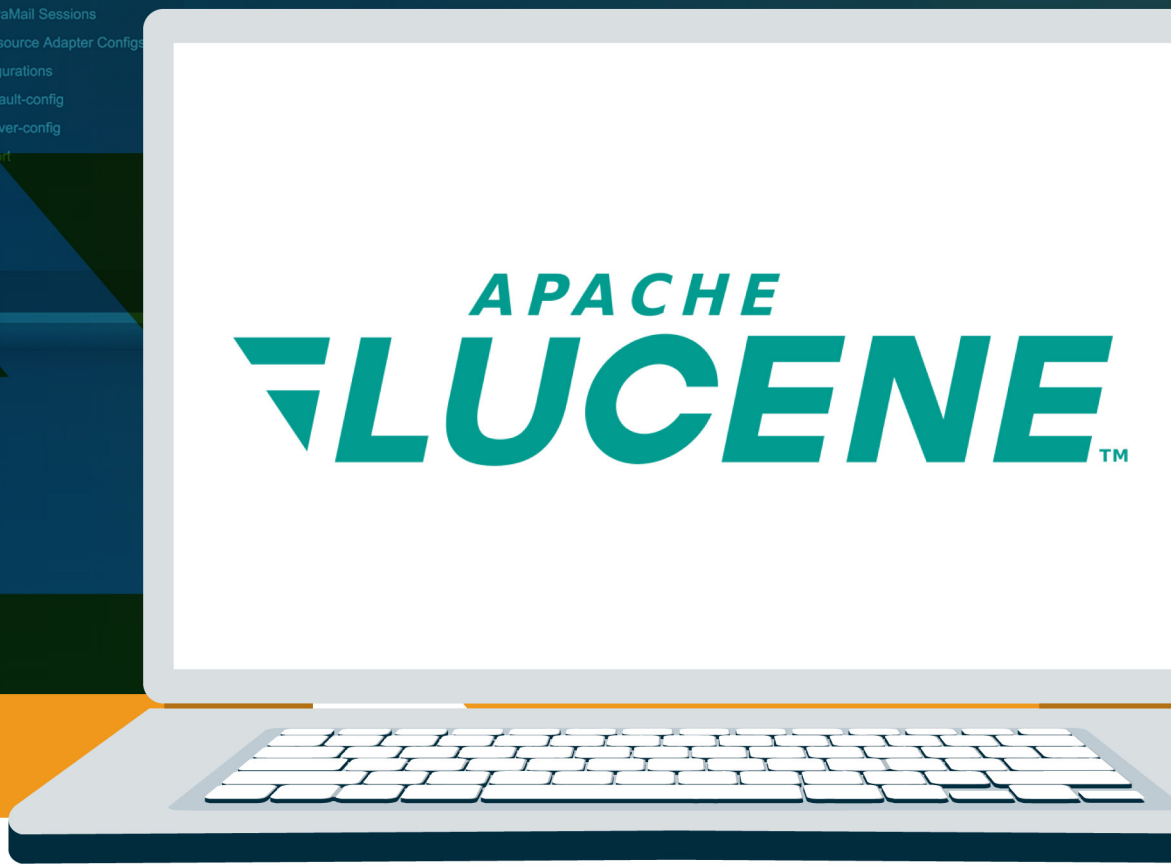
payara®

Full-Text Search in Jakarta EE with Apache Lucene



The Payara® Platform - Production-Ready, Cloud Native and Aggressively Compatible.

User Guide

Contents

Guide Updated: **September 2023**

Full-text search is a technique used to search a full-text database against user queries, as opposed to searching just the metadata or fixed sets of keywords. This allows for more complex and flexible search criteria and returns results that are more contextually relevant to the query. Full-text search is invaluable in applications that require extensive data retrieval features like content management systems, e-commerce platforms, or enterprise search engines.

Apache Lucene is a high-performance, scalable library for full-text search and indexing. It offers a robust set of features like advanced text parsing, ranking algorithms, and various types of queries out of the box. Lucene's Java-based architecture makes it easy to integrate with Jakarta EE applications, providing a powerful toolset for adding sophisticated search capabilities. Given that both are Java-centric technologies, the integration is seamless, offering robustness and scalability while minimising the need for third-party services.

The goal of this guide is to provide a quick introduction to integrating Apache Lucene into a Jakarta EE application for full-text search capabilities. By the end of this guide, you will be able to:

- Understand the core concepts of Apache Lucene and how it works for full-text search.
- Implement indexing and searching of Jakarta EE entities using Lucene.
- Optimise search results and keep the Lucene index in sync with your application's database and
- Address scalability and performance considerations.

Apache Lucene is a very powerful library that can be used to create sophisticated search experiences in your applications. This guide will only attempt to scratch the surface. You should read up on the project to familiarise yourself with all the features that it supports and how to use it for your own custom solutions. Let's start off by taking a look at what Apache Lucene is.

What is Apache Lucene?

Apache Lucene is an open-source search engine library written in Java, originally created by Doug Cutting in 1999. It has since become an integral part of the Apache Software Foundation and serves as the backbone for various search platforms, including Elasticsearch and Solr. Over the years, Lucene has gained a reputation for being a robust, scalable, and highly customizable search engine library.

Key Features

Apache Lucene comes with a lot of features out of the box. Some of the core ones include:

High-Performance Indexing

Lucene is engineered for speed, specifically in the domain of indexing large sets of textual data. Its indexing structure uses an inverted index, which significantly accelerates read operations. This high-speed indexing is essential for applications that need to update their search indexes frequently or in real-time. The optimised data structures and algorithms ensure that search queries are executed in milliseconds, providing a seamless user experience.

Flexible Query Language

One of the standout features of Lucene is its rich and flexible query language. It supports a wide variety of search queries, including but not limited to boolean queries, wildcard queries, and phrase queries. This flexibility allows you to implement complex search functionalities effortlessly, catering to the diverse requirements of modern applications.

Ranking Algorithms

Lucene comes with a built-in Term Frequency-Inverse Document Frequency (TF-IDF) ranking algorithm, a widely used method to rank search results based on their relevance. However, its architecture is open-ended, allowing you to plug in custom ranking algorithms. This is particularly useful when your applications require specialised ranking criteria, such as e-commerce websites where product rankings could be influenced by user reviews, availability, or other business-specific metrics.

Extensible and Customizable

Lucene's architecture is highly extensible, allowing you to create custom plugins, analyzers, and other extensions to meet your custom requirements. Whether you need a specialised tokenizer for text analysis or a custom field type for unique data, Lucene provides the means to extend its core functionalities. This extensibility makes it adaptable to a wide range of applications and use-cases.

Scalability

Designed with scalability in mind, Lucene is capable of handling extremely large datasets and high query volumes. Its architecture supports horizontal scaling, meaning you can distribute its index across multiple servers to handle increasing data loads. This is crucial for enterprise-level applications that require the ability to grow and handle large amounts of data efficiently.

Text Parsing and Analysis

Lucene excels in the area of text parsing and natural language processing. It comes with a variety of built-in text analyzers, supporting multiple languages. These analyzers can break down text into tokens, remove stop words, perform stemming, and execute other text normalisation tasks. This capability is particularly beneficial for applications that require intelligent text search features, such as sentiment analysis or language translation services.

Each of these key features contributes to making Apache Lucene a powerful, flexible, and efficient solution for integrating search functionalities into Jakarta EE and other types of applications.

Lucene Use Cases

Incorporating Apache Lucene into a database-backed Jakarta EE application can offer several advantages, especially when dealing with complex search requirements that go beyond the capabilities of traditional SQL databases. Here are some use-cases and benefits of using Apache Lucene in such an environment:

Full-Text Search

If your application needs to offer advanced search capabilities such as text-based content searches, Lucene can provide this functionality efficiently. This is particularly useful for applications like e-commerce websites, content management systems, and knowledge bases.

Faceted Search

Lucene can help in offering faceted search, which is a way to classify search results into categories. This is useful in scenarios like e-commerce sites where users might want to filter products by various attributes like price, brand, and rating.

Geospatial Search

If you have location-based data, Lucene can efficiently handle geospatial queries, which can be challenging for traditional databases.

Real-Time Search

Lucene can index and search data in near real-time, making it suitable for applications that require real-time analytics and reporting.

Document Retrieval

If your application involves working with a large number of documents, Lucene can be used to build a document retrieval system with features like ranking, highlighting, and categorization.

Log and Event Data Search

Applications that generate large volumes of log or event data can benefit from Lucene's efficient storage and fast retrieval capabilities.

Lucene's Architecture

At its core, Lucene's architecture is built around the concept of an Index. An index in Lucene is somewhat analogous to a database in RDBMS, serving as a data structure optimised for fast retrieval of documents. Let's see a simplified overview of how Lucene works:

- **Document and Fields:** The basic unit of search and index is a Document, which is a collection of Fields. Each field has a key-value pair where the key is the name of the field, and the value is the content you want to index or store.
- **Analyzers:** Before indexing, text data is processed through an Analyzer, which breaks down the text into Tokens (usually individual words) and may also transform the text (lowercasing, stemming, etc.).
- **Indexing Process:** During this process, the documents are added to an IndexWriter, which processes them through the analyzer and stores them in the index.
- **Query Parser:** To search the indexed documents, Lucene provides a query parser that translates human-entered text into a Lucene query.
- **Searcher:** The IndexSearcher class is used to perform the actual search on the index. It takes a query object, runs it against the index, and returns a list of Hits or results.
- **Scoring and Ranking:** Each hit is assigned a score based on the query, which determines its rank in the search results. You can roughly think of it as similar to Google's PageRank.
- **Retrieval:** Finally, the documents corresponding to the hits can be retrieved for display or further processing.

The above concepts are built on concrete artefacts in the Lucene library that you will use in your application. Let's take a look at these constructs below.

Index

In Lucene, an index is a data structure that improves the speed of search operations. It's similar to a database index in RDBMS, optimized for textual search. An index contains a series of documents that you can query.

Document

A document is the basic unit of search and index in Lucene. It represents a real-world entity like a book, a web page, or a database record. A document is a collection of fields.

Field

A field is a named section in a document that has been indexed. Each field has two main properties: a key (field name) and a value (field content). Fields can be stored, which means they can be retrieved later, or they can be only indexed, which means they can be involved in queries but aren't retrievable.

Term

A term is essentially an indexed field value. During the indexing process, text fields are broken down into terms, which are then used during the search.

Query

A query is a structured request to search for documents within an index. Lucene offers a rich set of query types, including term queries, boolean queries, and more complex ones like wildcard and phrase queries.

The Lucene API Overview

The Apache Lucene API provides a comprehensive set of classes and interfaces that allow for complex text search functionalities. While it offers a broad range of features, it is designed to be intuitive and easy to use. Below is an overview of some of the key classes and interfaces you'll frequently use when working with Lucene.

Directory

The Directory class abstracts the storage medium where the Lucene index resides. Lucene offers several out-of-the-box implementations of this class, such as FSDirectory for file system storage and RAMDirectory for in-memory storage. The choice of directory implementation can have implications for performance and persistence, so it's essential to choose the one that aligns with your application's needs.

IndexWriter

The IndexWriter class is your entry point for adding documents to a Lucene index. When you create an instance of IndexWriter, you specify the Directory where the index will be stored and the Analyzer to be used for text processing. Once instantiated, you add Document objects to it, each of which represents a set of data you want to be searchable. After adding documents, you commit changes to write them permanently to the index. The IndexWriter handles all the intricacies of breaking down the documents into indexable terms.

Document and Field

The Document and Field classes are straightforward data classes used during the indexing process. A Document is essentially a container for multiple Field objects, each representing a piece of data you want to index. For instance, in the context of indexing articles, a Field could be the title, another could be the content, and yet another could be the author. These fields get tokenized and stored in the index for quick retrieval during searches.

Analyzer

The Analyzer class is crucial for text processing, both during indexing and searching. An analyzer tokenizes the text and applies various transformations like lowercasing, stemming, and stop-word removal. Lucene provides several built-in analyzers, including StandardAnalyzer and WhitespaceAnalyzer, each suitable for different types of text data. However, you can also extend the Analyzer class to create custom analyzers tailored to your specific needs.

QueryParser

The QueryParser class is used for converting human-readable query strings into Lucene Query objects. For example, if you have a search box in your application where users can type in their search queries, you can use QueryParser to translate these strings into queries that Lucene can understand and execute. This makes it easier to implement search functionalities that are both powerful and user-friendly.

IndexSearcher

The IndexSearcher class is responsible for executing search queries against a Lucene index. Once you have a Query object, you pass it to an instance of IndexSearcher. The searcher then returns a TopDocs object, which contains an array of ScoreDoc objects. Each ScoreDoc represents a document that matches the search query, along with a score indicating the relevance of the match.

As you can see, the core architecture of Apache Lucene is quite simple. The different components make it easy to get started easily with the project, as seen in the next section where we add it to a Jakarta EE application.

Adding Lucene To A Jakarta EE Application

To add Apache Lucene to a Jakarta EE application, add the following dependencies in your pom.xml file. Remember to change the versions to the latest at the time you read this guide.

```
<dependency>
            <groupId>org.apache.lucene</groupId>
            <artifactId>lucene-core</artifactId>
            <version>9.7.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.lucene</groupId>
            <artifactId>lucene-queryparser</artifactId>
            <version>9.7.0</version>
         </dependency>
```

Integrating Lucene with Jakarta EE

There are different ways you can integrate the Lucene API in your Jakarta EE applications. THere is no right or wrong way. Whichever way you choose will depend on your project and personal tastes and preferences. That said, let's see how you can incorporate the API into your project using Jakarta EE constructs like Jakarta CDI producers, injection and MicroProfile Config Properties.

Where Does Lucene Fit in a Typical Jakarta EE Architecture?

In a standard Jakarta EE application architecture, Lucene typically fits into the service layer, acting as a specialised service for text search. Here's how it generally integrates:

- **Database Layer:** Your Jakarta EE application will likely interact with a relational or NoSQL database for CRUD (Create, Read, Update, Delete) operations.
- **Jakarta EE Services Layer:** This is where business logic resides. Lucene usually becomes a part of this layer, perhaps encapsulated within a dedicated SearchService class that handles indexing and search operations.
- **Controller Layer:** The Jakarta EE controllers or REST endpoints would interact with the SearchService to handle incoming search requests and return the search results.
- **User Interface:** The UI can be a Jakarta EE view technology like JSF, or a client-side technology like React or Angular. It interacts with the controller layer to initiate searches and display results.

As stated above, we will be using standard Jakarta EE constructs to fit Lucene into our service layer. Let's see how by first creating Jakarta CDI producers for IndexWriter and IndexReader.

Initializing IndexWriter

There are different ways an IndexWriter can be created. You can opt to create instances everywhere you need them. But would be unnecessarily repeating yourself. A much easier way would be to have a sort of Factory that produces instances on demand. The following code snippet shows the use of Jakarta CDI producer method to produce IndexWriter instances that can then be injected, effectively turning IndexWriter instances into CDI beans that can be managed by the Jakarta CDI container.

```
@ApplicationScoped
public class OpenAIFactory {

private Directory directory;
@Inject
@ConfigProperty(name = "index.path")
private String indexDirectoryPath;

@PostConstruct
private void init() {

directory = FSDirectory.open(Paths.get(indexDirectoryPath));
}


@Produces
@Dependent
public IndexWriter produceIndexWriter() {
Analyzer analyzer = new StandardAnalyzer();
IndexWriterConfig config = new IndexWriterConfig(analyzer);
return new IndexWriter(directory, config);
}
```

The `OpenAIFactory` class first initialises a Directory in the `@PostConstruct` callback listener. This method is called by the CDI runtime when this bean has been fully instantiated but before it's put into service. Within it we call the static open method on the `FSDirectory` class, passing it a Path object obtained from reading a configuration value injected using the MicroProfile Config API. The `produceIndexWriter` method creates an `IndexWriterConfig` with a `StandardAnalyzer`, passing it to the constructor of the IndexWriter, along with the created Directory.

With this in place, we can now inject IndexWriter instances with @Inject and have the CDI runtime consult this method to get an instance for us. The returned instances from the method are set to @Singleton scope, meaning we have a single instance of the IndexWriter across the application. Of course this is an architectural choice and your use case might require a different scope. In that case, all you have to do is change the scope of the method. IndexWriter is thread safe.

Initialising IndexSearcher

An IndexSearcher instance is responsible for executing queries against the index. Similar to the way we created the IndexReader above, we can also create IndexSearcher instances through CDI producers as shown below. The shown method snippet is also in the OpenAIFactory class shown above.

```
@Produces
@ApplicationScoped
public IndexSearcher produceIndexSearcher() {
IndexReader indexReader = indexReader = DirectoryReader.open(directory);
return new IndexSearcher(indexReader);
}
```

With our two core components ready, we are ready to start indexing data with Lucene.

Creating a Lucene Index

Creating a Lucene index essentially means populating it with documents that can later be searched. The steps involve creating a Document object, populating it with Fields and then passing it to the IndexWriter to index. For starters, let's see the class whose instances we'd want to index.

```
@Entity
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private UUID id;

    private String title;
    private String author;

    private String content;
    private LocalDate publishedDate;
    private BigDecimal price;
}
```

We have a Jakarta Persistence entity that we'd want to index for faster search. Essentially we want to have a search implementation in front of our database. So instead of directly querying our database, we'd want to query our index. To index instances of the Book entity, we create a controller that handles search related features for us. Let's explore it next.

```
@ApplicationScoped
public class SearchService {

@Inject
IndexWriter indexWriter;

public void indexBook(final Book book) {
Document document = new Document();
Jsonb jsonb = JsonbBuilder.create();
Map<String, Object> cityMap = jsonb.fromJson(jsonb.toJson(book), new
TypeLiteral<Map<String, Object>>() {
}.getType());
cityMap.forEach((k, v) -> document.add(new TextField(k, v.toString(), Field.
Store.YES)));

indexWriter.addDocument(document);
indexWriter.commit();



}
```

The SearchService application scoped singleton controller has an indexBook method that takes a Book instance and adds it to an index. The class declares a dependency on IndexWriter through @Inject annotation. This field will be instantiated with the instance returned from our producer method explored earlier. The method creates a key-value pair of the Book instance, and then for reach pair, creates a TextField instance to be added to the Document. The fully constructed document is then added to the IndexWriter and committed through the addDocument and commit methods, respectively.

The construct used in this method is an attempt to avoid having to manually add each and every field of the Book class to the index by calling get methods. Using the Jakarta JSON-B API, we can easily transform Java objects into JSON, and then transform the JSON back to a `java.util.Map` instance keyed to a String. This way, we don't have to change anything in the method when the fields of the Book class change. This style is a matter of taste and project dependent. There is really no right or wrong way to create a Document.

Real-Time Indexing and Data Synchronisation

In a dynamic application where the underlying data changes frequently, it could get tedious having to manually keep the Lucene index in sync with the Jakarta EE database. Out-of-sync data can lead to incorrect or outdated search results, which can significantly impact the user experience. We can use the Jakarta Persistence entity lister construct to get around this problem.

The simplest way to keep the Lucene index in sync with the database is to re-index the modified records whenever a Create, Read, Update, or Delete (CRUD) operation occurs. You can use Jakarta EE events or JPA entity listeners to trigger these indexing operations.

For instance, we can define a JPA PrePersist and PostPersist listeners to index newly created or updated entities as follows.

```
@EntityListeners(SearchService.class)
@Entity
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private UUID id;

    private String title;
    private String author;

    private String content;
    private LocalDate publishedDate;
    private BigDecimal price;
}
```

Our Book entity now has @EntityListener annotation that takes our SearchService class as an argument. The updated SearchService class method indexBook is shown below.

```
@PrePersist
@PreUpdate
public void indexBook(final Book book) {
Document document = new Document();
Jsonb jsonb = JsonbBuilder.create();
Map<String, Object> cityMap = jsonb.fromJson(jsonb.toJson(book), new
TypeLiteral<Map<String, Object>>() {
}.getType());
cityMap.forEach((k, v) -> document.add(new TextField(k, v.toString(),
Field.Store.YES)));

indexWriter.addDocument(document);
indexWriter.commit();
}
```

The method now has the @PrePersist and @PostConstruct annotations. These annotations are callback listeners that will be called by the Jakarta Persistence runtime when a new entity is created or updated. With these in place, we have automated the indexing of newly created or updated entities. Of course if you have a really big application, this may not be optimal. Later in this guide, we will touch on other, much more performant ways of keeping data in sync.

Searching with Lucene

After successfully indexing the data, the next step is to perform searches. Lucene offers a variety of query types to suit different needs. From basic searches like TermQuery to more advanced ones like FuzzyQuery, Lucene provides a powerful search mechanism that can be easily integrated into your applications.

Basic Search Queries

There are different types of queries that can be created. All queries are executed by the IndexSearcher. The following sections show code snippets of the various types of queries available.

TermQuery

This is the simplest form of query. It searches for documents that contain a specific term in a specified field. A basic TermQuery on the book title looks as follows.

```
Query termQuery = new TermQuery(new Term("title", "sample"));
```

BooleanQuery

This query combines multiple queries using boolean operators like AND, OR, and NOT.

```
BooleanQuery.Builder booleanQuery = new BooleanQuery.Builder();
booleanQuery.add(new TermQuery(new Term("title", "sample")), BooleanClause.
Occur.MUST);
booleanQuery.add(new TermQuery(new Term("author", "john")), BooleanClause.
Occur.MUST_NOT);
```

PhraseQuery

Searches for a sequence of terms within a specified field.

```
PhraseQuery phraseQuery = new PhraseQuery("title", "sample", "document");
```

Advanced Search Queries

FuzzyQuery

Searches for terms that are similar to the specified term.

```
FuzzyQuery fuzzyQuery = new FuzzyQuery(new Term("title", "sampl"));
```

WildcardQuery

Allows for wildcard characters (‘*’ and ‘?’) in the search term.

```
WildcardQuery wildcardQuery = new WildcardQuery(new Term("title", "sam*"));
```

RangeQuery

Searches for documents where the specified field has a value within a certain range.

```
Query rangeQuery = IntPoint.newRangeQuery("price", 10, 20);
```

Executing Queries

Executing any query entails handing it to the IndexSearcher and retrieving matched Documents from the search. The query method in the SearchService class shows how the various queries above can be executed through the IndexSearcher.

```
public List<Document> query(final Query query) {
    List<Document> documents = new ArrayList<>();

    TopDocs topDocs = indexSearcher.search(query, 20);
    StoredFields storedFields = indexSearcher.storedFields();
    for (ScoreDoc hit : topDocs.scoreDocs) {
    Document doc = storedFields.document(hit.doc);
    documents.add(doc);
    }

    return documents;
    }
```

The query method takes a Query object, the super class of all the queries types listed above. The search method of the IndexSearcher is passed the query, along with a number of hits to return. The method returns a TopDocs object that contains an array of ScoreDocs. The IndexSearcher has a storedFields method that returns a StoredFields object. The method then iterates over the scoreDocs array on the TopDocs object, calling and passing the doc on the ScoreDoc to the document method of the storedFields. Each returned Document is added to a list that is returned after. This is a very high level view of executing queries with the IndexSearcher that was injected into the class.

Improving Search Results

Once you've got the basics of searching down, the next logical step is to improve the quality of your search results. Lucene provides several mechanisms to fine-tune your search outcomes, ranging from scoring and analyzers to result highlighting.

Scoring Results

Scoring is the process by which Lucene ranks the search results. Each document in a search result has a score that determines its relevance to the query. Lucene uses a complex formula to calculate this score, but you can influence it using boosting. Boosting allows you to give more weight to certain terms or fields during indexing or searching.

Here's how you can apply boosting during query construction:

```
TermQuery termQuery = new TermQuery(new Term("title", "lord of the
rings"));
BoostQuery boostedQuery = new BoostQuery(termQuery, 2.0f);
```

In the above code snippet, TermQuery is being created with a Term that targets documents having the word "lord of the rings" in their "title" field. In Lucene, a Term is an object that represents a pair of field name and term text. So, the Term object here specifies that we are interested in the "title" field and are looking for documents that contain the word "lord of the rings". The TermQuery is one of the simplest yet most powerful types of queries in Lucene.

BoostQuery wraps around the TermQuery to modify its behaviour. The BoostQuery takes two parameters:

- The original query you want to boost (in this case, termQuery)
- A boost factor (in this case, 2.0f)

The boost factor is a floating-point number that adjusts the relevance score of the documents that match the query. A boost factor greater than 1 will make the query more important, thereby increasing the scores of the matching documents. In this example, the boost factor of 2.0f means that the score for the documents that match the term "lord of the rings" in the "title" field will be doubled. Another way to improve search results is through the use of Analyzers.

Scalability and Performance

When integrating Lucene with Jakarta EE applications, scalability and performance are paramount concerns. As your application grows, so does the volume of data to index and search. This section outlines best practices for scaling Lucene and offers performance-tuning tips to ensure your Jakarta EE application runs smoothly.

Lucene Jakarta EE Best Practices

Sharding

Divide your index into smaller parts, known as shards. Each shard is a self-contained index. This enables horizontal scaling and allows you to distribute your index across multiple servers or clusters.

Replication

For read-heavy workloads, replicate your shards to multiple nodes. This ensures high availability and fault tolerance.

Load Balancing

Use a load balancer to distribute search and index requests evenly across nodes. This is crucial for maintaining a responsive and fast application.

Caching

Implement caching strategies for frequent queries and filters using Lucene's LRUQueryCache or Jakarta EE's caching mechanisms.

Asynchronous Operations

Use asynchronous methods to handle indexing, especially when dealing with large volumes of data. This helps in not blocking the main application flow. The advent of Java 21 and virtual threads means you can cheaply offload indexing to a separate thread as well.

Performance Tuning Tips

Optimise Index

Lucene provides an IndexWriter.optimize() method, which merges all segments into a single segment, reducing the index size and improving search performance. However, this operation is I/O-intensive, so use it judiciously.

Use Bulk Operations

When adding or updating several documents, do it in bulk. Lucene is optimised for batch operations, which are far more efficient than single-document writes.

Tune RAM Buffer Size

The IndexWriterConfig class allows you to set the RAM buffer size, controlling how much data is buffered before it's written to disk. Increasing this size can improve indexing speed but uses more memory.

Selective Indexing

Be selective in what fields you index and store. Index only the fields that you'll search or sort on, and store only those you'll display. This reduces disk I/O and speeds up search.

Use DocValues for Sorting and Aggregations

If you often sort results or perform aggregations, consider using DocValues, a disk-based data structure that enables fast random access.

Security Considerations

While Lucene itself does not provide built-in security features for indexes, it's crucial to consider security aspects when integrating Lucene with Jakarta EE applications. Let's discuss ways to secure Lucene indexes and integrate these security measures with Jakarta EE's existing security features.

File System Permissions

The simplest way to secure Lucene indexes is by setting proper file system permissions. Limit access to the directory where the index files are stored to only the users who need it.

Encryption

To further secure your index, you can encrypt the index files. This can be done either at the file system level or by using custom encryption methods before writing to or reading from the index. Keep in mind that encryption can affect performance.

Secure Communication

If your architecture involves remote indexes, make sure the communication between the Jakarta EE application and the index server is encrypted, using protocols like HTTPS.

Authentication and Authorization

Implement strong authentication and authorization checks before allowing any operation on the index. This ensures that only authorised users can perform actions like read, write, or delete on the index.

Conclusion

Implementing full-text search in a Jakarta EE application using Apache Lucene is a powerful way to enhance your application's data retrieval capabilities. Throughout this guide, we've walked through the essential steps to achieve this integration effectively. From understanding the core concepts of Lucene to setting up your Jakarta EE project, we've covered indexing data, querying the index, and even real-time data synchronisation. Along the way, we've also touched upon important considerations for scalability, performance, and security.

Key points to remember:

- Lucene's flexibility and efficiency make it an excellent choice for implementing search capabilities in Jakarta EE applications.
- Effective indexing and querying are crucial for a robust search feature.
- Scalability and performance are vital for enterprise-level applications and can be achieved through sharding, replication, and careful query design.
- Security should not be an afterthought. Make sure to protect your indexes and integrate with Jakarta EE's security features.

By applying the best practices and tips shared in this guide, you're well on your way to implementing a robust, scalable, and secure full-text search functionality in your Jakarta EE applications. Happy coding!

Interested in Payara? Try Before You Buy

PAYARA SERVER FREE TRIAL

PAYARA CLOUD FREE TRIAL

sales@payara.fish

UK: +44 800 538 5490
Intl: +1 888 239 8941

www.payara.fish

Payara Services Ltd 2023 All Rights Reserved. Registered in England and Wales; Registration Number 09998946
Registered Office: Malvern Hills Science Park, Geraldine Road, Malvern, United Kingdom, WR14 3SZ