Spring Lucene Reference Guide

0.8-SNAPSHOT

Thierry Templier (Argia-Engineering)

Part I. Introduction

Chapter 1. Lucene

1.1. Lucene technology

According to the home page project, "Apache Lucene is a high-performance, full-featured text search engine library written entirely in Java. It is a technology suitable for nearly any application that requires full-text search, especially cross-platform".

The project is hosted by Apache and allows to make scalable architecture based on distributed indexes by providing supports for indexing and searching indexes. It provides several kinds of indexes (in-memory, file-system based, database based).

Here are the list of the Lucene related projects:

- Lucene Java (Java implementation),
- Nutch,
- Lucy (C implementation),
- Solr,
- Lucene.Net (.Net implementation),
- Tika,
- Mahout.

1.2. Related tools

1.2.1. Lucene Java

Lucene Java is the implementation of the Lucene technology with the Java language. Other languages like C and .Net are still supported with respectively the Lucy and Lucene.Net projects.

Lucene Java allows to interact and search the index using the Java language.

The tool is available at the url http://lucene.apache.org/java/

1.2.2. Solrj

Solr is an open source enterprise search server based on the Lucene Java search library, with XML/HTTP and JSON APIs, hit highlighting, faceted search, caching, replication, a web administration interface and many more features. It runs in a Java servlet container such as Tomcat.

The tool is available at the url http://lucene.apache.org/solr/

1.2.3. Tika

Apache Tika is a toolkit for detecting and extracting metadata and structured text content from various documents using existing parser libraries.

The tool is available at the url http://lucene.apache.org/tika/

Chapter 2. Other tools available

In this section, we will describe the other tools available in the Java community in order to make easier the use of the Lucene technology. For eath tool, we will show how the Spring Lucene support is different.

2.1. Compass

TODO

2.2. Hibernate Search

TODO

Chapter 3. Aim of the support

The aim of the Lucene support is to provide facilities in order to use the tools based on Lucene in a Spring environment. The support follows the principles than those used in the Spring framework.

With the current version of the tool, only Lucene Java is supported but the supports of Solrj and Tika will be added soon in the next version. The same features are provided in the context of these tools.

3.1. Architecture and structure

TODO: describe the high level architecture of the project

TODO: describe the approach to make easy the use of Lucene technologies in Spring applications

3.2. Abstraction layer upon Lucene

The support provides a thin layer upon the Lucene API which offers the same feature basing on interfaces. It enabled a level of indirection in order to make easier the resource management in different context and make possible unit tests of classes using Lucene. This thin layer covers both indexing and searching

With this feature, you can specify the resource management strategy declaratively in the Spring configuration according to the use context of Lucene. This aspect is supported for both indexing and searching with dedicated strategies. As a matter of fact, Lucene owns specific resources management to access indices. For instance, you can share searcher instances in order to search in a concurrent environment. However, only one instance of Lucene writer can access an index in order to update it.

3.3. Easy configuration

In addition, the support provides interesting facilities in order to configure these resources and the use strategies in a Spring environment thanks to a dedicated namespace. This configuration facility is available for both indexing and searching.

3.4. Template-driven approach

Besides this aspect, the support provides too a classic template-driven approach, this approach being currently used in the Spring integration of tools. Templates integrate all the technical plumbing management allowing you to concentrate on the specific code of your application.

The support integrate itself the declarative transaction management of Spring with dedicated implementations of `PlatformTransactionManager` interface related to Lucene tools. These features are based on the underlying API of these tools.

3.5. Document handler

On the other hand, the support provides a generic document handling feature in order to offer dedicated entities to create documents . This feature also manages the document and handler association. The handler has the responsibility to create a document from an object or an *InputStream* . The feature is particularly useful to manage the indexing of different file formats.

Chapter 4. History of the support

TODO

Chapter 5. Development

This section is dedicated to developers who want to get the sources, compile them and eventually contribute to the project.

5.1. Obtain the sources

TODO: describe how to obtain sources with svn

5.2. Modules

TODO: list and describe the modules of the projects

5.3. Compile and build the modules

TODO: describe how to compile and build the modules using Maven

Chapter 6. Quick start dedicated to the Lucene indexing support

6.1. Aim of the quickstart

The aim of this section is to provide quickly a short view of the way to implement indexing on a Lucene index using the Lucene support. It allows us to show the usage of the main entities of this support and how to configure them in a simply way.

6.2. Configure the index resources

The Lucene support of Spring provides a dedicated namespace in order to easily configure resources related to the index. It supports both in memory and filesystem indexes.

The first step of the configuration is to create a root structure of the Spring XML file integrating the lucene namespace. This configuration is shown in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:lucene="http://www.springframework.org/schema/lucene"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
            http://www.springframework.org/schema/lucene
                http://www.springframework.org/schema/lucene/lucene-index.xsd">
    (...)
</beans>
```

For the example, we configure a simple filesystem index whose the content is located in the classpath. The tag index of the namespace and its attributes must be used in order to configure this aspect, as shown in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans (...)>
    <bean id="analyzer" class="org.apache.lucene.analysis.SimpleAnalyzer"/>

    <lucene:index id="myDirectory" analyzer-ref="analyzer">
        <lucene:fsDirectory location="/org/springframework/lucene/config"/>
        <lucene:indexFactory/>
    </lucene:index>
</beans>
```

You can note that the simple analyzer of Lucene is used and configured as a bean in the previous code.

Now that you have configured the index with the index tag, the lucene support allows you to access a factory in order to obtain resources in order to interact with the index. This factory is automatically configured as a Spring bean with the identifier specified on the index tag.

6.3. Indexing documents

The next step is to use the configured factory in order to interact with the index, indexing documents in our case. In that purpose, we create an indexing service class named SampleIndexingService. The later used the abstract class LuceneIndexDaoSupport as parent class to have the setter method for the factory previously configured.

The following code shows the skeleton of the class SampleIndexingService:

```
public class SampleIndexingService extends LuceneIndexDaoSupport {

    public void indexDocuments() {
        (...)
    }
}
```

The configuration of this class in Spring is really simple, as described in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans (...)>
    (...)

    <lucene:index id="myDirectory" analyzer-ref="analyzer">
        (...)
    </lucene:index>

    <bean id="sampleIndexingService" class="org.springframework.lucene.sample.SampleIndexingService">
        <property name="indexFactory" ref="myDirectory-indexFactory"/>
    </bean>
</beans>
```

You have now a fully configured class for indexing. You can now use the central entity of the Lucene indexing support, the LuceneIndex>Template, in order to implement your indexing features. In the sample, we describe how to index two documents, as in the following code:

```
public void indexDocuments() {
    Document document1 = new Document();
    document1.add(new Field("field", "a sample 1", Field.Store.YES, Field.Index.ANALYZED));
    document1.add(new Field("sort", "2", Field.Store.YES, Field.Index.NOT_ANALYZED));
    getLuceneIndexTemplate().addDocument(document1);

    Document document2 = new Document();
    document2.add(new Field("field", "a sample 2", Field.Store.YES, Field.Index.ANALYZED));
    document2.add(new Field("sort", "1", Field.Store.YES, Field.Index.NOT_ANALYZED));
    getLuceneIndexTemplate().addDocument(document2);
}
```

6.4. Transactions

The support provides an implementation in order to use the declarative transaction support of Spring. This implementation, the class LuceneIndexTransactionManager, can be configured basing on the previous factory. The following code describes the configuration of the manager:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans (...)>
    (...)

    <bean id="luceneIndexTransactionManager"
          class="org.springframework.lucene.index.factory.LuceneIndexTransactionManager">
        <property name="indexFactory" ref="myDirectory-indexFactory"/>
    </bean>
</beans>
```

Then, the aop and transactional supports of Spring can be used in order to apply transactions on the indexDocuments method of the SampleIndexingService class, as described below:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<beans (...)>
    (...)

    <aop:config>
        <aop:advisor pointcut="execution(* *..SampleIndexingService.*(..))" advice-ref="txAdvice"/>
    </aop:config>

    <tx:advice id="txAdvice" transaction-manager="transactionManager">
        <tx:attributes>
            <tx:method name="indexDocuments"/>
        </tx:attributes>
    </tx:advice>
</beans>
```

Chapter 7. Quick start dedicated to the Lucene search support

7.1. Aim of the quickstart

The aim of this section is to provide quickly a short view of the way to implement search on a Lucene index using the Lucene support. It allows to show the usage of the main entities of this support and how to configure them in a simply way.

We assume that you run the previous quick start in order to populate the index and have documents that match the query.

7.2. Configure the search resources

The Lucene support of Spring provides a dedicated namespace in order to easily configure resources related to the index. It supports both in memory and filesystem indexes.

The first step of the configuration is to create a root structure of the Spring XML file integrating the lucene namespace. This configuration is shown in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:lucene="http://www.springframework.org/schema/lucene"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
            http://www.springframework.org/schema/lucene
                http://www.springframework.org/schema/lucene/lucene-index.xsd">
    (...)
</beans>
```

For the example, we configure a simple filesystem index whose the content is located in the classpath. The tag index of the namespace and its attributes must be used in order to configure this aspect. An instance of SearcherFactory must be configured too in order to use the search facility of Spring.

The following code shows the configutation of these entities:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans (...)>
    <bean id="analyzer" class="org.apache.lucene.analysis.SimpleAnalyzer"/>

    <lucene:index id="myDirectory" analyzer-ref="analyzer">
        <lucene:fsDirectory location="/org/springframework/lucene/config"/>
        <lucene:searcherFactory/>
    </lucene:index>
</beans>
```

You can note that the simple analyzer of Lucene is used and configured as a bean in the previous code.

Now that you have configured the index with the index tag, the lucene support allow you to access a factory in order to obtain resources in order to interact with the index. This factory is automatically configured as a Spring bean with the identifier specified on the index tag.

7.3. Search documents in the index

The next step is to use the configured factory in order to search the index. In that purpose, we create a search service class named SampleSearchService. The later used the abstract class LuceneSearchDaoSupport as parent class to have the setter method for the factory previously configured.

The following code shows the skeleton of the class SampleSearchService:

```
public class SampleSearchService extends LuceneSearchDaoSupport {

    public void searchDocuments() {
        (...)
    }
}
```

The configuration of this class in Spring is really simple, as described in the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans (...)>
    (...)

    <lucene:index id="myDirectory" analyzer-ref="analyzer">
        (...)
    </lucene:index>

    <bean id="sampleSearchService" class="org.springframework.lucene.sample.SampleSearchService">
        <property name="searcherFactory" ref="myDirectory-searcherFactory"/>
    </bean>
</beans>
```

You have now a fully configured class for searching. You can now use the central entity of the Lucene search support, the LuceneSearchTemplate, in order to implement your searching features. In the sample, we describe how to search documents basing a string query, as in the following code:

```
public void searchDocuments() {
    List<String> results = getLuceneSearchTemplate().search("field:lucene", new HitExtractor() {
        public Object mapHit(int id, Document document, float score) {
            return document.get("field");
        }
    });

    for (String field : results) {
        System.out.println("Result: " + field);
    }
}
```

Part II. Lucene Support Reference Documentation

Chapter 8. Lucene indexing support

8.1. Concepts

TODO

8.2. Configure indices

The support provides facilities in order to configure different kinds of directories in order to access indices. Both RAM and filesystem directories are supported thanks to the `RAMDirectoryFactoryBean` and `FSDirectoryFactoryBean` classes.

These two classes allow to handle correctly directories, i.e. they are closed on the shutdown of the application context of Spring. However, these entities don't allow the creation new indices when they don't exist. To do that, you must use an implementation of the `IndexFactory` interface like the `SimpleIndexFactory` class and its `create` property. In this case, the index structure is created at the first access.

The following code shows how to configure a RAM directory:

```
<bean id="ramDirectory" class="org.springframework.lucene.index.support.RAMDirectoryFactoryBean"/>

<bean id="indexFactory" class="org.springframework.lucene.index.factory.SimpleIndexFactory">
    <property name="directory" ref="ramDirectory"/>
    <property name="create" value="true"/>
</bean>
```

The following code shows how to configure a filesystem directory:

```
<bean id="fsDirectory" class="org.springframework.lucene.index.support.FSDirectoryFactoryBean"/>

<bean id="indexFactory" class="org.springframework.lucene.index.factory.SimpleIndexFactory">
    <property name="directory" ref="ramDirectory"/>
    <property name="create" value="true"/>
</bean>
```

8.3. Root entities

indirection level. change the strategy of management of resource in the configuration without any change in the code. TODO: IndexFactory and abstraction layer TODO: IndexReaderWrapper/IndexWriterWrapper and abstraction layer

The following code describes the content of the `IndexFactory` interface, entity allowing the access to the resources in order to interact with the index:

```
public interface IndexFactory {
    IndexReaderWrapper getIndexReader();
    IndexWriterWrapper getIndexWriter();
}
```

The following table shows the different implementations of the `IndexFactory` interface provided by the support:

Table 8.1.

Implementation	Description
SimpleIndexFactory	The simplest and by default implementation of the interface which is based on the `SimpleIndexReaderWrapper` and `SimpleIndexWriterWrapper` implementations. For each ask of a resource, a new one is created basing on the injected directory. This implementation supports too the creation of the structure for a new index and the locking resolution. In this case, you need to use respectively the `create` and `resolveLock` properties.
LockIndexFactory	The concurrent implementation based on a lock strategy. This class is a delegating implementation that encapsulate a target index factory. The implementation put a lock at the acquiring of a resource and leave it after its use.

8.4. Index root entities

The Lucene indexing support adds other abstractions in order to TODO:

Table 8.2.

Entity	Interface	Description
Document creator	`DocumentCreator`	This entity enables to create a document in order to use it in an indexing process (add or update).
Document creator using an InputStream	`InputStreamDocumentCreator`	This entity enables to create a document using an InputStream (related to a file or others) in order to use it in an indexing process (add or update).
Documents creator	`DocumentsCreator`	This entity enables to create several documents in order to use it in an indexing process (add or update).
Reader callback	`ReaderCallback`	This entity corresponds to a callback interface in order to enable the use of the underlying resource, Lucene reader wrapper, managed by the Lucene support.
Writer callback	`WriterCallback`	This entity corresponds to a

Entity	Interface	Description
		callback interface in order to enable the use of the underlying resource, Lucene writer wrapper, managed by the Lucene support.

The central entity of the support used to execute indexing operations is the Lucene indexing template. It offers several ways to configure indexing according to your needs and your knowledge of the underlying API of Lucene. In this context, we can distinguish three levels to implement indexing in the support:

- Using the abstraction level provided by the template;
- Using the template with the Lucene entities like `Document` and `Term`;
- Advanced indexing using the underlying writer instance.

The template allows you to handle several kinds of operations, as described in the following list:

- Add operations of document in the index;
- Update operations of documents in the index;
- Delete operations of documents in the index;
- Extra operations like optimizing the index.

We will describe now all these features with these different approaches in the following sections.

8.4.1. Simple indexing based on Lucene entities

The support lets you use directly the Lucene entities like the Document and/ Term classes in order to manipulate the index. In this case, you have the responsability to create the documents to add or update and the term. The latters are then passed to the `addDocument(s)` and `updateDocument(s)` methods. In this case, the template has the responsability to handle the underlying Lucene resources in order to manage the operation.

Related to the adding, the support provides the ability to add only one document with the `addDocument` methods. In order to add several documents, `addDocuments` methods are provided too.

The following example shows how to create a Lucene document and add it to the index using the `addDocument` method:

```
Document document = new Document();
document.add(new Field("field", "a sample 1", Field.Store.YES, Field.Index.ANALYZED));
document.add(new Field("sort", "2", Field.Store.YES, Field.Index.NOT_ANALYZED));
getLuceneIndexTemplate().addDocument(document);
```

The following example shows how to create a list of Lucene documents and add it to the index using the `addDocuments` method:

```
List<Document> documents = new ArrayList<Document>();

Document document1 = new Document();
document1.add(new Field("field", "a sample 1", Field.Store.YES, Field.Index.ANALYZED));
document1.add(new Field("sort", "1", Field.Store.YES, Field.Index.NOT_ANALYZED));
documents.add(document1);
```

```
Document document2 = new Document();
document2.add(new Field("field", "a sample 2", Field.Store.YES, Field.Index.ANALYZED));
document2.add(new Field("sort", "2", Field.Store.YES, Field.Index.NOT_ANALYZED));
documents.add(document2);

getLuceneIndexTemplate().addDocuments(documents);
```

In the case of updating documents, the template does a smart update because it checks if the term corresponds exactly to a document in the case of the `updateDocument` method and at least to one document in the case of the `updateDocuments` method.

The update methods of the template use internally the update methods of the index writer. These latters make a delete of documents based on the specified term and then add the document. Thus, the corresponding methods of the template follow the same mechanism.

The following code shows the use of the `updateDocument` method:

```
Document document = new Document();
document.add(new Field("field", "a Lucene sample", Field.Store.YES, Field.Index.ANALYZED));
document.add(new Field("sort", "2", Field.Store.YES, Field.Index.NOT_ANALYZED));
getLuceneIndexTemplate().updateDocument("field:lucene", document);
```

The following code shows the use of the `updateDocuments` method in order to update several documents with one operation of the template:

```
List<Document> documents = new ArrayList<Document>();

Document document1 = new Document();
document1.add(new Field("field", "a sample", Field.Store.YES, Field.Index.ANALYZED));
document1.add(new Field("sort", "1", Field.Store.YES, Field.Index.NOT_ANALYZED));
documents.add(document1);

Document document2 = new Document();
document2.add(new Field("field", "a Lucene sample", Field.Store.YES, Field.Index.ANALYZED));
document2.add(new Field("sort", "2", Field.Store.YES, Field.Index.NOT_ANALYZED));
documents.add(document2);

getLuceneIndexTemplate().updateDocuments("field:sample", documents);
```

The delete methods follow the same mechanisms of the update methods according the checks of documents. Both `deleteDocument` and `deleteDocuments` are provided by the template in order to delete one or more documents.

The following code shows the use of `deleteDocument` and `deleteDocuments` methods:

```
getLuceneIndexTemplate().deleteDocument("field:lucene");
getLuceneIndexTemplate().deleteDocuments("field:sample");
```

8.4.2. Indexing using the abstraction methods of the template

The template of the Lucene support provides too the ability to use an abstraction layer in the process of Lucene document creation. The main advantages of this approach consists in the handling of resources according to the exception eventually thrown during the document creation. The template supports both simple document creators and input stream based document creators.

The template offers the possibility to use this mechanism with `addDocument(s)` and `updateDocument(s)` methods. We will describe now this feature.

The following code describes the content of the simplest document creator, which provides a simple way to create a Lucene document:

```
public interface DocumentCreator {
    Document createDocument() throws Exception;
}
```

The following code shows the way to use the `DocumentCreator` interface with the `addDocument` method of the template:

```
getLuceneIndexTemplate().addDocument(new DocumentCreator() {
    public Document createDocument() throws Exception {
        Document document = new Document();
        document.add(new Field("field", "a Lucene sample", Field.Store.YES, Field.Index.ANALYZED));
        document.add(new Field("sort", "2", Field.Store.YES, Field.Index.NOT_ANALYZED));
        return document;
    }
});
```

The `InputStreamDocumentCreator` interface provides a dedicated entity in order to create a Lucene document from an `InputStream`. When using this interface, you need to specify how to obtain the `InputStream` and to create a document from this `InputStream`.

The template has the responsability to correctly handle the `InputStream` and to close it in every case. The following code describes the content of the `InputStreamDocumentCreator` interface:

```
public interface InputStreamDocumentCreator {
    InputStream createInputStream() throws IOException;
    Document createDocumentFromInputStream(InputStream inputStream) throws Exception;
}
```

The following code shows the way to use the `InputStreamDocumentCreator` interface with the `addDocument` method of the template:

```
template.addDocument(new InputStreamDocumentCreator() {
    public InputStream createInputStream() throws IOException {
        ClassPathResource resource = new ClassPathResource(
                    "/org/springframework/lucene/index/core/test.txt");
        return resource.getInputStream();
    }

    public Document createDocumentFromInputStream(InputStream inputStream) throws Exception {
        String contents = IOUtils.getContents(inputStream);

        Document document = new Document();
        document.add(new Field("field", contents, Field.Store.YES, Field.Index.ANALYZED));
        document.add(new Field("sort", "2", Field.Store.YES, Field.Index.NOT_ANALYZED));
        return document;
    }
});
```

Finally the support provides an entity in order to create a list of Lucene documents to be added, the `DocumentsCreator` interface. It is similar to the `DocumentCreator` interface. The following code describes this interface:

```
public interface DocumentsCreator {
    List<Document> createDocuments() throws Exception;
}
```

The following code shows the way to use the `DocumentsCreator` interface with the `addDocuments` method of the template:

```
template.addDocuments(new DocumentsCreator() {
    public List<Document> createDocuments() throws Exception {
        List<Document> documents = new ArrayList<Document>();

        Document document1 = new Document();
        document1.add(new Field("field", "a Lucene sample", Field.Store.YES, Field.Index.ANALYZED));
        document1.add(new Field("sort", "1", Field.Store.YES, Field.Index.NOT_ANALYZED));
        documents.add(document1);

        Document document2 = new Document();
        document2.add(new Field("field", "a sample", Field.Store.YES, Field.Index.ANALYZED));
        document2.add(new Field("sort", "2", Field.Store.YES, Field.Index.NOT_ANALYZED));
        documents.add(document2);

        return documents;
    }
});
```

8.4.3. Analyzer specification

There are two approaches in order to specify an analyzer during the indexing process. On one hand, you can set a global analyzer for the template which is used as default analyzer. Thus, with methods without an `analyzer` parameter, the global analyzer of the template is used. The following code describes the use of this approach:

```
Document document = new Document();
document.add(new Field("field", "a sample 1", Field.Store.YES, Field.Index.ANALYZED));
document.add(new Field("sort", "2", Field.Store.YES, Field.Index.NOT_ANALYZED));
getLuceneIndexTemplate().addDocument(document);
```

In the code above, the default analyzer injected in the template is used to add the document. If no analyzer is defined, an exception is thrown.

The support provides too methods with a `analyzer` parameter. In this case, the specified analyzer is used instead of the default one. The following code describes the use of this approach with the same operation:

```
SimpleAnalyzer analyzer = new SimpleAnalyzer();

Document document = new Document();
document.add(new Field("field", "a sample 1", Field.Store.YES, Field.Index.ANALYZED));
document.add(new Field("sort", "2", Field.Store.YES, Field.Index.NOT_ANALYZED));
getLuceneIndexTemplate().addDocument(document, analyzer);
```

8.4.4. Other indexing operations

TODO: describe the use of optimize, getXXX methods

8.4.5. Use of the underlying resource

The aim of the indexing template is to integrate and hide the use of Lucene API in order to make easy the indexing. This entity provides to the developer all the common operations in this context. However, if you need to go beyond these methods, the template enables to provide the underlying reader and writer entities in order to use it explicitely.

The feature is based on the `ReaderCallback` and `WriterCallback` interfaces described above. When using the entity, you need to implement respectively the `doWithReader` and `doWithWriter` methods which gives you the underlying instance corresponding the reader and the writer. These latter can be used in your indexing.

The Lucene support continues however to handle and manage these resources. The code below describes the content of the interface `ReaderCallback`:

```
public interface ReaderCallback {
    Object doWithReader(IndexReaderWrapper reader) throws Exception;
}
```

The code below describes the content of the interface `WriterCallback`:

```
public interface WriterCallback {
    Object doWithWriter(IndexWriterWrapper writer) throws Exception;
}
```

This entity can be used as parameter of the `read` and `writer` methods of the template, as shown in the following code:

```
getLuceneIndexTemplate().write(new WriterCallback() {
    public Object doWithWriter(IndexWriterWrapper writer) throws Exception {
        Document document = new Document();
        document.add(new Field("field", "a sample", Field.Store.YES, Field.Index.ANALYZED));

        writer.addDocument(document);

        return null;
    }
});
```

When using the underlying writer, the analyzer specified for the index template is not used. You need to explicitely specify it on your calls or configure it on the index factory used. In the latter case, the analyzer is set during the creation of the writer.

8.5. DAO support class

Like in the other dao supports of Spring, the Lucene support provides a dedicated entity in order to make easier the injection of resources in the entities implementing Lucene indexing. This entity, the `LuceneIndexDaoSupport` class, allows to inject a `IndexFactory` and an analyzer. You don't need anymore to create the corresponding injection methods.

In the same time, the class provides too the `getLuceneIndexTemplate` method in order to have access to the index template of the support.

The following code describes the use of the `LuceneIndexDaoSupport` in a class implementing a search:

```
public class SampleSearchService extends LuceneIndexDaoSupport {

    public void indexDocuments() {
        Document document = new Document();
        document.add(new Field("field", "a sample 1", Field.Store.YES, Field.Index.ANALYZED));
        document.add(new Field("sort", "2", Field.Store.YES, Field.Index.NOT_ANALYZED));
        getLuceneIndexTemplate().addDocument(document);
    }
}
```

You can note that the class makes possible to directly inject a configured `LuceneIndexTemplate` in Spring.

8.6. Resource management and transactions

Lucene has the particularity to allow the creation of only one index writer for an index simultaneously. That's why you need to be careful when indexing documents.

On the other hand, Lucene provides now a support of transactions when updating the index. The support provides a dedicated implementation of the Spring `PlatformTransactionManager` in order to use the transactional support of the framework.

This implementation, the `LuceneIndexTransactionManager` class, must be configured using an instance of `IndexFactory`, as shown in the following code:

```
<bean id="indexFactory" class="org.springframework.lucene.index.factory.SimpleIndexFactory">
    (...)
</bean>

<bean id="transactionManager" class="org.springframework.lucene.index.factory.LuceneIndexTransactionManager">
    <property name="indexFactory" ref="indexFactory"/>
</bean>
```

This implementation of the `PlatformTransactionManager` supports read-only transactions, which allows to extend the scope of a Lucene index reader but doesn't allow the use of an index writer.

Chapter 9. Lucene search support

9.1. Concepts

TODO

9.2. Root entities

TODO: SearcherFactory and abstraction layer

TODO: SearcherWrapper and abstraction layer

TODO: reopen a searcher

9.3. Search root entities

The Lucene search support adds other abstractions in order to TODO: LuceneSearcher

Table 9.1.

Entity	**Interface**	**Description**
Hit extractor	`HitExtractor`	This entity enables to extract informations of documents contained in the result of the search.
Query creator	`QueryCreator`	This entity specifies how to create a Lucene query basing the different supports of the tools.
Query result creator	`QueryResultCreator`	This entity specifies the strategy to be used in order to retrieve all or a subset of the results. It enables to easily handle pagination and the retrieving of the first results.
Searcher callback	`SearcherCallback`	This entity corresponds to a callback interface in order to enable the use of the underlying resource managed by the Lucene support.

The central entity of the support used to execute searches is the Lucene search template. It offers several ways to configure search contents according to your needs and your knowledge of the underlying API of Lucene. In this context, we can distinguish three levels to implement searches in the support:

- Driven by Lucene query requests as string;
- Custom query building using the `Query` and `QueryParser` classes and subclasses of Lucene;
- Advanced query building using the underlying searcher instance.

We will describe now all these approachs in the following sections.

9.3.1. Simple searches based on a query string

With this approach, you aren't tied to the Lucene API and can directly use search expressions as query strings. It enables you to use all the powerful of Lucene query string for the searches.

The simplest way to make a search is to specify the query string to the `search` method as a parameter. In this case, you need to explicitely specify the fields corresponding to expressions because no default field is used. The following code shows how to execute the query string `"field:lucene"`, i.e. all the documents in the index owing the `field` field containing the term `"lucene"`:

```
List<String> results = getLuceneSearchTemplate().search("field:lucene", new HitExtractor() {
    public Object mapHit(int id, Document document, float score) {
        return document.get("field");
    }
});
```

The support provides too the ability to specify one or several default fields for the search. In this case, there is no need to explicitely specify the fields the search is based on. The following code describes the same search as previously but using the approach using `field` a default field:

```
List<String> results = getLuceneSearchTemplate().search("field", "lucene", new HitExtractor() {
    public Object mapHit(int id, Document document, float score) {
        return document.get("field");
    }
});
```

When using several default fields, a table of strings can be passed as first parameter of the `search` methods.

9.3.2. Use of Lucene classes for the searches

TODO:

9.3.3. Use of the underlying resource

The aim of the search template is to integrate and hide the use of Lucene API in order to make easy the use of searches. This entity provides to the developer all the common operations in this context. However, if you need to go beyond these methods, the template enables to provide the underlying searcher entity in order to use it explicitely.

The feature is based on the `SearcherCallback` interface described above. When using the entity, you need to implement the `doWithSearcher` method which gives you the underlying instance of the searcher. This latter can be used for your search.

The Lucene support continues however to handle and manage this resource. The code below describes the content of this interface:

```
public interface SearcherCallback {
    Object doWithSearcher(SearcherWrapper searcher) throws Exception;
}
```

This entity can be used as parameter of a search method of the template, as shown in the following code:

```
List<String> results = (List<String>) getLuceneSearchTemplate().search(new SearcherCallback() {
    public Object doWithSearcher(SearcherWrapper searcher) throws Exception {
        LuceneHits hits = searcher.search(new TermQuery(new Term("field", "lucene")));
        List<String> results = new ArrayList<String>();
        for (int cpt=0; cpt<hits.length(); cpt++) {
            Document document = hits.doc(cpt);
            results.add(document.get("field"));
        }
        return results;
    }
});
```

9.3.4. Extract results of searches

The Lucene support offers several different strategies in order to extract datas from the result of searches:

- Extraction based of the `HitExtractor` interface, interface provided by the support;
- Configuration of the strategy based on the QueryResultCreator;
- Extraction based of the Lucene `HitCollector` interface.

The central interface used by the search template in order to return the documents of a search is the `HitExtractor` interface. This interface isn't a Lucene interface and is provided by the support. It offers a convenient way to implement mapping in order to convert document into data objects.

In contrary to the Lucene `HitCollector` interface, the `mapHit` method of this interface provides the current document of the iteration. The built object is automatical added in the collection returned.

The following code describes the content of the `HitExtractor` interface:

```
public interface HitExtractor {
    Object mapHit(int id, Document document, float score);
}
```

TODO: control the building of the result collection with QueryResultCreator

The following code describes the content of the `QueryResultCreator` interface:

```
public interface QueryResultCreator {
        List createResult(LuceneHits hits, HitExtractor hitExtractor) throws IOException;
}
```

TODO: Lucene HitCollector

The following code describes the content of the `HitCollector` interface:

```
public interface HitCollector {
    void collect(int doc, float score);
}
```

9.4. DAO support class

Like in the other dao supports of Spring, the Lucene support provides a dedicated entity in order to make easier the injection of resources in the entities implementing Lucene searches. This entity, the `LuceneSearchDaoSupport` class, allows to inject a `SearcherFactory` and an analyzer. You don't need anymore to create the corresponding injection methods.

In the same time, the class provides too the `getLuceneSearcherTemplate` method in order to have access to the search template of the support.

The following code describes the use of the `LuceneSearchDaoSupport` in a class implementing a search:

```
public class SampleSearchService extends LuceneSearchDaoSupport {

    public void searchDocuments() {
        List<String> results = getLuceneSearchTemplate().search("field:lucene", new HitExtractor() {
            public Object mapHit(int id, Document document, float score) {
                return document.get("field");
            }
        });
    }
}
```

You can note that the class makes possible to directly inject a configured `LuceneSearcherTemplate` in Spring.

9.5. Advanced searches

TODO

9.5.1. Support of queries

TODO: Queries as string

TODO: Programmatically building of queries

9.5.2. Filter and sort

TODO:

9.5.3. Pagination

TODO: first results

TODO: pagination with PagingQueryResultCreator

9.6. Object approach

TODO

Chapter 10. Configuration

The Lucene support provides Spring 2 namespaces in order to make easier the configuration of differents entities.

10.1. Lucene

In order to the namespace, you need to configure as a standard XML namespace on the `beans` tag in the Spring configuration. This way to do is the common one used by all the standard Spring 2 namespaces.

The identifier of the Lucene namespace is `http://www.springframework.org/schema/lucene` and the related xsd file `http://www.springframework.org/schema/lucene/lucene-index.xsd`. The following code describes the configuration of this namespace:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:lucene="http://www.springframework.org/schema/lucene"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
                            http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
                http://www.springframework.org/schema/lucene
                            http://www.springframework.org/schema/lucene/lucene-index.xsd">
    (...)
</beans>
```

Now, you can use the namespace in order to configure easily the Lucene resources of the support.

10.1.1. Resource configuration

The central of this namespace is the `index` tag which allows to configure the type of directory and the resources to interact with it to both index and search documents. The following tags can be used inside it:

- `ramDirectory` or `fsDirectory` in order to specify the type of directory used;
- `indexFactory` in order to configure the index factory resource used to index documents
- `searcherFactory` in order to configure the searcher factory resource used to index documents

Note that it's not mandatory to use both `indexFactory` and `searcherFactory` tags at the same time. If you only need to make searchs, you can only specify a `searcherFactory` tag.

The following describes a simple use of the `index` tag. You can note that a global Lucene analyzer is specified to the tag. This analyzer is used by both the index factory and searcher factory.

```
<bean id="analyzer" class="org.apache.lucene.analysis.SimpleAnalyzer"/>

<lucene:index id="myDirectory" analyzer-ref="analyzer">
    <lucene:fsDirectory location="/org/springframework/lucene/config"/>
    <lucene:indexFactory concurrent="lock" />
    <lucene:searcherFactory/>
</lucene:index>
```

Every tags used within the `index` tag can specify their own `id` attribute if necessary. In this case, they are used for the corresponding created beans. However, and you can see upper, you can specify an `id` on the `index` tag. This attribute is used to determine a default value for the identifiers of the inner tags if an `id` attribute is not specified. Here are the strategies of creating:

- For the `*Directory` tags, the value of the attribute is directly used;
- For the `indexFactory` tag, the value of the attribute is suffixed by `-indexFactory`;
- For the `searcherFactory` tag, the value of the attribute is suffixed by `-searcherFactory`;

When using the `fsDirectory` tag in order to configure a filesystem directory, you need to use the `location` attribute to specify its location. The Spring IO abstraction can be used in the attribute to locate the index in the classpath or in the filesystem for instance. If an in-memory index is configured with the `ramDirectory` tag, there is no need to use any attribute.

For both the `indexFactory` and `searcherFactory` tags, the `type` attribut can be used in order to specify the type of the corresponding implementation to used. By default, the value is `simple`.

The following table describes all the possible values of the `type` attribute for the both the later tags:

Table 10.1.

Tag	Value	Description
`indexFactory`	`simple`	Configure the `SimpleIndexFactory` implementation for the index factory. In this case, an index reader or writer is created for each method of the template unless you use the transactional support where the scope is tied to the transaction.
`searcherFactory`	`simple`	Configure the `SimpleSearcherFactory` implementation for the searcher factory. In this case, a index searcher is created for each method of the template.
`searcherFactory`	`single`	Configure the `SingleSearcherFactory` implementation for the searcher factory. In this case, a single index searcher is created and used for each method of the template. This searcher is destroyed when the application shutdown.

For the index factory resource, a `concurrent` attribut can be used in order to enable concurrent access to Lucene reader and writer. The only possible value is `lock` in order to configure a lock based approach.

10.1.2. Analyzer configuration

Both the index factory and searcher factory need a Lucene analyzer for their configuration. Like the `id` attribute, an analyzer can be configured in the global way according to the `analyzer-ref` attribute of the `index` tag. In this case, the configured analyzer is used by both index and searcher factories, as shown in the following code:

```
<bean id="analyzer" class="org.apache.lucene.analysis.SimpleAnalyzer"/>

<lucene:index id="myDirectory" analyzer-ref="analyzer">
    <lucene:fsDirectory location="/org/springframework/lucene/config"/>
    <lucene:indexFactory/>
    <lucene:searcherFactory/>
</lucene:index>
```

If you want to configure the analyzer bean as an inner bean, the `analyzer` inner tag can use as described in the following code:

```
<lucene:index id="myDirectory">
    <lucene:analyzer>
        <bean class="org.apache.lucene.analysis.SimpleAnalyzer"/>
    </lucene:analyzer>
    <lucene:fsDirectory location="/org/springframework/lucene/config"/>
    <lucene:indexFactory/>
    <lucene:searcherFactory/>
</lucene:index>
```

The namespace provides too the possibility to configure different analyzers for the index and search factories by using the `analyzer-ref` attribute of the `indexFactory` and `searcherFactory` tags. The following code describes the aspect:

```
<bean id="analyzer1" class="org.apache.lucene.analysis.SimpleAnalyzer"/>
<bean id="analyzer2" class="org.apache.lucene.analysis.SimpleAnalyzer"/>

<lucene:index id="">
    <lucene:fsDirectory location="/org/springframework/lucene/config"/>
    <lucene:indexFactory id="" concurrent="lock"  analyzer-ref="analyzer1"/>
    <lucene:searcherFactory id=""  analyzer-ref="analyzer2"/>
</lucene:index>
```

Like the global approach, both `indexFactory` and `searcherFactory` tags can configure an analyzer as an inner bean using the `analyzer` inner tag, as shown below:

```
<bean id="analyzer" class="org.apache.lucene.analysis.SimpleAnalyzer"/>

<lucene:index id="">
    <lucene:fsDirectory location="/org/springframework/lucene/config"/>
    <lucene:indexFactory id="" concurrent="lock" analyzer-ref="analyzer"/>
    <lucene:searcherFactory id="">
        <lucene:analyzer>
            <bean class="org.apache.lucene.analysis.SimpleAnalyzer"/>
        </lucene:analyzer>
    </lucene:searcherFactory>
</lucene:index>
```

Part III. Solr Support Reference Documentation

Part IV. Other Resources

In addition to this reference documentation, there are a number of other resources that may help you learn how to use OSGi and Spring Dynamic Modules. These additional, third-party resources are enumerated in this section.

Part V. Appendixes