANUMERIC character except ( or ) or space

Identifiers are used to represent field names and query names in the engine.

```
EXAMPLES
firstname, allOf, anyOf
```

Constant
A constant is any set of unicode characters between single quote. Back slash can be used to escape a single quote in the input. The reason to use Single Quote to represent constants in the engine is to allow easy embedding of the queries in JSON objects.

A constant is used to represent search values in a query.

**EXAMPLES**

'United Kingdom', Andrew's Car

**Variable**

A variable is an identifier preceded by a @ character.

Variables are used to represent dynamic values in a query. These values can be passed by user or can be calculated using scripts etc.

**EXAMPLES**

Example: @firstname, @exchangerate

**Switch**

A switch is a key value pair where the value part is optional. It is used for configuring the query behaviour.

**EXAMPLES**

-filter, -matchall, -constantscore '2'
Global Switch

These are global in sense that these can be applied to any query type.

MATCHALL SWITCH

This switch basically short circuits the query in case no value is provided for the field to be searched. This is useful if you don't want a condition to be applicable when there is no value to be searched.

```
anyOf(lastname, 'smith', 'doe') AND anyOf(firstname, @firstname, -matchall)
```

In the above example if the user doesn't provide a value for the variable `@firstname` then the condition will be ignored. The query will effectively be short circuited to:

```
anyOf(lastname, 'smith', 'doe') AND *
```

This construct is useful when preforming duplicate detection over a set of uncleansed data. So, the queries can easily handle missing values.

MATCHNONE SWITCH

This switch is basically the reverse of MatchAll switch, it forces no match in case no value is provided for the field to be searched. This is useful if you don't want a condition to be match anything when there is no value to be searched.

```
anyOf(lastname, 'smith', 'doe') AND anyOf(firstname, @firstname, -matchnone)
```

In the above example if the user doesn't provide a value for the variable `@firstname` then the condition will be force the clause to match no documents. The query will effectively be short circuited to:

```
anyOf(lastname, 'smith', 'doe') AND anyOf(firstname, 'non existing value')
```

MATCHFIELDDEFAULT SWITCH

This switch uses the field's default value in case no value is provided for the field to be searched. This is useful if you don't want a condition to be match anything but the field's default value when there is no value to be searched.

```
anyOf(lastname, 'smith', 'doe') AND anyOf(firstname, @firstname, -matchFieldDefault)
```

In the above example if the user doesn't provide a value for the variable `@firstname` then the condition will be force the clause to use null as the search value. The query will effectively transformed to:
If the field is a numeric type then the corresponding default numeric value will be used.

**USEDEFAULT SWITCH**

This switch uses the default value provided in the switch in case no value is provided for the field to be searched. This is useful if you don't want a condition to be match anything but the default value when there is no value to be searched.

```
anyOf(lastname, 'smith', 'doe') AND anyOf(firstname, 'null')
```

In the above example if the user doesn't provide a value for the variable `firstname` then the condition will be force the clause to use jimmy as the search value. The query will effectively transformed to:

```
anyOf(lastname, 'smith', 'doe') AND anyOf(firstname, 'jimmy')
```

**BOOST SWITCH**

This switch boosts the score of a matching condition by a factor provided as part of the switch.

```
anyOf(lastname, 'smith', 'doe') AND anyOf(firstname, 'roger', -boost '2')
```

In the above example the score of the 2nd anyOf condition will be increased by a factor of 2 if the firstname matches roger. This is useful to improve the relative priority of certain conditions compared to other conditions.

**CONSTANTSORE SWITCH**

This switch provides a constant score to a matching condition.

```
anyOf(lastname, 'smith', 'doe') AND anyOf(firstname, 'roger', -constantScore '2')
```

In the above example the score of the 2nd anyOf condition will have a score of 2 if the firstname matches roger.

**NOSCORE SWITCH**

This switch removes any score associated with a matching condition. This is equivalent to forcing the condition to act as a filter.
In the above example the 2nd anyOf condition will not contribute to the overall score.

This is useful to remove filtering clauses from contributing to the overall score. For example if you have a field called state which saves the state of the record. You would not like it to contribute to the overall score as most of the records will have it set to 'active'.

**Query Specific Switch**

These are specific to a particular query type and are used to fine tune the query behaviour. Applying these to unsupported query types will not result in an error but will definitly produce unexpected behaviour.

**Condition**

A condition is the smallest unit of a query which specifies the search criteria to be applied for a single field. A single condition is a valid query.

A condition at minimum requires:

- Name of the query
- The field on which the query is to be applied
- Atleast a single source of value for the query. This can come from a variable or constant.

```
anyOf(lastname, 'smith', 'doe') AND anyOf(firstname, 'roger', -noScore)
```

FlexSearch supports a number of query operators, more explanation about these can be accessed from the Query Types section.

**Query**

A query is basically a group of conditions which can be combined together with AND, OR and parentheses.
Purely negative queries (i.e. queries with top level Not operation) are not supported.

The parser implements operator precedence as \texttt{NOT} \texttt{>>} \texttt{AND} \texttt{>>} \texttt{OR}.

**EBNF Format**

Below is the Query syntax in EBNF format. Copy and paste it to [http://www.bottlecaps.de/rr/ui](http://www.bottlecaps.de/rr/ui) to generate the above railroad diagrams.

```ebnf
Query ::= Condition ('OR' Query | 'AND' Query)? | '(' Query ')'
Condition ::= 'NOT'? QueryOperator '(' FieldName (',' (Variable | Constant | Switch))+ ')
Identifier ::= ("Any ALPHANUMERIC character except ( or ) or space")+
Constant ::= "" ("Any UNICODE character except '"' | "\"" single quote")* ""
Variable ::= '@' Identifier
Switch ::= '-' Identifier Constant?
QueryOperator ::= Identifier
FieldName ::= Identifier
```

71
How query works?

In order to understand how the different query types work, it would be beneficial to understand the basic processing that happens before a query is processed. The following flow chart defines the steps that a query string goes through before being submitted to the underlying Lucene engine:
1. The query string is processed using a query parser which generates an abstract syntax tree from the input. Think of abstract syntax tree as a machine understandable representation of the original query string. The AST contains information about the query types and the clauses.

2. Process each individual condition from the AST.

3. Each condition goes through multiple validation steps which includes things like checking that the field name is valid, the query type is valid and valid field values are provided. If no valid value is specified then the query processor looks for the defined missing field behaviour.

4. Each query type has different behaviour associated with it. For example some query types are designed to work with multiple tokens, phrases, positional matching etc. Once we have identified the correct query type for the condition then it is checked if the query type supports search analyzer. Certain query types avoid search time analysis to avoid tampering with the input. For example `like` query type avoids search analyzer as the analyzer may remove certain special characters like `?` and `*`.

5. In case the query type supports an analyzer then the provided analyzer is used or the input text is converted to the internal representation for the given field type. For example Boolean fields are internally saved as `T` and `F` to represent true and false values. This is done to save disk space. These fields don't support sophisticated analysis so they are configured not to use an analyzer. In case the query type does not support a search time analyzer then the input value is simply converted into lowercase to force case insensitive matching.

6. The newly passed tokens are passed into the query type processor which generates a Lucene equivalent query from it. A query type processor is also responsible for processing query type specific switches.
7. Once all the query types are processed then a master Lucene query is built which is executed to produce search results.

**General Query Properties**

These properties are specified for each query type in the information box provided at the beginning of each query documentation.

**SEARCH ANALYZER SUPPORT & BOOLEAN GENERATION BEHAVIOUR**

This property signifies if a query type supports search time analysis or not? As explained earlier some query types bypass search time analysis to avoid removal of special characters from the input text.

To illustrate an example of search analyzer support, when the below query is executed:

```
allof(agriproducts, 'Rice wheat', 'BARLEY')
```

It will be converted into three queries as shown below:

```
allof(agriproducts, 'rice') AND allof(agriproducts, 'wheat') AND allof(agriproducts, 'BARLEY')
```

Here are two things have happened:

1. The original query has two tokens **Rice wheat** and **BARLEY**. After going through the search analyzer the two tokens will get converted into three tokens (this is assuming that a standard analyzer is used at search time). The three new tokens are: **rice**, **wheat** and **BARLEY**.

2. The second thing which happens is the Boolean behaviour. In this case the three tokens are joined using a **AND** operator. Each query type has different configured behaviour when it comes to generating Boolean query. Boolean queries are automatically generated when possible.

**POSITIONAL MATCH SUPPORT**

This property signifies if the query type supports positional matching of the tokens in the field value. Query types which do not support positional matching will match a given token or tokens anywhere in the corpus. Usually the query types which support positional matching will associate special meaning to the order in which the field values are specified for a given query type.

**FIELD VALUES ORDER**

This property signifies if the query type gives any special emphasis to the order in which multiple field values are specified. This is usually associated with the positional match support.

For example, the AllOf query supports multiple field values per clause and the order in which the field values are specified does not matter.
MULTIPLE FIELD VALUES PER CLAUSE

This property signifies if a user can pass multiple search values but clause. This is helpful in simplifying the query in case the same query type is to be used for multiple values.

In the above example the query type has support for multiple field values per clause. Here we have passed rice and wheat as two separate field values in the same query clause.

**AllOf**

**AllOf** query is the simplest of the term related queries which forces all the specified terms to match in a given input. This query does not take position in consideration and will match terms out of order.

<table>
<thead>
<tr>
<th>Information Box</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Search Analyzer</td>
<td>Supported</td>
</tr>
<tr>
<td>Boolean behaviour</td>
<td>And</td>
</tr>
<tr>
<td>Positional match</td>
<td>Unsupported</td>
</tr>
<tr>
<td>Field Value order</td>
<td>Does not matter</td>
</tr>
<tr>
<td>Multiple field values per clause</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**QUERY EXAMPLES**

The following search query returns all documents containing wheat and Rice both, in the agriproducts field.
AllOf single clause with a single token

```
allof(agriproducts, 'rice wheat')
```

The above query is semantically similar to the below queries:

```
allof(agriproducts, 'rice') and alloxof(agriproducts, 'wheat')
```

```
allof(agriproducts, 'rice', 'wheat')
```

```
allof(agriproducts, 'wheat rice')
```

AnyOf

AnyOf query is the simplest of the term related queries which forces one of the specified term to match in a given input. This query does not take position in consideration and will match terms out of order.

<table>
<thead>
<tr>
<th>Information Box</th>
<th>Supported</th>
<th>Unsupported</th>
<th>Does not matter</th>
<th>Yes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search Analyzer</td>
<td>Boolean behaviour</td>
<td>Positional match</td>
<td>Field Value order</td>
<td>Multiple field values per clause</td>
</tr>
</tbody>
</table>

QUERY EXAMPLES

The following search query returns all documents containing \texttt{wheat} or \texttt{Rice} or both, in the \texttt{agriproducts} field.
AnyOf single clause with a single token

\[ \text{anyOf} (\text{agriproducts}, 'rice\ wheat') \]

The content of the section is optimized for web viewing. Please access the content from the website.

The above query is semantically similar to the below queries:

\[ \text{anyOf} (\text{agriproducts}, 'rice') \text{ and } \text{anyOf} (\text{agriproducts}, 'wheat') \]

\[ \text{anyOf} (\text{agriproducts}, 'rice', 'wheat') \]

\[ \text{anyOf} (\text{agriproducts}, 'wheat\ rice') \]

Phrase Match

A Query that matches documents containing a particular sequence of terms.

<table>
<thead>
<tr>
<th>Information Box</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Search Analyzer</strong></td>
</tr>
<tr>
<td><strong>Boolean behaviour</strong></td>
</tr>
<tr>
<td><strong>Positional match</strong></td>
</tr>
<tr>
<td><strong>Field Value order</strong></td>
</tr>
<tr>
<td><strong>Multiple field values per clause</strong></td>
</tr>
</tbody>
</table>

**QUERY EXAMPLES**

The following search query returns all documents containing the 3 words [Federal parliamentary democracy](#) is exactly the same order.
Phrase search passing multiple words as single token

```
phraseMatch(governmenttype, 'federal parliamentary democracy')
```

Unlike the previous query types, phrase match input has positional relevance. Here instead of passing a single token as "Federal parliamentary democracy", if we pass them as three tokens the overall result will be different as the query will be treated as 3 phrase match queries and will be joined using an OR operator.

Be careful with phrase matches as the order of token and the number of tokens can affect the search results drastically.

Phrase search passing multiple words as multiple tokens

```
phraseMatch(governmenttype, 'federal', 'parliamentary', 'democracy')
```

Phrase query also supports `slop` parameter. By default the slop is set to 0 which means match in exact order. A minimum slop of 2 is required to change the order of the terms.

Specifying `slop` in phrase query does not maintain the order of the terms. The query is reduced to a term query with the terms being in the specified range of each other.

Phrase search with slop of 4

```
phraseMatch(governmenttype, 'parliamentary monarchy', -slop 4)
```

The content of the section is optimized for web viewing. Please access the content from the website.
Below query demonstrated the behaviour when slop is used to match the same words from the above query but in reverse order.

**Phrase search with slop of 4**

```
phraseMatch(governmenttype, 'monarchy parliamentary', -slop '4')
```

The content of the section is optimized for web viewing. Please access the content from the website.

Phrase match query also supports an additional switch: `multiphrase`. This switch can be used to enforce additional positional matching at the same position. For example let's say we want to match word `parliamentary` followed by either `democracy` or `system`. This can be easily accomplished by using `multiphrase` switch.

**Match both phrases containing 'parliamentary democracy' and 'parliamentary system'**

```
phraseMatch(governmenttype, 'parliamentary', 'democracy system', -multiphrase)
```

The content of the section is optimized for web viewing. Please access the content from the website.

**Match phrases containing 'parliamentary democracy', 'parliamentary system' and 'parliamentary constitutional'**

```
phraseMatch(governmenttype, 'parliamentary', 'democracy system constitutional', -multiphrase)
```

The content of the section is optimized for web viewing. Please access the content from the website.
Match phrases containing 'parliamentary monarchy' and 'constitutional monarchy'

```plaintext
phraseMatch(governmenttype, 'constitutional parliamentary', 'monarchy', -multiphrase)
```

The content of the section is optimized for web viewing. Please access the content from the website.

Fuzzy

Implements the fuzzy search query. The similarity measurement is based on the Damerau-Levenshtein (optimal string alignment) algorithm. At most, this query will match terms up to 2 edits. Higher distances, are generally not useful and will match a significant amount of the term dictionary.

Information Box

<table>
<thead>
<tr>
<th>Search Analyzer</th>
<th>Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean behaviour</td>
<td>Or</td>
</tr>
<tr>
<td>Positional match</td>
<td>Unsupported</td>
</tr>
<tr>
<td>Field Value order</td>
<td>Does not matter</td>
</tr>
<tr>
<td>Multiple field values per clause</td>
<td>Yes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prefixlength</td>
<td>0</td>
<td>int</td>
<td>Length of common (non-fuzzy) prefix.</td>
</tr>
<tr>
<td>slop</td>
<td>1</td>
<td>int</td>
<td>The number of allowed edits</td>
</tr>
</tbody>
</table>

QUERY EXAMPLES

The following search query returns all documents containing Iran and all documents containing Iran with 1 character difference, in the countryname field.
Fuzzy with default slop of 1

```
fuzzy(countryname, 'Iran')
```

The content of the section is optimized for web viewing. Please access the content from the website.

The following search query demonstrates the use of `slop` operator. It returns all countries similar to `China` with a difference of two characters.

Fuzzy with slop of 2

```
fuzzy(countryname, 'China', -slop '2')
```

The content of the section is optimized for web viewing. Please access the content from the website.

Like

Implements the wildcard search query. Supported wildcards are `*`, which matches any character sequence (including the empty one), and `?`, which matches any single character. Note this query can be slow, as it needs to iterate over many terms.

### Information Box

<table>
<thead>
<tr>
<th>Search Analyzer</th>
<th>Unsupported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean behaviour</td>
<td>Or</td>
</tr>
<tr>
<td>Positional match</td>
<td>Unsupported</td>
</tr>
<tr>
<td>Field Value order</td>
<td>Does not matter</td>
</tr>
<tr>
<td>Multiple field values per clause</td>
<td>Yes</td>
</tr>
</tbody>
</table>

In order to prevent extremely slow Wildcard Queries, a Wildcard term should not start with the wildcard `*`.

### QUERY EXAMPLES

The following search query returns all documents with `uni` coming anywhere in the word.
Like using "" operator

```
like(countryname, 'uni*')
```

The following query will match any word where it starts with \textit{Unit} followed by any single character and ends with \textit{d}.

Like with single character operator

```
like(countryname, 'unit?d')
```

The content of the section is optimized for web viewing. Please access the content from the website.

Regex

A fast regular expression query based on the \texttt{org.apache.lucene.util.automaton} package. Comparisons are fast.

<table>
<thead>
<tr>
<th>Information Box</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Search Analyzer</td>
<td>Unsupported</td>
</tr>
<tr>
<td>Boolean behaviour</td>
<td>Or</td>
</tr>
<tr>
<td>Positional match</td>
<td>Unsupported</td>
</tr>
<tr>
<td>Field Value order</td>
<td>Does not matter</td>
</tr>
<tr>
<td>Multiple field values per clause</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The term dictionary is enumerated in an intelligent way, to avoid comparisons. The supported syntax is documented in the Java RegExp class.

This query can be slow, as it needs to iterate over many terms. In order to prevent extremely slow RegexpQueries, a RegExp term should not start with the expression \texttt{\textasciitilde}.

QUERY EXAMPLES
The following search query matches all the documents containing silk and milk.

**Simple regex match**

```regex(agriproducts, '[ms]ilk')```

The content of the section is optimized for web viewing. Please access the content from the website.

**Numeric Range Operator**

A Query that matches numeric values within a specified range. To use this, you must first index the numeric values using Int, Long, DateTime, Date or Double.

**Information Box**

<table>
<thead>
<tr>
<th>Search Analyzer</th>
<th>Unsupported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean behaviour</td>
<td>Unsupported</td>
</tr>
<tr>
<td>Positional match</td>
<td>Unsupported</td>
</tr>
<tr>
<td>Field Value order</td>
<td>Should be the first token</td>
</tr>
<tr>
<td>Multiple field values per clause</td>
<td>No</td>
</tr>
</tbody>
</table>

Range supports **gt** (greater than), **ge** (greater or equal), **lt** (less than) and **le** (less or equal) functions.

**QUERY EXAMPLES**

- `gt(population, '1000000')`

- `ge(population, '1000000')`

- `lt(population, '1000000')`

- `le(population, '1000000')`

**Match All**
A query that matches all documents. It is a useful query to iterate over all documents in an index.

<table>
<thead>
<tr>
<th>Information Box</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Search Analyzer</td>
<td>Unsupported</td>
</tr>
<tr>
<td>Boolean behaviour</td>
<td>Unsupported</td>
</tr>
<tr>
<td>Positional match</td>
<td>Unsupported</td>
</tr>
<tr>
<td>Field Value order</td>
<td>Unsupported</td>
</tr>
<tr>
<td>Multiple field values per clause</td>
<td>Unsupported</td>
</tr>
</tbody>
</table>

**QUERY EXAMPLES**

The following search query matches all the documents in the index.

```
matchall(countryname, '*')
```

**Match None**

A query that matches no documents. It is a useful query to ensure that a clause never matched anything under specific conditions.

<table>
<thead>
<tr>
<th>Information Box</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Search Analyzer</td>
<td>Unsupported</td>
</tr>
<tr>
<td>Boolean behaviour</td>
<td>Unsupported</td>
</tr>
<tr>
<td>Positional match</td>
<td>Unsupported</td>
</tr>
<tr>
<td>Field Value order</td>
<td>Unsupported</td>
</tr>
<tr>
<td>Multiple field values per clause</td>
<td>Unsupported</td>
</tr>
</tbody>
</table>
Predefined Query

Contents
- Advantages
- Example
- Working with predefined queries

Predefined Query is an extension of normal searching capability of FlexSearch which allows central management of queries. It is also used by background duplicate matching. Think of it as a way to define a search criteria which is managed at the server and can be called from various systems without the need to specify the criteria as the as a part of the query. This allows easy management of queries across many systems. For example, you can define a query which can detect duplicates in your customer data, you can call this query from your various systems like data entry, point of sale etc. If you ever decide to update the criteria you don’t have to redefine the criteria in all the systems.

This is an extremely powerful and useful feature present in the engine. It also allows you to define various kinds of scripts which can be executed before or after the main query is processed, this gives you an easy way to extend the search pipeline.

ADVANTAGES
- Centralized system to store and manage all queries.
- Easily modify queries and the changes will be instant across all references to this query.
- Test it faster using the Search Studio application in the FlexSearch Portal.

Example

This is a part of the country index configuration, showing the predefined query agriSearch.
A predefined query is actually a saved [SearchQuery](#) object. The most important (and mandatory) properties are:

- **QueryName**: this property is only mandatory in the case of Predefined Queries. If this isn't populated, then a [ValidationException](#) error will be returned. It holds the name of the predefined query.
- **QueryString**: describes the query to submit to FlexSearch.
- **IndexName**: the index to execute the query against.
- **Columns**: this property isn't mandatory, though it will default to [], which means no fields would be returned from a search. This property holds the list of fields that you want to be returned by the search.

Here is how the definition would sound in plain English:

- We have a predefined query named **agriSearch**
- Whenever the query is invoked, we want it to return 2 columns: **countryname** and **agriproducts**
- We want the query to return at most **10** records
- The query will search against the **country** index
- The results will be ordered by their **score** in the **ascending** direction
- We will only consider the records that have a score higher than or equal to **0.0**
- We don't want to bring only the records that are distinct. We will take all records.
• We want to skip the first 0 records from the result, i.e. bring all records from the beginning.

• Use the following search query: `allof(agriproducts, 'wheat', 'corn', 'grapes') AND like(countryname, @countryName, -matchall)`

• **Yes**, please include the value of the score within each returned document

• We **don’t** want to use any *PreSearch Script*

• We **don’t** want to override the *predefined query* options that we’ve set here with the options that we submit in the search URL.

  E.g. calling `http://localhost:9800/indices/country?queryname=agrisearch&count=3` will not overwrite the count value of 10 with 3.

• **Yes**, please return an empty string if a field has the `null` value.

• We **aren’t** supplying any values for the variables in the query

  The name of the variable in this particular case (`countryName`) is just a coincidence it’s named the same as a field in the `country` index. You can have any name you want for a variable.

**Working With Predefined Queries**

Please refer to the links in the see also section.
Text Highlighting

Contents

FlexSearch supports text highlighting across all query types provided correct highlighting options are set in the request query. Text highlighting is supported only for **Text** field types.

PreTag and PostTag can be specified and the returned result will contain the matched text between pre and post tags. This is helpful in case the results are to be expressed in a web page.

The content of the section is optimized for web viewing. Please access the content from the website.
Rest
API Basics

Contents

- Formats
- JSONP support

FlexSearch follows the following basic URI convention:

```
[HTTP VERB] http://{servername}:{portnumber}/{resource}/{resourceId}/{sub-resource|operation}
```

For example if the users wants to check the status of index with name `test` then the indexname in the above URI will be replaced by `test` and the operation will become `status`:

```
GET /indices/test/status HTTP/1.1
```

The various Http verbs are mapped to the following operations:

<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>Get a resource</td>
</tr>
<tr>
<td>POST</td>
<td>Create a resource</td>
</tr>
<tr>
<td>PUT</td>
<td>Update/Create a resource</td>
</tr>
<tr>
<td>DELETE</td>
<td>Delete a resource</td>
</tr>
</tbody>
</table>

Formats

FlexSearch supports 2 response formats out of the box:

- JSON
- MessagePack

JSONP Support

To embed the response in a jsonp callback, append `?callback=myCallback`
API Reference

FlexSearch REST Services

Description of all the REST endpoints exposed by FlexSearch.

Methods

List Of Operations

- get /analyzers
- get /analyzers/{analyzerName}
- put /analyzers/{analyzerName}
- delete /analyzers/{analyzerName}
- post /analyzers/{analyzerName}/analyzeText
- get /indices
- post /indices
- get /indices/{indexId}/documents/{docId}
- put /indices/{indexId}/documents/{docId}
- delete /indices/{indexId}/documents/{docId}
- get /indices/{indexName}
- delete /indices/{indexName}
- put /indices/{indexName}/configuration
- post /indices/{indexName}/csv
- get /indices/{indexName}/documents
- post /indices/{indexName}/documents
- delete /indices/{indexName}/documents
- post /indices/{indexName}/duplicatedetection/{predefinedQueryName}
- get /indices/{indexName}/exists
- put /indices/{indexName}/fields
- put /indices/{indexName}/predefinedQuery
- put /indices/{indexName}/refresh
- get /indices/{indexName}/search
- post /indices/{indexName}/search
- get /indices/{indexName}/size
- post /indices/{indexName}/sql
- get /indices/{indexName}/status
- put /indices/{indexName}/status/{status}
- get /jobs/{jobId}
- get /memory
- get /ping
- put /setupdemo

GET ALL ANALYZERS
get /analyzers

Returns all the analyzers present in the system.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Method</td>
<td>get</td>
</tr>
<tr>
<td>SDK Method Name</td>
<td>getAllAnalyzers</td>
</tr>
<tr>
<td>Return type</td>
<td>getAllAnalyzersResponse</td>
</tr>
<tr>
<td>Responses</td>
<td>• 200: An array of analyzers</td>
</tr>
<tr>
<td></td>
<td>• 400: Unexpected error</td>
</tr>
</tbody>
</table>

GET ANALYZER

get /analyzers/{analyzerName}

Returns an analyzer by name.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Method</td>
<td>get</td>
</tr>
<tr>
<td>SDK Method Name</td>
<td>getAnalyzer</td>
</tr>
<tr>
<td>Path parameters</td>
<td>• analyzerName (required): Analyzer name</td>
</tr>
<tr>
<td>Return type</td>
<td>getAnalyzerResponse</td>
</tr>
<tr>
<td>Responses</td>
<td>• 200: OK</td>
</tr>
<tr>
<td></td>
<td>• 400: Bad Request</td>
</tr>
</tbody>
</table>

CREATE OR UPDATE AN ANALYZER

put /analyzers/{analyzerName}

This service is idempotent.
DELETE AN ANALYZER

```
delete /analyzers/{analyzerName}
```

Deletes an analyzer by name.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Method</td>
<td>delete</td>
</tr>
<tr>
<td>SDK Method Name</td>
<td>deleteAnalyzer</td>
</tr>
<tr>
<td>Path parameters</td>
<td>analyzerName (required) : Analyzer name</td>
</tr>
<tr>
<td>Request body</td>
<td>analyzer (required)</td>
</tr>
<tr>
<td>Return type</td>
<td>deleteAnalyzerResponse</td>
</tr>
<tr>
<td>Responses</td>
<td>200 : OK, 400 : Bad Request</td>
</tr>
</tbody>
</table>

[ Jump to Methods ]

ANALYZE INPUT TEXT

```
post /analyzers/{analyzerName}/analyzetext
```

This endpoint is useful to understand the effect of a particular analyzer on the input text. You can use the service with both custom and built-in analyzers. The returned response contains the tokenized input.
**GET ALL INDICES**

```
get /indices
```

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Method</td>
<td><code>get</code></td>
</tr>
<tr>
<td>SDK Method Name</td>
<td><code>getAllIndices</code></td>
</tr>
<tr>
<td>Return type</td>
<td><code>getAllIndicesResponse</code></td>
</tr>
</tbody>
</table>
| Responses     | • `200` : OK  
                • `400` : Bad Request |

[ Jump to Methods ]

**CREATE A NEW INDEX**

```
post /indices
```

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Method</td>
<td><code>post</code></td>
</tr>
<tr>
<td>SDK Method Name</td>
<td><code>analyzeText</code></td>
</tr>
<tr>
<td>Path parameters</td>
<td>• <code>analyzerName</code> (required) : Analyzer name</td>
</tr>
<tr>
<td>Request body</td>
<td><code>analyzeText</code> (required)</td>
</tr>
<tr>
<td>Return type</td>
<td><code>analyzeTextResponse</code></td>
</tr>
</tbody>
</table>
| Responses     | • `200` : OK  
                • `400` : Bad Request |

[ Jump to Methods ]
CREATE OR UPDATE A DOCUMENT

```
put /indices/{indexId}/documents/{docId}
```

It is advisable to use create document endpoint when you are sure that the document does not exist in an index. This service will always perform an ID based lookup to determine if a document already exists. In case of non-unique ID based index, this will replace all the documents with the currently passed document. This endpoint can be used with concurrency control semantics.
### DELETE A DOCUMENT

```
delete /indices/{indexId}/documents/{docId}
```

In case of non-unique ID field, this will delete all the documents associated with that ID.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Method</td>
<td>delete</td>
</tr>
<tr>
<td>SDK Method Name</td>
<td>deleteDocument</td>
</tr>
</tbody>
</table>
| Path parameters | • indexId (required) : Index name  
| Request body   | document (required) |
| Return type    | deleteDocumentResponse |
| Responses      | • 200 : OK 
|               | • 400 : Bad Request |

[ Jump to Methods ]

---

### GET AN INDEX

```
get /indices/{indexName}
```

This service will return a status of 404 when index is not present on the server.
### DELETE AN INDEX

**delete /indices/{indexName}**

Index deletion happens in two parts, first the index configuration file is deleted from the configurations folder, then the index is deleted from the data folder. In case any error is encountered the cleanup will be performed on the server restart.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Method</td>
<td>delete</td>
</tr>
<tr>
<td>SDK Method Name</td>
<td>deleteIndex</td>
</tr>
<tr>
<td>Path parameters</td>
<td>· IndexName (required): Index name</td>
</tr>
<tr>
<td>Return type</td>
<td>deleteIndexResponse</td>
</tr>
<tr>
<td>Responses</td>
<td>· 200: OK</td>
</tr>
<tr>
<td></td>
<td>· 400: Bad Request</td>
</tr>
</tbody>
</table>

### UPDATE THE CONFIGURATION OF AN INDEX

**put /indices/{indexName}/configuration**

The Index Version cannot be modified.
### CONNECTOR FOR IMPORTING CSV FILE DATA INTO THE SYSTEM.

**post** /indices/{indexName}/csv

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Method</td>
<td><strong>post</strong></td>
</tr>
<tr>
<td>SDK Method Name</td>
<td><strong>csv</strong></td>
</tr>
<tr>
<td>Path parameters</td>
<td><strong>indexName</strong> (required) : Index name</td>
</tr>
<tr>
<td>Request body</td>
<td><strong>csvIndexingRequest</strong> (required)</td>
</tr>
<tr>
<td>Return type</td>
<td><strong>csvIndexingResponse</strong></td>
</tr>
<tr>
<td>Responses</td>
<td>• <strong>200</strong> : OK</td>
</tr>
<tr>
<td></td>
<td>• <strong>400</strong> : Failure</td>
</tr>
</tbody>
</table>

[ Jump to Methods ]

### RETURNS TOP 10 DOCUMENT FROM THE INDEX

**get** /indices/{indexName}/documents

This endpoint is useful to determine the structure of the documents indexed. At times it is quicker to get the count of all the documents present in the index using the service rather than using the search API.
CREATE A DOCUMENT

```
post /indices/{indexName}/documents
```

Creates a new document. Unlike a database system FlexSearch doesn’t impose the requirement of a unique ID per document. You can add multiple documents by the same ID but this can impose a problem while adding or retrieving them. You can enforce a unique ID check by using the 'timestamp' field. To understand more about ID check and concurrency control, please refer to the article `concurrency control` under concepts section.

DELETE ALL DOCUMENTS

```
delete /indices/{indexName}/documents
```

[ Jump to Methods ]
This will remove all the documents present in an index. This is useful when you want to re-index all the documents.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Method</td>
<td>delete</td>
</tr>
<tr>
<td>SDK Method Name</td>
<td>deleteAllDocuments</td>
</tr>
<tr>
<td>Path parameters</td>
<td>indexName (required) : Index name</td>
</tr>
<tr>
<td>Return type</td>
<td>deleteAllDocumentsResponse</td>
</tr>
</tbody>
</table>
| Responses           | 200 : OK  
                      | 400 : Bad Request |

DUPLICATE DETECTION HANDLER

post /indices/{indexName}/duplicatedetection/{predefinedQueryName}

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Method</td>
<td>post</td>
</tr>
<tr>
<td>SDK Method Name</td>
<td>duplicateDetection</td>
</tr>
</tbody>
</table>
| Path parameters     | indexName (required) : Index name  
                      | predefinedQueryName (required) : Predefined query name |
| Request body        | duplicateDetectionRequest (required) |
| Return type         | duplicateDetectionResponse |
| Responses           | 200 : Job ID  
                      | 400 : Bad Request |

CHECK IF AN INDEX EXISTS

get /indices/{indexName}/exists

This endpoint can be used to check if an index is present in the system. This endpoint is a lighter alternative to accessing the index by an ID as the response is smaller in size.
### UPDATE THE INDEX FIELDS

```
put /indices/{indexName}/fields
```

Any analyzer which is to be used as part of an index field should be defined before adding the field to the index. Always re-index the data after a field update, otherwise you may get unexpected results. New fields added as part of fields update will not have any data available for the older records, in such cases if the indexing is not done the engine will use default values for the field type. If an existing field is removed then the data associated with that field will not be accessible even though the data will not be removed from the index itself.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Method</td>
<td><strong>put</strong></td>
</tr>
<tr>
<td>SDK Method Name</td>
<td><code>updateIndexFields</code></td>
</tr>
<tr>
<td>Path parameters</td>
<td>• <code>{indexName}</code> (required) : Index Name</td>
</tr>
<tr>
<td>Request body</td>
<td><code>fieldsUpdateRequest</code> (required)</td>
</tr>
<tr>
<td>Return type</td>
<td><code>updateIndexFieldsResponse</code></td>
</tr>
<tr>
<td>Responses</td>
<td>• 200 : OK</td>
</tr>
<tr>
<td></td>
<td>• 400 : Bad Request</td>
</tr>
</tbody>
</table>

[ Jump to Methods ]
**REFRESHES AN INDEX**

```plaintext
put /indices/{indexName}/refresh
```

Refreshes an index so that committed data can be visible when searching.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Method</td>
<td><code>put</code></td>
</tr>
<tr>
<td>SDK Method Name</td>
<td><code>refreshIndex</code></td>
</tr>
<tr>
<td>Path parameters</td>
<td>• <code>{indexName}</code> (required) : Index name</td>
</tr>
<tr>
<td>Return type</td>
<td><code>refreshIndexResponse</code></td>
</tr>
<tr>
<td>Responses</td>
<td>• <strong>200</strong> : OK</td>
</tr>
<tr>
<td></td>
<td>• <strong>400</strong> : Bad Request</td>
</tr>
</tbody>
</table>

[ Jump to Methods ]

**SEARCH IN A INDEX**

```plaintext
get /indices/{indexName}/search
```

Search across the index for documents. Any parameter passed as part of query string takes precedence over the same parameter in the request body. This operation supports both GET & POST verbs.
<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Method</td>
<td><strong>get</strong></td>
</tr>
<tr>
<td>SDK Method Name</td>
<td><strong>getSearch</strong></td>
</tr>
<tr>
<td>Path parameters</td>
<td><em>IndexName</em> (required) : Index name</td>
</tr>
</tbody>
</table>
| Query parameters | *q* : Short hand for 'QueryString'.  
|               | *cols* : Columns to be retrieved. Use * to retrieve all columns.  
|               | *count* : Count parameter. Refer to 'Search Query' properties.  
|               | *skip* : Skip parameter. Refer to 'Search Query' properties.  
|               | *orderby* : Order by parameter. Refer to 'Search Query' properties.  
|               | *orderbydirection* : Order by Direction parameter. Refer to 'Search Query' properties. |
| Request body  | **searchQuery** (required) |
| Return type   | searchResponse |
| Responses     | *200* : OK  
|               | *400* : Bad Request |

**SEARCH IN A INDEX**

```
post /indices/{indexName}/search
```

Search across the index for documents. Any parameter passed as part of query string takes precedence over the same parameter in the request body. This operation supports both GET & POST verbs.
**Property Name** | **Description**
---|---
HTTP Method | post
SDK Method Name | search
Path parameters |*
- **indexName** (required) : Index name
Query parameters |*
- **q**: Short hand for 'QueryString'.
- **c**: Columns to be retrieved. Use * to retrieve all columns.
- **count**: Count parameter. Refer to 'Search Query' properties.
- **skip**: Skip parameter. Refer to 'Search Query' properties.
- **orderby**: Order by parameter. Refer to 'Search Query' properties.
- **orderbydirection**: Order by Direction parameter. Refer to 'Search Query' properties.

Request body | searchQuery (required)
Return type | searchResponse
Responses |*
- **200**: OK
- **400**: Bad Request

[ Jump to Methods ]

RETURNS THE SIZE OF AN INDEX

```
get /indices/{indexName}/size
```

The return size may be higher than the actual size of the documents present in the index. The return value includes the space occupied by the transaction logs and older segment files which are not cleaned up as part of the last comment.

**Property Name** | **Description**
---|---
HTTP Method | get
SDK Method Name | getIndexSize
Path parameters |*
- **indexName** (required) : Index name
Return type | getIndexSizeResponse
Responses |*
- **200**: OK
- **400**: Bad Request

[ Jump to Methods ]

CONNECTOR FOR IMPORTING DATA FROM MICROSOFT SQL
INTO THE SYSTEM.

```
post /indices/{indexName}/sql
```

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Method</td>
<td>post</td>
</tr>
<tr>
<td>SDK Method Name</td>
<td>sql</td>
</tr>
<tr>
<td>Path parameters</td>
<td>·  <code>IndexName</code> (required) : Index Name</td>
</tr>
<tr>
<td>Request body</td>
<td>·  <code>sqlIndexingRequest</code> (required)</td>
</tr>
<tr>
<td>Return type</td>
<td>·  <code>sqlIndexingResponse</code></td>
</tr>
<tr>
<td>Responses</td>
<td>·  <code>200</code> : OK</td>
</tr>
<tr>
<td></td>
<td>·  <code>400</code> : Failure</td>
</tr>
</tbody>
</table>

[ Jump to Methods ]

RETURNS THE STATUS OF AN INDEX

```
get /indices/{indexName}/status
```

This endpoint can be used to determine if an index is online or off-line.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Method</td>
<td>get</td>
</tr>
<tr>
<td>SDK Method Name</td>
<td>getIndexStatus</td>
</tr>
<tr>
<td>Path parameters</td>
<td>·  <code>IndexName</code> (required) : Index Name</td>
</tr>
<tr>
<td>Return type</td>
<td>·  <code>getStatusResponse</code></td>
</tr>
<tr>
<td>Responses</td>
<td>·  <code>200</code> : OK</td>
</tr>
<tr>
<td></td>
<td>·  <code>400</code> : Bad Request</td>
</tr>
</tbody>
</table>

[ Jump to Methods ]

UPDATE THE STATUS OF AN INDEX

```
pay /indices/{indexName}/status/{status}
```

This endpoint can be used to set an index online or off-line.
## updateIndexStatus

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Method</td>
<td><code>put</code></td>
</tr>
<tr>
<td>SDK Method Name</td>
<td>updateIndexStatus</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Path parameters</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>indexName</td>
<td>(required) : Index Name</td>
</tr>
<tr>
<td>status</td>
<td>(required) : offline / online</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return type</th>
<th>updateIndexStatusResponse</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Responses</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>OK</td>
</tr>
<tr>
<td>400</td>
<td>Bad Request</td>
</tr>
</tbody>
</table>

**[Jump to Methods]**

## GET /jobs/{jobId}

RETURNS JOB INFORMATION

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Method</td>
<td><code>get</code></td>
</tr>
<tr>
<td>SDK Method Name</td>
<td>getJob</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Path parameters</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>jobId</td>
<td>(required) : Job ID</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return type</th>
<th>getJobResponse</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Responses</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Job</td>
</tr>
</tbody>
</table>

**[Jump to Methods]**

## GET /memory

RETURNS MEMORY USED BY THE SERVER

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Method</td>
<td><code>get</code></td>
</tr>
<tr>
<td>SDK Method Name</td>
<td>getServerMemoryDetails</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Return type</th>
<th>getServerMemoryDetailsResponse</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Responses</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>OK</td>
</tr>
<tr>
<td>400</td>
<td>Bad Request</td>
</tr>
</tbody>
</table>
PING SERVER

`get /ping`

A simple endpoint which can be used to check the server is running. This is useful for checking the status of the server from a load balancer or fire-wall.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Method</td>
<td><code>get</code></td>
</tr>
<tr>
<td>SDK Method Name</td>
<td><code>ping</code></td>
</tr>
<tr>
<td>Return type</td>
<td>pingResponse</td>
</tr>
<tr>
<td>Responses</td>
<td><code>200</code>: OK</td>
</tr>
</tbody>
</table>

SETUP A DEMO INDEX

`put /setupdemo`

This endpoint is useful for setting up a demo index which can be used to explore the capabilities of FlexSearch. This is an in-memory index which gets wiped out on server restart.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP Method</td>
<td><code>put</code></td>
</tr>
<tr>
<td>SDK Method Name</td>
<td><code>setupDemo</code></td>
</tr>
<tr>
<td>Return type</td>
<td>setupDemoResponse</td>
</tr>
<tr>
<td>Responses</td>
<td><code>200</code>: OK</td>
</tr>
<tr>
<td></td>
<td><code>400</code>: OK</td>
</tr>
</tbody>
</table>

Models

List Of Models

- `OperationMessage`
- CsvIndexingRequest
- SqlIndexingRequest
- DuplicateDetectionRequest
- FieldsUpdateRequest
- IndexStatusRequest
- Tokenizer
- Filter
- Analyzer
- ShardConfiguration
- IndexConfiguration
- Field
- HighlightOption
- SearchQuery
- Document
- Index
- SearchResults
- Job
- CreationId
- IndexExists
- MemoryDetails
- NoBody
- GetAllAnalyzersResponse
- GetAnalyzerResponse
- CreateOrUpdateAnalyzerResponse
- DeleteAnalyzerResponse
- AnalyzeText
- AnalyzeTextResponse
- GetDocumentResponse
- CreateOrUpdateDocumentResponse
- DeleteDocumentResponse
- GetDocumentsResponse
- CreateDocumentResponse
- DeleteAllDocumentsResponse
- SearchResponse
- GetIndexResponse
- DeleteIndexResponse
- GetAllIndicesResponse
- CreateIndexResponse
- IndexExistsResponse
- GetIndexSizeResponse
- RefreshIndexResponse
- GetStatusResponse
- UpdateIndexStatusResponse
- UpdateIndexConfigurationResponse
- UpdateIndexFieldsResponse
- UpdateIndexPredefinedQueryResponse
- GetJobResponse
- GetServerMemoryDetailsResponse
- PingResponse
- **SetupDemoResponse**
- **DuplicateDetectionResponse**
- **CsvIndexingResponse**
- **SqlIndexingResponse**

**OPERATIONMESSAGE**

Represents an operational message returned from the operation. This can be used to represent both success and failure messages.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>⭕ properties : map[String, String]</td>
<td>The list of key value pairs providing information about the operation.</td>
</tr>
<tr>
<td>⭤ message : String</td>
<td>The actual message returned by the operation.</td>
</tr>
<tr>
<td>⭤ operationCode : String</td>
<td>Operation Code associated with the message.</td>
</tr>
</tbody>
</table>

[ Jump to Models ]

**CSVINDEXINGREQUEST**

Represents a request which can be sent to CSV connector to index CSV data.
### Property Name | Description
--- | ---
indexName: String | The name of the index on which the operation is to be performed.

hasHeaderRecord: Boolean | Indicates if the passed CSV file(s) has a header record

headers: array[String] | The headers to be used by each column. This should only be passed when there is no header in the csv file. The first column is always assumed to be id field. Make sure in your array you always offset the column names by 1 position.

path: String | The path of the folder or file to be indexed. The service will pick up all files with .csv extension.

---

**SQLINDEXINGREQUEST**

Represents a request which can be sent to SQL connector to index SQL data

### Property Name | Description
--- | ---
indexName: String | Name of the index

query: String | The query to be used to fetch data from SQL server

connectionString: String | Connection string used to connect to the server

forceCreate: Boolean | Signifies if all updates to the index are create

createJob: Boolean | Signifies if the connector should create a job for the task and return a jobID which can be used to check the status of the job.
DUPLICATEDETECTIONREQUEST

Represents a request to execute dedupe against a given index and predefined query

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>selectionQuery</td>
<td>The query that filters the records to include in the duplicate detection.</td>
</tr>
<tr>
<td>fileName</td>
<td>The file path of the CSV to submit for this session.</td>
</tr>
<tr>
<td>displayName</td>
<td>The name of the field used to display the results.</td>
</tr>
<tr>
<td>threadCount</td>
<td>The number of threads on which to run the duplicate detection.</td>
</tr>
<tr>
<td>indexName</td>
<td>The name of the index on which the operation is to be performed.</td>
</tr>
<tr>
<td>predefinedQueryName</td>
<td>The name of the query used for duplication detection.</td>
</tr>
<tr>
<td>maxRecordsToScan</td>
<td>The maximum number of records to scan for duplication detection.</td>
</tr>
<tr>
<td>duplicatesCount</td>
<td>The maximum number of duplicates to include in the result</td>
</tr>
</tbody>
</table>

[ Jump to Models ]

FIELDSUPDATEREQUEST

Represents a request to update a field in an index

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fields</td>
<td>The list of fields containing the new properties</td>
</tr>
</tbody>
</table>
INDEXSTATUSRESPONSE

Represents the response of an Index status request - tells whether an index is online, off-line, opening, etc.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>indexStatus: String</td>
<td>The status of an index. The property is an enum which has the following valid values:</td>
</tr>
<tr>
<td>opening</td>
<td></td>
</tr>
<tr>
<td>recovering</td>
<td></td>
</tr>
<tr>
<td>online</td>
<td></td>
</tr>
<tr>
<td>onlinefollower</td>
<td></td>
</tr>
<tr>
<td>offline</td>
<td></td>
</tr>
<tr>
<td>closing</td>
<td></td>
</tr>
<tr>
<td>faulted</td>
<td></td>
</tr>
</tbody>
</table>

TOKENIZER

Tokenizer breaks up a stream of text into tokens, where each token is a sub-sequence of the characters in the text. An analyzer is aware of the field it is configured for, but a tokenizer is not.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tokenizerName: String</td>
<td>Name of the tokenizer</td>
</tr>
<tr>
<td>parameters: map[String, String]</td>
<td>Key value pair to be used to configure object's properties.</td>
</tr>
</tbody>
</table>

FILTER

Filters consume input and produce a stream of tokens. In most cases a filter looks at each token in the stream sequentially and decides whether to pass it along, replace it or discard it. A filter may also do more complex analysis by looking ahead to consider multiple tokens at once, although this is less common.
### Property Name | Description
--- | ---
filterName: String | Name of the filter

- parameters: map[String, String] | Key value pair to be used to configure object's properties.

**ANALYZER**

An analyzer examines the text of fields and generates a token stream.

### Property Name | Description
--- | ---
analyzerName: String | Name of the analyzer
tokenizer: tokenizer

- filters: array[filter]

**SHARDCONFIGURATION**

Allows to control various Index Shards related settings.

### Property Name | Description
--- | ---
shardCount: Integer | Total number of shards to be present in the given index.

**INDEXCONFIGURATION**

Allows to control various Index related settings.
<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>commitTimeSeconds : Integer</td>
<td>The amount of time in seconds that FlexSearch should wait before committing changes to the disk. This is only used if no commits have happened in the set time period otherwise CommitEveryNFlushes takes care of commits</td>
</tr>
<tr>
<td>deleteLogsOnClose : Boolean</td>
<td>Determines whether to clear all transaction logs before closing an index. This setting is for advance use and should be left to default.</td>
</tr>
<tr>
<td>commitOnClose : Boolean</td>
<td>Determines whether to commit first before closing an index</td>
</tr>
<tr>
<td>autoCommit : Boolean</td>
<td>Determines whether to enable auto commit functionality or not</td>
</tr>
<tr>
<td>directoryType : String</td>
<td>A Directory is a flat list of files. Files may be written once, when they are created. Once a file is created it may only be opened for read, or deleted. Random access is permitted both when reading and writing. The property is an enum which has the following valid values:</td>
</tr>
<tr>
<td></td>
<td>- FileSystem</td>
</tr>
<tr>
<td></td>
<td>- MemoryMapped</td>
</tr>
<tr>
<td></td>
<td>- Ram</td>
</tr>
<tr>
<td>defaultWriteLockTimeout : Integer</td>
<td>The default maximum time to wait for a write lock (in milliseconds).</td>
</tr>
<tr>
<td>ramBufferSizeMb : Integer</td>
<td>Determines the amount of RAM that may be used for buffering added documents and deletions before they are flushed to the Directory.</td>
</tr>
<tr>
<td>maxBufferedDocs : Integer</td>
<td>The number of buffered added documents that will trigger a flush if enabled.</td>
</tr>
<tr>
<td>refreshTimeMillisseconds : Integer</td>
<td>The amount of time in milliseconds that FlexSearch should wait before reopening index reader. This helps in keeping writing and real time aspects of the engine separate.</td>
</tr>
<tr>
<td>Property Name</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>__autoRefresh : Boolean</td>
<td>Determines whether to enable auto refresh or not</td>
</tr>
<tr>
<td>__indexVersion : String</td>
<td>Corresponds to Lucene Index version. There will always be a default codec associated with each index version. The property is an enum which has the following valid values:</td>
</tr>
<tr>
<td></td>
<td>• FlexSearch_1A</td>
</tr>
<tr>
<td></td>
<td>• FlexSearch_1B</td>
</tr>
<tr>
<td>__allowReads : Boolean</td>
<td>Signifies if the index supports reading back of indexed data.</td>
</tr>
<tr>
<td>__allowWrites : Boolean</td>
<td>Signifies if the index supports modification of data.</td>
</tr>
</tbody>
</table>

FIELD

Represents a group of fields.
<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>_allowSort: Boolean</td>
<td>Enable sorting for the field</td>
</tr>
<tr>
<td>_fieldName: String</td>
<td>Name of the field</td>
</tr>
<tr>
<td>_fieldType: String</td>
<td>Type of the field The property is an enum which has the following valid values:</td>
</tr>
<tr>
<td></td>
<td>• Int</td>
</tr>
<tr>
<td></td>
<td>• Double</td>
</tr>
<tr>
<td></td>
<td>• Float</td>
</tr>
<tr>
<td></td>
<td>• Keyword</td>
</tr>
<tr>
<td></td>
<td>• Text</td>
</tr>
<tr>
<td></td>
<td>• Bool</td>
</tr>
<tr>
<td></td>
<td>• Date</td>
</tr>
<tr>
<td></td>
<td>• DateTime</td>
</tr>
<tr>
<td></td>
<td>• Stored</td>
</tr>
<tr>
<td></td>
<td>• Long</td>
</tr>
<tr>
<td>_indexAnalyzer: String</td>
<td>Analyzer to be used during indexing</td>
</tr>
<tr>
<td>_searchAnalyzer: String</td>
<td>Analyzer to be used during searching</td>
</tr>
<tr>
<td>_similarity: String</td>
<td>Similarity defines the components of scoring. Similarity determines how engine weights terms. FlexSearch interacts with Similarity at both index-time and query-time. The property is an enum which has the following valid values:</td>
</tr>
<tr>
<td></td>
<td>• BM25</td>
</tr>
<tr>
<td></td>
<td>• TFIDF</td>
</tr>
</tbody>
</table>

[HIGHLIGHTOPTION](#)

Used for configuring the settings for text highlighting in the search results
**Property Name** | **Description**
---|---
fragmentsToReturn : `Integer` | Total number of fragments to return per document

highlightedFields : `array[String]` | The fields to be used for text highlighting

postTag : `String` | Post tag to represent the ending of the highlighted word

preTag : `String` | Pre tag to represent the ending of the highlighted word

---

**SEARCHQUERY**

Search query is used for searching over a FlexSearch index. This provides a consistent syntax to execute various types of queries. The syntax is similar to the SQL syntax. This was done on purpose to reduce the learning curve.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
</table>
queryName : `String` | Unique name of the query. This is only required if you are setting up a predefined query. |
columns : `array[String]` | Columns to be returned as part of results. `+*` - return all columns `+[]` - return no columns `[columnName]` - return specific column |
count : `Integer` | Count of results to be returned |
highlights : `highlightOption` |
indexName : `String` | Name of the index |
<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>orderBy : String</td>
<td>Can be used to order the results by score or specific field</td>
</tr>
<tr>
<td>orderByDirection : String</td>
<td>Can be used to determine the sort order. The property is an enum which has the following valid values:</td>
</tr>
<tr>
<td></td>
<td>• Ascending</td>
</tr>
<tr>
<td></td>
<td>• Descending</td>
</tr>
<tr>
<td>cutOff : Double</td>
<td>Can be used to remove results lower than a certain threshold. This works in conjunction with the score of the top record as all the other records are filtered using the score set by the top scoring record.</td>
</tr>
<tr>
<td>distinctBy : String</td>
<td>Can be used to return records with distinct values for the given field. Works in a manner similar to Sql distinct by clause.</td>
</tr>
<tr>
<td>skip : Integer</td>
<td>Used to enable paging and skip certain pre-fetched results.</td>
</tr>
<tr>
<td>queryString : String</td>
<td>Query string to be used for searching</td>
</tr>
<tr>
<td>returnScore : Boolean</td>
<td>If true then scores are returned as a part of search result.</td>
</tr>
<tr>
<td>preSearchScript : String</td>
<td>Script that is executed before submitting the search to Lucene. It can be used to modify the incoming Variables</td>
</tr>
<tr>
<td>overridePredefinedQueryOptions : Boolean</td>
<td>Can be used to override the configuration saved in the predefined query with the one which is passed as the Search Query</td>
</tr>
<tr>
<td>returnEmptyStringForNull : Boolean</td>
<td>Returns an empty string for null values saved in the index rather than the null constant</td>
</tr>
</tbody>
</table>
variables: `map[String, String]`  
The mapping between the variable names (the ones prefixed by '@') given in the query string and their actual values.

**DOCUMENT**

A document represents the basic unit of information which can be added or retrieved from the index. A document consists of several fields. A field represents the actual data to be indexed. In database analogy an index can be considered as a table while a document is a row of that table. Like a table a FlexSearch document requires a fixed schema and all fields should have a field type.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fields: <code>map[String, String]</code></td>
<td>Represents a group of fields.</td>
</tr>
<tr>
<td>id: <code>String</code></td>
<td>The unique id of the document</td>
</tr>
<tr>
<td>indexName: <code>String</code></td>
<td>The name of the index on which the operation is to be performed.</td>
</tr>
<tr>
<td>timeStamp: <code>Long</code></td>
<td>Represents the time at which the document was last indexed.</td>
</tr>
<tr>
<td>modifyIndex: <code>Long</code></td>
<td>Represents the operation number associated with the operation in the global order of the operations. This allows causal ordering of the events. A documents with a lower ModifyIndex can be assumed to be modified before another with a higher number. ModifyIndex is used for optimistic concurrency control.</td>
</tr>
<tr>
<td>highlights: <code>array[String]</code></td>
<td>Returns any highlighted segments of the document. This will only be returned as a part of Search result.</td>
</tr>
<tr>
<td>score: <code>Double</code></td>
<td>The score associated with the document. This will only be used when documents are returned from a Search Query.</td>
</tr>
</tbody>
</table>
INDEX

Options

Controls how much information is stored in the postings lists.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>indexName : String</td>
<td>The name of the index on which the operation is to be performed.</td>
</tr>
<tr>
<td>fields : array[field]</td>
<td>Represents a group of fields.</td>
</tr>
<tr>
<td>predefinedQueries : array[searchQuery]</td>
<td>Array of Predefined Queries</td>
</tr>
<tr>
<td>shardConfiguration : shardConfiguration</td>
<td>Allows to control various Index Shards related settings.</td>
</tr>
<tr>
<td>indexConfiguration : IndexConfiguration</td>
<td>Allows to control various Index related settings.</td>
</tr>
<tr>
<td>active : Boolean</td>
<td>Represents if an Index is active or not?</td>
</tr>
</tbody>
</table>

SEARCHRESULTS

Represents the output of a Search Query
Property Name | Description
---|---
 documents : array[document] | Array of Documents
 recordsReturned : Integer | Total number of records returned as part of the Search Query
 bestScore : Float | The highest score returned by any document as part of the search result.
 totalAvailable : Integer | The total number of documents available as part of the search query. Note the actual records returned could be lower than the total available in the index.

[ Jump to Models ]

**JOB**

Represents a job which can be scheduled or executed on the server
<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>__jobId : String</td>
<td>The unique ID associated with the Job</td>
</tr>
<tr>
<td>__totalItems : Integer</td>
<td>The total number of items to be processed as part of the job.</td>
</tr>
<tr>
<td>__processedItems : Integer</td>
<td>The total number of items processed so far.</td>
</tr>
<tr>
<td>__failedItems : Integer</td>
<td>The total number of items which have failed.</td>
</tr>
<tr>
<td>__jobStatus : String</td>
<td>The current status of the job. The property is an enum which has the following valid values:</td>
</tr>
<tr>
<td></td>
<td>• Initializing</td>
</tr>
<tr>
<td></td>
<td>• Initialized</td>
</tr>
<tr>
<td></td>
<td>• InProgress</td>
</tr>
<tr>
<td></td>
<td>• Completed</td>
</tr>
<tr>
<td></td>
<td>• CompletedWithError</td>
</tr>
<tr>
<td>__message : String</td>
<td>Any operation message associated with the job.</td>
</tr>
</tbody>
</table>

[ Jump to Models ]

**CREATIONID**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id : String</td>
<td></td>
</tr>
</tbody>
</table>

[ Jump to Models ]

**INDEXEXISTS**

Signifies if an index exists on the server or not?
### Property Name | Description
---|---
exists : **Boolean** | Status signifying if the index exists.

**Jump to Models**

### MEMORYDETAILS

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>usedMemory : <strong>Long</strong></td>
<td></td>
</tr>
<tr>
<td>totalMemory : <strong>Long</strong></td>
<td></td>
</tr>
<tr>
<td>usage : <strong>Double</strong></td>
<td></td>
</tr>
</tbody>
</table>

**Jump to Models**

### NOBODY

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
</table>

**Jump to Models**

### GETALLANALYZERSRESPONSE

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data :</td>
<td></td>
</tr>
<tr>
<td>array[analyzer]</td>
<td></td>
</tr>
<tr>
<td>error :</td>
<td></td>
</tr>
<tr>
<td>operationMessage</td>
<td></td>
</tr>
</tbody>
</table>

**Jump to Models**
GETANALYZERRESPONSE

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>data</strong></td>
<td>analyzer</td>
</tr>
<tr>
<td><strong>error</strong></td>
<td>operationMessage</td>
</tr>
</tbody>
</table>

[ Jump to Models ]

CREATETORUPDATEANALYZERRESPONSE

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>data</strong></td>
<td>Boolean</td>
</tr>
<tr>
<td><strong>error</strong></td>
<td>operationMessage</td>
</tr>
</tbody>
</table>

[ Jump to Models ]

DELETEANALYZERRESPONSE

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>data</strong></td>
<td>Boolean</td>
</tr>
<tr>
<td><strong>error</strong></td>
<td>operationMessage</td>
</tr>
</tbody>
</table>

[ Jump to Models ]

ANALYZETEXT

A request object which can be used to analyze a text input using an analyzer.
<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>⚫ text: String</td>
<td>The text to be analyzed by the analyzer.</td>
</tr>
<tr>
<td>⚫ analyzerName: String</td>
<td>The name of the analyzer.</td>
</tr>
</tbody>
</table>

[Jump to Models]

**ANALYZETEXTRESPONSE**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data: array[String]</td>
<td></td>
</tr>
<tr>
<td>error: operationMessage</td>
<td></td>
</tr>
</tbody>
</table>

[Jump to Models]

**GETDOCUMENTRESPONSE**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data: document</td>
<td></td>
</tr>
<tr>
<td>error: operationMessage</td>
<td></td>
</tr>
</tbody>
</table>

[Jump to Models]

**CREATEORUPDATEDOCUMENTRESPONSE**
### DELETEDOCUMENTRESPONSE

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>data</strong></td>
<td>Boolean</td>
</tr>
<tr>
<td><strong>error</strong></td>
<td>operationMessage</td>
</tr>
</tbody>
</table>

[Jump to Models]

### GETDOCUMENTSRESPONSE

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>data</strong></td>
<td>searchResults</td>
</tr>
<tr>
<td><strong>error</strong></td>
<td>operationMessage</td>
</tr>
</tbody>
</table>

[Jump to Models]

### CREATEDOCUMENTRESPONSE
## DELETEALLDOCUMENTSRESPONSE

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>data</em></td>
<td>Boolean</td>
</tr>
<tr>
<td><em>error</em></td>
<td>operationMessage</td>
</tr>
</tbody>
</table>

[Jump to Models]

## SEARCHRESPONSE

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>data</em></td>
<td>searchResults</td>
</tr>
<tr>
<td><em>error</em></td>
<td>operationMessage</td>
</tr>
</tbody>
</table>

[Jump to Models]

## GETINDEXRESPONSE
### DELETEINDEXRESPONSE

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>data</code></td>
<td>Boolean</td>
</tr>
<tr>
<td><code>error</code></td>
<td>operationMessage</td>
</tr>
</tbody>
</table>

[Jump to Models]

### GETALLINDICESRESPONSE

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>data</code></td>
<td>array[index]</td>
</tr>
<tr>
<td><code>error</code></td>
<td>operationMessage</td>
</tr>
</tbody>
</table>

[Jump to Models]

### CREATEINDEXRESPONSE
## REST

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>data</em> : Boolean</td>
<td></td>
</tr>
<tr>
<td><em>error</em> : operationMessage</td>
<td></td>
</tr>
</tbody>
</table>

[Jump to Models]

### INDEXEXISTSRESPONSE

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>data</em> : indexExists</td>
<td></td>
</tr>
<tr>
<td><em>error</em> : operationMessage</td>
<td></td>
</tr>
</tbody>
</table>

[Jump to Models]

### GETINDEXSIZERESPONSE

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>data</em> : BigDecimal</td>
<td></td>
</tr>
<tr>
<td><em>error</em> : operationMessage</td>
<td></td>
</tr>
</tbody>
</table>

[Jump to Models]

### REFRESHINDEXRESPONSE
<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>data</em> : Boolean</td>
<td></td>
</tr>
<tr>
<td><em>error</em> : operationMessage</td>
<td></td>
</tr>
</tbody>
</table>

[ Jump to Models ]

### GETSTATUSRESPONSE

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>data</em> : IndexStatusResponse</td>
<td></td>
</tr>
<tr>
<td><em>error</em> : operationMessage</td>
<td></td>
</tr>
</tbody>
</table>

[ Jump to Models ]

### UPDATEINDEXSTATUSRESPONSE

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>data</em> : Boolean</td>
<td></td>
</tr>
<tr>
<td><em>error</em> : operationMessage</td>
<td></td>
</tr>
</tbody>
</table>

[ Jump to Models ]

### UPDATEINDEXCONFIGURATIONRESPONSE
<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>data</em></td>
<td>Boolean</td>
</tr>
<tr>
<td><em>error</em></td>
<td>operationMessage</td>
</tr>
</tbody>
</table>

[Jump to Models]
<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data: Job</td>
<td></td>
</tr>
<tr>
<td>error: operationMessage</td>
<td></td>
</tr>
</tbody>
</table>

[ Jump to Models ]

**GETSERVERMEMORYDETAILSRESPONSE**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data: memoryDetails</td>
<td></td>
</tr>
<tr>
<td>error: operationMessage</td>
<td></td>
</tr>
</tbody>
</table>

[ Jump to Models ]

**PINGRESPONSE**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>data: Boolean</td>
<td></td>
</tr>
<tr>
<td>error: operationMessage</td>
<td></td>
</tr>
</tbody>
</table>

[ Jump to Models ]

**SETUPDEMORESPONSE**
### DUPLICATEDETECTIONRESPONSE

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>data</strong></td>
<td>Boolean</td>
</tr>
<tr>
<td><strong>error</strong></td>
<td>operationMessage</td>
</tr>
</tbody>
</table>

[Jump to Models]

### CSVINDEXINGRESPONSE

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>data</strong></td>
<td>String</td>
</tr>
<tr>
<td><strong>error</strong></td>
<td>operationMessage</td>
</tr>
</tbody>
</table>

[Jump to Models]

### SQLINDEXINGRESPONSE

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>data</strong></td>
<td>String</td>
</tr>
<tr>
<td><strong>error</strong></td>
<td>operationMessage</td>
</tr>
</tbody>
</table>

[Jump to Models]
<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>data</em> :</td>
<td>String</td>
</tr>
<tr>
<td><em>error</em> :</td>
<td>operationMessage</td>
</tr>
</tbody>
</table>

[ Jump to Models ]

## DataTypes

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Byte</td>
<td>0 .. 255</td>
</tr>
<tr>
<td>Integer</td>
<td>-2,147,483,648 .. 2,147,483,647</td>
</tr>
<tr>
<td>Float</td>
<td>-3.402823e38 .. 3.402823e38</td>
</tr>
<tr>
<td>Double</td>
<td>-1.79769313486232e308 .. 1.79769313486232e308</td>
</tr>
<tr>
<td>String</td>
<td>A string of Unicode characters.</td>
</tr>
<tr>
<td>Boolean</td>
<td>True or False.</td>
</tr>
</tbody>
</table>
Examples

Contents

- Index API Examples
- Creating an index without any data
- Creating an index with two fields and
- Duplicate indices cannot be created
- Getting all available indices
- Getting an index by name
- Getting a non-existing index returns an error
- Checking if an index exists
- Getting the status of an index
- Updating an index configuration
- Deleting an index
- Deleting a non-existing index will return an error
- Changing the status of an index
- Document API examples
- Getting top 10 documents from an index
- Getting a document by ID
- Modifying an existing document
- Creating a new document
- Getting a non-existing document by ID returns
- Search API Examples
- Create or Update a

Index API Examples

CREATING AN INDEX WITHOUT ANY DATA

The content of the section is optimized for web viewing. Please access the content from the website.

CREATING AN INDEX WITH TWO FIELDS **FIRSTNAME** AND **LASTNAME**

The content of the section is optimized for web viewing. Please access the content from the website.

DUPLICATE INDICES CANNOT BE CREATED

The content of the section is optimized for web viewing. Please access the content from the website.

GETTING ALL AVAILABLE INDICES
GETTING AN INDEX BY NAME

GETTING A NON EXISTING INDEX RETURNS AN ERROR

CHECKING IF AN INDEX EXISTS

GETTING THE STATUS OF AN INDEX

UPDATING AN INDEX CONFIGURATION

DELETING AN INDEX

DELETING A NON EXISTING INDEX WILL RETURN AN ERROR

CHANGING THE STATUS OF AN INDEX

In this particular case we are going to bring an index **online**, meaning we are going to set its status to **online**.

Document API Examples
GETTING TOP 10 DOCUMENTS FROM AN INDEX

The content of the section is optimized for web viewing. Please access the content from the website.

GETTING A DOCUMENT BY ID

The content of the section is optimized for web viewing. Please access the content from the website.

MODIFYING AN EXISTING DOCUMENT

The content of the section is optimized for web viewing. Please access the content from the website.

CREATING A NEW DOCUMENT

The content of the section is optimized for web viewing. Please access the content from the website.

GETTING A NON-EXISTING DOCUMENT BY ID RETURNS NOT FOUND

The content of the section is optimized for web viewing. Please access the content from the website.

Search API Examples

CREATE OR UPDATE A **PREDEFINED QUERY**

Whether you want to create or update a **predefined query**, the method is the same.

AllOf 2 clauses with single tokens

\[
\text{allOf}(\text{agriproducts}, \text{'rice'}) \text{ and allOf}(\text{agriproducts}, \text{'wheat'})
\]

The content of the section is optimized for web viewing. Please access the content from the website.
AllOf single clause with a single token

allOf(agriproducts, 'rice wheat')

The content of the section is optimized for web viewing. Please access the content from the website.

AllOf single clause with 2 tokens

allOf(agriproducts, 'rice', 'wheat')

The content of the section is optimized for web viewing. Please access the content from the website.

AnyOf 2 clauses with single tokens

anyOf(agriproducts, 'rice') OR anyOf(agriproducts, 'wheat')

The content of the section is optimized for web viewing. Please access the content from the website.

AnyOf single clause with a single token

anyOf(agriproducts, 'rice wheat')

The content of the section is optimized for web viewing. Please access the content from the website.
AnyOf single clause with 2 tokens

```
anyOf(agriproducts, 'rice', 'wheat')
```

The content of the section is optimized for web viewing. Please access the content from the website.

Fuzzy with default slop of 1

```
fuzzy(countryname, 'Iran')
```

The content of the section is optimized for web viewing. Please access the content from the website.

Fuzzy with slop of 2

```
fuzzy(countryname, 'China', -slop '2')
```

The content of the section is optimized for web viewing. Please access the content from the website.

Like using ‘*’ operator

```
like(countryname, 'uni*')
```

The content of the section is optimized for web viewing. Please access the content from the website.
Like with single character operator

```python
like(countryname, 'unit?d')
```

The content of the section is optimized for web viewing. Please access the content from the website.

Matching inside a word using like

```python
like(countryname, '*uni*')
```

The content of the section is optimized for web viewing. Please access the content from the website.

Matchall to get all documents back

```python
matchall(countryname, '*')
```

The content of the section is optimized for web viewing. Please access the content from the website.

Matchnone will not match any documents

```python
matchnone(countryname, '*')
```

The content of the section is optimized for web viewing. Please access the content from the website.
<table>
<thead>
<tr>
<th>Greater than 'gt' operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>gt(population, '1000000')</td>
</tr>
<tr>
<td>The content of the section is optimized for web viewing. Please access the content from the website.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Greater than or equal to 'ge' operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>ge(population, '1000000')</td>
</tr>
<tr>
<td>The content of the section is optimized for web viewing. Please access the content from the website.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Less than 'lt' operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>lt(population, '1000000')</td>
</tr>
<tr>
<td>The content of the section is optimized for web viewing. Please access the content from the website.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Less than or equal to 'le' operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>le(population, '1000000')</td>
</tr>
<tr>
<td>The content of the section is optimized for web viewing. Please access the content from the website.</td>
</tr>
</tbody>
</table>
Phrase search passing multiple words as single token

\[
\text{phraseMatch(governmenttype, 'federal parliamentary democracy')}
\]

The content of the section is optimized for web viewing. Please access the content from the website.

Phrase search passing multiple words as multiple tokens

\[
\text{phraseMatch(governmenttype, 'federal', 'parliamentary', 'democracy')}
\]

The content of the section is optimized for web viewing. Please access the content from the website.

Phrase search with slop of 4

\[
\text{phraseMatch(governmenttype, 'parliamentary monarchy', -slop '4')}
\]

The content of the section is optimized for web viewing. Please access the content from the website.

Phrase search with slop of 4

\[
\text{phraseMatch(governmenttype, 'monarchy parliamentary', -slop '4')}
\]

The content of the section is optimized for web viewing. Please access the content from the website.
Match both phrases containing 'parliamentary democracy' and 'parliamentary system'

\texttt{phrasetMatch(governmenttype, 'parliamentary', 'democracy system', -multiphrase)}

The content of the section is optimized for web viewing. Please access the content from the website.

Match phrases containing 'parliamentary democracy', 'parliamentary system' and 'parliamentary constitutional'

\texttt{phrasetMatch(governmenttype, 'parliamentary', 'democracy system constitutional', -multiphrase)}

The content of the section is optimized for web viewing. Please access the content from the website.

Match phrases containing 'parliamentary monarchy' and 'constitutional monarchy'

\texttt{phrasetMatch(governmenttype, 'constitutional parliamentary', 'monarchy', -multiphrase)}

The content of the section is optimized for web viewing. Please access the content from the website.

Simple regex match

\texttt{regex(agriproducts, '[ms]ilk')}
Sdk
CSharp Client

Contents

- Namespaces
- Request Model
- Guidelines on implementing your own FlexSearch Request
- Implementing the Method
- Using and attributes
- Using and attributes
- Response Model
- Guidelines on implementing your own
- Usage example

FlexSearch comes with a C# client that is automatically generated from the Swagger definition. You can find the C# client dll in the FlexSearch.Clients.*.zip package under the name:

FlexSearch.Api.dll

All you need to do to start using it is to reference this DLL in your project.

Namespaces

The C# client is split into three parts / namespaces:

1. **Model**
   Contains all of the requests, responses, enums and helper models that you will need for working with the out-of-box FlexSearch functionality.

2. **Api**
   Contains all of the necessary methods for accessing FlexSearch HTTP endpoints. The APIs are grouped by Swagger tags into:
   - **Indices API** - has methods for working with Indices: create, update, delete, check, etc.
   - **Search API**
   - **Documents API**
   - **Analyzer API**
   - **Common API** - has methods for working with the most populat FlexSearch endpoints such as creating an index, searching, getting documents, as well as accessing the FlexSearch connectors.
   - **Jobs API**
   - **Server API** - has methods for setting up a demo index, getting server stats, ping-ing, etc.

3. **Client**
   Contains the code that handles the HTTP calls. The Client is used by the Api's during instantiation. It can be instantiated by using either a base path (e.g. http://localhost:9800) or an HttpResponseMessageHandler

Request Model
All FlexSearch `Request` objects derive from `IUserDataTransferObject`. This forces the users to implement a validation method for each request type. The method (`Validate()`) will be called by FlexSearch at the beginning of any HTTP handler.

**Guidelines On Implementing Your Own FlexSearch Request**

**IMPLEMENTING THE `VALIDATE()` METHOD**

The `Validate()` method should populate the `IUserDataTransferObject` fields in the following way:

- `Validated` - should be used to indicate if the `Validate()` method has been called at least once.
- `ErrorField` - should hold the name of the Field that was identified as having a problem.
- `ErrorDescription` - should contain an explanation of the problem.

**USING `DATAContract` AND `DataMember` ATTRIBUTES**

In order to make your request fully compatible with FlexSearch, you should use:

- `System.Runtime.Serialization.DataContract` attribute on your request
- `System.Runtime.Serialization.DataMember` attribute on the properties you want to expose.

Typically, these would be the new properties you create, not the ones implemented from `IUserDataTransferObject`.

**USING `JsonProperty` AND `JsonIgnore` ATTRIBUTES**

We suggest using the `JsonIgnore` attribute on the properties implemented from `IUserDataTransferObject`. The `JsonProperty` attribute should be used on the new properties you've created, being careful to use `camelCase` for the `PropertyName`.

**Response Model**

All FlexSearch `Responses` have two Properties:

- `Data` - this contains the actual data that you want to retrieve when making a request
  
  E.g. The ID of the newly created Index or the list of `Analyzer` DTOs when you’re getting all Analyzers

- `Error` - In case there was an error when running the request, this property will be populated. It contains helpful information to help you identify what the problem is.

  These are the pieces of information present in the `Error` property:

  1. `Message` - It contains a human readable description of the error
  2. `OperationCode` - It contains a code that identifies the category of the error
  3. `Properties` - This is a list of the key pieces of data that concern this error.

  Example:
When implementing your own response, you don't need to worry about the **Data** and **Error** part. FlexSearch will handle that for you.

You will just need to focus on annotating your object with the attributes **DataContract**, **DataMember**, and **JsonProperty**, like instructed in the Request Model.

## Usage Example

In this example, I'm going to do a search for any of the words "most", "prosperous" and "countries" inside the field **background** of the **country** index.

```csharp
// First initialize a client to connect to our FlexSearch URL
var searchApi = new SearchApi("http://localhost:9800")

// Next create the request that we want to send. In this case, the
// request is of type 'SearchQuery'
var query = new SearchQuery("country", "anyOf(background, 'most prosperous
countries')");

// We want to get back the fields "countryname" and "background"
query.Columns = new string[] { "countryname", "background" };

// Now we submit the request using a POST
var response = api.Search(query, query.IndexName);

// We interpret the response
if (response.Error?.Message != null)
    Console.WriteLine($"Got an error: {response.Error.Message}");
else
    Console.WriteLine($"Got back {response.Data.RecordsReturned} records");
```
TypeScript Client

Contents

- Summary
- Initializing the APIs
- Example
- Accessing the API methods
- Example
- Complex example

Summary

The TypeScript client can be found in the FlexSearch.Clients.*.zip package in the clients/ts folder.

The client also comes with a definition file called api.d.ts to give you intellisense.

The TypeScript APIs are grouped by swagger tags, just like the C# client is. You therefore have an `AnalyzerApi`, `ServerApi`, `SearchApi` and so on.

Initializing The APIs

Initializing an API is as easy as providing it a reference to Angular’s `ng.IHttpService` service ($http). This would use the base path of `http://localhost:9800` by default.

```javascript
let api = new API.Client.AnalyzerApi($http);
```

There are also several other `optional` parameters that you can pass to the api constructor:

- `httpParamSerializer` - Angular’s params serializer
- `basePath` - the base path to use in the requests. E.g. `http://localhost:9800`
- `bs`, `q` and `errorHandler` -
  These 3 params need to be specified in a group. They were created specifically for Angular apps that use FlexSearch’s base template (Main). They are used in conjunction with the API methods that have a name ending in *Handled (e.g. `getAnalyzerHandled`).

Whenever a *Handled API method is called, if it returns an error (i.e. has the Error property populated) then an Angular Material $mdBottomSheet will be displayed with the error details. This makes for very easy error handling.

The 3 parameters mean the following:

- `bs` is the reference to the $mdBottomSheet that will be displayed
- `q` is the reference to Angular’s $q service that is used for rejecting a promise
**ErrorHandler** is the reference to FlexSearch's error handling mechanism. In order to gain access to FlexSearch's instance of this function, you just need to make sure you reference `srcjs/src/common/partials/error.ts`.

**EXAMPLE**

Initializing an instance of a FlexSearch TypeScript client as a service in an Angular app, that displays the errors using FlexSearch's framework can be done in the following way:

```typescript
angular.module('myModule', ['ngMaterial'])
  .service('indicesApi', ['$http', '$mdBottomSheet', '$q',
    function($http, $mdBottomSheet, $q) {
      return new API.Client.IndicesApi($http, null, null, $mdBottomSheet, $q, errorHandler);
    }]);
```

### Accessing The API Methods

Each API method (web service) has an overload with the `Handled` suffix. The `Handled` methods handle errored calls automatically using FlexSearch's framework. This only works if you've initialized the API with the error handling parameters.

For example, you will have the following methods in the `AnalyzerApi`:

- `getAnalyzer`
- `getAnalyzerHandled`
- `deleteAnalyzer`
- `deleteAnalyzerHandled`

The outcome of the `Handled` methods is that if an API method call returns an error, then a window will pop up from the bottom of the screen showing the error details. The method call will still return an error promise and you can further handle it if you want.

You can just as well call the normal methods (without `Handled`). The return object of the method will be the same. The only difference is that the error details won't be displayed on the screen.

**EXAMPLE**

Let's try to get all the Analyzers registered in FlexSearch and submit an analysis request against the first of them.

We assume we have an already instantiated `analyzerApi` with the error handling arguments populated.
Complex Example

In order to see a complex example have a look at the FlexSearch Dashboard app.

In the `index.ts` file you will see how the `indices`, `server` and `documents` APIs are initialized in the error handling mode.

In the `cluster.ts` file you will see how we call different methods from the API, how we use the results in a strongly typed manner and how we handle the errors in order to produce a series of stats about the FlexSearch server.
Custom Plugins

Contents

- Definition
- Writing your own FlexSearch Plugin
- Implementing the HTTP endpoint
- Creating the AngularJS app
- Adding the setup.* files
- Deploying your app
- Installing a FlexSearch Plugin

Definition

Custom plugins are pieces of code that enrich the capabilities of what the FlexSearch engine can do. They are loaded automatically when the engine starts.

The plugin can be composed of two parts:

1. A *.dll that contains the implementation of the interface/abstract class you want to use.
   Such interfaces can be `HttpHandlerBase<T, U>` or `IQueryFunction`.
   Dlls are loaded automatically from FlexSearch's ./Plugins/ folder.

2. An AngularJS app that will be included automatically in the FlexSearch portal. Examples of such apps are the `Analyzer Testing` app, the `Search Studio` app, the `Dashboard`, etc.
   Apps are loaded automatically from FlexSearch's ./Web/apps/ folder.

Examples of already implemented plugins can be found in the FlexSearch GitHub organization:

- SQL Connector
- CSV Connector
- El Machina (just for demo purposes)

Writing Your Own FlexSearch Plugin

Let's assume we want to add a new HTTP endpoint (web service) that when accessed returns the machine name that's running the FlexSearch engine. We will also want to create an HTML screen to display this information. Let's name this plugin El Machina. You can find the source code of this sample app at https://github.com/FlexSearch/el-machina.

In order to kickstart your custom plugin development, the FlexSearch team has provided a template for writing custom plugins. It helps you reference `FlexSearch.Core.dll`, `FlexSearch.Api.dll` and also provides build scripts for your [F#] or [AngularJS] app.

The template is in the form of a github repository called PluginLibrary.

The plugin will be structured in the following way (let's assume that the root folder is `el-machina`):
The **lib** folder will contain the **PluginLibrary** repo, the **src** folder the F# project and the **srcjs** folder the AngularJS app.

In order to include the **PluginLibrary** repo as a submodule in your local git repo, you will need to run the following command from the root of your repo:

```
git submodule add https://github.com/FlexSearch/PluginLibrary lib
```

Next we will add the **src** and **srcjs** folders

```
mkdir src
mkdir srcjs
```

**IMPLEMENTING THE HTTP ENDPOINT**

Now we need to create the C# project that will implement the new HTTP endpoint. For this I will create a Class Library project called **ElMachina** in the **src** folder. The folder structure will look like this:

```
/el-machina/src/
|-- ElMachina/
     |-- ElMachina.csproj
     |-- Handler.cs
     |-- ElMachina.sln
```

We now need to add a reference to FlexSearch.Core and FlexSearch.Api from the **/lib/src/** folder, and then to FSharp.Core.

In the **Handler.cs** file we add the following code:
using FlexSearch.Api.Model;
using Microsoft.FSharp.Core;
using System;
using static FlexSearch.Core.Http;

namespace ElMachina
{
    public class GetMachineName : HttpHandlerBase<NoBody, string>
    {
        public GetMachineName()
        {
            base(new FSharpOption<bool>(false), new FSharpOption<bool>(false))
        }

        public override ResponseContext<string> Process(RequestContext request, FSharpOption<NoBody> body)
        {
            return ResponseContext<string>.SuccessResponse.NewSuccessResponse(Environment.MachineName, Ok);
        }
    }
}

From this code you can see that we've implemented the abstract base class `HttpHandlerBase`, where we've specified that we have no body (`NoBody`) for the request and the response returns a `string`. The function override always returns a Success response containing the Machine name and the OK HTTP status code.

We need to add a `.nuget` folder that holds the Nuget executable, needed by Fake to build the app. We will put this in `el-machina/src/.nuget`.

**CREATING THE ANGULARJS APP**

The code needed for this should be written in TypeScript and put in the `el-machina/srcjs` folder. This code is just small part of the entire AngularJS app, representing the portal. It references code from the `el-machina/lib/srcjs/src` folder.

When the entire app is building, the code from `el-machina/srcjs` will be copied over in `el-machina/lib/srcjs/src/apps/<name-of-your-app>`. Only then will the gulp script be ran.

We will need 4 files in total for this app:

```
/el-machina/srcjs/
|-- info.json
|-- index.ts
|-- elmachina.ts
|-- elmachina.html
```
- `info.json` holds the basic information needed to display this app on the FlexSearch portal homepage.
- `index.ts` holds the angular module definition with the states, services, controllers, etc.
- `elmachina.ts` is the controller that calls the newly created web service
- `elmachina.html` holds the HTML template that is rendered for that controller.

**ADDING THE SETUP:* FILES**

The `setup.*` files tie all these pieces of code together, build them and package them in a zip file.

The `setup.fsx` file is an F# script that uses Fake to copy files, build the assembly and package everything.

The `setup.bat` file downloads Fake from nuget if it's necessary, then calls `setup.fsx`.

Hopefully, the only things you would need to change in your app are the references to the name of the application. For example, if your app is called MyApp, you would just have to search for `elmachina` in `setup.fsx` and replace those occurrences appropriately.

In order to build your application you just need to run the following command from the `/el-machina` folder:

```
..\setup.bat
```

You can run Fake targets individually by calling:

```
..\setup.bat target=<target-name>
```

For example, you can just build the C# application by calling:

```
..\setup.bat target=BuildApp
```

**DEPLOYING YOUR APP**

Just run `..\setup.bat` from the `/el-machina` folder. This will generate a .zip file in `/el-machina/deploy/`.

The .zip file has two parts in it that you will need to deploy to your FlexSearch engine folder:

1. The `*.dll` containing your server-side code. You will need to put this into:

   `<flexsearch-engine-folder>/Plugins/`

2. A folder containing the AngularJS application. In our case the folder is called `elmachina`. Copy this folder into:

   `<flexsearch-engine-folder>/Web/apps/`

When you restart your FlexSearch engine, the new plugin and portal app will be picked up automatically. Just navigate to your portal homepage and see for yourself.

**Installing A FlexSearch Plugin**
In order to install a plugin you just need to take the *.dll file from your package and put it in the Plugins folder. The plugin will be picked up the next time FlexSearch starts.

For example, for the CSV Connector, you would take the CsvConnector.dll file from the generated zip and put it in the Plugins folder.
CSV Connector

Contents
- Summary
- Request object
- Important Notes:
- Endpoint
- Example

Summary
The CSV Connector is nothing more than a FlexSearch plugin. It allows users to import data into a FlexSearch index from a CSV file or folder by POST-ing a request to a specific endpoint.

Request Object
You can find the CsvIndexingRequest request object in the Reference section of the REST API.

IMPORTANT NOTES:
- The Path parameter supports passing either a folder or a .csv file. If a folder is passed, then all files with a *.csv extension will be processed.
- If the HasHeaderRecord parameter is set to False, then the Headers parameter MUST be specified.
- If the HasHeaderRecord parameter is set to True, then the Headers parameter will be ignored.
- The first column in the CSV file will be taken as the index ID. Therefore, make sure that you offset your column names by 1 in the Headers array.

Endpoint
Assuming you want to import data into the contact index, you would need to POST a CsvIndexingRequest to the following endpoint:

```
<flexsearch_url_with_port_no>/indices/contact/csv
```

Example
Please have a look at the sample file from the CSV Connector repository. It shows how to create the client, instantiate a request, send the request to FlexSearch, then check the status of that request.
SQL Connector

Contents

- Summary
- Request object
- Important Notes:
- Endpoint
- Example

Summary

The SQL Connector is nothing more than a FlexSearch plugin. It allows users to import data into a FlexSearch index from a SQL database by POST-ing a request to a specific endpoint.

Request Object

You can find the SqlIndexingRequest request object in the Reference section of the REST API.

IMPORTANT NOTES:

- The connection string should be in a supported .NET format. Example:
  ```
  data source=<server_name>;initial catalog=<db_name>;Integrated Security=True
  ```

- The query is a SQL SELECT statement in which the names of the columns should match the names of the index fields. Also, a very important note is that the first column acts as a unique identifier for the document within the index.

Endpoint

Assuming you want to import data into the contact index, you would need to POST a SqlIndexingRequest to the following endpoint:

```
<flexsearch_url_with_port_no>/indices/contact/sql
```

Example

Please have a look at the sample file from the SQL Connector repository. It shows how to create the client, instantiate a request, send the request to FlexSearch, then check the status of that request.
Checking Index Status

Contents

- Checking the status of an index using the C# client
- Bonus - Checking if an index exists
- Checking the status of an index by writing the HTTP request ourselves

An index goes through different statuses during its lifecycle:

```
Opening --> Online --> Closing --> Offline
```

- **Opening** - an index will be in the *Opening* status while it loads its configuration, analyzers, queries, etc. and initializes its *index writers* and *readers*.
- **Online** - an index is in the *Online* status after it's finished *opening*. It will continue to stay in this status until it's manually closed (a.k.a put *offline*) or the FlexSearch server is closed.
- **Closing** - an index is in the *Closing* status while it's unloading, releasing and disposing all the objects used during the *opening* phase and acquired while it was *online*.
- **Offline** - an index is in the *Offline* status if it was manually closed. An offline index cannot receive any commands apart from activating it (using the `UpdateStatus("online")` web service)

You can check the status of an index in FlexSearch in two ways:

1. Using one of the FlexSearch clients (C#, TypeScript or JavaScript)
2. By submitting the HTTP request yourself (using a tool like Fiddler, for example)

For demonstration purposes, let's assume we already have an active / opened index named *contact*.

Checking The Status Of An Index Using The C# Client

First you need to get hold of an API that has the `GetIndexStatus` web service method. You can use the `IndicesApi`.

```
var indicesApi = new IndicesApi("http://localhost:9000");
```

Now we can call the `GetIndexStatus` method, supplying the *name of the index* as a `string`. This method returns an option from the `IndexStatus` enum, i.e. one of the lifecycle statuses.
var response = indicesApi.GetIndexStatus("country");

if (response.Error?.Message != null)
    Console.WriteLine(response.Error.Message);
else
{
    IndexStatus status = response.Data.IndexStatus;
}

BONUS - CHECKING IF AN INDEX EXISTS

To check if an index exists in FlexSearch you need to use the `IndexExists` method from the `IndicesApi`. It returns `true` or `false`.

var indicesApi = new IndicesApi("http://localhost:9800");

var response = indicesApi.IndexExists("some-index-name");

if (response.Error?.Message != null)
    Console.WriteLine(response.Error.Message);
else
{
    bool exists = response.Data.Exists;
}

Checking The Status Of An Index By Writing The HTTP Request Ourselves

Please have a look at the REST documentation for Index Management APIs.
Closing An Index

Contents

- Closing an index using the C# client
- Closing an index by writing the HTTP request ourselves

Closing an index will just make the index unavailable for any changes or searches. It will not delete or modify any data. You can reopen / activate an index at any time.

You can close an index in FlexSearch in two ways:

1. Using one of the FlexSearch clients (C#, TypeScript or JavaScript)
2. By submitting the HTTP request yourself (using a tool like Fiddler, for example)

For demonstration purposes, let's assume we already have an active / opened index named contact.

Closing An Index Using The C# Client

First you need to get hold of an API that has the UpdateIndexStatus web service method. You can use the IndicesApi:

```csharp
var indicesApi = new IndicesApi("http://localhost:9800");
```

Now we can call the UpdateIndexStatus method, supplying the name of the index and the status as a string. The status can have one of the following values: online or offline. We are, of course, going to set the status to offline.

The response doesn't contain any data, it just contains the Error part.

```csharp
var response = indicesApi.UpdateIndexStatus("contact", "offline");

// This method doesn't return any significant data. It just reports any errors.
if (response.Error?.Message != null)
    Console.WriteLine(response.Error.Message);
```

Closing An Index By Writing The HTTP Request Ourselves

Please have a look at the REST documentation for Index Management APIs.
Creating An Index

Contents

• Creating an index using the C# client
• Creating an index by writing the HTTP request ourselves

You can create an index in FlexSearch in two ways:

1. Using one of the FlexSearch clients (C#, TypeScript or JavaScript)
2. By submitting the HTTP request yourself (using a tool like Fiddler, for example)

For demonstration purposes, let's create an index named contact that has a name column of type Text and an age column of type Int.

Creating An Index Using The C# Client

First you need to get hold of an API that has the CreateIndex web service method. You can use either the CommonApi or the IndicesApi:

```csharp
var indicesApi = new IndicesApi("http://localhost:9800");
```

After initializing the C# client, we need to create a FlexSearch.Api.Model.Index object that contains all the configuration needed to create the index.

```csharp
var index = new Index("contact")
{
    Fields = new Field[]
    {
        new Field("name", FieldType.Text),
        new Field("age", FieldType.Int)
    }
};
```

Now we can call the CreateIndex method, supplying the index object as a parameter. The response should return true in the Data part if everything went ok.

```csharp
var response = indicesApi.CreateIndex(index);
Debug.Assert(response.Error == null);
Debug.Assert(response.Data == true);
```

Creating An Index By Writing The HTTP Request Ourselves
Please have a look at the REST documentation for Index Management APIs.
Deleting Documents

You can delete documents in FlexSearch in two ways:

1. Using one of the FlexSearch clients (C#, TypeScript or JavaScript)
2. By submitting the HTTP request yourself (using a tool like Fiddler, for example)

For demonstration purposes, let’s delete an existing document with ID “7” from the index named contact.

Deleting Documents Using The C# Client

First you need to get hold of an API that has the DeleteDocument web service method. You can use either the CommonApi or the DocumentsApi:

```csharp
var documentsApi = new DocumentsApi("http://localhost:9800");
```

After initializing the C# client, we just need to call the DeleteDocument method supplying the index name and the document ID that we want to delete.

The response doesn’t contain any data, it just contains the Error part.

```csharp
var response = documentsApi.DeleteDocument("contact", "7");

// This method doesn’t return any significant data. It just reports any errors.
if (response.Error?.Message != null)
    Console.WriteLine(response.Error.Message);
```

Deleting Documents By Writing The HTTP Request Ourselves

Please have a look at the REST documentation for Document Management APIs.
Indexing Documents

Contents

- Indexing documents using the C# client
- Indexing documents by writing the HTTP request ourselves

You can index documents in FlexSearch in two ways:

1. Using one of the FlexSearch clients (C#, TypeScript or JavaScript)
2. By submitting the HTTP request yourself (using a tool like Fiddler, for example)

For demonstration purposes, let's add documents to an already existing index named **contact** that has a **name** column of type **Text** and an **age** column of type **Int**.

### Indexing Documents Using The C# Client

First you need to get hold of an API that has the **CreateDocument** web service method. You can use either the **CommonApi** or the **DocumentsApi**:

```csharp
var documentsApi = new DocumentsApi("http://localhost:9800");
```

After initializing the C# client, we need to get hold of the **FlexSearch.Api.Model.Document** objects that we want to index. Here I'm just creating 10 sample documents with **name** and **age** columns populated.

```csharp
public IEnumerable<Document> GetSampleDocuments(int count, string indexName)
{
    for (var i = 1; i < count + 1; i++)
    {
        var documentId = i.ToString();
        var d = new Document(documentId, indexName);

        // Assume the given index has 2 fields: 'name' and 'age'
        d.Fields.Add("name", "name-number-" + i);
        d.Fields.Add("age", (i + 20).ToString());

        yield return d;
    }
}
```

Now we just need to index these documents using the **CreateDocument** method.
Indexing Documents By Writing The HTTP Request Ourselves

Please have a look at the REST documentation for Document Management APIs.
Modifying Documents

You can modify documents in FlexSearch in two ways:

1. Using one of the FlexSearch clients (C#, TypeScript or JavaScript)
2. By submitting the HTTP request yourself (using a tool like Fiddler, for example)

For demonstration purposes, let's assume we already have an index named `contact` that has a `name` column of type `Text` and an `age` column of type `Int`. This index contains the following document:

```json
{
    _id: 7,
    name: 'Vladamir',
    age: 26
}
```

We want to modify this document in order to correct the name to 'Vladimir'.

Modifying Documents Using The C# Client

First you need to get hold of an API that has the `ModifyDocument` web service method. You can use either the `CommonApi` or the `DocumentsApi`:

```csharp
var documentsApi = new DocumentsApi("http://localhost:9800");
```

After initializing the C# client, we need create a `FlexSearch.Api.Model.Document` instance containing ALL the fields that we want the document to contain, not just the fields that we want to modify. FlexSearch updates all fields. If you only supply one field, then FlexSearch will update that field to the new value and set all the other fields (apart from the system fields, ofc) to their default value.

```csharp
var correctedDocument = new Document("7", "contact");
// We need to supply *ALL* fields, even if we're not changing them
correctedDocument.Fields.Add("name", "Vladimir");
correctedDocument.Fields.Add("age", "26");
```

Now we just need to update the index with the corrected document using the `CreateOrUpdateDocument` method.
var response = documentsApi.CreateOrUpdateDocument(correctedDocument, "contact", "7");

// This method doesn't return any significant data. It just reports any errors.
if (response.Error?.Message != null)
    Console.WriteLine(response.Error.Message);

FlexSearch's document update method is actually an upsert - if the document with the specified ID doesn't exist, a new one will be created.

Modifying Documents By Writing The HTTP Request Ourselves

Please have a look at the REST documentation for Document Management APIs.
Removing an index will delete all its data and configuration files. You cannot bring back a deleted index.

You can remove an index in FlexSearch in two ways:

1. Using one of the FlexSearch clients (C#, TypeScript or JavaScript)
2. By submitting the HTTP request yourself (using a tool like Fiddler, for example)

For demonstration purposes, let’s assume we already have an index named contact.

You might also consider just closing an index, as opposed to deleting it. By closing an index you keep all the data, but the index cannot be changed or searched in any way. See closing an index sample. You can reopen that index at any time.

Removing An Index Using The C# Client

First you need to get hold of an API that has the `DeleteIndex` web service method. You can use either the `CommonApi` or the `IndicesApi`:

```csharp
var indicesApi = new IndicesApi("http://localhost:9800");
```

Now we can call the `DeleteIndex` method, supplying the name of the index as a parameter. The response doesn’t contain any data, it just contains the `Error` part.

```csharp
var response = indicesApi.DeleteIndex("contact");

// This method doesn’t return any significant data. It just reports any errors.
if (response.Error?.Message != null)
    Console.WriteLine(response.Error.Message);
```

Removing An Index By Writing The HTTP Request Ourselves

Please have a look at the REST documentation for Index Management APIs.
Searching Data

Contents

- Search using the C# Client
- Get the Search API
- Create your SearchQuery that contains the query string
- Submitting the search to FlexSearch
- Search by writing the HTTP request ourselves
- Query examples

You can search data in FlexSearch in two ways:

1. Using one of the FlexSearch clients (C#, TypeScript or JavaScript)
2. By submitting the HTTP request yourself (using a tool like Fiddler, for example)

Search Using The C# Client

For demonstration purposes, let's use the country index that already has data in it. Please see the Setting up the demo index section in order to initialize the country index. You can find the source code of this example in the Samples Github repository.

Let's say we want to get all countries that have a population greater than 100 million and have a name that contains United.

GET THE SEARCH API

For accessing the Search web services, you will need an instance of the SearchApi.

```
var searchApi = new SearchApi("http://localhost:9800");
```

CREATE YOUR SEARCHQUERY THAT CONTAINS THE QUERY STRING

For this you simply need to create a SearchQuery object and pass the name of the index and the query string. In the case of our example, the query string will be:

```
gt(population, '100000000') AND allOf(countryname, 'United')
```

By default no columns will be returned from the search, so you also need to specify which columns to bring back. For this example we will bring all columns. This results in the following C# code:
var queryString = "gt(population, '100000000') AND allOf(countryname, 'United')";
var query = new SearchQuery("country", queryString)
{
  Columns = new string[] { "*" }
};

SUBMITTING THE SEARCH TO FLEXSEARCH

For this you just need to call the Search method on the SearchApi:

var response = searchApi.Search("country", query);
InterpretResponse(response);

The InterpretResponse function is defined here:

private void InterpretResponse(SearchResponse response)
{
  if (response.Error?.Message != null)
  {
    Console.WriteLine(response.Error.Message);
  }
  else
  {
    var records = response.Data.Documents;
    var countryNameOfFirstRecord = records[0].Fields["countryname"];
    var totalAvailableNumberOfRecords = response.Data.TotalAvailable;
  }
}

Search By Writing The HTTP Request Ourselves

Please have a look at the REST documentation for Search APIs.

Query Examples

Please refer to the Query Functions to go through search examples.
Search With Predefined Query

Contents

- Get the Search API
- Create your SearchQuery that calls the Predefined Query
- Calling the predefined query without any Variables using the C# client
- Calling the predefined query while passing a variable using the C# client
- Searching using a Predefined Query by writing the HTTP request ourselves
- 1. Doing a request
- 2. Doing a request

You can search using predefined queries in FlexSearch in two ways:

1. Using one of the FlexSearch clients (C#, TypeScript or JavaScript)
2. By submitting the HTTP request yourself (using a tool like Fiddler, for example)

For demonstration purposes, let's use the country index that already has a predefined query named 'agriSearch'. Here is the 'agriSearch' predefined query definition:

```
"predefinedQueries": [
{
  "queryName": "agriSearch",
  "columns": [
    "countryname",
    "agriproducts"
  ],
  "count": 10,
  "indexName": "country",
  "orderBy": "score",
  "orderByDirection": "Ascending",
  "cutOff": 0.0,
  "distinctBy": "",
  "skip": 0,
  "queryString": "allof(agriproducts, 'wheat', 'corn', 'grapes') AND like(countryname, @countryName, -matchall)",
  "returnScore": true,
  "preSearchScript": "",
  "overridePredefinedQueryOptions": false,
  "returnEmptyStringForNull": true,
  "variables": {}
}
]
```

To see how this definition would sound in plain English, see the Predefined Queries documentation.

Get The Search API
For accessing the **Search** web services, you will need an instance of the **SearchApi**.

```csharp
var searchApi = new SearchApi("http://localhost:9800");
```

### Create Your SearchQuery That Calls The Predefined Query

For this you simply need to create a **SearchQuery** object and pass the name of the index and the name of the Predefined Query.

```csharp
var query = new SearchQuery()
{
    IndexName = "country",
    QueryName = "agrisearch"
};
```

### Calling The Predefined Query Without Any Variables Using The C# Client

What will happen in this particular case is that the `countryName` variable (the one referenced by `@countryName` in the `QueryString`) will be blank. The `-matchall` switch will see that the variable is blank and simply *ignore* the `like()` clause (by replacing it with a `matchall` query, hence the name of the switch). We will expect a query equivalent to:

(a) `allof(agriproducts, 'wheat', 'corn', 'grapes')`

Here is how we call the **Search** method:

```csharp
var response = searchApi.Search("country", query);
InterpretResponse(response);
```

The **InterpretResponse** function is defined here:
private void InterpretResponse(SearchResponse response)
{
    if (response.Error?.Message != null)
    {
        Console.WriteLine(response.Error.Message);
    }
    else
    {
        var records = response.Data.Documents;
        var countryNameOfFirstRecord = records[0].Fields["countryname"];
        var totalAvailableNumberOfRecords = response.Data.TotalAvailable;
    }
}

Calling The Predefined Query While Passing A Variable Using The C# Client

We will use romania as the value for the countryName variable. Variable names are case insensitive. Since the @countryName variable has a value, the -matchall switch won't kick in anymore. We would therefore expect a query equivalent to:

```
(b) allof(agriproducts, 'wheat', 'corn', 'grapes') AND like(countryname, 'romania')
```

Here is how we initialize the variable and call the PostSearch method:

```
query.Variables.Add("countryname", "romania");
var response = searchApi.Search("country", query);
InterpretResponse(response);
```

Searching Using A Predefined Query By Writing The HTTP Request Ourselves

When searching you have two options:

1. Do a GET request on the search endpoint, specifying URL parameters.
2. Do a POST request on the search endpoint, putting all parameters in the body as a SearchQuery object.

1. DOING A GET REQUEST

In order to call the agriSearch predefined query without specifying any parameters, you can just call:
This would be the equivalent of calling query [a] defined earlier.

If you want to specify variables, then your only option is to use the POST request.

2. DOING A POST REQUEST

In order to do the POST request you need to build your SearchQuery in JSON and submit it as the body of the request. Let's first build it without specifying any variables.

```
$ http POST http://localhost:9800/indices/country/search -d
{
    queryName: "agrisearch"
}
```

The above would be the equivalent of calling query [a] defined earlier. Now let's add the `countryName` variable with the value of `romania`.

```
$ http POST http://localhost:9800/indices/country/search -d
{
    queryName: "agrisearch",
    variables: {
        countryname: "romania"
    }
}
```

The above would be the equivalent of calling query [b] defined earlier. As you can see from the JSON, you don't need to specify all the properties of the SearchQuery object.

For more examples, please have a look at the REST documentation for Search APIs.
Working With Predefined Query

You can create or update predefined queries within a FlexSearch index in two ways:

1. Using one of the FlexSearch clients (C#, TypeScript or JavaScript)
2. By submitting the HTTP request yourself (using a tool like Fiddler, for example)

For demonstration purposes, let’s use the country index that already has a predefined query named 'agriSearch'. We want to modify the query string of this predefined query to throw an error if the countryname parameter is not supplied.

Therefore we want to modify it from:

```plaintext
allof(agriproducts, 'wheat', 'corn', 'grapes') AND like(countryname, @countryName, -matchall)
```

to:

```plaintext
allof(agriproducts, 'wheat', 'corn', 'grapes') AND like(countryname, @countryName)
```

Creating Or Updating A Predefined Query Using The C# Client

First you need to get hold of an API that has the UpdateIndexPredefinedQuery web service method. You can use the IndicesApi for it. An important thing to note about this method is that acts as an upsert - if the predefined query doesn’t exist on the index, then a new one will be created. The matching is done based on the QueryName property.

```csharp
var indicesApi = new IndicesApi("http://localhost:9800");
```

After initializing the C# client, we need create a FlexSearch.Api.Model.SearchQuery instance containing the new query. Make sure you keep the old query name if you want this to be an update.
var updatedQuery =
    new SearchQuery("country", "alof(agriproducts, 'wheat', 'corn', 'grapes')
    AND like(countryname, @countryName")
    {
        QueryName = "agriSearch"
    };

Now we just need to call the UpdateIndexPredefinedQuery method to register this query against the country index.

    var response = indicesApi.UpdateIndexPredefinedQuery(updatedQuery, "country");

    // This method doesn’t return any significant data. It just reports any errors.
    if (response.Error?.Message != null)
        Console.WriteLine(response.Error.Message);

Creating Or Updating A Predefined Query By Writing
The HTTP Request Ourselves

Refer to API Reference for more information about Document Management APIs.