

Search-Engine Documentation

The search engine should be a Spring application (a JAR file that runs on any server or computer) that works with a locally installed MySQL database, has a simple web interface and API through which it can be controlled and receive search results for a request.

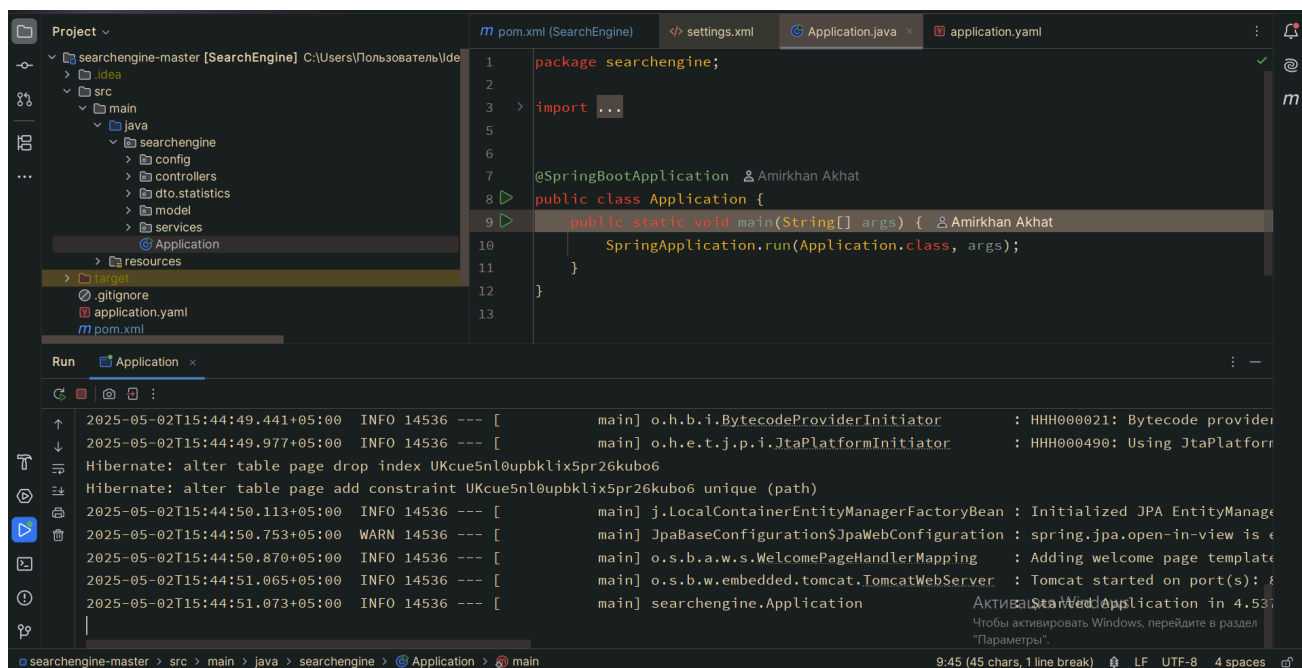
Principles of the search engine

1. In the configuration file, before launching the application, the addresses of the sites that the engine should search are specified.
2. The search engine should independently crawl all the pages of the specified sites and index them (create a so-called index) so that it can then find the most relevant pages for any search query.
3. The user sends a request through the engine API. The request is a set of words by which you need to find the pages of the site.
4. The request is transformed in a certain way into a list of words translated into a basic form. For example, for nouns - nominative case, singular.
5. The index is searched for pages on which all these words are found.
6. The search results are ranked, sorted and returned to the user.

Below you will find all the technical details of the search engine implementation that will help you create a working application. They are divided into several small stages. Each stage is described in detail what needs to be done and how to check the final result.

The first step is to download a project template (a simple Spring Boot application) from the repository. https://github.com/AmirkhanAkhat/search_engine. Open the project and you will see the hierarchy of the source code of the search engine. Next step is to run the main Application.java file which will be located in the src folder.


Run it and the local Apache Tomcat server will be available on your machine by the url <http://localhost:8080/>





Description of the web interface

The web interface (frontend component) of the project is a single web page with three tabs:

- **Dashboard.** This tab opens by default. It displays general statistics for all sites, as well as detailed statistics and status for each site (statistics obtained by requesting `/api/statistics`).

 DASHBOARD

 MANAGEMENT


 SEARCH

DASHBOARD

3
sites

16
pages

2095
lemmas


Metanit - <https://www.metanit.com/>  INDEXING ▲


Status time: 29.4.2025 22:25:24

Pages: 8

Lemmas: 678


Error: null


Kaspi - <https://www.kaspi.kz/>  INDEXING ▼


TengriNews - <https://www.tengrinews.kz/>  INDEXING ▼

Активация Windows
Чтобы активировать Win

- **Management.** This tab provides search engine management tools - starting and stopping full indexing (reindexing), as well as the ability to add (update) a separate page via a link:

 DASHBOARD

 MANAGEMENT

 SEARCH

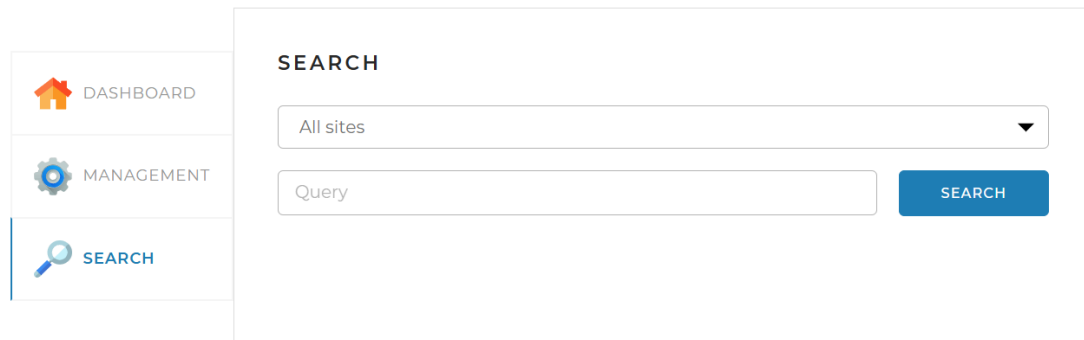
MANAGEMENT

START INDEXING

Add/update page:

ADD/UPDATE

- **Search.** This page is designed for testing the search engine. It contains a search field, a drop-down list with a choice



All information on the tabs is loaded by requests to your application's API. When you click buttons, requests are also sent.

The project is implemented based on Spring Boot and the Maven builder, so the pom.xml file specifies several dependencies that connect Spring Boot itself, the Thymeleaf template engine, and the Lombok library, which provides the ability to use convenient annotations.

All classes are located in the **src/main/java/searchengine** folder. This is necessary for the correct assembly and launch of the application using Maven.

The application launch begins with the main method, located in the Application class, marked with the @SpringBootApplication annotation.

Since Spring Boot includes not only the Spring framework, but also the Apache Tomcat web server, the launched application immediately begins to "listen" to port 8080 (by default), and when you go to http://localhost:8080/ in the browser, the main page of the application begins to open. Below we will analyze in detail how to make such a page open.

The web page itself (the index.html file) is located in the resources/templates folder, so it can be connected in the controller using the Thymeleaf template engine.

There are two controllers in the controllers folder: DefaultController and ApiController. In fact, the index method is created in DefaultController with

the `@RequestMapping("/")` annotation, which means that this method should be called when requesting the main page of the application.

The index method itself contains the return "index". Since the Thymeleaf template engine is running in the project, this code automatically connects and returns as a response the code of the web page of the same name (index.html), located in the resources/templates folder.

In the `StatisticsServiceImpl` class, the data from the configuration file is first connected. The configuration file (application.yaml) is located in the application's resources folder and contains a list of sites with their names and addresses. Open it and see what parameters are written in it.

To get the data from the configuration file into the service, the application does the following:

The `Site` and `SiteList` classes are implemented. They are located in the config folder. They have lombok annotations `@Setter` and `@Getter`, which add setters and getters for all fields to the classes.

The `SitesList` class is marked with the annotations `@Component` and `@ConfigurationProperties` (prefix = "indexing-settings"). Note the prefix value - this is the name of the configuration key, inside which the list of sites is located. The annotations lead to automatic initialization of the object of this class with data from the application.yaml file.

The `StatisticsServiceImpl` service class is marked with the lombok annotation `@RequiredArgsConstructor`, which adds a constructor with arguments corresponding to the uninitialized final fields of the class. This class has only one such field — `sitesList`, so a constructor with only this argument will be added. When creating an object of the `StatisticsServiceImpl` class, an object of the `SitesList` class will be passed to the constructor, which, as mentioned above, is automatically initialized based on the configuration data.

In the `getStatistics` service method, a `StatisticsResponse` class object is gradually assembled from site data, as well as some random information and two arrays specified at the beginning of the method.

Note that all classes on the basis of which the service assembles the final object are located in the `dto` folder and the `statistics` subfolder. The abbreviation DTO stands for Data Transfer Object

Install MySQL server on your computer if it is not already installed and create an empty **search_engine** database in it. Create a user to connect to the database. This can be the root user, who has access to all databases (created when installing MySQL server), or a separate user, who has access only to the database you created, at your discretion.

- Connect dependencies to work with the database:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <scope>runtime</scope>
</dependency>
```

- Create an `application.yaml` config file in the root of the project and explicitly specify the port through which the web application will be available, as well as the access data to the MySQL server:

```
server:
  port: 8080

spring:
  datasource:
    username: user
    password: pass
    url:
      jdbc:mysql://localhost:3306/search_engine?useSSL=false&requireSSL=false
      &allowPublicKeyRetrieval=true
  jpa:
    properties:
      hibernate:
        dialect: org.hibernate.dialect.MySQL8Dialect
    hibernate:
      ddl-auto: update
```

```
show-sql: true
```

- The project already has a configuration file **application.yaml** created in the **resources** folder.

Database structure

site — information about sites and their indexing statuses

id INT NOT NULL AUTO_INCREMENT;
status ENUM('INDEXING', 'INDEXED', 'FAILED') NOT NULL — current status of full site indexing, reflecting the readiness of the search engine to search the site — indexing or reindexing is in progress, the site is fully indexed (ready for searching) or it could not be indexed (the site is not ready for searching and will not be until errors are fixed and indexing is restarted);
status_time DATETIME NOT NULL — status date and time (in case of INDEXING status, the date and time should be updated regularly when each new page is added to the index);
last_error TEXT — indexing error text or NULL if there was none;
url VARCHAR(255) NOT NULL — address of the main page of the site;
name VARCHAR(255) NOT NULL — site name.

page — indexed pages of the site

id INT NOT NULL AUTO_INCREMENT;
site_id INT NOT NULL — website ID from the site table;
path TEXT NOT NULL — page address from the site root (must start with a slash, for example: /news/372189/);
code INT NOT NULL — HTTP response code received when requesting the page (for example, 200, 404, 500, or others);
content MEDIUMTEXT NOT NULL — page content (HTML code).

The path field must be indexed so that searching by it is fast when it contains many links. Indexes are discussed in the course "SQL Query Language".

lemma — lemmas found in texts (see reference: en.wikipedia.org/wiki/Lemmatization).

id INT NOT NULL AUTO_INCREMENT;
site_id INT NOT NULL — website ID from the site table;
lemma VARCHAR(255) NOT NULL — normal form of the word (lemma);
frequency INT NOT NULL — number of pages where the word occurs at least once.
The maximum value cannot exceed the total number of words on the site.

index — search index

id INT NOT NULL AUTO_INCREMENT;
page_id INT NOT NULL — page identifier;
lemma_id INT NOT NULL — lemma identifier;
rank FLOAT NOT NULL — number of this lemma for this page.

The site indexing service was written with code that would take the list of sites from the application configuration and for each site:

- delete all existing data on this site (records from the site and page tables);
- create a new record with the INDEXING status in the site table;
- crawl all pages, starting with the main one, add their addresses, statuses and content to the database in the page table;
- while crawling, constantly update the date and time in the status_time field of the site table to the current one;
- upon completion of the crawl, change the status (status field) to INDEXED;
- if an error occurred and the crawl could not be completed, change the status to FAILED and enter understandable information about the error that occurred in the last_error field.
- To follow the next link, a new thread must be created using **Fork-Join**. This thread will receive the contents of the page and the list of links that are on

this page (the values of the href attributes of the **HTML <a> tags**) using **JSOUP**.

Web page indexing system

Description:

Indexing is the process of forming a search index for a certain amount of information. A search index is a specially organized database (in our case, a MySQL database) that allows you to quickly and conveniently search this information. The speed of searching by a search index in any search engine, as a rule, takes a short time (usually a fraction of a second) compared to a regular search by enumeration over the entire array of information. Remember the difference in speed between a search by enumeration in a simple array and a binary search in a sorted one.

The convenience of storing information in an index is achieved through a properly organized storage structure - a database. In particular, you will not be able to search for information by ordinary text or a set of texts taking into account the morphology of the Russian language and simultaneously evaluate the relevance of the results if you do not have a specially organized search index.

At this stage, you will need to connect the so-called lemmatizer - a library that allows you to get lemmas of words - their original forms. For example, for nouns - this is a word in the nominative case and singular. Lemmatization is convenient to use in search engines, since it allows you to search for the necessary information taking into account morphology. For example, the word "horses" is found on the page, and you enter the search query "horse". A simple search for the presence of the word in the text will not find this page, but if all words are reduced to "horse", then all pages on which this word is present in the original or modified version will be found. There are many [lemmatizers](#). We will use a lemmatizer (the link is provided for reference, you do not need to use the codes from this repository), which is used in the widely used Apache Solr search engine (in particular, it is used as a Wikipedia search engine).

- Add a block to the pom.xml file indicating the path to the repository we created (dependencies will be loaded from it):

```
<repositories>
  <repository>
    <id>skillbox-gitlab</id>
    <url>https://gitlab.skillbox.ru/api/v4/projects/263574/packages/maven</url>
  </repository>
</repositories>
```

- Also add all libraries from this repository to the dependencies section of the pom.xml file:

```
<dependency>
  <groupId>org.apache.lucene.morphology</groupId>
  <artifactId>morph</artifactId>
  <version>1.5</version>
</dependency>
<dependency>
  <groupId>org.apache.lucene.analysis</groupId>
  <artifactId>morphology</artifactId>
  <version>1.5</version>
</dependency>
<dependency>
  <groupId>org.apache.lucene.morphology</groupId>
  <artifactId>dictionary-reader</artifactId>
  <version>1.5</version>
</dependency>
<dependency>
  <groupId>org.apache.lucene.morphology</groupId>
  <artifactId>english</artifactId>
  <version>1.5</version>
</dependency>
<dependency>
  <groupId>org.apache.lucene.morphology</groupId>
  <artifactId>russian</artifactId>
  <version>1.5</version>
</dependency>
```

- Now you need to specify a token to access this Maven repository, because GitLab prohibits public access to libraries. To specify the token, find or create the settings.xml file.

- In Windows, it is located in the directory
C:/Users/<Your User Name>/m2
- In Linux, in the directory
/home/<Your User Name>/m2
- In macOS, at
/Users/<Your User Name>/m2

If there is no settings.xml file, create it and paste the code into it:

```

1  <settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
2      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3      xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
4      http://maven.apache.org/xsd/settings-1.0.0.xsd">
5
6      <servers>
7          <server>
8              <id>skillbox-gitlab</id>
9              <configuration>
10                 <httpHeaders>
11                     <property>
12                         <name>Private-Token</name>
13                         <value>glpat-Viu1C6oUSddYB3JdKviW</value>
14                     </property>
15                 </httpHeaders>
16             </configuration>
17         </server>
18     </servers>
19 </settings>

```

Code was written that will receive the HTML code of the transmitted web page, save it in the database in the page table, convert it into a set of lemmas and their quantities, and then save this information in the lemma table and index database as follows:

The lemmas should be added to the lemma table. If the lemma is not in the table yet, it should be added with a frequency value, condition 1. If the lemma is already in the table, the frequency of the number should be increased by 1. The frequency of the number for each lemma should ultimately correspond to the number of pages on which this lemma occurs at least once.

The anchor lemmas and pages should be added to the table index. For each lemma-page pair, one record should be created in this table indicating the number of this lemma on the page in the rank field.

Test the indexing on each page by specifying the path to it in the web interface of your application and starting its indexing. Remember that when you add a page to the database, it must be linked to records in the site table, which must either already be there or must be created based on one of the site list pages in your application's tables.

In case of an attempt to index a page from some other site, the API command should return an error accordingly.

If the transferred page has already been indexed, before indexing it, it is necessary to remove all information about it from the page, lemma and index tables.

Stage 4. Search system

Goal

Implement an information search system using the created search index.

It is necessary to add code to the program that will, based on a search query (text string), return search results in the form of a list of objects with their properties.

Split the search query into separate words and form a list of unique lemmas from these words, excluding interjections, conjunctions, prepositions and particles. Use the code you wrote in the previous step for this.

Exclude lemmas from the resulting list that are found on too many pages. Experiment and determine this percentage yourself.

Sort lemmas in order of increasing frequency of occurrence (ascending the value of the frequency field) - from the rarest to the most common.

For the first, rarest lemma from the list, find all the pages on which it is found. Then look for matches of the next lemma from this list of pages, and then repeat the operation for each subsequent lemma. The list of pages should decrease at each iteration.

If in the end there are no pages left, then output an empty list.

For example: Word Rank of «horse» is **4.2** and Word Rank «runs» is **3.1**. The absolute relevance is i.e 10.3.

$$R_{abs} = 4,2 + 3,1 = 7,3$$

Relative relevance can be obtained by dividing the absolute relevance for a particular page by the maximum absolute relevance among all pages in a given search result:

$$R_{rel} = 7,3 / 10,3 = 0,7087$$

Sort pages by decreasing relevance (from highest to lowest) and output as a list of objects with the following fields:

- uri — path to the page like /path/to/page/6784;
- title — page title;
- snippet — text fragment in which matches were found (see below);
- relevance — page relevance (see calculation formula above).

API Specification

Launching full indexing — GET /api/startIndexing

The method starts a full indexing of all sites or a full reindexing if they are already indexed.

If indexing or reindexing is already running, the method returns the corresponding error message.

Parameters:

Method without parameters

Response format in case of success:

```
{  
    'result': true  
}
```

Response format in case of error:

```
{  
    'result': false,  
    'error': "Indexing has already started"  
}
```

Stop current indexing – GET /api/stopIndexing

The method stops the current indexing (reindexing) process. If indexing or reindexing is not currently occurring, the method returns the appropriate error message.

Parameters:

Method without parameters

Response format in case of success:

```
{  
    'result': true  
}
```

Response format in case of error:

```
{  
    'result': false,  
    'error': "Indexation not started"  
}
```

Adding or updating a single page — POST /api/indexPage

The method adds to the index or updates a separate page, the address of which is passed in the parameter.

If the page address is passed incorrectly, the method should return the appropriate error.

Parameters:

- url - address of the page you need reindex.

Response format if successful:

```
{
  'result': true
}
```

Response format in case of error:

```
{
  'result': false,
  'error': "This page is outside of the sites
           specified in the configuration file"
}
```

Statistics - GET /api/statistics

The method returns statistics and other service information about the state of search indexes and the engine itself.

If there are no indexing errors for a particular site, you do not need to set the error key.

Parameters:

Method without parameters.

Response format:

```

{
  'result': true,
  'statistics': {
    "total": {
      "sites": 10,
      "pages": 436423,
      "lemmas": 5127891,
      "indexing": true
    },
    "detailed": [
      {
        "url": "http://www.site.com",
        "name": "Site name",
        "status": "INDEXED",
        "statusTime": 1600160357,
        "error": "Indexing error: main
                  website page is unavailable"
        "pages": 5764,
        "lemmas": 321115
      },
      ...
    ]
  }
}

```

Retrieving data from a search query - GET /api/search

The method searches for pages by transferred search query (query parameter).

To display results in portions, you can also set the parameters offset (offset from the beginning of the list of results) and limit (the number of results that need to be displayed).

The response displays the total number of results (count), which does not depend on the values of the offset and limit parameters, and a data array with the search results. Each result is an object containing the properties of the search result (see below for the structure and description of each property).

If the search query is not specified or there is no ready index (the site we are searching for, or all sites are not indexed at once), the method should return the appropriate error (see example below). Error texts must be clear and reflect the essence of the errors.

Parameters:

- query — search query;
- site — the site to search on (if not specified, the search should occur across all indexed sites); specified in address format, for example:
http://www.site.com (no slash at the end);
- offset — offset from 0 for paged output (optional parameter; if not set, the default value is zero);
- limit — the number of results that need to be displayed (optional parameter; if not set, the default value is 20).

Response format if successful:

```
{
  'result': true,
  'count': 574,
  'data': [
    {
      "site": "http://www.site.com",
      "siteName": "Site name",
      "uri": "/path/to/page/6784",
      "title": "The title of the page,
                which we display"
      "snippet": "Snippet of text,
                in which they were found
                matches<b>highlighted
                bold</b>in HTML format",
      "relevance": 0.93362
    },
    ...
  ]
}
```

Response format in case of error:

```
{
  'result': false,
  'error': "Empty search query specified"
}
```

Answers in case of errors

All API commands must implement correct responses in case of errors. Any API method can return an error if it occurs. In this case, the response should look standard:

```
{  
  'result': false,  
  'error': "The specified page was not found"  
}
```

Such responses must be accompanied by appropriate status codes. It is advisable to limit yourself to using codes 400, 401, 403, 404, 405 and 500 when the corresponding error types occur.