University of West Bohemia
Faculty of Applied Sciences
Department of Computer Science and Engineering

# Master Thesis

# Fulltext Search in the Database and in Texts of Social Networks

Pilsen, 2013

Jan Koreň

# Declaration

I hereby declare that this master thesis is completely my own work and that I used only the cited sources.

Pilsen, May 15, 2013

Jan Koreň

# Acknowledgements

I would like to thank to my thesis supervisor Ing. Jan Štěbeták for his assistance, support and inspiring suggestions.

# Abstract

This thesis is focused on design and implementation of the full text search in the EEG/ERP Portal. This full text search capability includes searching text in the EEG/ERP Portal database as well as in related data from social networks, such as LinkedIn or Facebook. With the development of the Portal and the increasing amount of processed data, a proper full text search mechanism for information retrieval is necessary for improving the user experience by enabling efficient document retrieval.

# Contents

# 1 | Introduction

Accessing required information from a large set of data in a quick and user-friendly manner is no longer an unachievable goal. Advancements in the field of information retrieval in the last few decades have made its applications very common. Full-text search, as one of such applications, has in fact become an essential part of everyday's life in a modern society.

This work deals with the topic of full-text search over data belonging to the domain of the EEG/ERP Portal, a piece of software which is being developed at the University of West Bohemia in Pilsen.

The work is organized into two main parts. The first part, theoretical part, includes chapters 2-4 and covers theoretical knowledge used throughout the thesis. Chapter 2 deals with the problematics of full-text search, its core concepts are introduced and a comparison between full-text search and relational database systems is made. Specific open-source full text search engines and libraries are listed in Chapter 3. In Chapter 4, the EEG/ERP Portal together with its underlying technologies, which are currently used for its development, are presented.

The practical part of the thesis is dedicated to creation of the full-text search functionality and is formed by chapters 5-8. Chapter 5 includes analysis of the state of the EEG/ERP Portal before any changes were made, and collecting full text-search requirements. Based on the requirements, a full-text search solution is chosen and the overall system architecture is proposed. Chapter 6 is focused on creating a document model for indexed data, and on all necessary configuration related to indexing and searching these data. Chapter 7 is devoted to implementation of the full-text search functionality into the EEG/ERP Portal application. In Chapter 8, unit tests that confirm the functionality of the created code are described.

The final chapter, Chapter 9, contains a summary of the thesis and presentation of results.

# Part I

# Theoretical Part

# 2 | Full Text Search

The amount of information has grown rapidly in the last few years due to the information explosion caused mainly by the World Wide Web. The result is that nowadays, people are exposed to much more information than they used to be. In order to manage such amount of data and obtain relevant information very quickly (in the order of milliseconds), new powerful techniques operating on vast collections of data were needed. The aim of this chapter is to explain basic concepts applied in full-text search, one of the methods dealing with the problem of searching information.

## 2.1 Information retrieval

Full-text search can be considered as a part of a subdiscipline of computer science known as *information retrieval* (IR) [1]. There is a number of available definitions of information retrieval. According to [2], information retrieval (IR) is loosely defined as

> *"the subfield of computer science that deals with the automated storage and retrieval of documents"*

This definition as well as the definitions from other sources (e.g. from [1]) sum up the purpose of IR in a very general way. There also exist stricter IR definitions that explicitly mention working with unstructured data. One such definition of IR can be found in [3]:

> *"Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers)."*

All IR systems - and full-text search engines are no exception - are based on the same architecture described in Section 2.2.1 which is adapted to requirements

that the specific systems have. In addition, these systems share common IR terminology, whose most important terms are explained in this chapter.

## 2.2 Principles of Full-text Search

The field of full-text search covers a wide range of topics, including efficient algorithms and data structures that enable fast and reliable full-text search over large amount (in practice gigabytes) of data. It is not the aim of this thesis to provide a deeper, more complex insight into this problematics. The final implementation of the full-text search feature will be based on an existing full-text search engine. However, there are several terms and concepts that must be at least briefly explained so that the reader can fully understand the latter text.

### 2.2.1 Full-text Search Engine Architecture

As can be seen in schema in Figure 2.1, full-text search engines comprise several steps in order to provide a user with search results to a given *query*. *Query* is in this case a text phrase, optionally enriched by special operators which serve for refining the query. It is a *user* of the IR system who comes up with the query, expecting that the system will fulfill his or her *information need* by returning relevant search results.

#### Documents

Search engines need to ingest source data first to have something to search on. The basic informational unit which is processed by the search engine and returned to the user in case of match with the query is called *document*. Documents consist of one or more fields with content. In this context, documents inserted to the system represent real documents, such as HTML, PDF files or relational database data. Therefore, a series of data transformations must be made first in order to extract all desired searchable information from different data sources. These transformation steps are discussed in the practical part of the thesis in Chapter 7 and are for reasons of clarity not depicted in the schema.

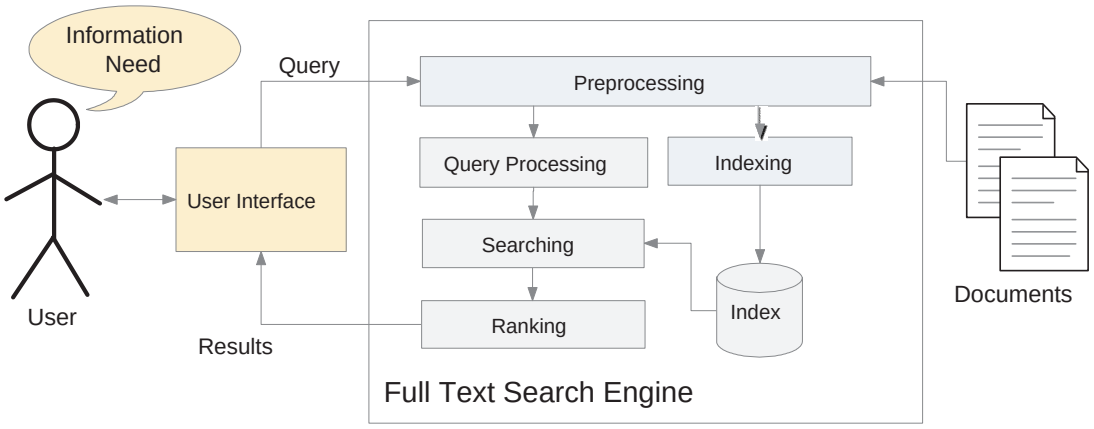


**Figure 2.1:** IR Architecture Schema. Adapted from [4, 5].

### Preprocessing

Both input query and documents usually undergo several preprocessing steps. These steps transform the original query or document content to searchable text units called *terms*. If documents are processed, the created terms are then together with document metadata passed on for indexing. It is important to mention that in order to get correct results, the same preprocessing steps must be applied for both search queries and documents.

The first preprocessing step is called *tokenization*. During the process of tokenization, the document content is treated as a character stream. It is broken into basic elements called *tokens*, in some sources also referred to as *words*, by using appropriate *tokenizers*. Tokenization happens typically at the word level, so a token can be seen as a simple word of ordinary text in most cases. Apart from the obvious creation of tokens by splitting the input stream by the whitespace characters, tokenizer implementations also remove punctuation and may include more sophisticated strategies that are oriented to a given problem domain [3].

After a token stream is created, those tokens that bring no or very little information are often filtered out, thus not indexed. These tokens, mainly articles, prepositions and conjunctions, such as *"the"*, *"an"*, *"to"*, *"and"* etc. are normally called *stop words* and are included in the so-called *stop list*. Although eliminating tokens listed in the stop list can reduce the final index size and speed up processing, it will disable finding phrases made of stop words only, such as *"to be or not to be"*.

The next step concerns *token normalization*. It is a process that includes token

transformations, such as *case-folding* (converting all letters to their lowercase equivalents or vice versa), removal of accents and diacritics etc. The output of token normalization is the creation of *equivalence classes*. The equivalence classes ensure that *"matches occur despite superficial differences in the character sequences of the tokens"* [3].

Sometimes it is useful to retrieve documents containing not only an exact searched keyword, but also one of its possible variations, such as *walk*, *walks*, *walking*, *walked* for the keyword *walk*. The corresponding transformation based on reducing words to their root forms (or *stems*) is known as *stemming*. One of the stemming positive side-effects is a decreased index size due to storing only the created stems in the index. Using stemming, however, may lead to obtaining false positives, if different words are assigned the same stem, or false negatives, if different forms of the same word get different stems assigned. There are several stemming algorithms available and more information about them can be found e.g. in [2].

It is also possible to do *synonym expansion* of tokens. The synonym expansion can be triggered either during indexing, or during querying. Similarly to defining stop words, synonyms are stored typically in a *list of synonyms*. The usage of synonyms is very domain-specific, so irrelevant query results might be obtained if the usage of synonyms is not optimized.

### Indexing

Indexing deals with effective storage of created document terms. Therefore, indexing involves transformation of documents to a more convenient data structure for the needs of information retrieval - the *index*.

### Index

Index is defined by Frakes [2] the following way:

> *"A collection of terms with pointers to places where information about them can be found."*

Based on information found in [1, 2, 3], there are three main methods of indexing – *inverted index*, *signature file* and *bitmaps*. Based on the comparisons made in [1] and in [6], using inverted indexes should be the preferred way due to their search

efficiency and lower index size they require. The remaining two alternatives are recommended to be used only in certain circumstances which are very rare in practice.

### Inverted Index

This data structure, sometimes referred to as the *inverted file* [2], can be thought of as the well-known index at the end of a book. It consists of a *vocabulary* (sometimes also called as *lexicon*) of indexed terms. If the IR terminology is followed, the inverted index can be characterized more precisely. One of such more precise characteristics can be found in [1] and claims that:

> *"An inverted file contains, for each term in the lexicon, an inverted list that stores a list of pointers to all occurrences of that term in the main text, where each pointer is, in effect, the number of a document in which that term appears."*

To visualize the basic idea of the inverted index, Figure 2.2 shows a few indexed terms in the dictionary and their corresponding *inverted lists* (also known as *postings lists* [4]). Inverted lists contain typically document identifiers that point to documents themselves. Inverted indexes are usually stored in a highly compressed form on disk and this why many index compression techniques targeted to their compression have been studied [7].

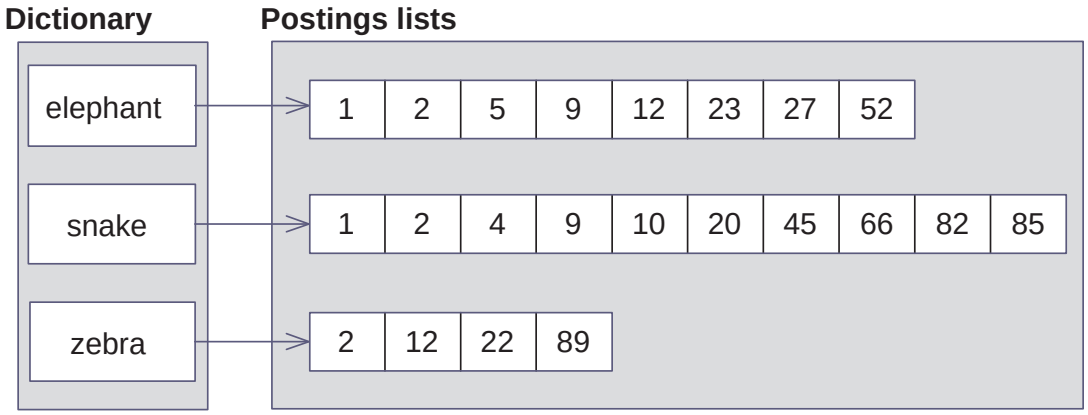


**Figure 2.2:** Inverted Index Schema. Adapted from [3].

### Query Processing, Searching and Ranking

In addition to the steps done in the *processing* phase, a query based on the usage of an inverted index must go through the following three steps [5]:

1. The query terms are searched in the index vocabulary.
2. From each term found in the vocabulary, all the posting lists are retrieved and decoded.
3. The lists are then manipulated so that results of the query are received.

The type of manipulation depends on the type of the used model of information retrieval.

In the case of the *Boolean model*, the query terms are combined with three basic Boolean operators - intersection (AND), union (OR) and difference (NOT). *"The answers, or responses, to the query are those documents that satisfy the stipulated condition."* [1]. This model suffers from several disadvantages. First, it is difficult for many users to formulate correct Boolean queries in order to get expected results [8]. Furthermore, only exact matches are retrieved (document either matches the Boolean expression or not) and all terms are equally important, so the model gives no information about the relevance of retrieved documents.

More advanced retrieval models use *term weights* which provide additional information about importance of individual terms in the documents. The weights are derived from statistical information about a term, usually by calculating the *term frequency* (the number of occurrences of a term in a document) and *inverse document frequency* (the inverse-like value of the number of documents that contain a specific term) for each query term contained in a document. More information about the term weight calculation can be found e.g. in [9, 3]

One of the retrieval models that uses term weights is the *vector space model*. This model represents queries and documents as vectors in a multidimensional vector space, in which each dimension belongs to one term. The final vector size and orientation is composed of individual term weights for each dimension (see Figure 2.3). The vector sizes and angles between them can be then compared by using appropriate similarity measures. Consequently, measuring similarity of a query vector and found document vectors enables *ranking* of the query results.

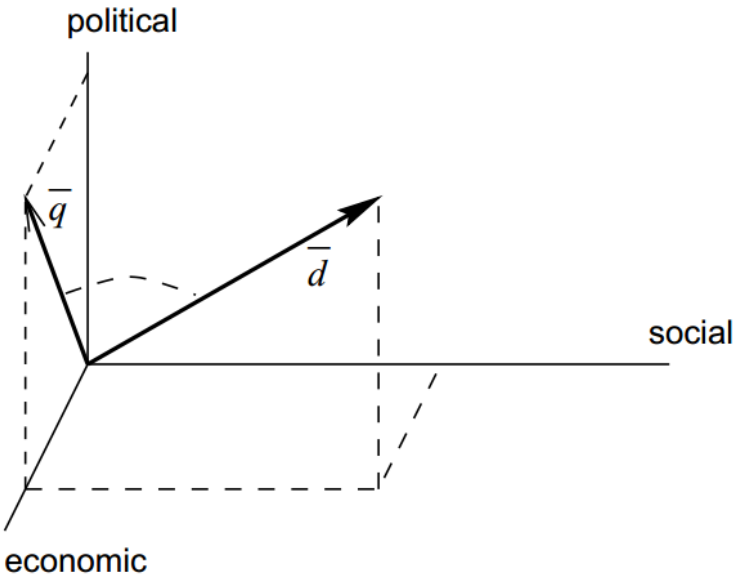


**Figure 2.3:** *Query and Document Representation in the Vector Space Model.* Adapted from [9].

Another IR model used in practice by several search engines is the *probabilistic model*. This model is based on probabilistic theory and its more detailed explanation, which is out of scope of this work, can be found e.g. in [10].

## 2.3 Full-text versus RDBMS Searching

Relational database management systems (RDBMS) are a proved solution to storing large volumes of structured data, in which they excel. As long as there is no need to search in a lot of data stored in a RDBMS, relational databases are a recommended solution for our domain. However, the usage of a classical RDBMS to search a phrase in a block of text, for example in a text column, may become unacceptable due to too much time needed to scan a huge amount of data. Traditional search engines offer the SQL (Structured Query Language) `LIKE` operator. The `LIKE` operator is used for searching a given pattern in a specific column. When using `LIKE`, full table scan is needed to be performed [11]. It means that each row is examined to check if it matches the searched string or not. With the increasing amount of data in the table, it takes more time to process them by the full table scan.

Another disadvantage of using SQL queries containing the `LIKE` operator is the fact that if more terms are searched, the associated SQL query grows in complexity. Since it is intended to store normalized data in RDBMS to avoid data redundancy and ensure their consistency, searched data are often stored in multiple tables. This is why multiple JOIN operations must be used to fetch all data from the corresponding tables. Furthermore, the query must be written the way to ensure finding the records with terms not necessarily next to each other, too. As the query gets more complex, it therefore takes more time to get query results. Another reason why the query execution slows down is that the query needs to match each term individually.

It is important for a user to know how much the found results match the input query. RDBMS simply output the records matching the criteria. This way the found results of the output result set do not contain any information about the relevancy to the searched terms.

Full-text search engines, on the other hand, use different data structures especially designed for full-text searching needs. If there is a lot of data to be searched, inverted index provides a fast way to return matching documents by looking up the corresponding terms in the index dictionary which point to the target documents. Documents are usually stored in a denormalized form, meaning that there may exist duplicate, redundant information across the document collection. In this aspect they are more similar to NoSQL document-oriented databases, such as MongoDB. Furthermore, by using methods described in section 2.2.1, full-text search engines output ranked search results that do not have to match the input query completely.

# 3 | Available Search Engines

According to the comparisons of fulltext search engines in [5] and [12], there are several reasonable options to choose from. Unfortunately, there is not enough comparative benchmarks on performance of search engines. Available comparisons are often not so objective since, for example, only the default settings of search engines are used in the tests or just one use case is applied (as in [12]). This may lead to false conclusions obtained from these benchmarks because search engines' capabilities might not have been fully demonstrated and tested.

It is also worth mentioning that some comparisons were made a few years ago and are likely to be out-dated since these technologies evolve quite quickly. All these experiments showed, however, that relational database full-text search performance is very slow compared to full-text search engines. [13, 14].

In the following paragraphs, the reader can gain a basic overview of several full-text search engines which are considered to be fairly performant and usable for common full-text search scenarios.

## 3.1 Indri

Indri [15] is an academic C++ based text search engine developed at the University of Massachusetts and is a part of the Lemur Project. The engine is interesting because of its implementation as it combines inference networks with language modeling. Its API is accessible also from other languages such as Java or C#. Great support of scaling and support of true multithreaded operations, enabling concurrent adding, querying and deleting documents, belong to its main features. In the technical paper [16] it was shown that Indri is very performant and in comparison with Lucene it achieves even better results in terms of retrieval effectiveness for short queries, index size and performance.

## 3.2 Lucene

Lucene [17] is an open source Java-based search library. It can enrich applications by its ability to index data and search over them. These two main items form what is commonly reffered to as Lucene core. Apart from the core, its latest version comprises useful features related to full-text search problems (e.g. result highlighting). Lucene is highly modularized and its API enables a user to extend its functionality relatively easily [18].

It stores its index as files on disk. When the search is made, segments of indexes are copied to memory, thus its memory consumption is high and mostly stable.

According to [5, 16], it belongs to the fastest engines and its performance is high especially when querying one- or two-word phrases. Its small size of index it creates is also a plus when the lack of memory could be an issue. Due to its portability and its active development and active and numerous community, it has become the leading open source search engine used in many successful projects [19].

According to its official documentation [20], *"Lucene scoring uses a combination of the Vector Space Model (VSM) of Information Retrieval and the Boolean model"*. The Boolean model is used to filter the documents that are then scored.

The current version of Lucene, Lucene 4.0, *„represents a significant milestone in the development of Lucene due to a number of new features and efficiency improvements as compared to previous versions of Lucene.“* [21]

## 3.3 Sphinx

Sphinx [22] is an open source search server distributed under GPL license. It consists of an indexing tool, which is also referred to as indexer, and a searching daemon. Sphinx is written in C++ and is especially designed for indexing database systems (it integrates well with MySQL). The reason behind its connection with MySQL RDBMS is to enable efficient full-text search on a large amount of database data [23]. It allows a user to issue queries with SQL-like syntax for indexing the database content and searching in the index. Besides database indexing, it also supports an XML format for arbitrary data. Considering results from all

found benchmarks involving Sphinx [15, 5, 24, 14], its indexing speed is quick and its search speed belongs to the fastest ones. It offers API bindings for several platforms including Java. Among its strengths we can name quick installation and deployment and a perfect online documentation for an open source project. Extensibility is quite an issue if our application requires additional features [25].

## 3.4 Xapian

According to information found on its official website [26], it is an open-source search library written in C++ based on a probabilistic retrieval model. It provides a user with bindings that allow to use it from a couple of other languages, including Java, C# or Ruby. Its index size is generally larger when compared to other search engines, but this extra piece of information contained in the index allows Xapian to delete or update a document in a more correct way. It is done by storing a list of elements in each document in the termlist table [27]. The benchmarks in [12] from 2009 show that its search performance is said to be equally good as the one of Lucene.

## 3.5 Zettair

Another search engine that received a very positive rating in [5] is Zettair [28]. The comparison in this paper concludes that Zettair has the fastest indexer, provides fast querying speeds and has also the best retrieval effectiveness on average. Zettair is being developed at the Australian university and is written entirely in C. One of its main features is the ability to handle large data sets (100 GB and more) due to its scalability to large collections [28].

There are currently no ports to other languages available, so the full-text search, when using Zettair, must be implemented in C.

## 3.6 Lucene-based Search Solutions

Popularity of Lucene is reflected in the existence of several search solutions which are based on this search library. Because Lucene itself as a search library provides

the core functionality, i.e. indexing and searching documents, its direct integration in applications without any additional enhancements can be cumbersome in the most common use cases. Unless there are special requirements involving access to low-level APIs of Lucene, opting for the already proved search solutions is recommended [29].

The solutions built on top of Lucene can be divided into *search tools* and *search servers*. Search tools extend Lucene in a specific way to cover the problem domain and can be embedded to an application in a form of a library. They are usually dependent on other technologies which must be included in the target application. A widely used example of such search tool is *Hibernate Search*. Search servers, on the other hand, are fully independent standalone applications which communicate with the application typically by using the HTTP protocol. The representatives of this category are *Solr* and *Elasticsearch*.

### Hibernate Search

Hibernate Search [30] is an enterprise search tool that forms a bridge between Hibernate and Lucene. The aim of Hibernate Search is to enable full-text search over persistent domain model by overcoming certain mismatches between the domain model and simple Lucene documents. In order to make Hibernate Search work, Hibernate or JPA must be used as an object-relational mapping between Java objects and database tables. It manipulates with persisted Hibernate entities and makes them searchable by adding them in the Lucene index. By default, Hibernate Search automatically updates Lucene index after an object is saved or updated by using Hibernate. In order to achieve indexing of Hibernate entities, created model classes must be enriched of specific Hibernate Search annotations for full-text search purposes.

### Solr

Solr [31] is an open source search server built on top of Lucene which runs within a servlet container, such as Jetty or Tomcat. Because it can be considered as a Lucene extension, most of the Lucene terminology is used for Solr as well. Unlike Lucene, document fields have to be defined in the application schema file *schema.xml*. Apart from obligatory field definitions in the schema file, it is also

defined how the values of different fields are processed.

Solr extends Lucene by providing many useful features related to full-text search, e.g. keyword highlighting, faceted search, rich document handling or the did-you-mean feature, just to name a few. Since Solr runs as a separate process, it communicates with applications via HTTP requests, which represent query data, and HTTP responses, representing search results found in the index, by exposing its REST-like API. This technology is configuration-driven and compared to Lucene, many full-text search aspects can be set up with no need of writing a single line of Java code.

### Elasticsearch

Elasticsearch [32] is another promising young technology build on Lucene, designed from its early beginnings as a highly scalable solution suitable for big data.

Unlike Solr, it provides a more flexible approach of data definition. Elasticsearch is a one-man project with a steadily growing community, but compared to Solr, it is relatively immature in terms of some of its features [33, 34].

# 4 | EEG/ERP Portal

This chapter describes the motivation behind the creation of the EEG/ERP Portal. Crucial technologies and frameworks that have been used in development of the EEG/ERP Portal are also described.

## 4.1 About EEG/ERP Portal

The EEG/ERP Portal [35] (Figure 4.1) is a web-based application which serves to neuroinformatics researchers as a means of managing, sharing and evaluating measured data. The application also comprises advanced features designed specifically for the needs of the EEG/ERP researchers, such as tools for manipulation with EEG signals.

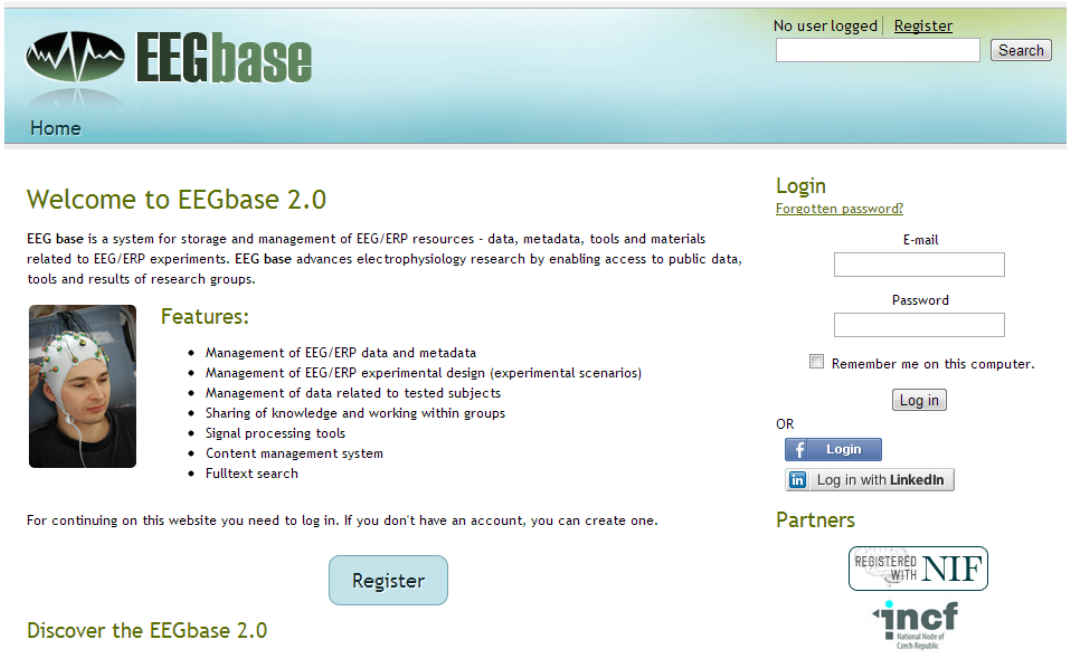


**Figure 4.1:** EEG/ERP Portal Welcome Page.

## 4.2 Hibernate

Hibernate [36] is an open-source object-relational mapping framework, whose purpose is to facilitate storage and retrieval of Java domain objects. It is used in the EEG/ERP Portal to map its data model to the object model and vice versa.

### 4.2.1 Hibernate loading strategies

Hibernate provides two strategies of fetching data from the database to their object representations. As an analogy to database relationships, POJO (Plain Old Java Object) objects can maintain associations to other objects. The strategies differ in the way they treat these object associations, both having their advantages and disadvantages.

The first possible way is to load all associated object collections at the same time when the database record corresponding to the object is fetched. This is called *eager loading*. The second way is to return only the data belonging directly to the object and not to fetch the collections until they are required to be fetched. This approach is known as *lazy loading*.

Lazy loading is generally considered a preferred way in most of the applications. The main reason behind this is performance effectiveness. When a certain POJO object (actually the underlying table record) is accessed, it is sufficient for most of the use cases to fetch only the fields of primitive types, because associated collections are not needed. By using lazy loading in such scenarios, many unnecessary `JOIN` and `SELECT` operations are often avoided. In the end, the operation savings reflect both in lower time and memory costs which results in faster application responses.

On the contrary, overusing eager loading can considerably slow down the application and may lead to the *n+1 selects problem* described e.g. in [37]. However, there are many reasonable situations when all associated object data have to be always available. As an example, one can imagine a requirement to always display writers together with all the books they have written. Then it is completely legitimate to apply eager loading to load the books for each of their authors because of the certainty specified by the requirement.

There are also cases when it is desired to load otherwise lazily initialized collections

eagerly. Hibernate also allows enforcing temporary eager loading.

To understand how lazy loading works in Hibernate, it is important to briefly explain the dynamic proxy pattern.

### Dynamic proxy

Hibernate uses dynamic proxies to implement lazy loading of object properties. The *"Design Patterns"* book written by *Gang of Four* (GOF) [38] describes the proxy pattern the following way:

> *"Allows for object level access control by acting as a pass through entity or a placeholder object."*

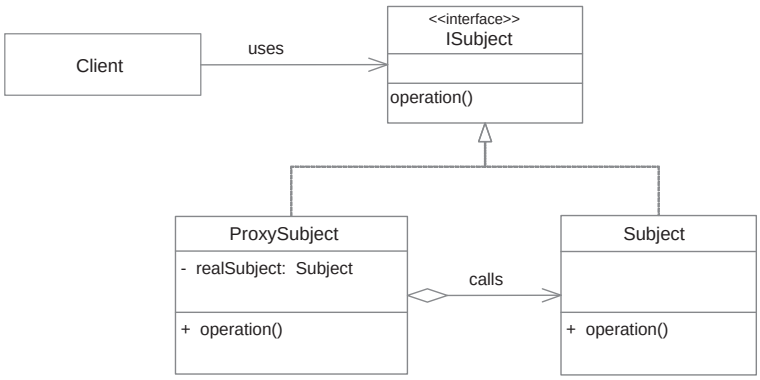


**Figure 4.2:** Proxy Design Pattern.

When an object is required to be loaded, some of its object properties are not actually fetched from the database. Instead, they are represented by their corresponding proxy objects. The proxy object is usually referred to as *stub* and does not hold any actual information. Its only ability is to call the real object it represents. The UML diagram depicted in figure 4.2 helps visualize the whole mechanism. Since these stub objects are in the case of Hibernate created dynamically at runtime by using the bytecode libraries *javassist* or *CGLIB*, we talk about dynamic proxies.

## 4.3 Spring Framework

The Spring Framework [39] facilitates creating Java-based enterprise applications. It applies a principle called *dependency injection* which helps make the classes loosely coupled. The idea of this principle is to let the framework do all necessary wiring of objects, so that the objects can focus on their core responsibilities. By reducing dependencies among application components, the code becomes more testable and reusable. A thorough explanation of the dependency injection principle can be found for example in Martin Fowler's infamous article [40].

### 4.3.1 Spring Social

Spring Social is one of the Spring Framework extensions that was created to enable easier connection of Spring-based applications with Software-as-a-Service (SaaS) providers, like Facebook, LinkedIn, Twitter and others. The EEG/ERP Portal uses Spring Social to interact with LinkedIn. Spring Social offers a variety of methods which wrap the existing LinkedIn REST API calls.

#### LinkedIn REST API

LinkedIn provides REST API to access various information. *Representational State Transfer* (REST) is pragmatically defined in [41] as *"a set of principles that define how Web standards, such as HTTP and URIs, are supposed to be used"*. REST takes the advantage of the HTTP protocol to describe the action that should be performed on a given resource. Resource is an entity that can be identified by URL. Internal domain model of LinkedIn, which includes entities like people, companies and jobs, is mapped to REST resources.

Required information one wishes to obtain can be specified by URI parameters in the JSON format. Examples of its usage are contained in the practical part of the thesis in Chapter 7.

## 4.4 Wicket

Wicket [42] is a component-based framework for creating user interface of web applications. The EEG/ERP Portal's presentation layer is being developed in Wicket by the time of writing this thesis. In the case of the EEG/ERP Portal, it is a replacement of JSP pages and Spring MVC. This decision was made due to clearer separation of application logic and markup which, in effect, leads to more readable and maintainable code.

A web page created by using the Wicket framework consists of a HTML file that determines the page appearance, and a paired Wicket `WebPage` class instance that contains a component hierarchy. The binding of HTML files and their relevant Java classes is done by specifying component identifiers. In case of an HTML page, a value set to the special `wicket:id` tag attribute identifies a Wicket component. The same value of the component name must be then specified in the matching component in Java code.

# Part II

# Practical Part

# 5 | Analysis

This chapter focuses on the analysis of the state of the EEG/ERP Portal before making any changes in the application code.

## 5.1 Current State of Full Text Search

Currently, the EEG/ERP Portal application uses Hibernate Search as a mechanism to index chosen data stored in Oracle RDBMS. Hibernate search provides an annotation interface which serves for indexing purposes. A subset of these annotations is used to mark indexed entities and fields withing them which should form the document in the index. Furthermore, a few field annotations enable to configure how the fields are later processed by specifying *analyzers* to be applied on those fields.

By using Hibernate Search to implement full-text search, the application is enforced to use Hibernate or JPA for data persistence [43]. Using any other technologies than these two results in a malfunctioning application. The main drawback of using Hibernate Search in our case the inability to index data from different sources than the database. Since one of the requirements is to enable searching data from social networks, the current solution can hardly fulfill this requirement.

In the current state of full-text search, the indexed data are stored in memory. Keeping an in-memory index provides fast performance of searching because accessing data in memory is much faster than if data are stored on disk. A disadvantage of the in-memory index is that these data are lost every time the application server stops. This is the reason why the index must be created each time the application starts running. Furthermore, as the size of the index gets bigger, the available amount of memory may be insufficient. The lack of memory then results in disk swapping which causes serious performance degradation.

Since Hibernate Search uses Lucene as a search library which is responsible for indexing and full-text search, the created classes that represent full-text search logic in the EEG/ERP Portal make a heavy use of the Lucene API. Direct Lucene

API usage is considered low-level for these purposes, the created code is more difficult to maintain and it is more likely that new bugs are introduced. In the current state, for example, text highlighting of a subset of found results does not work as expected in some cases.

It is shown in Figure 5.1 that the current implementation of full-text search is, apart from its dependency on Hibernate, tightly coupled with the whole application.

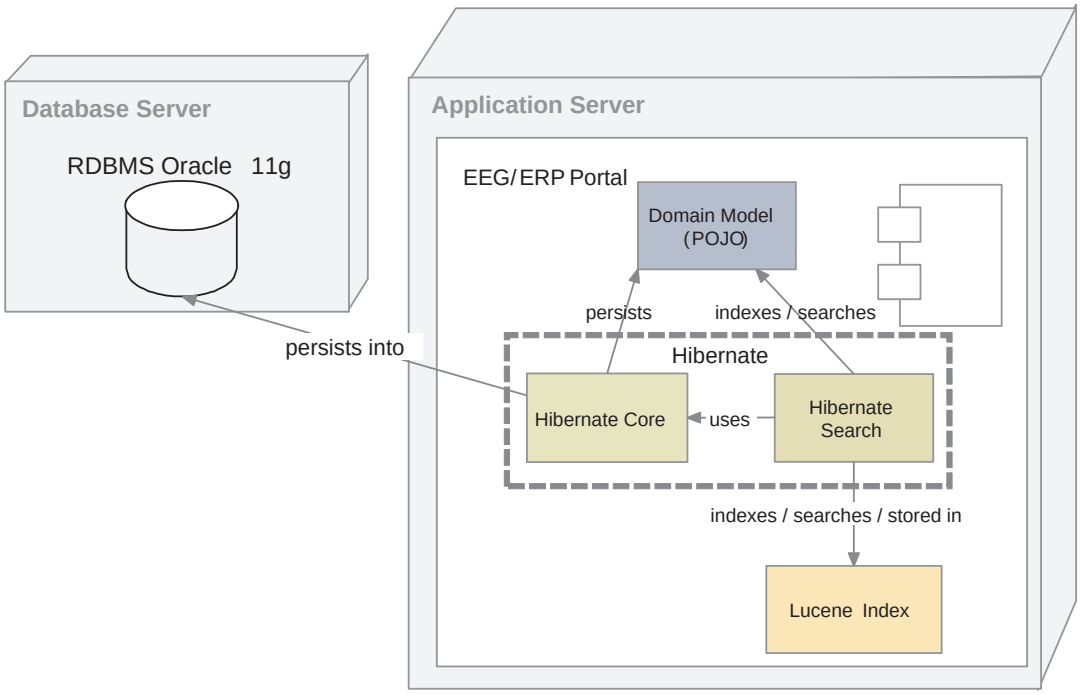


**Figure 5.1:** Architecture with Hibernate Search.

## 5.2 Current State of Integration with Social Networks

Currently, the EEG/ERP Portal is successfully integrated with its LinkedIn group. A user can use the EEG/ERP Portal to publish LinkedIn articles as well as to see all of them. Users can also log in to the EEG/ERP Portal via LinkedIn. Since LinkedIn uses the OAuth 2.0 security protocol for authentication, the OAuth authentication flow is implemented by using Spring Social.

### 5.2.1 Desired Improvements of Full Text Search

#### Appearance

The user interface of the page with search results should basically remain the same. As far as search forms are concerned, it is expected to have:

- one search text field on the search page,
- another one in the header of each web page so the search can be executed from any page of the website.

#### Search Functionality

In terms of search features, there is a need to enable easier navigation among found results. This is why faceting search based on result categories should be made. Furthermore, synonym search is desired to allow finding also the results that contain defined synonymic keywords. Moreover, wildcard search as well as using Boolean AND, OR and NOT operators for querying is required.

### 5.2.2 Social Network Search

Since Hibernate Search is used as a technology responsible, among others, for data indexing, only the data stored in the relational database can be indexed. However, articles and other information found in the LinkedIn group of the EEG/ERP Portal have no direct connection to the database, so Hibernate Search cannot reach these data and therefore cannot save them to the index it keeps. A possible solution which partially solves this problem would be to keep duplicate records about the articles in the EEG/ERP Portal database. Apart from the problem with keeping three copies of basically the same information (the original article published on LinkedIn, its copy in the database and its representation in the Lucene index), those articles published directly from LinkedIn and not from the EEG/ERP Portal via a form would get unnoticed by the application and their corresponding documents in the index would not exist.

## 5.3 Choice of Full Text Search Solution

From the search engines listed in previous parts of the thesis and technologies on which the EEG/ERP Portal is based, the choice of the search engine can be restricted by the following criteria:

- *speed* - Based on information provided in Chapter 3, it turns out that the speeds of compared search engines differ only slightly and the observable differences depend on a specific use case. However, due to their overall great performance, all listed search engines can be considered a suitable solution with respect to this criterion.

- *integration with the EEG/ERP Portal* - The EEG/ERP Portal is based on Java technologies and this is why search engines providing Java API are easier to be integrated to the working infrastructure and therefore preferred.

- *other features and extensions* - Because full-text search engines as they are take care of indexing and searching data, it is desirable to have a set of built-in features, such as result highlighting, faceted search, synonym search or more-like-this search, to make building full-text search easier. It is taken for granted by the end users to use the full-text search with some of such features implemented.

- *independence on data sources* - The chosen search engine must be able to accept data from various sources and to not be limited to only one specific data source, such as relational database. The reason behind this is the mentioned need to index LinkedIn articles as well as to enable further possible indexing scenarios in the future, such as indexing .pdf or XML files.

- *independence on other technologies* - This criterion means that the search engine should not rely on a specific technology to be used. Dependence of Hibernate Search on Hibernate or heavy orientation of Sphinx on MySQL may serve as the examples of the search engines which perform well if certain conditions are met, but cannot run or do not perform well if not.

- *community* - Numerous and active developer community also plays a big role in the final choice. The bigger community around the search engine is, the higher is the chance that the engine development will not stop early, new

features will be introduced and found bugs will be resolved quickly. Although relatively new one-man projects can look very promising and their future development is more likely to be managed by a still growing community, their stability is not guaranteed and there is a higher risk of stopping the project. A well-documented project, many available tutorials, active user groups and a large number of users are good signs of project stability and ensure that there will be probably someone from the community willing to help solve given problems.

The search engines were evaluated based on these criteria. Since it was stated for the speed criterion that its differences for the discussed search engines were not significant, it is not included in the final evaluation. This evaluation can be seen for reasons of clarity in table 5.1

**Table 5.1:** Comparison of Full Text Search Engines.

| Search Engine | Integration | Extension | Independence | Community |
|---|---|---|---|---|
| Indri | ✔ | ✗ | ✔ | 1/5 |
| Sphinx | ✔ | ✗ | ✔ | 3/5 |
| Lucene | ✔ | ✔ | ✔ | 5/5 |
| Zettair | ✗ | ✗ | ✔ | 1/5 |
| Xapian | ✔ | ✔ | ✔ | 1/5 |

It is worth mentioning that the last criterion, *Community*, cannot be evaluated in an exact manner. This criterion involves the size of mailing lists, the number of search results about the search engines found on Google, the number of related blog posts as well as the number of posts on specialized websites, such as *Stack Overflow*.

The result of the comparison is to base the the EEG/ERP Portal full-text search feature on the Lucene search library. As mentioned in section 3.6, the Lucene-based search applications are in practice built by using more sophisticated search solutions (also mentioned in section 3.6).

### 5.3.1 Chosing a Lucene-based Solution

From the solutions based on Lucene listed in section 3.6, it is obvious that Hibernate Search cannot be used due to the reasons that can be found in section

5.1. Although both Solr and ElasticSearch are likely to be good full-text search solutions for the EEG/ERP Portal, it was decided to choose a more proved and mature alternative (by the time of making this decision, in November 2012). This is why the preference was given to Solr.

## 5.4 Using Solr

This section briefly explains the specifics of Solr and is devoted particularly to its deployment. It is needed to provide the explanation of some of its aspects upon which the forthcoming text is based.

### 5.4.1 Installation

The whole Solr distribution can be downloaded from the Solr website [31]. It comes in a form of a directory that contains everything needed to run the application, including its .war archive. Hence, the whole directory can be copied to the target destination.

### 5.4.2 Configuration

Apart from the Solr application itself, the distribution also includes a set of configuration files. Solr can be configured in many ways by modifying its .xml files. In `solr/collection1/conf` directory, there are two important .xml files that are customized in Chapter 6:

- `schema.xml` - This file contains information about document fields and their processing in indexing and querying phases.
- `solrconfig.xml` - It contains server-related configuration. Particularly, request handlers and response writers are set in this file. They handle incoming requests and generate responses of a search, respectively.

Solr provides an administration user interface which is depicted in Figure 5.2. It helps a user analyze and optimize a current Solr configuration by running and analyzing queries.

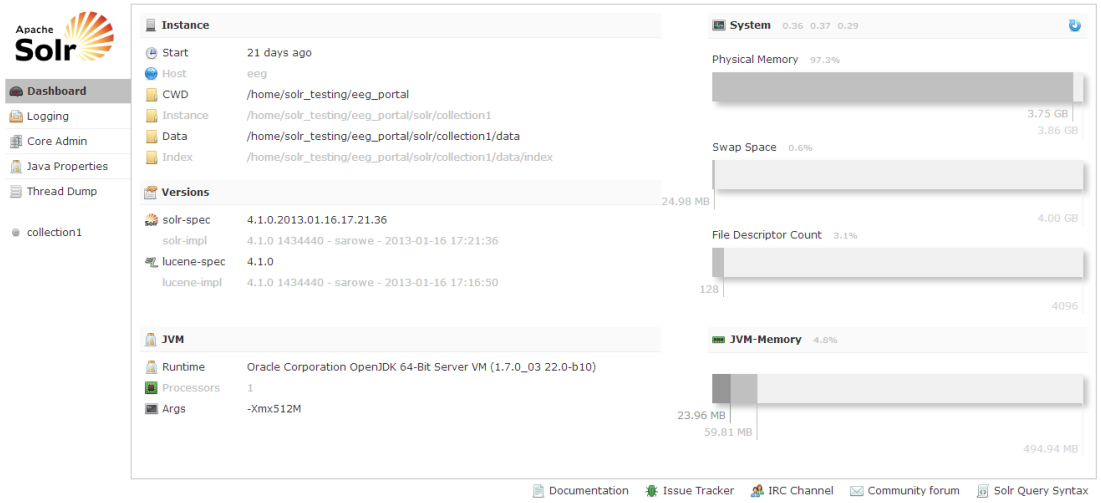


**Figure 5.2:** Solr User Interface.

### 5.4.3 Running Solr

By default, Solr runs on the embedded Jetty server which can be started by running `java -jar start.jar` relatively to the Solr home directory. Two Solr instances can be found at the host address `http://147.228.63.134/solr` of the EEG/ERP Portal server on ports 8585 and 8686 for production and testing, respectively. In order to run the instances on the server automatically, the shell scripts (one for each instance) were made. The mentioned port numbers are set in these scripts by specifying the `jetty.port` system property, such as `-Djetty.port=8686` for the testing Solr server.

### 5.4.4 Solr Integration Possibilities

There are basically two ways of how to integrate Solr with an existing application. The first way is based on configuration of periodically executed SQL queries that handle database indexing. To do this, the *DataImportHandler* (DIH) tool is used. Another way is to apply a programmatic approach and to use a Java API called *SolrJ*.

### DataImportHandler

*DataImportHandler* (DIH) offers fast indexing of data from a variety of sources, including relational databases. The extraction of database data for indexing is based on creating custom SQL queries. The SQL queries serve as a mapping mechanism between query target attributes and document fields.

Although DIH configuration is straightforward to set and its indexing performance is excellent, it has several limitations and disadvantages:

- *data model changes* - In order to index new data and to enable delta imports, i.e. to index only the changed data since last indexing, it is necessary to add a new time stamp column for each table whose data we wish to index. Each record of these new columns contains a time value of the last indexing activity. Another issue that needs to be solved on the database level is managing deleted database records. There exist several methods to reflect the changes in the index and can be found for example in the Rafał Kuć's article [44].

- *dealing with changes of the database schema* - If the database schema changes, all affected SQL queries must be rewritten in order to index the database correctly again.

- *code separation* - The logic in the form of the SQL queries is out of reach of the EEG/ERP Portal application logic since the queries must be written in a special configuration file managed by the Solr server.

- *for indexing only* - DIH is just an indexing tool that creates index documents. It cannot be used for retrieving the documents, so their searching must be implemented in a different way.

### SolrJ

Solr provides the official Java API for integration of Java applications with the Solr server called SolrJ. SolrJ client can be used not only for indexing, but also for integration of an existing Java application with the Solr server. Compared to DIH, SolrJ offers richer debugging possibilities.

SolrJ offers two possible ways of running Solr. Apart from the remote communication with the Solr server provided by the `HttpSolrServer` class, Solr can also be embedded within the EEG/ERP Portal application by running as the `EmbeddedSolrServer` instance. The latter option is generally not a recommended way to run Solr. As the Solr documentation [45] says:

> *"The simplest, safest, way to use Solr is via Solr's standard HTTP interfaces. Embedding Solr is less flexible, harder to support, not as well tested, and should be reserved for special circumstances."*

Following the stated requirements, the embedded version of Solr is not a suitable option for the EEG/ERP Portal either.

### Maven Dependencies

EEG/ERP Portal uses the Maven build automation tool. For both Solr and SolrJ, there are available Maven dependencies that can be added to the Maven *pom.xml* file. Listing 5.1 shows these added dependencies.

**Listing 5.1:** Solr and SolrJ Maven Dependencies.

```
<dependency>
  <groupId>org.apache.solr</groupId>
  <artifactId>solr-core</artifactId>
  <version>4.1.0</version>
  <exclusions>
    <exclusion>
      <artifactId>slf4j-jdk14</artifactId>
      <groupId>org.slf4j</groupId>
    </exclusion>
  </exclusions>
</dependency>

<dependency>
  <groupId>org.apache.solr</groupId>
  <artifactId>solr-solrj</artifactId>
  <version>4.1.0</version>
</dependency>
```

### 5.4.5 Configuring SolrJ with Spring

In order to use SolrJ, it is needed to create a `SolrServer` class instance to enable communication with the remote Solr server. This is why a Spring bean defining the `HttpSolrServer` instance has to be created in the Spring application context. The code sample in Listing 5.2 shows how to configure the bean.

**Listing 5.2:** Configuration of the Solr Server Bean.

```
1 <bean name="solrServer" class="org.apache.solr.client.solrj.impl.↩
     HttpSolrServer">
2   <constructor-arg name="baseURL" value="${solr.serverUrl}"/>
3   <constructor-arg name="client" ref="httpClient"/>
4   <property name="connectionTimeout" value="${solr.↩
       connectionTimeout}"/>
5 </bean>
```

The bean specifies a reference to the Apache Commons's HttpClient bean which was created due to the need for secure communication.

### 5.4.6 Securing Solr

Solr is not secured after the installation. Hence, anyone can use its web services to access and manipulate indexed data, which is a serious security risk. In order to enable authorized communication with the server, it has been decided to use HTTP Basic Authentication to protect the access to the Solr server on the path level.

The description of setting basic authentication of Solr running under Jetty is omitted because the exact configuration steps can be found in the corresponding section of Solr documentation [46].

On the EEG/ERP Portal application side, the created `BasicAuthHttpClient` class extends `DefaultHttpClient` by providing preemptive basic authentication. It means that the credentials are automatically passed on the first request. Listing 5.3 shows the corresponding lines in the constructor of the `BasicAuthHttpClient` class.

**Listing 5.3:** Preemptive Basic Authentication.

```
1 BasicCredentialsProvider credsProvider = new ↩
     BasicCredentialsProvider();
```

```
2 credsProvider.setCredentials(new AuthScope(url.getHost(),
3             AuthScope.ANY_PORT),
4             new UsernamePasswordCredentials(username, password));
5 setCredentialsProvider(credsProvider);
```

All parameters necessary to use the `BasicAuthHttpClient` instance are set in its Spring bean as shown in Listing 5.4 below. Note that in order to achieve thread-safe communication, `ThreadSafeClientConnManager` is used as its connection manager.

**Listing 5.4:** `BasicAuthHttpClient` Spring Bean.

```
1  <bean id="httpClient" class="cz.zcu.kiv.eegdatabase.logic.util.↩
       BasicAuthHttpClient">
2    <constructor-arg name="url"  value="${solr.serverUrl}"/>
3    <constructor-arg name="username"  value="${solr.username}"/>
4    <constructor-arg name="password" value="${solr.password}"/>
5    <constructor-arg name="connManager">
6      <bean class="org.apache.http.impl.conn.tsccm.↩
         ThreadSafeClientConnManager">
7        <property name="defaultMaxPerRoute" value="${solr.↩
           defaultMaxConnectionsPerHost}" />
8        <property name="maxTotal" value="${solr.maxTotalConnections}"↩
           />
9      </bean>
10   </constructor-arg>
11 </bean>
```

## 5.5 Proposed System Architecture

Based on the text from the preceding paragraphs, a new system architecture was proposed. The architecture schema is depicted in Figure 5.3. SolrJ API is used for both indexing and the needed interaction with the EEG/ERP Portal. There are two Solr Server instances running. One of them is used for maintaining indexed data of the production environment. The second one is utilized for testing purposes. Using a separate Solr test server was preferred to its application-embedded version due to the reasons described in previous text.

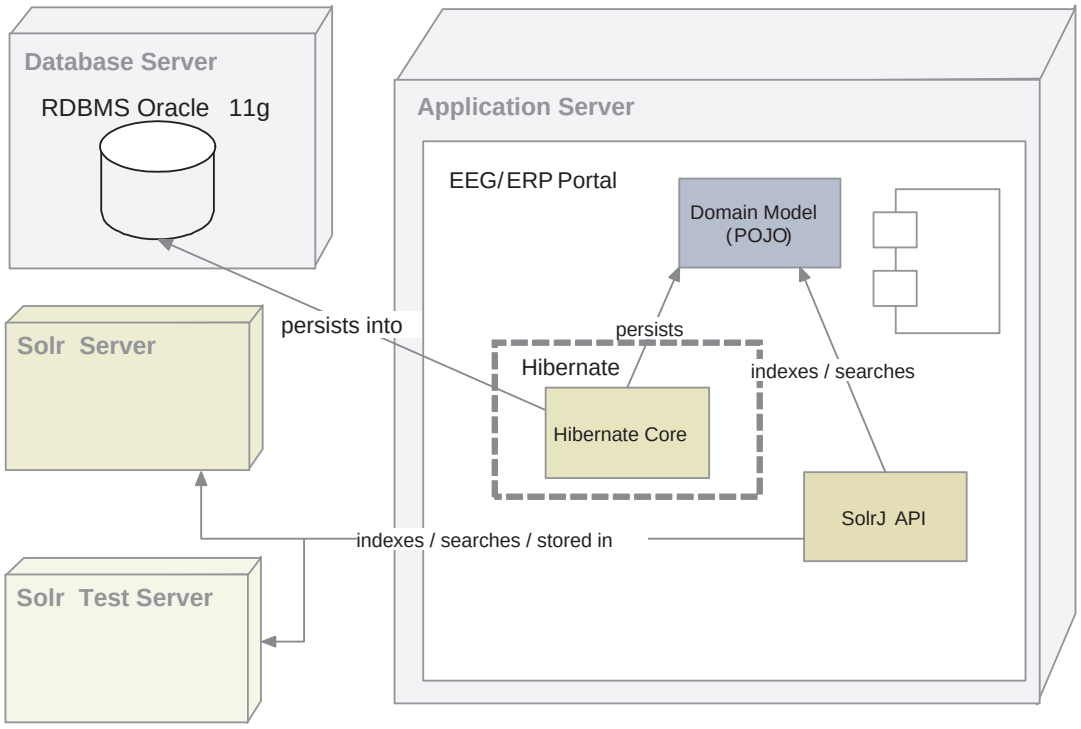


**Figure 5.3:** Architecture with Solr Included.

# 6 | Index Design

The aim of the following chapter is to design an index structure for the data to be searched. Based on the described limitations and collected requirements from Chapter 5, the index structure is proposed and its advantages and disadvantages are mentioned in the text.

## 6.1 Identifying domain entities

To design a document structure properly, it is crucial to know what kind of information is going to be searched and what should be displayed to a user. This is the reason why the EEG/ERP Portal application's domain model must be explored first.

Based on the search requirements, the full-text search functionality will be oriented on searching and displaying information about five main domain entities: articles, experiments, persons, research groups and scenarios.

### 6.1.1 Relation to POJO classes

The corresponding POJO representations of the domain entities are, however, logically split into multiple POJO classes. Nevertheless, there are several classes which contain the core data belonging to their representing domain entities. These POJO classes are of our main interest and will be considered parent classes in the full text search context. There exist one-to-one or one-to-many associations among parent POJO classes and some other, child classes. The Solr documents created from the class hierarchy, which is identified in the text below, will mainly consist of textual data. In practice, it means that mostly `String` object fields will be stored in the documents.

The following list captures relations between entities of the domain model and their respective parent POJO objects:

- *article* - Articles are represented by the `Article` class. It contains the `title` and `text` string fields that hold values of article title and its text, respectively. Articles can be commented on, so each article may have one or more article comments associated. Text of the comments themselves is contained in the `ArticleComment` instances.

- *experiment* - The `Experiment` class has the `environmentNote` field to describe the experiment environment. Apart from this field, it also refers to related `Weather`, `Disease`, `DataFile`, `Hardware` and `Software` objects that include information about weather, diseases of a tested subject, related data files, and used hardware and software during an experiment, respectively.

- *person* - The `Person` class contains a person's name, surname and a note about the person. Although it also has associations to other classes, it is not necessary to include them for indexing and searching purposes.

- *research group* - The `ResearchGroup` class keeps a name, title and description of a research group and as in the case of the `Person` class, its structure matches the structure of its domain entity, so no other referenced class instances are needed.

- *scenario* - The class `Scenario` contains its name, title and description. The class itself corresponds to its domain entity.

After inspecting the aforementioned classes, it turns out that there are several fields that most of the objects have in common. For example, many POJO classes possess the `title` and `description` fields. It is also worth noticing that the class fields `description`, `text` and `note` have a very similar meaning in this context. This repeatability and similarity of class fields can be conveniently used to create a single type of a Solr document.

On the other hand, there are several class fields that are distinctive only for a few or for a single domain entity (such as the `temperature` field in the `Experiment` class or the `type` field in the `Hardware` class to specify parameter data type attribute). Unlike a database, not populating some document fields has no additional overhead [29].

### 6.1.2 Single versus Multiple Solr Cores

Solr can maintain multiple schemas with their own configuration stored in different *cores* that run within a single Solr instance. In this case it is necessary to decide whether it is better to create one universal document type and use a single Solr core or to use multiple Solr cores for each of the domain classes, each managing its own document type.

Both alternatives have their strengths and weaknesses. If separated searching of multiple document types is expected (e.g. in the case of different applications using the same Solr server instance, or having multiple language mutations of an application) then it is more reasonable to use multiple cores with different schemas, since the core can run independently on each other. To support searching by using a single search box, the final choice depends strongly on data heterogeneity, i.e. how much different the data in the document types are.

According to the Solr wiki [47], *"The more heterogeneous (different kinds of data) you have in one field or in one index, the less useful it is."* If different types of things are put into a common field, the field terms that are frequent for one entity do not have to appear frequently anymore. It affects scoring, because Solr uses document frequencies for its calculation. Thus, with the increasing heterogeneity of data, the quality of final document scores decreases [29].

Another factor worth considering is the expected total number of documents. Due to scalability reasons, it is in practice recommended to split documents to multiple cores if their number stored in a single index exceeds one million [29].

In the EEG/ERP Portal, the are some fields of different entities that can be put together under one document field. Furthermore, the document types tend to be similar and the total number of documents is not expected to be over one million.

Thus, the choice of having a single core, meaning using a single index, will be preferred.

## 6.2 Ensuring Document Uniqueness

When storing different types of documents under a single index, a problem of document uniqueness arises. Although a relational database record can be, for

example, uniquely identified by its primary key and a source table, its information about the source table is lost after its corresponding Lucene/Solr document is created. Hence, the original uniqueness is lost as there might be documents created from records coming from different tables and having the same id. This is why a custom mechanism ensuring a unique id for each document was implemented, as can be read in chapter 7.

The need of document uniqueness is reflected on the document level as well. The following fields related to unique identification of documents were added in the Solr schema file:

- *id* - The purpose of the `id` field is to keep the original object ID value.
- *class* - Identifies a type of an indexed document. There are two reasons of having this field. First, in combination with the `id` field value, it provides a means of identification of a specific object. Therefore, a link to the found record in the application logic can be created. Second, the `class` field value can be used for displaying a category of a found result to a user as well as for enabling categorization of search results.
- *uuid* - This field serves as a unique identifier for each document in the index. It consists of two concatenated parts. The first part is the whole class name of an indexed object, the second part is the actual ID value of the object. To provide an example, the `uuid` value of an article with ID 20 is `cz.zcu.kiv.eegdatabase.data.pojo.Article20`.

These Solr fields were added to the `schema.xml` configuration file, whose incriminated lines are displayed in Listing 6.1.

**Listing 6.1:** Configuration of Identification Fields in the Solr Schema.

```
1 <field name="uuid" type="string" indexed="true" stored="true" ↩
     required="true" multiValued="false" />
2 <field name="id" type="int" indexed="false" stored="true" required=↩
     "true" multiValued="false" />
3 <field name="class" type="string" indexed="true" stored="true" ↩
     omitNorms="true"/>
```

## 6.3 Proposed Document Structure

Based on the object hierarchy and the fields these objects contain, the following document fields were configured:

- `title` - Title of a parent object.
- `text` - A longer textual sequence, such as description or a note, of a parent object.
- `name` - It contains a person's name (both first name and last name).
- `source` - This field determines a source of indexed documents. It can be set either to `linkedin`, if a LinkedIn article was indexed, or to `database`, if a POJO instance was indexed.
- `temperature` - Stores temperature during an experiment.
- `param_datatype` - It represents a data type or another type field of a parent POJO class.
- `file_mimetype` - Stores `mimetype` values of parent POJOs. In this case, it concerns the `Scenario` class.
- `child_title` - It is a multivalued field which stores all titles of child objects.
- `child_text` - This field is analogous to the `text` field. The difference is that it is multivalued and stores all textual values of child objects.
- `child_param_datatype` - It is used for the same reasons as the `param_datatype` field, except that it is used for child object values.

## 6.4 Result Highlighting

In order to enable highlighting of search results, the `solrconfig.xml` configuration file had to be modified. This modification in a form of default highlighting settings is shown in Listing 6.2. The fields that will be highlighted are surrounded by the `str` tag with the `hl.fl` value of its `name` attribute (see line 5 in Listing 6.2).

For more information about the used configuration properties, see Chapter 5 or refer to the official Solr documentation at [48].

**Listing 6.2:** Highlighting configuration.

```
<requestHandler name="/select" class="solr.SearchHandler">
  <lst name="defaults">
    ...
    <str name="hl">true</str>
    <str name="hl.fl">title text name source child_title child_text↩
        child_param_datatype</str>
    <str name="hl.requireFieldMatch">false</str>
    <str name="hl.usePhraseHighlighter">true</str>
    <str name="hl.highlightMultiTerm">true</str>
    <str name="hl.snippets">10</str>
    <str name="f.text.hl.snippets">5</str>
    <str name="hl.fragSize">100</str>
    <str name="hl.mergeContiguous">true</str>
    <str name="hl.encoder">html</str>
    <str name="hl.simple.pre">&lt;b&gt;</str>
    <str name="hl.simple.post">&lt;/b&gt;</str>
    <str name="f.title.hl.alternateField">title</str>
    <str name="f.text.hl.alternateField">text</str>
  </lst>
  ...
</requestHandler>
```

## 6.5 Handling Synonyms

Solr has a built-in synonym expansion support and used synonyms are defined in a plain text file. The file `synonyms.txt` is used and some of its synonym definitions look as shown in Listing 6.3.

**Listing 6.3:** Example of Synonym Definitions.

```
experimentation, test => experiment
holiday, vacation
```

In the first line in Listing 6.3, the terms on the left side of "=>" are replaced by the term `experiment`, but not vice versa. The rule in the second line treats all terms equally and does expansion of each term to the remaining synonyms.

Synonym handling can be configured in Solr in schema field type definitions. Listing 6.4 shows the configuration of synonym handling for the `text_general` field type which is used for storing searched text.

**Listing 6.4:** Synonyms configuration.

```
<analyzer type="index">
  ...
  <filter class="solr.SynonymFilterFactory" synonyms="synonyms.txt"↩
      ignoreCase="true" expand="false"/>
  ...
</analyzer>
```

In this case, synonym expansion happens at index time. Since the `expand` attribute is set to `false`, all equivalent synonyms will be reduced to the first one in the list [49].

Index time synonym expansion should be preferred to query time synonym expansion [29, 49]. The main reason of doing so is to support using multi-word synonyms. In query time, the query parser performs whitespace tokenization that always precedes the analyzer chain. Therefore, multi-word synonyms are split into separate tokens and are not recognized. However, index-time synonym expansion is less flexible. If any changes in the synonym file are made, the whole index must be rebuilt in order to apply the changes.

## 6.6 Autocomplete Support

In order to enable the autocomplete functionality, an extra document field needed to be added. The field `autocomplete` contains all searched phrases which are copied from selected source fields. It is done by utilizing the `copyField` element for all affected source fields. The excerpt from the schema.xml configuration file can be seen in Listing 6.5 below.

**Listing 6.5:** Configuration of Autocomplete.

```
<field name="autocomplete" type="text_autocomplete" indexed="true" ↩
   stored="true" multiValued="true" />
...
<copyField source="title" dest="autocomplete"/>
<copyField source="name" dest="autocomplete"/>
<copyField source="child_title" dest="autocomplete"/>
```

```
<copyField source="file_mimetype" dest="autocomplete"/>
<copyField source="param_datatype" dest="autocomplete"/>
<copyField source="source" dest="autocomplete"/>
```

### 6.6.1 Autocomplete Field

A new Solr field designed for text completion was created. The configuration of the autocomplete field type can be seen in Listing 6.6: The tokens stored in this field are processed both in index and query phase. Both phases have one tokenizer and one filter class in common. First, the usage of `KeywordTokenizerFactory` which, according to the Lucene/Solr documentation, *"Emits the entire input as a single token."* ensures treating stored field data as a single phrase. To prevent letter case mismatches, `LowerCaseFilterFactory` was applied.

During the search phase, the terms related to autocomplete are additionally made by using Solr `EdgeNGramFilterFactory` class. This class creates n-grams for an input term. Based on these n-gram tokens, the target phrase will be suggested. The minimal and maximal n-gram lengths are set to 2 and 50, respectively, which allows meaningful phrase suggestions up to the total string length of 50 characters. In practice, it means that, for example, the filter output for the input string *"hello"* will be 4 tokens: *"he"*, *"hel"*, *"hell"*, and *"hello"*. By typing a string that matches one of these n-gram tokens, the original *"hello"* string will be suggested.

**Listing 6.6:** Autocomplete Field Type.

```
<fieldType name="text_autocomplete" class="solr.TextField" ↩
    positionIncrementGap="100">
  <analyzer type="index">
    <tokenizer class="solr.KeywordTokenizerFactory"/>
    <filter class="solr.LowerCaseFilterFactory"/>
    <filter class="solr.EdgeNGramFilterFactory" minGramSize="2" ↩
        maxGramSize="50" />
  </analyzer>
  <analyzer type="query">
    <tokenizer class="solr.KeywordTokenizerFactory" />
    <filter class="solr.LowerCaseFilterFactory"/>
  </analyzer>
</fieldType>
```

### 6.6.2 Using Edismax Parser

The default query parser set in the Solr configuration is `LuceneQueryParser`. Although it offers a lot of possibilities, any Solr syntax mistakes result in `ParseException`. To avoid checking every potential state which leads to a syntax error, such as an odd number of parentheses, it is recommended to use `ExtendedDismaxParser` [29]. This parser is more tolerant to syntax errors and guarantees that an exception will not be thrown in most of the cases. Instead, an empty search result set is returned. Compared to the default parser, its performance is generally the same [29]. Therefore, the Edismax parser is an appropriate choice for common search use cases since it offers the same query possibilities as the default parser. Listing 6.7 depicts the lines responsible for configuration of Edismax parser.

**Listing 6.7:** Configuration of the Edismax parser.

```
1 <requestHandler name="/select" class="solr.SearchHandler">
2   <lst name="defaults">
3     ...
4     <str name="defType">edismax</str>
5     ...
6   </lst>
7 <\requestHandler>
```

# 7 | Implementation

This chapter describes the implementation steps to create a full text search. It is based on the designed index schema and Solr configuration described in Chapter 6.

## 7.1 Indexing Data

It was mentioned in Chapter 5 that it was needed to index data coming from different data sources. Although the ways to build input Solr documents differ, the principle of indexing as the whole is the same for all data sources and can be divided into the following steps:

1. A Solr document is created from input data.
2. A connection between the Solr server and the EEG/ERP Portal application is established.
3. The document is sent to the Solr server.
4. Information about the created document is logged.

It is obvious that the steps 2,3 and 4 are common for all kinds of data sources. To separate common functionality of indexing from the specialized step during which a Solr document is built, the *Template Method* design pattern [38] was used. By applying this design pattern, the class hierarchy depicted in Figure 7.1 was implemented.

The parent abstract class `Indexer` takes care of performing the generic steps only by providing a `HttpSolrServer` instance and implementing the `index()`, `indexAll()` and `logCommitResponse()` methods, respectively. The step of document creation is handed over to its corresponding implementation of the `prepareForIndexing()` method in one of the child classes.

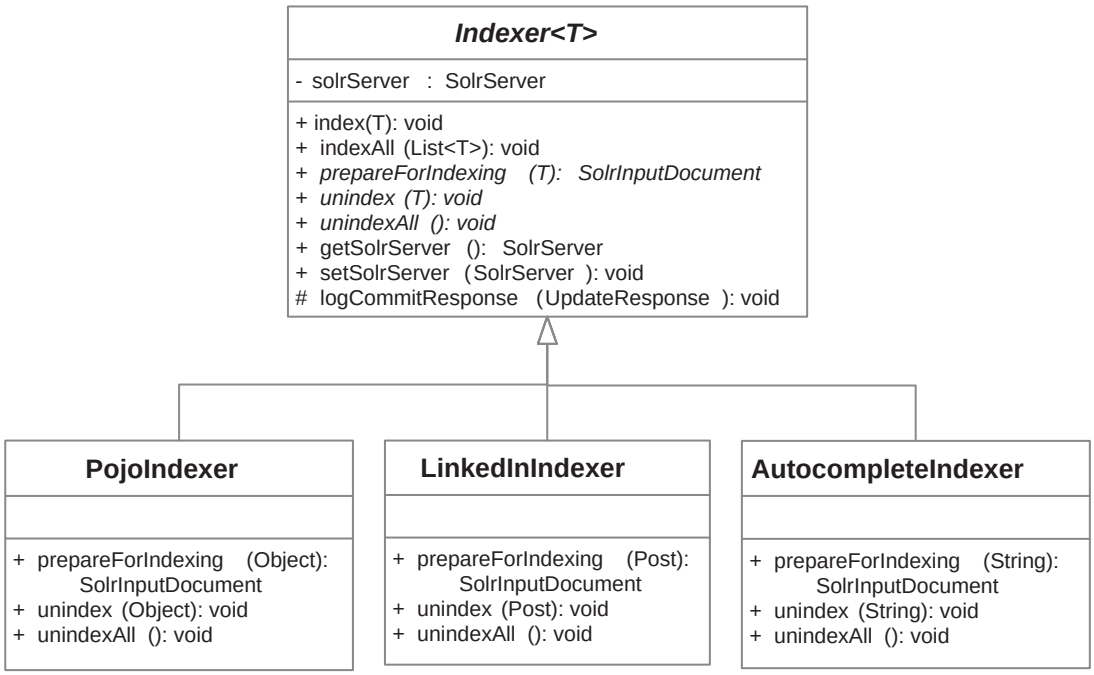


**Figure 7.1:** UML Diagram of Indexer Classes

There were three child classes (indexers) created and can be found in the `cz.zcu.kiv.eegdatabase.logic.indexing` package of the project. Each of the classes is responsible for processing different input data:

- `PojoIndexer` - indexer of POJO classes,
- `LinkedInIndexer` - indexer of LinkedIn articles,
- `AutocompleteIndexer` - indexer of searched phrases that improves the autocomplete functionality.

The following sections focus on the provided solutions to creating index documents from input data. Therefore, three different implementations of the `prepareForIndexing()` method in the classes listed above will be described.

### 7.1.1 Database Indexing

Generally speaking, the structure of data stored in RDBMS is not suitable for index of full-text search engines. The reasons behind this were already covered in Chapter 2. Therefore, a transformation from relational data to document data

had to be made. The domain model, the related POJO classes and their fields of our interest were identified in Chapter 5. Based on this identification, the Solr schema fields were created (described in Chapter 5).

### Possibilities of Indexing

Primarily, it is necessary to choose the method of mapping the POJO fields into Solr schema fields. By using the SolrJ API, it can be done by the following mechanisms:

- *`@Field` annotation* - The purpose of the `@Field` annotation is to mark the POJO fields that are to be indexed. In combination with the `addBean()` and `addBeans()` methods provided by SolrJ API, Solr documents can be created.

  However, this approach can be used only in a limited number of cases where a direct mapping of single-POJO fields is satisfactory. In more advanced scenarios, such as mapping an object hierarchy to a Solr document, another alternative must be chosen. This limitation lies in the inability to use the `@Field` annotation to mark nested objects or object collections.

  Furthermore, the usage of provided annotations brings a problem of ambiguity of mapped objects. Documents stored in the Solr index must possess an identifier which is unique across all stored documents. Object IDs are unique only in the class scope, so global uniqueness is not ensured.

- *using the SolrJ's `addField()` method* - Although this alternative is not so comfortable as the previous one, it gives more control of creating a Solr document. This is because merging multiple POJOs into one Solr document can be achieved. Nevertheless, using this method only would result in creating a less flexible code which is harder to maintain.

- *integration with Hibernate Search and using its annotation mechanisms* - The core idea of this proposed alternative is to use Hibernate Search capabilities, such as advanced indexing annotation support and automatic indexing of all collected changes, for the initial phase of indexing. Though it is possible to combine Hibernate Search and Solr (one of such ways to combine them can be found in [50]), there would be an extra undesired dependency on another technology.

- *custom annotation mechanism* - By exploiting possibilities offered by Java Reflection and combining them with custom Java annotations, a universal solution covering all needs can be implemented. This way, the problem of creating a Solr document from a class hierarchy, as well as the problem of the same document IDs, can be overcome. This alternative is undoubtedly the most challenging one, but its implementation can assure covering all specified needs.

For the above-mentioned reasons, it was decided to create a custom annotation mechanism for mapping POJO fields to their respective Solr document fields.

### Java Reflection

Java Reflection was frequently used to implement the new full-text search solution. Therefore, this mechanism will be described in the next section. Then, its usage throughout the created code of full-text search will be demonstrated.

### Introducing Java Reflection

Reflection is the ability to inspect the code and make its modifications at runtime. It is a feature that makes staticly-typed languages like Java more dynamic. Its heavy usage can be found especially in modern frameworks such as Spring or Hibernate, that both use reflection for instantiating classes from information in configuration files.

A very common use case of reflection in Java is the usage with annotations. This combination opens many possibilities of manipulating class metadata. In *JUnit 4*, for example, the `@Test` annotation was introduced. The JUnit framework looks up all methods marked by this marker annotation and call them in each execution of running unit tests.

The root class of the Java object hierarchy, the Object class, has the `getClass()` method providing the corresponding `Class` object, meaning that all Java classes can be invoked or inspected by means of reflection.

### Annotation Interface

In order to inspect classes and collect required field values, the classes as well as the some of the fields within them need to be marked by annotations. Three different annotations serving different purposes were created:

- `@Indexed` - both class (type) and field annotation that can be used in two ways: When it is used to annotate classes, it marks the parent POJO classes; when used on the field level, it marks child POJO classes or their collections,
- `@SolrField` - a field annotation which marks the fields whose values should be contained in a Solr document,
- `@SolrId` - a field annotation that determines the fields belonging to the unique identifier of indexed classes.

The annotation interfaces can be found in the `cz.zcu.kiv.eegdatabase.data.annotation` package of the EEG/ERP Portal.

Listing 7.1 provides an example of creating the `@SolrField` field annotation. Note that this annotation has the `name` attribute which serves to specify the matching field name in the Solr schema. All used Solr field names are defined as enumeration constant values of the `IndexField` enumeration type to avoid mistyping them.

**Listing 7.1:** Example of Creating the `@SolrField` Annotation.

```
1 @Target(ElementType.FIELD)
2 @Retention(RetentionPolicy.RUNTIME)
3 public @interface SolrField {
4     IndexField name();
5 }
```

The usage of the defined annotations is demonstrated on the code of the `Experiment` class in Listing 7.2 (irrelevant lines are omitted). It is worth noting that the associated POJO collections, whose fields logically belong to the same Solr document, are annotated accordingly.

**Listing 7.2:** Example of Using Indexing Annotations in the `Experiment` Class.

```
1 @Indexed
2 public class Experiment implements Serializable {
3   @SolrId
```

```
4   private int experimentId;
5   @Indexed
6   private Weather weather;
7   @SolrField(name = IndexField.TEMPERATURE)
8   private int temperature;
9   @SolrField(name = IndexField.TEXT)
10  private String environmentNote;
11  @Indexed
12  private Set<Hardware> hardwares = new HashSet<Hardware>(0);
13  @Indexed
14  private Set<Pharmaceutical> pharmaceuticals = new HashSet<↩
        Pharmaceutical>(0);
15  @Indexed
16  private Set<Disease> diseases = new HashSet<Disease>(0);
17  @Indexed
18  private Set<Software> softwares = new HashSet<Software>(0);
19  ...
20 }
```

The first practical application of Java Reflection discussed in this text is strongly related to the created set of annotations. To leverage the semantics hidden behind the used annotations, it is necessary to use reflection to introduce indexing logic by inspecting these annotated classes.

### Indexing Algorithm

In the `prepareForIndexing()` method, it is first found out if the type of a given `Object` instance, sent as the method parameter, has the `@Indexed` annotation assigned. If it does, it can continue traversing the class fields by searching a field with the `@SolrId` annotation. The value of such annotated field is used for assembling Solr document fields for later identification of a document. These document fields were described in Chapter 6.

The next step of the algorithm is of recursive nature and happens on two levels - on the parent and the child level. In both cases, all values of those fields containing the `@SolrField` annotation are extracted. On the parent level, in addition, the fields with the `@Indexed` annotation are searched to identify the associated child `Collection` or `Object` classes which are then traversed.

The schema in Figure 7.2 visually depicts the main idea of the described algorithm.

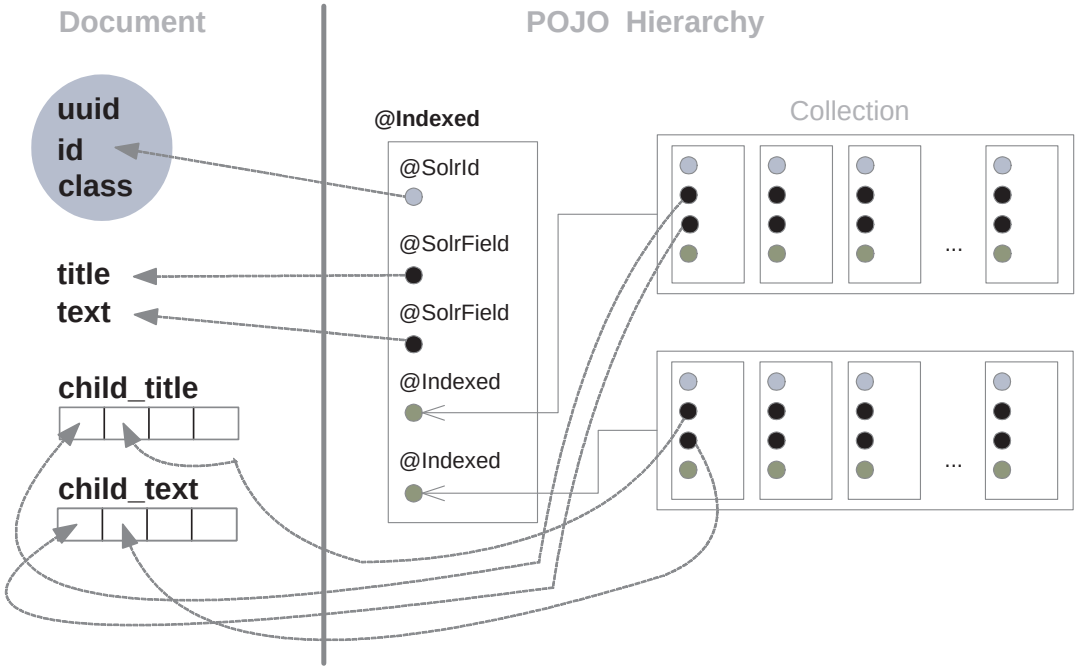


**Figure 7.2:** POJO Indexing Algorithm Schema.

### Ensuring eager loading

In order to make the indexing algorithm work, associated POJO collections, whose object field values are desired to be indexed, have to be already initialized. This condition, however, is not always met. As written earlier in Chapter 4, it is desired to apply lazy loading of object collections for as many cases as possible. If it is necessary, switching to the eager loading strategy can be done using the following ways:

- *change of Hibernate mapping configuration* - This modification involves setting the `lazy="false"` attribute for collections to be eagerly loaded. It means that all affected 1:N relationships are always loaded together with the parent object. This approach is not very flexible, because no proxies are created and the lazy loading behavior cannot be therefore configured at runtime.

- *custom HQL queries or Hibernate criteria that force eager fetching* - If HQL queries are used, one can specify eager fetching using the `fetch` keyword. In case of using the Criteria Query API, the `setFetchMode()` method with its fetch mode attribute set to `FetchMode.EAGER` does the job. This alternative

is more flexible than the previous one since lazy object initialization, which is set by default, is overridden by the eager fetch mode. The created proxy objects call the associated real objects to fetch necessary data. The drawback of this solution is the necessity to create custom HQL queries or Hibernate criteria for each entity that will cause collections to be loaded lazily. As the created queries can differ a lot, it is very difficult to apply this solution in a generic way.

- *usage of the `Hibernate.initialize()` method* - This method is used to initialize lazy-loaded collections. Its parameter takes an object that is to be fetched by the parent object. The method also ensures that all already initialized objects will be omitted. The power of the `Hibernate.initialize()` method lies in its universality. When used together with Java Reflection, a universal solution enforcing eager loading of collections for any POJO object can be achieved. It can be used even if the Hibernate session is closed [37].

Based on the aforementioned possibilities and the current requirements, using the `Hibernate.initialize()` method in combination with the power of Java Reflection was preferred.

This decision leads to modification of the `SimpleGenericDao` class. It was enriched of a new generic method which implements the chosen eager loading strategy. The method, namely the `getAllRecordsFull()` method, works on the principle of finding out all proxied object fields, i.e. those fields which are not initialized yet. All fields are accessed by reflective calls of their respective getter methods available in a given class. The fields are then initialized by adding them as the parameters of subsequent `Hibernate.initialize()` method calls.

The following Listing 7.3 shows the implementation of eager loading on demand described in the paragraph above.

**Listing 7.3:** Enforcing Eager Loading by Using Java Reflection.

```
1 public List<T> getAllRecordsFull() {
2   List<T> records = getAllRecords();
3   for(T record : records) {
4     Method[] methods = record.getClass().getDeclaredMethods();
5     for (Method method : methods) {
6       if(method.getName().startsWith("get") && !method.↩
            getReturnType().isPrimitive()) {
7         try {
```

```
8             initializeProperty(method.invoke(record));
9           } catch (IllegalAccessException e) {
10            log.error(e);
11          } catch (InvocationTargetException e) {
12            log.error(e);
13          } catch (Exception e) {
14            log.error(e);
15          }
16        }
17      }
18    }
19    return records;
20  }
21
22  ...
23
24  protected void initializeProperty(Object property) {
25    if(!Hibernate.isInitialized(property)) {
26      getHibernateTemplate().initialize(property);
27    }
28  }
```

### 7.1.2 LinkedIn Indexing

#### Using Spring Social

The methods provided by Spring Social are configured to give a user a set of default fields that are appropriate for some use cases. Although it is very convenient to have such layer of abstraction, sometimes there is a need to obtain other fields or to omit some of them. For example, if the Spring Social method for getting all articles in a group is called, some information, such as article summaries and their time stamps, is missing.

In such cases, a custom LinkedIn REST call must be created and put as a parameter of the `restOperations().getForObject()` method provided by Spring Social. The call can contain field selectors which are used to specify fields to return in the response.

The following example in Listing 7.4 shows the usage of field selectors in the REST call. The REST call in the listing gets full information about twenty latest

LinkedIn articles published in the EEG/ERP Portal group. Note the `group-id` placeholder which is later substituted by the real value of the EEG/ERP Portal group ID.

**Listing 7.4:** Example of a LinkedIn REST API call.

```
1 http://api.linkedin.com/v1/groups/{group-id}/posts:
2 (creation-timestamp,title,summary,id,creator:(first-name,last-name)↩
     )?
3 count=20&start=0&order=recency"
```

In the application code, LinkedIn REST calls are wrapped in the methods of the `LinkedInManager` class. The class includes these methods using direct LinkedIn REST calls:

- `getGroupPostsWithMoreInfo(int count, int start)` - It uses a very similar REST call as the one in Listing 7.4 to obtain LinkedIn articles. The number of retrieved articles and the index position of the first retrieved articles are defined in the method parameters.
- `getPostById(String id)` - Gets a LinkedIn post by its unique ID.
- `deletePost(String id)` - Deletes a post by specifying its ID as the method parameter.

Apart from the methods mentioned above, the following methods were added to the `LinkedInManager` class as well:

- `getLastPost()` - This method returns the last post added to the EEG Portal LinkedIn group.
- `getLastPostId()` - Returns the id value of the last post added to the EEG Portal LinkedIn group.

Usage of these methods is related to indexing LinkedIn articles which are created by using the EEG/ERP Portal.

#### Article-document Mapping

Compared to POJO-document mapping, mapping from the LinkedIn article to the created Solr document structure is straightforward. Received LinkedIn `Post` objects store only unique string ID, article title and summary values, so converting them to Solr documents involves copying these values to the respective `uuid`, `text` and `title` document fields. Apart from that, the document fields `class` and `source` must be populated in order to find these articles successfully while doing a full-text search.

### 7.1.3 Indexing Searched Phrases

It was decided to make searched phrases indexable. Indexing them enables the phrases to appear in the autocomplete box. Moreover, by indexing search phrases, search frequency of each phrase can be found out and stored together with the phrase in the document. This way, indexed phrase frequencies enable suggesting the most queried phrases first.

Internally, a phrase and its search frequency are stored together in the `autocomplete` Solr field and are separated by the number sign (#) symbol.

### 7.1.4 Changing Indexed Data

When some data stored in RDBMS are removed or edited, the made changes need to be reflected in the Solr index. In case if parent POJO objects are changed, the analogous document changes can be performed by using the indexer's `index()` and `unindex()` methods. The first method simply overwrites a stored document having the same `uuid` field as the new inserted one. The latter method removes a document determined by an object set as the method parameter.

However, child objects have no direct document equivalents, because they form only a part of the documents of parent objects. Although partial updates are possible since Solr 4.0, they are not implemented. The reasons of not implementing them are the following:

- It is expected that the EEG/ERP Portal application is designed to treat the child objects always in the context of their parent object.

- Adding the partial update support brings a new level of complexity into the application. Due to the application nature and its requirements, the added complexity was not worth the effort in this case.

Nevertheless, if a change of a child object happens, the index will become inconsistent. This is why periodic indexing of database data has to be ensured to prevent potential inconsistent index states and to reach the so-called *eventual consistency*.

Almost the same applies to LinkedIn articles where the problem is extended by adding, updating and removing the articles by the means of the LinkedIn website.

### 7.1.5 Periodic indexing

#### LinkedIn Articles

There are two ways to add an article to the EEG/ERP Portal group: either indirectly by filling in the form on the EEG/ERP Portal website or directly from LinkedIn.

In the first case, articles can be indexed immediately after publishing them because their times of publishing are known due to the interaction with the EEG/ERP Portal. The last published article in the LinkedIn group can be fetched by the means of LinkedIn REST API. Its object can be then used by the indexer, which adds information about the article to the index.

The latter case is more complicated as there is no interaction with the EEG/ERP Portal. So in order to index all LinkedIn articles, the article data must be retrieved first. It can be done by using LinkedIn REST API calls to receive their object representation. Since there is no available information of when articles are uploaded to LinkedIn, the REST calls must be done periodically. This way, periodic indexing of all articles published on LinkedIn can be achieved.

Although the obvious disadvantage of periodic indexing is the existence of a delay between publishing times of some articles and their indexing times (which is equal to the indexing period in the worst case), this method assures that all articles get indexed in the end.

### Scheduling in Spring

The Spring Framework has a native support of task scheduling and asynchronous calls. Since its version 3.0, methods can be scheduled and also run asynchronously by using annotations, namely the `@Scheduled` and `@Async` annotations.

The first mentioned annotation, when added to a method, makes the method schedulable by Spring. Usage of this annotation is restricted to the void methods with no parameters. The `@Scheduled` annotation has to contain a piece of metadata to tell Spring how to plan the method scheduling.

Currently there are three available attributes for the `@Scheduled` annotation, from which the most flexible option is specifying a cron expression to trigger a task as shown on the following lines:

**Listing 7.5:** Example of a Cron Expression.

```
1 @Scheduled(cron=* 0 22 * * SAT-SUN)
2 public void indexAll()
```

This way, the method `indexAll()` will be scheduled to run always at 10 pm only on Saturdays and Sundays. The cron syntax allows a user to create more sophisticated scheduling scenarios, but discussing the syntax is out of scope of this work. Since Spring is internally using Quartz as a scheduler, an interested reader can find all necessary information about the syntax in the Quartz documentation [51].

The `@Async` annotation is used to mark the methods to be invoked asynchronously. It is very easy to use for methods having void return values. Listing 7.6 shows the usage of the `@Async` annotation for the `indexLinkedIn()` and `indexDatabase()` methods that handle LinkedIn-articles and RDBMS indexing, respectively.

**Listing 7.6:** Using the `@Async` Annotation.

```
1 @Async
2 public void indexLinkedIn()
3 ...
4 @Async
5 public void indexDatabase()
```

In order to enable annotation-based scheduling, it is necessary to add a new element in the application context file as well as the `task` namespace to which the element belongs (see Listing 7.7).

**Listing 7.7:** Using `task` Namespace

```
1 <xmlns:task="... http://www.springframework.org/schema/task"
2 xsi:schemaLocation="... http://www.springframework.org/schema/task/↩
    spring-task.xsd">
3 ...
4 <task:annotation-driven executor="indexingExecutor" scheduler="↩
    indexingScheduler"/>
```

The annotation-driven element requires `executor` and `scheduler` attributes to be set to handle tasks represented by methods marked by `@Async` and `@Scheduled` annotations, respectively. The related lines can be seen in Listing 7.8:

**Listing 7.8:** Setting Spring Executer and Scheduler.

```
1 <task:executor id="indexingExecutor" pool-size="5"/>
2 <task:scheduler id="indexingScheduler" pool-size="1"/>
```

### 7.1.6 UML Class Diagram

The relationships among classes that create the indexing part of the full-text search are depicted in the simplified UML diagram in Figure 7.3.

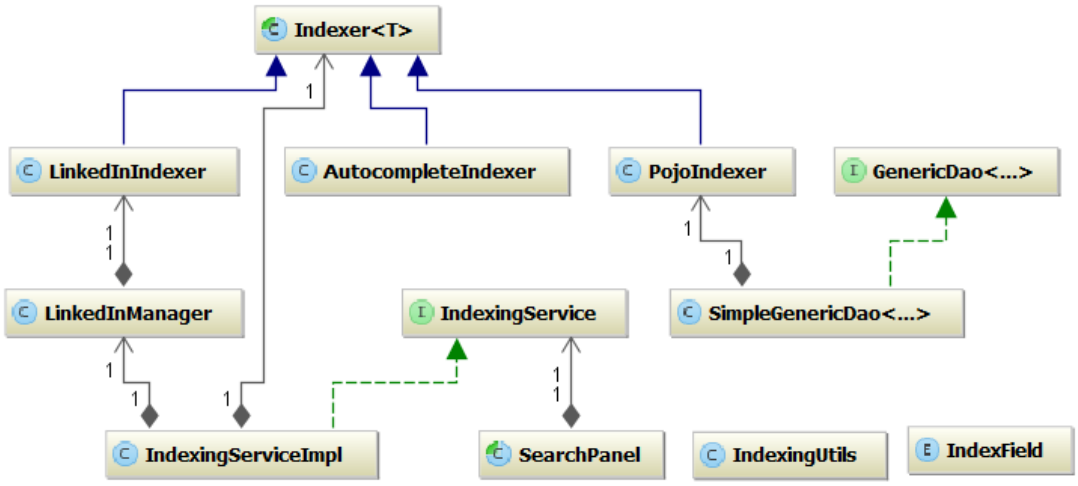


**Figure 7.3:** UML Class Diagram of the Indexing Part.

There can be seen these most important classes:

- `IndexingService` - Contains the indexing methods `indexDatabase()` and `indexLinkedIn()` described above that handle periodic indexing.

- `Indexer` - Direct usage of its `index()` and `unindex()` methods is made when any of POJO objects or LinkedIn `Posts` is created, updated or deleted. Thus every domain object must be associated with its indexing in the first two cases and its removal from the index in the last case. It is achieved by calling these methods in the `create()`, `update()` and `delete()` in the `SimpleGenericDao` class and in the `publish()` and `deletePost()` methods of the `LinkedInManager` class.

## 7.2 Searching

This section deals with the search user interface. The interface was implemented by using the Wicket Framework.

### 7.2.1 Search Logic

Most of the search logic can be found in the `FulltextSearchService` class. This method is responsible for:

- returning search results with highlighted keywords for a given query - method `getResultsForQuery()`
- returning suggested search phrases for autocomplete - method `getTextTo-Autocomplete()`
- returning a total number of found documents for a given query - method `getTotalNumberOfDocumentsForQuery()`
- faceting by document categories - method `getCategoryFacets(String solrQuery)`

The method `getResultsForQuery` returns `FullTextResult` objects that represent results received from the Solr server for a query. The `FullTextResult` class, in fact, contains fields that match those contained in the Solr document, i.e. the fields whose information has a value for users - document title, text fragments with highlighted searched keywords, type of the document, class of a Wicket target page which contains further details, id of a corresponding POJO, and time of indexation.

More details about the mentioned methods can be found in the JavaDoc documentation in the attached DVD (see Attachment A).

### 7.2.2 Full-text Search User Interface

Objects of the `FullTextResult` class are handed over to classes that form the full-text search user interface. The classes are located in the `cz.zcu.kiv.eegdatabase.wui.ui.search` package of the project.

Namely, these classes include:

- `SearchFacets` - It creates faceted search by showing number of found results for each of the document categories, allowing to filter the found results by the document category. It uses the `FacetCategory` class, which represents the facet category by the `name` and `count` parameters, and the enumeration type `ResultCategory`.
- `SearchPanel` - This abstract class represents a search panel that provides all necessary functionality, including the autocomplete capability. The reason why the class is abstract is that Wicket enforces to have a separate class for each HTML markup. Since it is required to have one search panel in the page header and another one, which is different by appearance, in the search page, their corresponding classes had to be created. There is the `MenuSearchPanel` class for the search panel located in the page header. For displaying the search panel in the search page, the `PageSearchPanel` class was made.
- `SearchResultPanel` - It is responsible for displaying search results. The search results are rendered by using the created `SearchResultDataProvider` class that communicates with the `FulltextSearchService` class to obtain the matching `FullTextResult` instances. Besides that, it also displays the number of found results per each document category by utilizing an instance of `SearchFacets`.
- `SearchPage` - It represents the search page. It uses the listed `SearchResultPanel` and `PageSearchPanel` classes to display necessary information.

The appearance of the search user interface is depicted in the screenshot in Figure 7.4.


**Figure 7.4:** Search User Interface.

### 7.2.3 UML Class Diagram

The described classes related to searching documents and associations among them are depicted in the UML class diagram in Figure 7.5

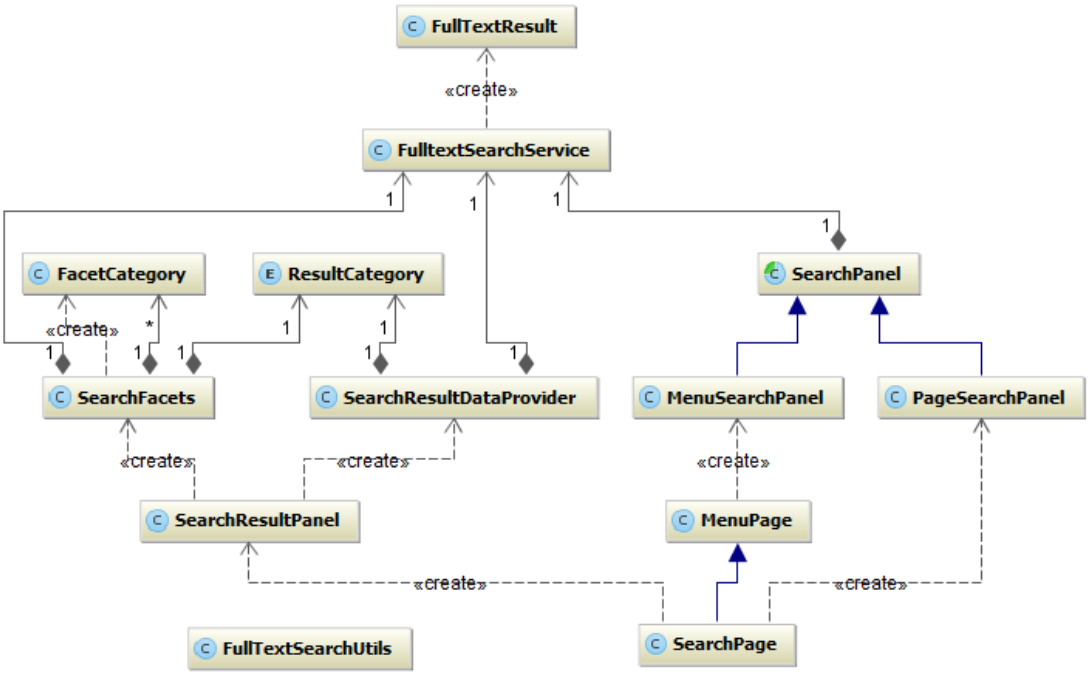


**Figure 7.5:** UML Class Diagram of the Search Part.

# 8 | Testing

This chapter presents created tests that prove the correct functionality of the implementation of the full text search for the EEG/ERP Portal. These tests are JUnit tests and their main purpose is to verify expected functionality of various classes related to the full-text search.

All tests use test data from the EEG/ERP Portal database server and interact with an independently running Solr testing server to access indexed data. The server runs at the address `147.228.63.134/solr` on port 8686 and its testing data come from the EEG database as a result of one-time indexing.

## 8.1 Unit Tests

There were several test cases created by using JUnit4 and they are discussed in this section. Integration of JUnit with Spring was already done in the past. In effect, the defined Spring beans can be used in the tests.

The created test classes can be found in the `cz.zcu.kiv.eegdatabase.search` package in the `test` directory of the EEG/ERP Portal application.

### 8.1.1 FulltextSearchServiceTest.java

This test class uses the `FulltextSearchService` class to test its autocomplete and faceting functionality. The `getAutocompleteResultsSuccess()` and `getAutocompleteResultsFail()` tests the autocomplete part. As their names say, these methods test returning the expected output for two different input strings. In case of the first mentioned test, the test passes if any autocomplete suggestions are found for the query string. On the contrary, the other test passes if a given search string returns no autocomplete results.

Faceted search is covered by the `getDocumentCountForEmptyQuery()`, `facetAllTest()` and `facetNoneTest()` tests. `getDocumentCountForEmptyQuery()` tests

obtaining the zero number of search results if an empty string is searched. The `facetAllTest()` and `facetNoneTest()` methods test creating faceted categories. The `facetAllTest()` test passes if the number of documents in each category is greater than zero when all documents are retrieved (by using the * sign as the input query). `facetNoneTest()` It uses the empty search string that should not return any documents. Therefore, this test succeeds if the number of documents in each category is equal to zero.

### 8.1.2 IndexingServiceTest.java

This class contains tests of methods of the `IndexingService` class. The `testDatabaseIndexing()` method verifies proper indexing of database records. The second test, `testLinkedInIndexing()`, checks correct indexing of LinkedIn articles.

### 8.1.3 IndexingTest.java

It contains methods that test indexing of created sample documents and their removal from the Solr index. For example, the `indexSampleIndexablePojos()` method is a test case that creates an instance of each indexable parent class and performs a query that returns all documents representing these instances. This test succeeds if all created documents are returned by the query.

### 8.1.4 LinkedInIndexingTest.java

This test class contains JUnit tests that index and unindex created test posts. The tests also check the correct behavior of manipulation with received LinkedIn articles via the LinkedIn REST API.

# 9 | Conclusion

The aim of this thesis has been to enrich the EEG/ERP Portal of a new full-text search functionality which enables full-text search over data from a relational database and social networks. This chapter summarizes the work presented.

In the theoretical part (chapters 2-4), the basic principles of full-text search systems were explained and their differences compared with relational database management systems were discussed. Afterwards, several open-source full-text search engines were presented. It turned out that all of them were more suited for full-text search than relational database systems and that there was no search engine, whose overall performance was significantly better.

The practical part starts with Chapter 5, in which the previous EEG/ERP Portal's full-text search solution was analyzed and the full-text search requirements were collected. Furthermore, the introduced search engines and libraries were compared with respect to the stated requirements. The comparison showed that the Lucene-based Solr search server was the most suitable solution. Therefore, Solr was chosen to be used for the full-text search implementation and the overall system architecture was proposed.

Chapter 6 was focused on designing the Solr index document. There was a need to index data, that were coming from different sources, into a unified structure. These sources are the EEG database and LinkedIn, the only social network integrated with the EEG/ERP Portal so far. After the analysis of indexable data in the database tables and in LinkedIn articles, an appropriate Solr schema design was made. Additional fields for autocomplete support were defined in the Solr schema as well. Furthermore, search result highlighting was configured.

Chapter 7 discussed the implementation aspects concerning indexing and searching data. Several approaches of indexing different data were analyzed. Different data sources led to creating indexers for both RDBMS data and LinkedIn articles. Furthermore, to improve the autocomplete functionality, the indexer of searched phrases was added. Relational database data were indexed by applying a transformation lying in denormalization of its tabular data. A custom annotation

mechanism was made to handle object-document mapping. Also optional eager loading, which is necessary for database indexing, was implemented. LinkedIn data were indexed by utilizing LinkedIn REST API. Apart from real-time indexing of modified or created data, indexing runs periodically by using the Spring Framework scheduling capabilities.

The second part of this chapter describes the user interface created by using the Wicket framework, and the search logic which, apart from displaying search results, handles faceted search by document categories, highlighting of searched keywords and keyword suggestions.

In the final chapter of the practical part, Chapter 8, the created full-text search was covered by JUnit tests to prove its validity. The created tests concern checking proper indexing and searching.

The work meets all set requirements. However, the results that come from this thesis can be further extended. The created solution is open to indexing other data sources, such as PDF files or data from social networks like Facebook and Twitter.

# Bibliography


[1] I. H. Witten, A. Moffat, and T. C. Bell, *Managing gigabytes (2nd ed.): compressing and indexing documents and images*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999.

[2] W. B. Frakes and R. Baeza-Yates, *Information Retrieval: Data Structures and Algorithms*. Prentice Hall PTR, June 1992.

[3] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge, UK: Cambridge University Press, 2008.

[4] S. Büttcher, C. L. A. Clarke, and G. V. Cormack, *Information Retrieval: Implementing and Evaluating Search Engines*. MIT Press, 2010.

[5] C. Middleton and R. Baeza-Yates, "A comparison of open source search engines," tech. rep., 10 2007. Available at `http://wrg.upf.edu/WRG/dctos/Middleton-Baeza.pdf`.

[6] J. Zobel, A. Moffat, and K. Ramamohanarao, "Guidelines for presentation and comparison of indexing techniques," *SIGMOD Rec.*, vol. 25, pp. 10–15, Sept. 1996.

[7] J. Zhang and T. Suel, "Efficient query evaluation on large textual collections in a peer-to-peer environment," in *Peer-to-Peer Computing, 2005. P2P 2005. Fifth IEEE International Conference on*, pp. 225–233, 2005.

[8] G. Salton, *Automatic text processing: the transformation, analysis, and retrieval of information by computer*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1989.

[9] D. Hiemstra, *Using Language Models for Information Retrieval*. CTIT Ph.D. thesis series, Taaluitgeverij Neslia Paniculata, 2001.

[10] C. Van Rijsbergen, *Information retrieval*. Butterworths, 1979.

[11] A. Ali, *Sphinx Search Beginner's Guide*. Packt Publishing, Limited, 2011.

[12] V. Singh, "A comparison of open source search engines." Available at http://zooie.wordpress.com/2009/07/06/a-comparison-of-open-source-search-engines-and-indexing-twitter/, July 2009. [Online; accessed: 4-December-2012].

[13] A. Arslan and O. Yilmazel, "Quality benchmarking relational databases and lucene in the trec4 adhoc task environment," in *Computer Science and Information Technology (IMCSIT), Proceedings of the 2010 International Multiconference on*, pp. 365 –372, oct. 2010.

[14] K. Jayant, "Benchmarking results of mysql, lucene and sphinx." Available at http://jayant7k.blogspot.cz/2006/06/benchmarking-results-of-mysql-lucene.html, Jun 2006. [Online; accessed: 6-December-2012].

[15] "Indri homepage." Available at http://www.lemurproject.org/indri.php. [Online; accessed: 4-December-2012].

[16] S. A. R. Howard Turtle, Yatish Hegde, "Yet another comparison of lucene and indri performance," in *Proceedings of the SIGIR 2012 Workshop on Open Source Information Retrieval*, vol. 46, pp. 64–67, August 2012.

[17] "Lucene homepage." Available at http://lucene.apache.org/. [Online; accessed: 4-December-2012].

[18] M. McCandless, E. Hatcher, and O. Gospodnetic, *Lucene in Action, Second Edition: Covers Apache Lucene 3.0.* Greenwich, CT, USA: Manning Publications Co., 2010.

[19] "Lucene-java wiki." Available at http://wiki.apache.org/lucene-java/PoweredBy. [Online; accessed: 21-February-2013].

[20] "Apache lucene - scoring." Available at http://lucene.apache.org/core/old_versioned_docs/versions/3_2_0/scoring.html. [Online; accessed: 27-April-2013].

[21] G. I. Andrzej Bialecki, Robert Muir, "Apache lucene 4," in *Proceedings of the SIGIR 2012 Workshop on Open Source Information Retrieval*, vol. 46, pp. 17–24, August 2012.

[22] "Sphinx homepage." Available at http://sphinxsearch.com/. [Online; accessed: 4-December-2012].

[23] A. Aksyonoff, *Introduction to Search with Sphinx: From Installation to Relevance Tuning*. O'Reilly Media, Incorporated, 2011.

[24] A. Gupta, "Lucene vs sphinx - a showdown on a large dataset." Available at http://jayant7k.blogspot.cz/2006/06/benchmarking-results-of-mysql-lucene.html, Aug 2009. [Online; accessed: 6-December-2012].

[25] "Elasticsearch, sphinx, lucene, solr, xapian. which fits for which usage?." Available at http://stackoverflow.com/questions/2271600/elasticsearch-sphinx-lucene-solr-xapian-which-fits-for-which-u\-sa\-ge. [Online; accessed: 28-Oct-2012].

[26] "Xapian homepage." Available at http://xapian.org/. [Online; accessed: 4-December-2012].

[27] O. Betts, "Index size comparison." Available at http://thread.gmane.org/gmane.comp.search.xapian.devel/2022, May 2012. [Online; accessed: 6-December-2012].

[28] "Zettair homepage." Available at http://www.seg.rmit.edu.au/zettair/. [Online; accessed: 4-December-2012].

[29] D. Smiley and D. Pugh, *Apache Solr 3 Enterprise Search Server: Enhance Your Search with Faceted Navigation, Result Highlighting, Relevancy Renked Sorting, and More*. Community experience distilled, Packt Publishing, Limited, 2011.

[30] "Hibernate search homepage." Available at http://www.hibernate.org/subprojects/search.html. [Online; accessed: 4-December-2012].

[31] "Solr homepage." Available at http://lucene.apache.org/solr/. [Online; accessed: 4-December-2012].

[32] "Elasticsearch homepage." Available at http://www.elasticsearch.org/. [Online; accessed: 25-April-2013].

[33] "Solr or elastisearch?." Available at http://www.searchtechnologies.com/elasticsearch-solr-lucene.html. [Online; accessed: 12-Dec-2012].

[34] "Why we chose solr 4.0 instead of elasticsearch." Available at http://www.sentric.ch/blog/why-we-chose-solr-4-0-instead-of-elasticsearch. [Online; accessed: 25-Jan-2013].

[35] "Eeg/erp portal - overview." Available at http://software.incf.org/software/eeg-erp-portal. [Online; accessed: 11-May-2013].

[36] "Hibernate home page." Available at http://www.hibernate.org/. [Online; accessed: 26-April-2013].

[37] C. Bauer and G. King, *Hibernate in Action (In Action series)*. Greenwich, CT, USA: Manning Publications Co., 2004.

[38] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 ed., Nov. 1994.

[39] "Spring framework home page." Available at http://www.springsource.org/. [Online; accessed: 26-April-2013].

[40] M. Fowler, "Inversion of Control Containers and the Dependency Injection pattern." Available at http://www.martinfowler.com/articles/injection.html, Jan. 2004. [Online; accessed: 30-April-2013].

[41] S. Tilkov, "A brief introduction to rest." Available at http://www.infoq.com/articles/rest-introduction. [Online; accessed: 27-April-2013].

[42] "Wicket home page." Available at http://wicket.apache.org/. [Online; accessed: 30-April-2013].

[43] E. Bernard and J. Griffin, *Hibernate Search in Action*. Greenwich, CT, USA: Manning Publications Co., 2008.

[44] R. Kuć, "Data Import Handler – removing data from index." Available at http://solr.pl/en/2011/01/03/data-import-handler-%E2%80%93-removing-data-from-index/, Jan. 2011. [Online; accessed: 30-April-2013].

[45] "Solr Wiki - Embedded Solr." Available at http://wiki.apache.org/solr/EmbeddedSolr, Sept. 2009. [Online; accessed: 10-Dec-2012].

[46] "Solr Security." Available at http://wiki.apache.org/solr/SolrSecurity, Sept. 2013. [Online; accessed: 25-Apr-2013].

[47] "Schema Design." Available at http://wiki.apache.org/solr/SchemaDesign, Jan. 2013. [Online; accessed: 01-May-2013].

[48] "Solr Wiki - Highlighting Parameters." Available at http://wiki.apache.org/solr/HighlightingParameters, Sept. 2013. [Online; accessed: 28-Mar-2013].

[49] "Analyzers, Tokenizers, and Token Filters." Available at http://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters, Sept. 2013. [Online; accessed: 7-May-2013].

[50] A. Levy, "Combining the power of hibernate search and solr." Available at anotherjavaday.blogspot.com/2012/01/combining-power-of-hibernate-search-and.html. [Online; accessed: 21-April-2013].

[51] "Crontrigger tutorial." http://www.quartz-scheduler.org/documentation/quartz-2.1.x/tutorials/crontrigger. [Online; accessed: 4-April-2013].

# List of Figures

# List of Tables

# Listings

# Attachments

## I DVD Content

The attached DVD contains the following folders:

- `javadoc` - This folder contains the generated JavaDoc documentation of the source code.
- `solr` - In this folder, a configured Solr server is present.
- `src` - It contains source code of the EEG/ERP Portal which includes the full-text search implementation.
- `thesis` - The PDF document of this thesis can be found here. All figures that appear in the thesis are included in the `figures` subfolder. The `latex` subfolder contains Latex source files as well as other necessary files needed to generate the output PDF file.