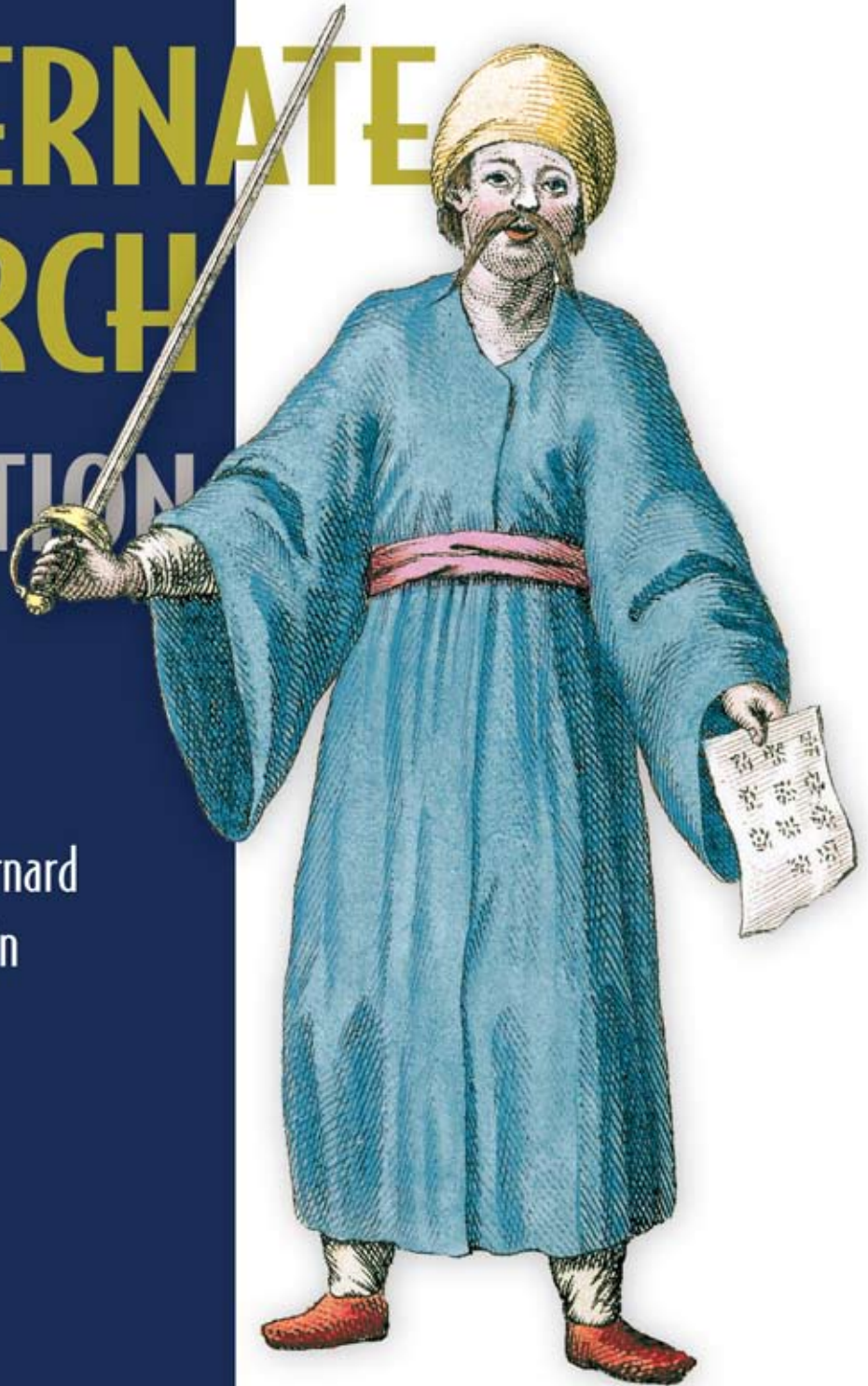


Sample Chapter

# HIBERNATE SEARCH IN ACTION

Emmanuel Bernard  
John Griffin





***Hibernate Search in Action***

by Emmanuel Bernard  
and John Griffin

**Chapter 2**

Copyright 2009 Manning Publications

# *brief contents*

---

<b>PART 1</b>	<b>UNDERSTANDING SEARCH TECHNOLOGY .....</b>	<b>1</b>
	1 ■ State of the art	3
	2 ■ Getting started with Hibernate Search	28
<b>PART 2</b>	<b>ENDING STRUCTURAL AND SYNCHRONIZATION MISMATCHES</b>	<b>61</b>
	3 ■ Mapping simple data structures	63
	4 ■ Mapping more advanced data structures	88
	5 ■ Indexing: where, how, what, and when	115
<b>PART 3</b>	<b>TAMING THE RETRIEVAL MISMATCH .....</b>	<b>159</b>
	6 ■ Querying with Hibernate Search	161
	7 ■ Writing a Lucene query	201
	8 ■ Filters: cross-cutting restrictions	251
<b>PART 4</b>	<b>PERFORMANCE AND SCALABILITY .....</b>	<b>273</b>
	9 ■ Performance considerations	275
	10 ■ Scalability: using Hibernate Search in a cluster	310
	11 ■ Accessing Lucene natively	327

## BRIEF CONTENTS

<b>PART 5</b>	<b>NATIVE LUCENE, SCORING, AND THE WHEEL.....</b>	<b>351</b>
12	■ Document ranking	353
13	■ Don't reinvent the wheel	399
appendix	Quick reference	

# Getting started with *Hibernate Search*

---

## ***This chapter covers***

- What is Hibernate Search?
- How to set up and configure Hibernate Search
- An introduction to mapping your domain model
- An introduction to indexing your data
- An introduction to doing full-text queries
- How to use Luke

In the chapter 1, we discussed difficulties of integrating a full-text search engine such as Apache Lucene into a Java application centered on a domain model and using Hibernate or Java Persistence to persist data. More specifically, we saw three mismatches:

- *Structural mismatch*—How to convert the object domain into the text-only index; how to deal with relations between objects in the index.

- *Synchronization mismatch*—How to keep the database and the index synchronized all the time.
- *Retrieval mismatch*—How to get a seamless integration between the domain model-centric data-retrieval methods and full-text search.

Hibernate Search leverages the Hibernate ORM and Apache Lucene (full-text search engine) technologies to address these mismatches. This chapter will give you an overview of Hibernate Search: how to use it, how to express full-text queries, and how it fits into the Hibernate programmatic model.

Hibernate Search is a project that complements Hibernate Core by providing the ability to do full-text search queries on persistent domain models. Hibernate Core is probably the most famous and most used ORM tool in the Java industry. An ORM lets you express your domain model in a pure object-oriented paradigm, and it persists this model to a relational database transparently for you. Hibernate Core lets you express queries in an object-oriented way through the use of its own portable SQL extension (HQL), an object-oriented criteria API, or a plain native SQL query. Typically, ORMs such as Hibernate Core apply optimization techniques that an SQL hand-coded solution would not: transactional write behind, batch processing, and first- and second-level caching. Hibernate Core is released under an open source license and can be found at <http://hibernate.org>.

Hibernate Search's full-text technology entirely depends on Apache Lucene. Lucene is a powerful full-text search engine library hosted at the Apache Software Foundation (<http://lucene.apache.org/java>). It has rapidly become the de facto standard for implementing full-text search solutions in Java. This success comes from several factors:

- It is free and open source.
- It has low-level and very powerful APIs.
- It is agnostic as to the kind of data indexed and searched.
- It has a good record of performance and maturity.
- It has a vibrant community.

All these qualities make Lucene the perfect information-retrieval library for building search solutions. These reasons are why Hibernate Search is built on top of Lucene.

Hibernate Search, which is also released under an open source license, is a bridge that brings Lucene features to the Hibernate world. Hibernate Search hides the low-level and sometimes complex Lucene API usage, applies the necessary options under the hood, and lets you index and retrieve the Hibernate persistent domain model with minimal work. This chapter should give you a good understanding of how Hibernate Search fits into the Hibernate programmatic model and describe how to quickly start and try Hibernate Search.

To demonstrate this integration, we'll start by writing a DVD store application. We won't write the whole application but rather focus on the domain model and the core engine, specifically the search engine.

Our object model will be quite simple and contain an `Item` entity. The `Item` entity represents a DVD. We want to let our users search by some of the `Item` properties. In this chapter, we'll show how to set up Hibernate Search, describe the metadata to make `Item` a full-text searchable entity, index the items stored in the database, and query the system to retrieve the matching DVDs.

## 2.1 **Requirements: what Hibernate Search needs**

Hibernate Search has been developed with Java 5 and needs to run on the Java Development Kit (JDK) or Java Runtime Environment (JRE) version 5 or above. Aside from this limitation, Hibernate Search runs everywhere Hibernate Core runs, especially in the architecture and environment of your choice. While it's next to impossible to list all the possible environments Hibernate and Hibernate Search run on, we can list a few typical ones:

- Full-featured applications (web based or not) deployed on a Java EE application server
- Simpler web-based applications on a servlet container
- Web-based applications using JBoss Seam
- Swing applications
- So-called lightweight dependency injection frameworks such as Spring Framework, Guice, or Web Beans
- Applications built on Java SE
- Frameworks or platforms that use Hibernate, such as Grails

Hibernate Search integrates well into the Hibernate platform. More specifically, you can use any of the following mapping strategies and APIs while using Hibernate Search:

- Hibernate Core APIs and `hbm.xml` files
- Hibernate Core APIs and Hibernate Annotations
- Hibernate `EntityManager` APIs and `hbm.xml` files
- Hibernate `EntityManager` APIs and Hibernate Annotations

In other words, Hibernate Search is agnostic to your choice of mapping metadata (XML or annotations) and integrates with both Hibernate native APIs and Java Persistence APIs.

While Hibernate Search has few restrictions, this chapter has some. The authors expect the reader to understand the basics of Hibernate. The reader must be familiar with the object-manipulation APIs from the Hibernate `Session` or the Java Persistence `EntityManager` as well as the query APIs. She also must be familiar with association mappings and the concept of bidirectional relationships. These requirements are nothing unusual for someone having a few months of experience with Hibernate.

In this book, most examples will use Hibernate Annotations as the mapping metadata. Annotations have some advantages over an XML deployment descriptor:

Metadata is much more compact, and mixing the class structure and the metadata greatly enhances behavior readability. Besides, modern platforms, including the Java platform, are moving away from XML as the preferred choice for code-centric metadata descriptors, which is reason enough for the authors to leave XML alone. Remember, while Hibernate Search uses annotations for its metadata, it works perfectly with hbm.xml-based domain models, and it should be simple to port the examples.

## 2.2 Setting up Hibernate Search

Configuring Hibernate Search is fairly easy because it integrates with the Hibernate Core configuration lifecycle. That being said, we'll go through the steps of adding Hibernate Search in a Hibernate-based application. We'll add the libraries to the classpath and add the configuration properties. But first you need to download Hibernate Search at <http://www.hibernate.org> or use the JBoss Maven repository (<http://repository.jboss.org/maven2/org/hibernate/hibernate-search>). It's useful to download the Apache Lucene distribution as well, which is available at <http://lucene.apache.org/java/>. It contains both documentation and a contribution section containing add-ons that aren't bundled with Hibernate Search. Make sure you use the same Lucene version that Hibernate Search is based on. You can find the correct version in the Hibernate Search distribution in `lib/readme.txt`.

### 2.2.1 Adding libraries to the classpath

Add Hibernate Search's necessary JARs (Java Archives) into your classpath. Hibernate Search requires three JARs:

- *hibernate-search.jar*—The core API and engine of Hibernate Search
- *lucene-core.jar*—Apache Lucene engine
- *hibernate-commons-annotations.jar*—Some common utilities for the Hibernate project

All three JARs are available in the Hibernate Search distribution, and pulling them from there is the safest way to have a compatible trio. Thus far Hibernate Search has been staying as close as possible to the latest Lucene version to benefit from bug fixes, performance improvements, and new features of the Lucene community.

You can also add the optional support for modular analyzers by adding the following JARs to your classpath:

- *solr-common.jar*
- *solr-core.jar*
- *lucene-snowball.jar*

These JARs (available in the Hibernate Search distribution) are a subset of the Solr distribution and contain analyzers. While optional, we recommend adding these JARs to your classpath because it greatly simplifies the use of analyzers. This feature is available beginning with Hibernate Search 3.1.



**NOTE** You can put the full Solr distribution instead of the version provided by Hibernate Search in your classpath if you wish to.

Hibernate Search is not compatible with all versions of Hibernate Core and Hibernate Annotations. It's best to refer to the compatibility matrix available on the [hibernate.org](http://hibernate.org) download page. At the time this book was written, the compatibility matrix tells us that:

- Hibernate Search 3.0.x is compatible with Hibernate Core 3.2.x starting from 3.2.2, Hibernate Annotations 3.3.x, and Hibernate EntityManager 3.3.x.
- Hibernate Search 3.1.x is compatible with Hibernate Core 3.3.x, Hibernate Annotations 3.4.x, and Hibernate EntityManager 3.4.x.

**NOTE** You can find dependencies that Hibernate Search has been built on and initially tested on in the Hibernate Search distribution or in the Maven dependency file (POM). Hibernate Search is published to the JBoss Maven repository (<http://repository.jboss.org/maven2/org/hibernate/hibernate-search>).

If you use Hibernate Annotations, `hibernate-commons-annotations.jar` is already present in your classpath.

Adding a JAR to your classpath depends on your deployment environment. It's virtually impossible to describe all likely deployments, but we'll go through a few of them.

In an SE environment, the JAR list is provided to the virtual machine thanks to a command-line argument:

```
# on Windows platforms
java -classpath hibernate-search.jar;lucene-core.jar
    ;hibernate-commons-annotations.jar;solr-core.jar ... my.StartupClass

# on Unix, Linux and Mac OS X platforms
java -classpath hibernate-search.jar:lucene-core.jar:
    hibernate-commons-annotations.jar:solr-core.jar ... my.StartupClass
```

If you happen to deploy your Hibernate application in a WAR (Web Archive) either deployed in a naked servlet container or a full-fledged Java EE application server, things are a bit simpler; you just need to add the necessary JARs into the `lib` directory of your WAR.

```
<WAR ROOT>
WEB-INF
  classes
    [contains your application classes]
  lib
    hibernate-search.jar
    lucene-core.jar
    hibernate-commons-annotations.jar
    solr-core.jar
```

```

solr-common.jar
lucene-snowball.jar
[contains other third party libraries]
...

```

You could also put Hibernate Search-required JARs as a common library in your servlet container or application server. The authors don't recommend such a strategy because it forces all deployed applications to use the same Hibernate Search version. Some support or operation teams tend to dislike such a strategy, and they'll let you know it.

If you deploy your application in an EAR (Enterprise Archive) in a Java EE application server, one of the strategies is to put the third-party libraries in the EAR's lib directory (or in the library-directory value in META-INF/application.xml if you happen to override it).

```

<EAR_ROOT>
  myejbjar1.jar
  mywar.war
  META-INF
  ...
  lib
    hibernate-search.jar
    lucene-core.jar
    hibernate-commons-annotations.jar
    solr-core.jar
    solr-common.jar
    lucene-snowball.jar
    [contains other third party libraries]
  ...

```

Unfortunately, this solution works only for Java EE 5 application servers and above. If you're stuck with a J2EE application server, you'll need to add a Class-Path entry in each META-INF/MANIFEST.MF file of any component that depends on Hibernate Search. Listing 2.1 and listing 2.2 describe how to do it.

#### Listing 2.1 MANIFEST.MF declaring a dependency on Hibernate Search

```

Manifest-Version: 1.0
Class-Path: lib/hibernate-search.jar lib/lucene-core.jar
           ➡ lib/hibernate-commons-annotations.jar lib/solr-core.jar ...

```

#### Listing 2.2 Structure of the EAR containing Hibernate Search

```

<EAR_ROOT>
  myejbjar1.jar
    META-INF/MANIFEST.MF (declaring the dependency on Hibernate Search)
  mywar.war
  META-INF
  ...
  lib
    hibernate-search.jar
    lucene-core.jar

```

```
hibernate-commons-annotations.jar
solr-core.jar
solr-common.jar
lucene-snowball.jar
[contains other third party libraries]
...
```

The Class-Path entry is a space-separated list of JARs or directory URLs relative to where the referencing archive is (in our example, EAR root).

Believe it or not, you just did the hardest part of the configuration! The next step is to tell Hibernate Search where to put the Lucene index structure.

## 2.2.2 *Providing configuration*

Once Hibernate Search is properly set up in your classpath, the next step is to indicate where the Apache Lucene indexes will be stored. You will place your Hibernate Search configuration in the same location where you placed your Hibernate Core configuration. Fortunately, you do not need another configuration file.

When you use Hibernate Core (possibly with Hibernate Annotations), you can provide the configuration parameters in three ways:

- In a `hibernate.cfg.xml` file
- In the `/hibernate.properties` file
- Through the configuration API and specifically `configuration.setProperty(String, String)`

The first solution is the most commonly used. Hibernate Search properties are regular Hibernate properties and fit in these solutions. When you use `Hibernate EntityManager`, the standard way to provide configuration parameters is to use the `META-INF/persistence.xml` file. Injecting Hibernate Search properties into this file is also supported. This is good news for us, in that there's no need to think about yet another configuration file to package!

What kind of configuration parameters does Hibernate Search need? Not a lot by default. Hibernate Search has been designed with the idea of configuration by exception in mind. This design concept uses the 80 percent-20 percent rule by letting the 80 percent scenarios be the default configuration. Of course, it's always possible to override the default in case we fall into the 20 percent scenarios. The configuration-by-exception principle will be more visible and more useful when we start talking about mapping. Let's look at a concrete example. When using Hibernate Search, you need to tell the library where to find Apache Lucene indexes. By default, Hibernate Search assumes you want to store your indexes in a file directory; this is a good assumption because it provides a good trade-off between performance and index size. However, you'll probably want to define the actual directory where the indexes will be stored. The property name is `hibernate.search.default.indexBase`, so depending on the configuration strategy used, the configuration will be updated as shown in listing 2.3.

**Listing 2.3** Hibernate Search configuration

```
#hibernate.properties ← hibernate.properties file
#regular Hibernate Core configuration ← Define your Hibernate Core properties
hibernate.dialect org.hibernate.dialect.PostgreSQLDialect
hibernate.connection.datasource jdbc/test
#Hibernate Search configuration ← Define Hibernate Search-specific properties
hibernate.search.default.indexBase /users/application/indexes

<?xml version="1.0" encoding="UTF-8"?> ← hibernate.cfg.xml file
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<!-- hibernate.cfg.xml -->
<hibernate-configuration>
    <session-factory name="dvdstore-catalog">
        <!-- regular Hibernate Core configuration -->
        <property name="hibernate.dialect">
            org.hibernate.dialect.PostgreSQLDialect
        </property>
        <property name="hibernate.connection.datasource">
            jdbc/test
        </property>
        <!-- Hibernate Search configuration --> ← Hibernate Search properties
        <property name="hibernate.search.default.indexBase">
            /users/application/indexes
        </property>
        <!-- mapping classes --> ← List your entities
        <mapping class="com.manning.dvdstore.model.Item"/>
    </session-factory>
</hibernate-configuration>

<?xml version="1.0" encoding="UTF-8"?> ← META-INF/persistence.xml
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
    version="1.0">

    <!-- example of a default persistence.xml -->
    <persistence-unit name="dvdstore-catalog">
        <jta-data-source>jdbc/test</jta-data-source>

        <properties>
            <!-- regular Hibernate Core configuration -->
            <property name="hibernate.dialect"
                value="org.hibernate.dialect.PostgreSQLDialect"/>

            <!-- Hibernate Search configuration -->
```

```

        <property name="hibernate.search.default.indexBase"
            value="/users/application/indexes"/>
    </properties>

    </persistence-unit>
</persistence>

```

**Hibernate Search  
properties** ←

This is the last time you'll see the XML headers (doctype and schema) in this book. They should always be there, but for conciseness we'll drop them in future examples.

This is the only configuration property we need to set to get started with Hibernate Search. Even this property is defaulted to `./`, which is the JVM current directory, but the authors think it's more appropriate to explicitly define the target directory.

Another property can be quite useful, especially in test environments: the Lucene directory provider. Hibernate Search stores your indexes in a file directory by default. But it can be quite convenient to store indexes only in memory when doing unit tests, especially if, like the authors, you prefer to use in-memory databases like HSQLDB, H2, or Derby to run your test suite. It makes the tests run faster and limits side effects between tests. We'll discuss this approach in section 5.1.3 and section 9.5.2.

**NOTE** **IN-MEMORY INDEX AND UNIT TESTING** We'd like to warn you of a classic error we're sure you'll be bitten by that can cost you a few hours until you figure it out. When you run a test on your index, make sure it is on par with the database you're testing on. Classically, unit tests clear the database and add a fresh set of data. Every so often you'll forget to update or clear your file system's Lucene directory. Your results will look confusing, returning duplicate or stale data. One elegant way to avoid that is to use in-memory directories; they're created and destroyed for every test, practically isolating them from one another.

As you can see, configuring Hibernate Search is very simple, and the required parameters are minimal. Well, it's not entirely true—we lied to you. If your system uses Hibernate Annotations 3.3.x and beyond, these are truly the only parameters required. But if your system uses Hibernate Core only, a few additional properties are required.

**NOTE** **HOW DO I KNOW WHETHER TO USE HIBERNATE ANNOTATIONS OR SIMPLY HIBERNATE CORE?** There are three very simple rules:

- If your domain model uses Hibernate Annotations or Java Persistence annotations, you're using Hibernate Annotations.
- If your application uses the Hibernate `EntityManager` API (the Java Persistence API really), you're also using Hibernate Annotations under the cover.
- If you're still unsure, check whether you create a `Configuration` object or an `AnnotationConfiguration` object. In the former case, you're using Hibernate Core. In the latter case, you're using Hibernate Annotations.

Why is that? Hibernate Annotations detects Hibernate Search and is able to autowire Hibernate event listeners for you. Unfortunately this is not (yet) the case for Hibernate Core. If you're using only Hibernate Core, you need to add the event listener configuration, as shown in listing 2.4.

**Listing 2.4 Enable event listeners if you don't use Hibernate Annotations**

```
<hibernate-configuration>
  <session-factory>
    ...
    <event type="post-update">
      <listener
class="org.hibernate.search.event.FullTextIndexEventListener"/>
      </event>
    <event type="post-insert">
      <listener
class="org.hibernate.search.event.FullTextIndexEventListener"/>
      </event>
    <event type="post-delete">
      <listener
class="org.hibernate.search.event.FullTextIndexEventListener"/>
      </event>
    <event type="post-collection-recreate">
      <listener
class="org.hibernate.search.event.FullTextIndexEventListener"/>
      </event>
    <event type="post-collection-remove">
      <listener
class="org.hibernate.search.event.FullTextIndexEventListener"/>
      </event>
    <event type="post-collection-update">
      <listener
class="org.hibernate.search.event.FullTextIndexEventListener"/>
      </event>
    </session-factory>
  </hibernate-configuration>
```

Now each time Hibernate Core inserts, updates, or deletes an entity, Hibernate Search will know about it.

**NOTE** If you use Hibernate Search 3.0.x, you need a slightly different configuration. Listing 2.5 describes it.

**Listing 2.5 Enable event listeners for Search 3.0 if you don't use Annotations.**

```
<hibernate-configuration>
  <session-factory>
    ...
    <event type="post-update">
      <listener
class="org.hibernate.search.event.FullTextIndexEventListener"/>
      </event>
    <event type="post-insert">
```

```

        <listener
class="org.hibernate.search.event.FullTextIndexEventListener" />
    </event>
    <event type="post-delete">
        <listener
class="org.hibernate.search.event.FullTextIndexEventListener" />
    </event>
    <event type="post-collection-recreate"> ← Collection event listener differs
        <listener
class="org.hibernate.search.event.FullTextIndexCollectionEventListener" />
    </event>
    <event type="post-collection-remove">
        <listener
class="org.hibernate.search.event.FullTextIndexCollectionEventListener" />
    </event>
    <event type="post-collection-update">
        <listener
class="org.hibernate.search.event.FullTextIndexCollectionEventListener" />
    </event>
    </session-factory>
</hibernate-configuration>

```

This event listener configuration looks pretty scary, but remember: You don't even need to think about it if you use Hibernate Annotations or Hibernate `EntityManager` 3.3.x or above, a good reason to move to these projects!

We'll discuss additional Hibernate Search parameters when the need arises, but what you know right now is more than enough to get started and suits a great many production systems.

## 2.3 *Mapping the domain model*

Now that Hibernate Search is configured properly, we need to decide which entity and which property will be usable in our full-text searches. Indexing every single entity and every single property doesn't make much sense. Putting aside that such a strategy would waste CPU, index size, and performance, it doesn't make a lot of business sense to be able to search a DVD by its image URL name. Mapping metadata will help define what to index and how: It will describe the conversion between our object-oriented domain object and the string-only flat world of Lucene indexes.

Hibernate Search expresses this mapping metadata through annotations. The choice of annotations was quite natural to the Hibernate Search designers because the metadata is closely related to the Java class structure. Configuration by exception is used extensively to limit the amount of metadata an application developer has to define and maintain.

### 2.3.1 *Indexing an entity*

Let's go practical now. What's needed to make a standard entity (a mapped plain old Java object [POJO] really) full-text searchable? Let's have a look at listing 2.6.

**Listing 2.6 Mapping a persistent POJO**

```

package com.manning.hsia.dvdstore.model;

@Entity
@Indexed ← Mark for indexing
public class Item {

    @Id @GeneratedValue
    @DocumentId ← Mark id property shared by Core and Search
    private Integer id;

    @Field ← Mark for indexing using tokenization
    private String title;

    @Field
    private String description;

    @Field(index=Index.UN_TOKENIZED, store=Store.YES) ← Mark for indexing without tokenization
    private String ean;

    private String imageURL; ← This property is not indexed (default)
    //public getters and setters
}

```

The first thing to do is to place an `@Indexed` annotation on the entity that will be searchable through Hibernate Search. In the previous section, you might have noticed that nowhere did we provide a list of indexed entities. Indeed, Hibernate Search gathers the list of indexed entities from the list of persistence entities marked with the `@Indexed` annotation, saving you the work of doing it manually. The index for the `Item` entity will be stored in a directory named `com.manning.hsia.dvdstore.model.Item` in the `indexBase` directory we configured previously. By default, the index name for a given entity is the fully qualified class name of the entity.

The second (and last) mandatory thing to do is to add a `@DocumentId` on the entity's identity property. Hibernate Search uses this property to make the link between a database entry and an index entry. Hibernate Search will then know which entry (*document* in the Lucene jargon) to update in the index when an item object is changed. Likewise, when reading results from the index, Hibernate Search will know to which object (or database row) it relates. That's it for the necessary steps: Add `@Indexed` on the entity, and add `@DocumentId` on the identifier property. But of course, as it is, it wouldn't be really useful since none of the interesting properties are indexed.

### 2.3.2 Indexing properties

To index a property, we need to use an `@Field` annotation. This annotation tells Hibernate Search that the property needs to be indexed in the Lucene document. Each property is indexed in a field that's named after the property name. In our example, the `title`, `description`, and `ean` properties are indexed by Lucene in, respectively, the `title`, `description`, and `ean` fields. While it's possible to change the



default Lucene field name of a property, it's considered a bad practice and will make querying more unnatural, as you'll see in the query section of this chapter. `imageUrl`, which is not marked by `@Field`, won't be indexed in the Lucene document even if Hibernate stores it in the database.

**NOTE** An object instance mapped by Hibernate roughly corresponds to a table row in the database. An object property is roughly materialized to a table column. To make the same analogy in the Lucene index, an object instance mapped by Hibernate roughly corresponds to a Lucene document, and an object property is roughly materialized to a Lucene field in the document. Now take this analogy with a grain of salt because this one-to-one correspondence isn't always verified. We'll come to these more exotic cases later in this book.

The `ean` property is indexed slightly differently than the others. While we still use `@Field` to map it, two new attributes have been defined. The first one, `index`, specifies how the property value should be indexed. While we have decided to chunk title and description into individual words to be able to search these fields by word (this process is called *tokenization*), the `ean` property should be treated differently. EAN, which stands for European Article Number, is the article bar code that you can see on just about any product sold nowadays. EAN is a superset of the UPC (Universal Product Code) used in North America. It would be fairly bad for the indexing process to tokenize a unique identifier because it would be impossible to search by it. That's why the index attribute is set to `Index.UN_TOKENIZED`; the EAN value won't be chunked during the indexing process.

The second particularity of the `ean` property is that its value will be stored in the Lucene index. By default, Hibernate Search doesn't store values in the index because they're not needed in most cases. As a result, the Lucene index is smaller and faster. In some situations, though, you want to store some properties in the Lucene index, either because the index is read outside of Hibernate Search or because you want to execute a particular type of query—projection—that we'll talk about later in the book. By adding the `store` attribute to `Store.YES` in the `@Field` annotation, you ask Hibernate Search to store the property value in the Lucene index.

The example shows annotations placed on fields. This isn't mandatory; you can place annotations on getters as well. If the annotation is on the getter, Hibernate Search will access the property value through the getter method. Indeed, this is the authors' preferred access strategy. To keep the example as short and readable as possible, this book will show annotations only on fields.

**NOTE** SHOULD I USE GETTER OR FIELD ACCESS? There's no performance impact in using one or the other, nor is there any advantage with regard to Hibernate Search. Choosing is more a matter of architectural taste. The authors tend to prefer getter access because it allows an abstraction over the object state. Also, the Java Persistence specification requires accessing data through getters for maximum portability. In any case, consistency is

the rule you should follow. Try to use the same access strategy for both Hibernate Core and Hibernate Search, because it will save you from some unwanted surprises.

We'll now show how to use Hibernate Search on an existing XML-based mapping structure (hbm.xml files).

### 2.3.3 What if I don't use Hibernate Annotations?

The previous example shows the use of Hibernate Search in conjunction with Hibernate Annotations, but the same example would work perfectly with hbm.xml files as well. This is particularly useful if you try to use Hibernate Search on an existing Hibernate Core-based application where the mapping is defined in XML. Have a look at listing 2.7.

**Listing 2.7 Mapping a persistent POJO using an hbm.xml file**

```
package com.manning.hsia.dvdstore.model;

@Indexed
public class Item {
    @DocumentId
    private Integer id;

    @Field
    private String title;

    @Field
    private String description;

    @Field(index=Index.UN_TOKENIZED, store=Store.YES)
    private String ean;

    private String imageURL;
    //public getters and setters
}

<hibernate-mapping package="com.manning.hsia.dvdstore.model">
  <class name="Item">
    <id name="id">
      <generator class="native"/>
    </id>
    <property name="title"/>
    <property name="description"/>
    <property name="ean"/>
    <property name="imageURL"/>
  </class>
</hibernate-mapping>
```

**No Java Persistence annotations are present**

**Mapping externalized in hbm.xml files**

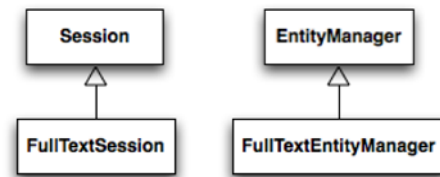
It's currently not possible to express the Hibernate Search metadata using an XML descriptor, but it might be added to a future version of the product, depending on user demand.

## 2.4 Indexing your data

We've just shown how the object model will be mapped into the index model, but we haven't addressed when the object model is indexed. Hibernate Search listens to Hibernate Core operations. Every time an entity marked for indexing is persisted, updated, or deleted, Hibernate Search is notified. In other words, every time you persist your domain model to the database, Hibernate Search knows it and can apply the same changes to the index. The index stays synchronized with the database state automatically and transparently for the application. That's good news for us because we don't have anything special to do!

What about existing data? Data already in the database may never be updated, and so Hibernate Search will then never be able to receive a notification from Hibernate Core. Because in most scenarios the index needs to be initially populated with existing and legacy data, Hibernate Search proposes a manual indexing API.

This is our first look at the Hibernate Search API. Hibernate Search extends the Hibernate Core main API to provide access to some of the full-text capabilities. A `FullTextSession` is a subinterface of `Session`. Similarly, a `FullTextEntityManager` is a subinterface of `EntityManager` (see figure 2.1). Those two subinterfaces contain the same methods and especially the one interesting us at the moment: the ability to manually index an object.



**Figure 2.1** `FullTextSession` and `FullTextEntityManager` extend `Session` and `EntityManager`, respectively.

Where can we get an instance of these interfaces? Internally, the `FullTextEntityManager` and `FullTextSession` implementations are wrappers around an `EntityManager` implementation or a `Session` implementation. Hibernate Search provides a helper class (`org.hibernate.search.jpa.Search`) to retrieve a `FullTextEntityManager` from a Hibernate `EntityManager` as well as a helper class to retrieve a `FullTextSession` from a `Session` (`org.hibernate.search.Search`). Listing 2.8 shows how to use these helper classes.

### Listing 2.8 Retrieving a `FullTextSession` or a `FullTextEntityManager`

```

Session session = ...;
FullTextSession fts =
    org.hibernate.search.Search.getFullTextSession(session);

EntityManager em = ...;
FullTextEntityManager ftem =
    org.hibernate.search.jpa.Search.getFullTextEntityManager(em);
  
```

← Wrap a Session object

← Wrap an EntityManager object

**NOTE** `getFullTextSession` and `getFullTextEntityManager` were named `createFullTextSession` and `createFullTextEntityManager` in Hibernate Search 3.0.

The two full-text APIs have a method named `index` whose responsibility is to index or reindex an already persistent object. Let's see in listing 2.9 how we would index all the existing items.

### Listing 2.9 Manually indexing object instances

```
FullTextEntityManager ftem = Search.getFullTextEntityManager(em);

ftem.getTransaction().begin();

@SuppressWarnings("unchecked")
List<Item> items = em.createQuery("select i from Item i").getResultList();

for (Item item : items) {
    ftem.index(item);
}

ftem.getTransaction().commit();
```

Manually index an item instance

Index is written at commit time

In this piece of code, `items` is the list of `Item` objects to index. You'll discover in section 5.4.2 a more efficient solution to massively indexing data, but this one will be good enough for now. The `index` method takes an `item` instance and indexes it. The Lucene index will thus contain the necessary information to execute full-text queries matching these items. The initial dataset indexed, subsequent changes, and whether it is item creation, item update, or item deletion will be taken care of by the Hibernate event system. The index and the database stay synchronized.

We now have an up-to-date index ready to be queried, which leads to the next question: How do I query data using Hibernate Search?

## 2.5 Querying your data

Hibernate Search tries to achieve two somewhat contradictory goals:

- Provide a seamless integration with the Hibernate Core API and programmatic model
- Give the full power and flexibility of Lucene, the underlying full-text engine

To achieve the first goal, Hibernate Search's query facility integrates into the Hibernate query API (or the Java Persistence query API if you use the `EntityManager`). If you know Hibernate Core, the query-manipulation APIs will look very familiar to you; they're the same! The second key point is that Hibernate Search returns Hibernate managed objects out of the persistence context; in more concrete terms it means that the objects retrieved from a full-text query are the same object instances you would have retrieved from an HQL query (had HQL the same full-text capabilities). In particular, you can update those objects, and Hibernate will synchronize any changes to the

database. Your objects also benefit from lazy loading association and transparent fetching with no additional work on the application programmer's side.

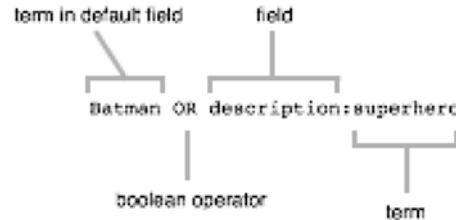
**NOTE** WHAT IS A PERSISTENCE CONTEXT? While the Hibernate Session is the API that lets you manipulate the object's state and query the database, the persistence context is the list of objects Hibernate is taking care of in the current session lifecycle. Every object loaded, persisted, or reattached by Hibernate will be placed into the persistence context and will be checked for any change at flush time. Why is the persistence context important? Because it's responsible for object unicity while you interact with the session. Persistence contexts guarantee that a given entry in the database is represented as one object and only one per session (that is, per persistence context). While usually misunderstood, this is a key behavior that saves the application programmer a lot of trouble.

To achieve the second goal, Hibernate Search doesn't try to encapsulate how Lucene expresses queries. You can write a plain Lucene query and pass it to Hibernate Search as it is.

Let's walk through the steps to create a query and retrieve the list of matching objects. For Lucene beginners, don't worry; no prerequisite knowledge of Lucene is necessary. We'll walk with you each and every step.

### 2.5.1 *Building the Lucene query*

The first thing we need to do is determine what query we're willing to execute. In our example, we want to retrieve all items matching a given set of words regardless of whether they are in the title properties or the description properties. The next step is to write the Lucene query associated with this request. We have a few ways to write a Lucene query. For starters, we'll use the simpler-to-understand query parser solution. Lucene comes bundled with an example of a query parser that takes a string as a parameter and builds the underlying Lucene query from it. The full description of the query syntax is available in any Lucene distribution at [docs/queryparsersyntax.html](http://docs/queryparsersyntax.html), but let's have a quick look at it. Figure 2.2 describes the components of a query.



**Figure 2.2** Query parser syntax

A query is composed of terms to look for (words) targeted in a given Lucene document field. The field name is followed by a colon (:) and the term to look for. To query more than one term, Boolean operators such as OR, AND, or NOT (they must be capitalized) can be used between terms. When building a query parser, a default field name is provided. If the term is not preceded by a field name, the default field name applies. When you need to apply some approximation searching to a word (maybe because you don't know the exact spelling), it needs to be followed by a tilde (~). For example:

```
title:hypernate~ OR description:persistence
```

To learn more about the Lucene query syntax, have a look at chapter 7, the Lucene documentation, or the excellent book *Lucene in Action* by Erik Hatcher and Otis Gospodnetić.

How does a field map back to our domain model mapping? Hibernate Search maps each indexed property into a field of the same name (unless you explicitly override the field name). This makes a query quite natural to read from an object-oriented point of view; the property `title` in our `Item` class can be queried by targeting the `title` field in a Lucene query. Now that we can express our queries, let's see how to build them (listing 2.10).

**NOTE** You might be afraid that the query syntax is not one your customer is willing or even able to use. The Lucene query parser is provided here to give you a quick start. Most public-faced applications define their own search syntax and build their Lucene queries programmatically. We'll explore this approach later in this book.

#### Listing 2.10 Building a Lucene query

```
String searchQuery = "title:Batman OR description:Batman";  <— Query string

QueryParser parser = new QueryParser(
    "title",
    new StandardAnalyzer()  <— Analyzer used
);

org.apache.lucene.search.Query luceneQuery;
try {
    luceneQuery = parser.parse(searchQuery);  <— Build Lucene query
}
catch (ParseException e) {
    throw new RuntimeException("Unable to parse query: " + searchQuery, e);
}
```

Diagram annotations for Listing 2.10:

- Query string**: Points to the `searchQuery` variable.
- Analyzer used**: Points to the `new StandardAnalyzer()` constructor.
- Default field**: Points to the `"title"` argument in the `QueryParser` constructor.
- Build a query parser**: Points to the `new QueryParser()` constructor.
- Build Lucene query**: Points to the `parser.parse(searchQuery)` call.

Once you've expressed the query as a string representation, building a Lucene query with the query parser is a two-step process. The first step is to build a query parser, define the default field targeted in the query, and define the analyzer used during the query building. The default field is used when the targeted fields are not explicit in the query string. It turns out that the authors don't use this feature very often. Next we'll present a more interesting solution. Analyzers are a primary component of Lucene and a key to its flexibility. An analyzer is responsible for breaking sentences into individual words. We'll skip this notion for now and come back to it in greater detail in section 5.2, when you will be more familiar with Hibernate Search and Lucene. The query parser is now ready and can generate Lucene queries out of any syntax-compliant query string. Note that the query hasn't yet been executed.

Lucene provides an improved query parser that allows you to target more than one field at a time automatically. Because Hibernate Search, by default, matches one property to one Lucene field, this query parser turns out to be very useful as a way to finely target which property to search by. Let's see how to use it (see listing 2.11).

**Listing 2.11 Using the MultiFieldQueryParser**

```
String searchQuery = "Batman";
String[] productFields = {"title", "description"}; ← Targeted fields

Map<String,Float> boostPerField = new HashMap<String,Float>(2); ← Boost
boostPerField.put( "title", (float) 4);                      factors
boostPerField.put( "description", (float) 1);

QueryParser parser = new MultiFieldQueryParser( ← Build
    productFields,                                     multifield
    new StandardAnalyzer(),                             query parser
    boostPerField
);

org.apache.lucene.search.Query luceneQuery;
try {
    luceneQuery = parser.parse(searchQuery);
}
catch (ParseException e) {
    throw new RuntimeException("Unable to parse query: " + searchQuery, e);
}
```

The `MultiFieldQueryParser` allows you to define more than one default field at a time. It becomes very easy to build queries that return all objects matching a given word or set of words in one or more object properties. In our example, the query will try to find *Batman* in either the title or the description field. The `MultiFieldQueryParser` also allows you to express the intuitive idea that title is more important than description in the query results. You can assign a different weight (also called *boost factor*) to each targeted field.

**2.5.2 Building the Hibernate Search query**

Our Lucene query is now ready to be executed. The next step is to wrap this query into a Hibernate Search query so that we can live in the full object-oriented paradigm. We already know how to retrieve a `FullTextSession` or `FullTextEntityManager` from a regular `Session` or `EntityManager`. A `FullTextSession` or a `FullTextEntityManager` is the entry point for creating a Hibernate Search query out of a Lucene query (see listing 2.12).

**Listing 2.12 Creating a Hibernate Search query**

```
FullTextSession ftSession = Search.getFullTextSession(session);

org.hibernate.Query query = ftSession.createFullTextQuery(
    luceneQuery,
    Item.class); ← Return matching Items

query = ftSession.createFullTextQuery(
    luceneQuery); ← Return all matching indexed entities

query = ftSession.createFullTextQuery(
    luceneQuery,
    Item.class,
    Actor.class); ← Return matching
                    Items and Actors
```

```

FullTextEntityManager ftEm =
    Search.getFullTextEntityManager(entityManager);

javax.persistence.Query query = ftEm.createFullTextQuery(
    luceneQuery,
    Item.class);
                                     ← Return matching Items

javax.persistence.Query query = ftEm.createFullTextQuery(
    luceneQuery);
                                     ← Return all matching indexed entities

javax.persistence.Query query = ftEm.createFullTextQuery(
    luceneQuery,
    Item.class,
    Actor.class);
    ← Return matching Items and Actors

```

The query-creation method takes the Lucene query as its first parameter, which isn't really a surprise, but it also optionally takes the class targeted by the query as an additional parameter (see our first example). This method uses a Java 5 feature named varargs. After the mandatory Lucene query parameter, the method can accept any number of Class parameters (from zero to many). If no class is provided, the query will target all entities indexed. If one or more classes are provided, the query will be limited to these classes and their subclasses (Hibernate Search queries are polymorphic, just like Hibernate Query Language [HQL] queries). While most queries target one class, it can be handy in some situations to target more than one entity type and benefit from the unstructured capabilities of Lucene indexes. Note that by restricting the query to a few entity types (and especially one entity type), Hibernate Search can optimize query performance. This should be your preferred choice.

The second interesting point to note is that the query objects are respectively of type `org.hibernate.Query` or `javax.persistence.Query` (depending whether you are targeting the Hibernate APIs or the Java Persistence APIs). This is very interesting because it enables a smooth integration with existing Hibernate applications. Anybody familiar with Hibernate or Java Persistence's queries will have no problem executing the query from this point.

### 2.5.3 Executing a Hibernate Search query

Executing and interacting with a Hibernate Search query is exactly like executing and interacting with an HQL or Java Persistence Query Language (JPQL) query simply because they share the same concepts and the same APIs.

Listing 2.13 demonstrates this.

#### Listing 2.13 Executing a Hibernate Search query

```

//Hibernate Core Query APIs
query.setFirstResult(20).setMaxResults(20);  ← Set pagination
List results = query.list();  ← Execute the query

//Java Persistence Query APIs
query.setFirstResult(20).setMaxResults(20);  ← Set pagination
List results = query.getResultList();  ← Execute the query

```



```

for (Item item : (List<Item>) results) {
    display( "title: " + item.getTitle() + "\nDescription: " +
            item.getDescription() );
}

```

There's no difference here between executing an HQL or JPA-QL query and executing a Hibernate Search query. Specifically, you can use pagination as well as execute the query to return a list, an iterator, or a single result. The behavior and semantic are the same as for the classic queries. Specifically, the returned result is composed of objects from your domain model and not Documents from the Lucene API.

The objects returned by the query are part of the Hibernate persistence context. Hibernate will propagate every change made to the returned objects into the database and the index transparently. And, more important, navigating through lazy associations (collections or single-ended associations) is possible transparently, thanks to Hibernate.

While building a query seems like a lot of steps, it's a very easy process. In summary:

- 1 Build the Lucene query (using one of the query parsers or programmatically).
- 2 Wrap the Lucene query inside a Hibernate Search query.
- 3 Optionally set some query properties (such as pagination).
- 4 Execute the query.

Unfortunately (or fortunately, if you like challenges), queries don't always return what you expect them to return. This could be because indexing didn't happen, or the query you've written doesn't do what you think it should. A tremendously useful tool is available that allows you to have an inside look at the Lucene index and see what queries return. Its name is Luke.

## 2.6 **Luke: inside look into Lucene indexes**

The most indispensable utility you can have in your arsenal of index troubleshooting tools—in fact it may be the *only* one you need—is Luke. With it you can examine every facet of an index you can imagine. Some of its capabilities are these:

- View individual documents.
- Execute a search and browse the results.
- Selectively delete documents from the index.
- Examine term frequency.

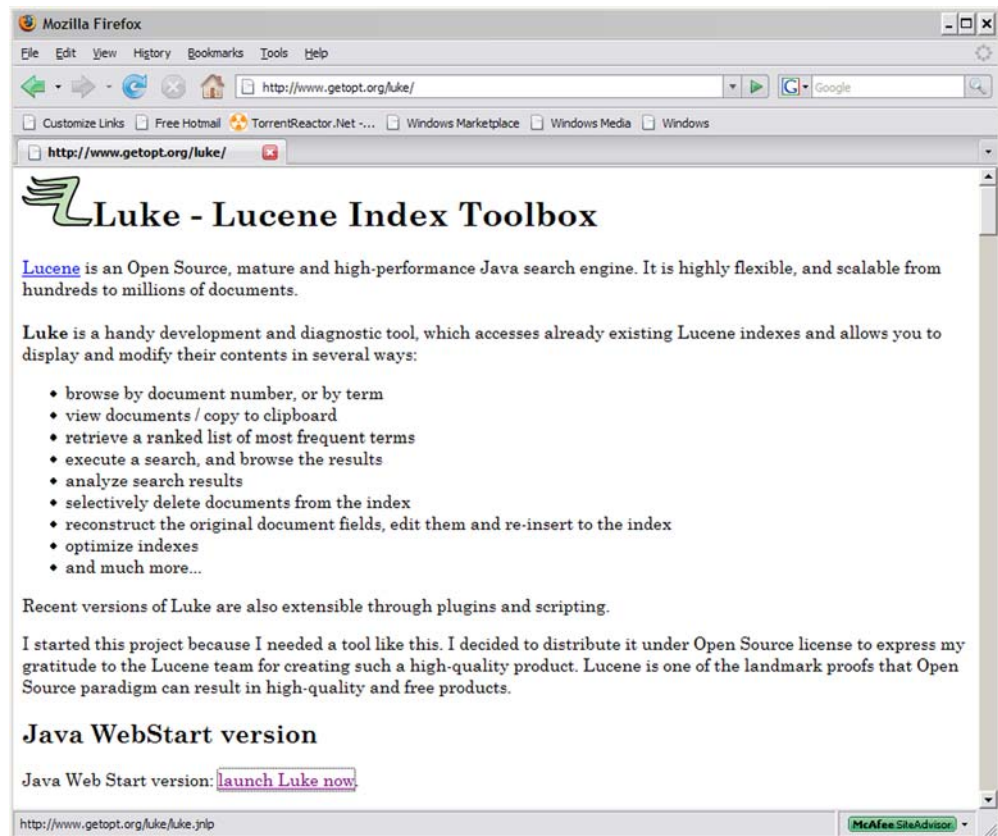
Luke's author is Andrzej Bialecki, and he actively maintains Luke to keep up with the latest Lucene version. Luke is available for download at <http://www.getopt.org/luke/>, shown in figure 2.3. You can download Luke in several different formats. A Java WebStart JNLP direct download of the most current version is the easiest to retrieve; it's basically automatic. You can also download several .jar file compilations and place them in your classpath any way you want them.

- *lukeall.jar*—Contains Luke, Lucene, Rhino JavaScript, plug-ins, and additional analyzers. This JAR has no external dependencies. Run it with `java -jar luke-all.jar`.

- *lukemin.jar*—A standalone minimal JAR, containing Luke and Lucene. This JAR has no external dependencies either. Run it with `java -jar lukemin.jar`.
- *Individual jars:*
  - *luke.jar*
  - *lucene-core.jar*
  - *lucene-analyzers.jar*
  - *lucene-snowball.jar*
  - *js.jar*

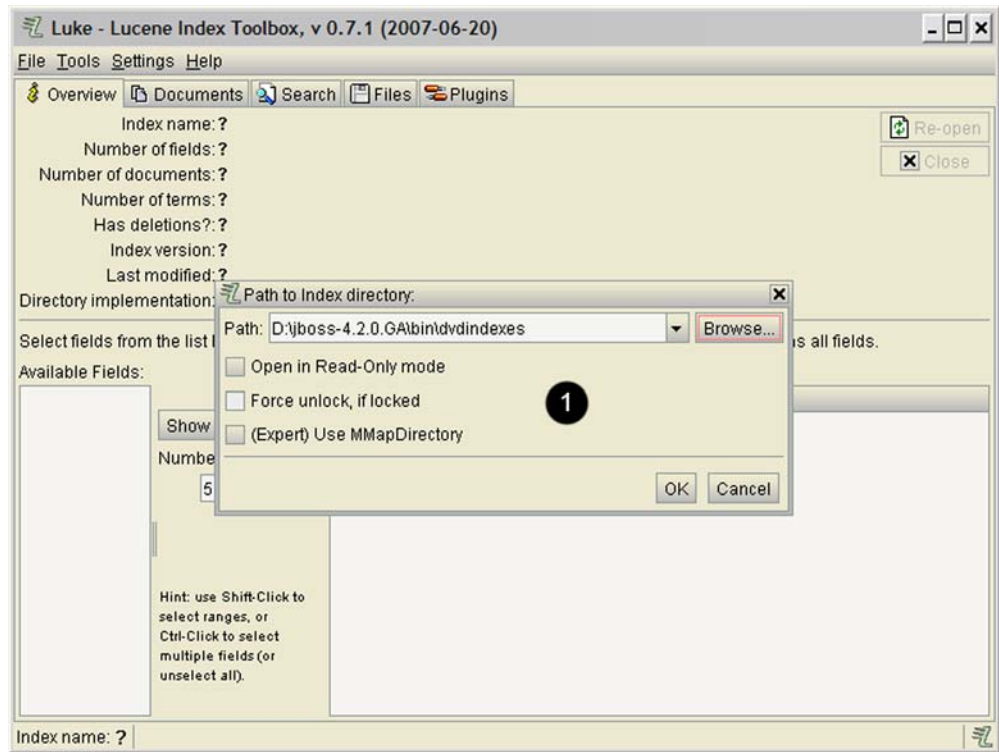
Minimum requirements are that at least the core JARs be in your classpath, for example, `java -classpath luke.jar;lucene-core.jar org.getopt.luke.Luke`. Be careful to use the right Luke version for your Lucene version, or Luke might not be able to read the Lucene index schema.

Luke's source code is also available for download from the website shown in figure 2.3 for those of you who want to dig into the real workings of the application.



**Figure 2.3** Luke's website with download links in various formats along with source code downloads

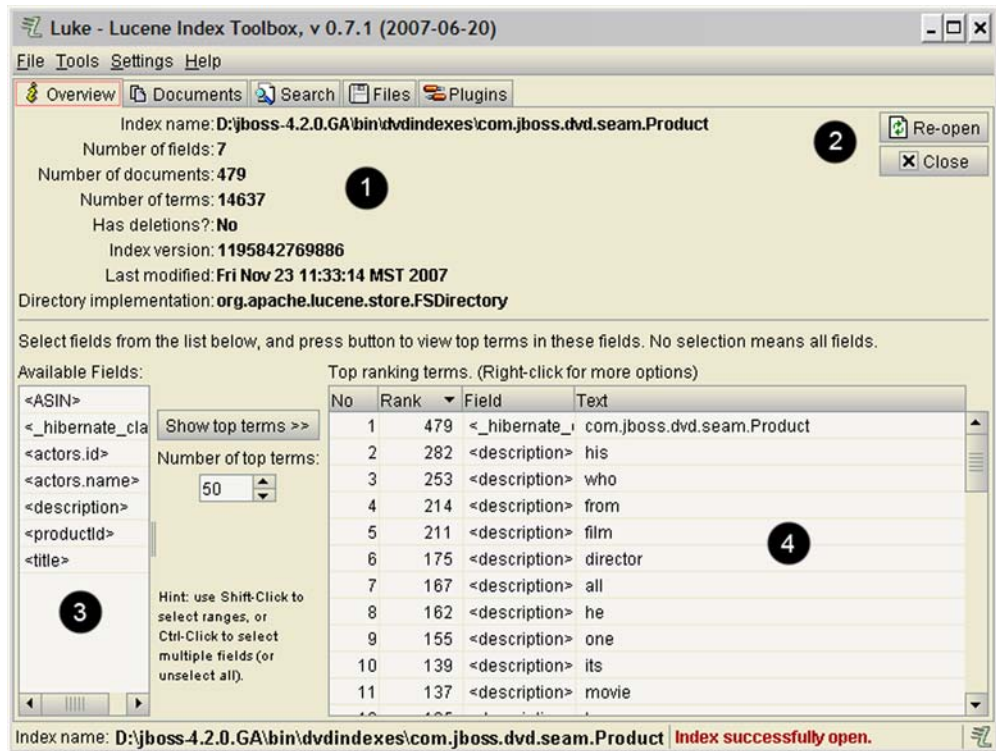
Let's go on the fifty-cent tour of Luke. We'll start with figure 2.4. The Overview tab is the initial screen when Luke is first started.



**Figure 2.4** Luke's opening screen containing the index path and modes to open it

① The Path field contains the operating system path to the index's location and modes you can choose to open the index. A convenient filesystem browser makes navigation easier. Utilizing open modes, you can force *unlock* on an index that may be locked. This could be useful from an administration point of view. Also, you can open the index in read-only mode to prevent making accidental changes. An advanced option allows you to open the index using an `MMapDirectory` instance, which uses nonblocking I/O (NIO) to memory map input files. This mode uses less memory per query term because a new buffer is not allocated per term, which may help applications that use, for example, wildcard queries.

Behind this subwindow you can see the other tabs: Overview, Documents, Search, Files, and Plugins, which are all coming up shortly. Let's move on to the Overview tab. Looking at figure 2.5, you can see a wealth of information in this tab alone.



**Figure 2.5** The Overview tab showing the index's statistics, fields, and an order ranking of the top terms in the index

① The top section is a comprehensive listing of the index's statistics, including last modification date, total number of documents in the index, number of fields, and so on. ② is the Re-open button.

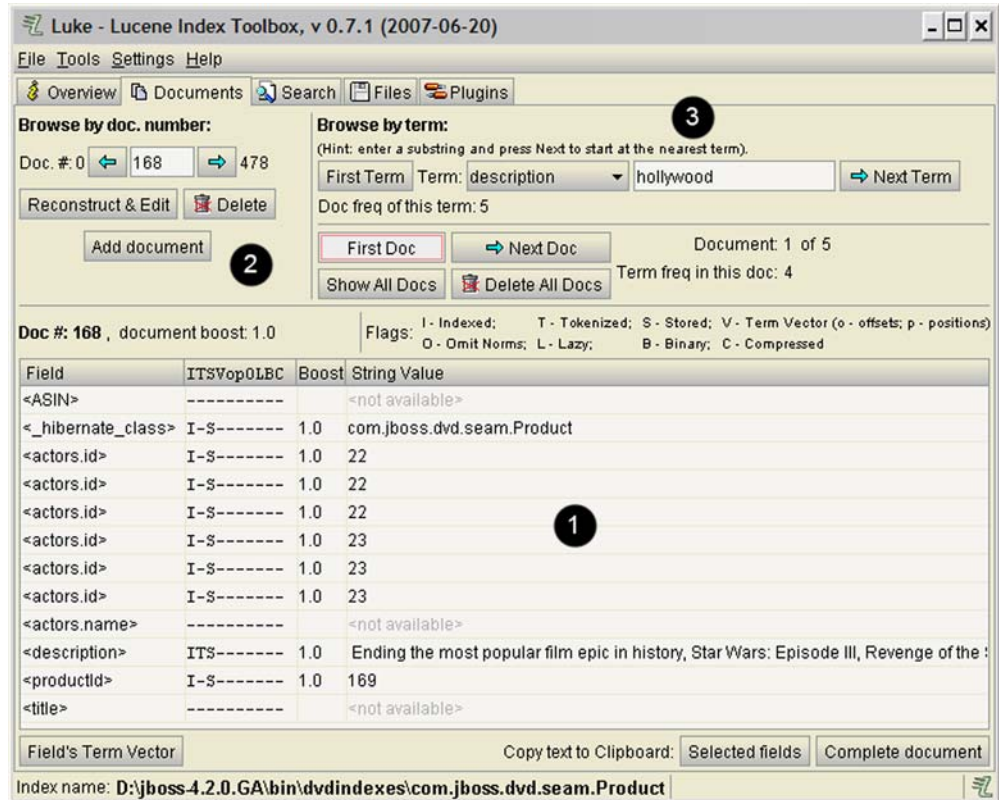
**NOTE** Documents deleted or updated (a delete followed by an insert in Lucene) are not seen until the index is reopened. When you open an index, you have a snapshot of what was indexed at the time it was opened.

A list of all available fields in the documents is shown at ③. The field name is the string enclosed in brackets.

④ is an ordered listing, from the most frequently occurring to the least, of the top terms in the index. From this quadrant of the tab you can do several things. Double-clicking a term transfers you to the Documents tab (we'll talk about that tab next) and automatically inserts the term you double-clicked into the Browse by Term text box. Right-clicking a term in this quadrant gives you several options. Browse Term Docs does the same thing as double-clicking the term: It transfers you to the Documents tab and automatically inserts the term you double-clicked into the Browse by Term text

box. Show All Term Docs transfers you to the Search tab (we'll talk about that shortly) and automatically inserts a search based on the Field and Text data in 3.

Let's move on to the next tab in Luke, the Documents tab. This is shown in figure 2.6.

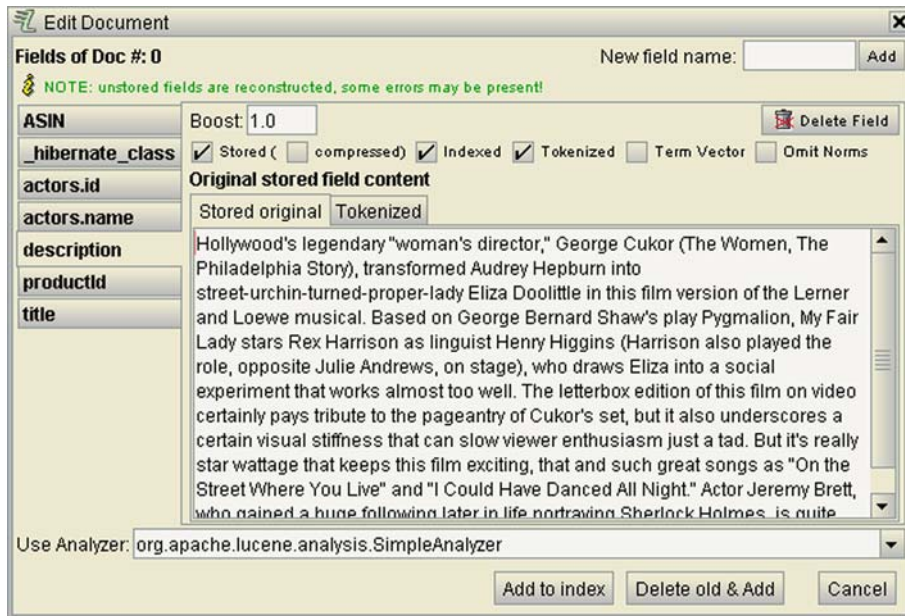


**Figure 2.6** The Documents tab, where documents can be stepped through, deleted, and examined

2 allows you to browse the index's documents by stepping through them one at a time. They are ordered by Lucene's internal document number. You can even add, edit, and delete individual documents by using this quadrant of the tab. The Edit Document screen is shown in figure 2.7. Here you can edit any field in the document. The Stored Original tab will show you the data stored in the index for this document's selected field.

**NOTE** If the data is not stored in the index via the `Store.YES` or `Store.COMPRESS` setting, you won't be able to access that data because it's simply not there! You'll see `<not available>` instead.





**Figure 2.7** The Luke Edit Document screen showing the Stored Original tab

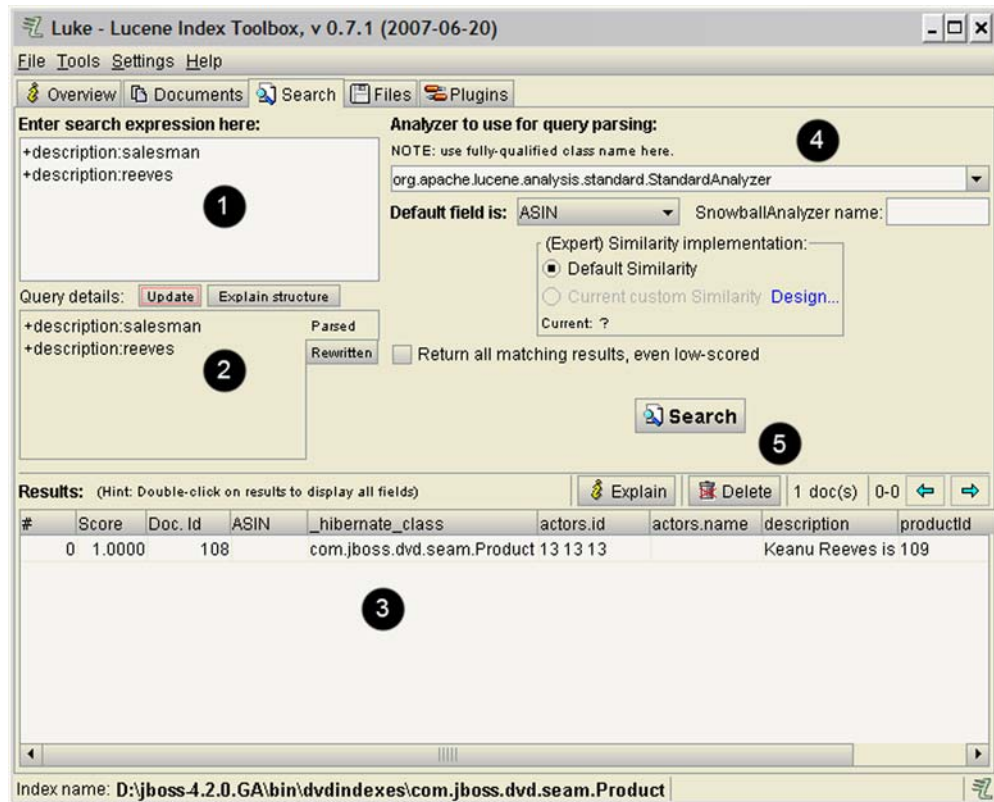
One of the really neat features here is the Tokenized tab. For fields that do store their data in the index, this tab shows the tokenized text based on the analyzer used to index the field.

This is all well and good, but suppose we wanted to browse the index by its terms in term order and not in the order of frequency, as listed in the Documents tab. This is done in the upper-right quadrant ③ (in figure 2.6) of the tab, Browse by Term, but is not as straightforward as it sounds. Clicking First Term takes you to the first term for that field in the index (alphabetical, numbers first). Of course, Next Term continues forward. Below this button is the document browser. Clicking First Doc takes you to the first document ① containing the term in the Browse by Term text box we just talked about. The Show All Docs button takes you to the Search Tab and automatically inserts a search for that field and term in the search window.

**NOTE** Be careful with the Delete All Docs button. Be positive about which index you have open before clicking this button. It would be a sad day if you forgot you were looking at the production index!

You may have noticed several terms scattered about this tab like *Doc freq of this term* and *Term freq in this doc*. We'll explain these terms and use them a lot in chapter 12, but for now, don't worry about them. Their meaning will become clear, and they'll mean a lot more then.

We're finally going to discuss the Search tab. This is important because the authors find themselves on this tab the vast majority of time when they're testing the effect of different analyzers, what the stop word effect will be, and the general behavior of a query. Figure 2.8 shows this tab.



**Figure 2.8** The Search tab showing the search window, details, and the results window

① is the search window. Here you enter search expressions based on the current index. Searches are expressed utilizing the Lucene query parser syntax. In this example the search is for two terms, both of which must occur in the Description field, and both terms are required (that's what the + signs indicate). A complete discussion of the syntax is given on the Lucene website at <http://lucene.apache.org/java/docs/queryparsersyntax.html> or in the Lucene documentation available in each distribution. We'll also cover the essential part of this syntax in chapter 7. You must specify the field name of a term (using the `field_name:` syntax) if you wish to query a different field than the one specified in the Default field just under ④. The uses of the +, -, ~ and other symbols are explained on the website.

After you enter your query in ❶, select the analyzer ❷ to use to parse your query. The default analyzers supported by Luke are:

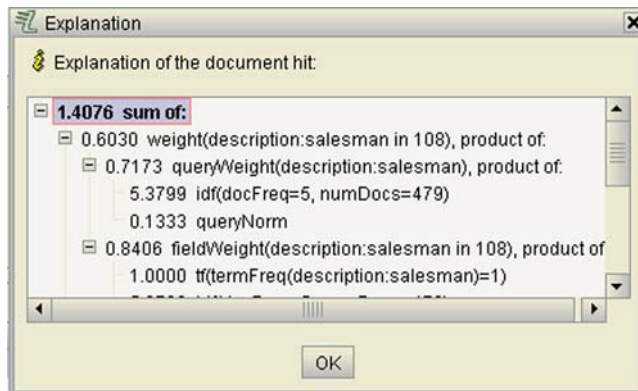
- StandardAnalyzer
- StopAnalyzer
- WhitespaceAnalyzer
- SimpleAnalyzer

When Luke initializes, it looks in the classpath for any additional analyzers and adds them to the drop-down list. That's how you can test any analyzer that you may have written. All you have to do then is select it from the drop-down list. Then click the Update button between ❶ and ❷, and the parsed query will be shown in the Parsed window ❸. We strongly recommend you get into the habit of doing this because, many times, the analyzer does things a little differently than you think it would. This will save you from heartaches.

The Search button ❹ executes the search displayed in ❷ against the current index and displays the search results in ❸. In this example, a search on the Description field for *salesman* and *reeves* resulted in one matching document, with Lucene ID 108. Double-clicking any matching document will take you back to the Documents tab, with the appropriate information concerning that document being displayed, such as the document number and a vertical listing of the document's fields. Also at ❹ is the Explain button. Clicking this button brings up the Explain window, shown in figure 2.9.

This window shows how the score of a document was calculated against a particular query and what factors were considered. This may not mean much to you now, but when you get into the middle of chapter 12 and we show you exactly what this provides, you'll appreciate it much more. This is especially true if you're one of those who want to modify the way documents are scored.

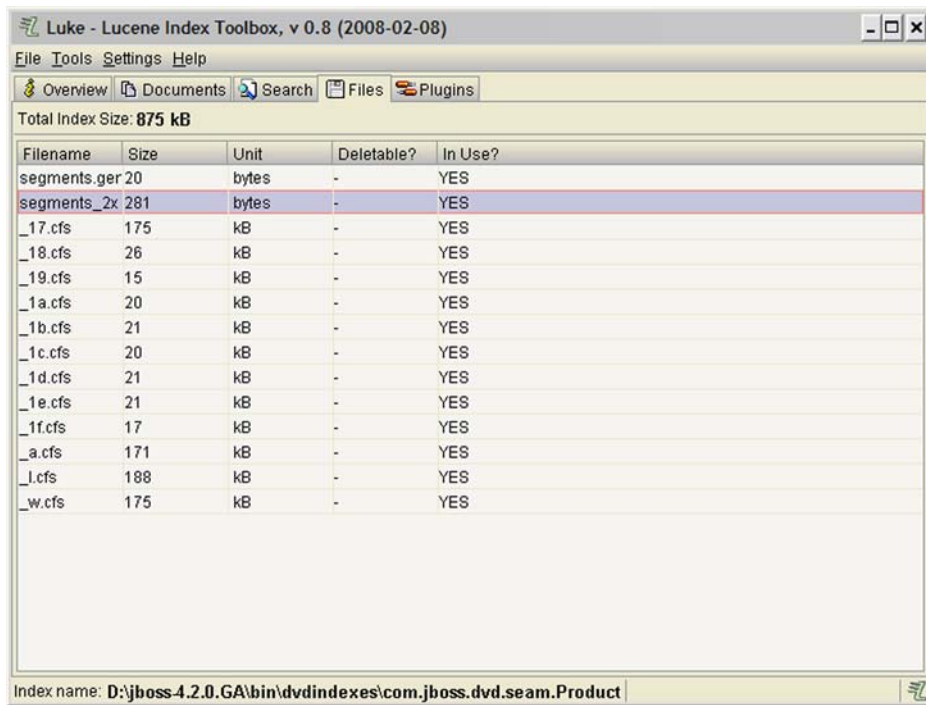
The next-to-last tab of Luke is the Files tab, shown in figure 2.10. From an administration point of view, this presents a lot of information. First, the Total Index Size value could be important for disk space considerations. Below that is a listing of all the



**Figure 2.9** The Explain window showing how the document's score was calculated



files associated with the index. There are no right-clicks or double-clicks here. What you see is what you get. This file listing helps with determining whether or not the index needs to be optimized.



Luke - Lucene Index Toolbox, v 0.8 (2008-02-08)

File Tools Settings Help

Overview Documents Search Files Plugins

Total Index Size: 875 kB

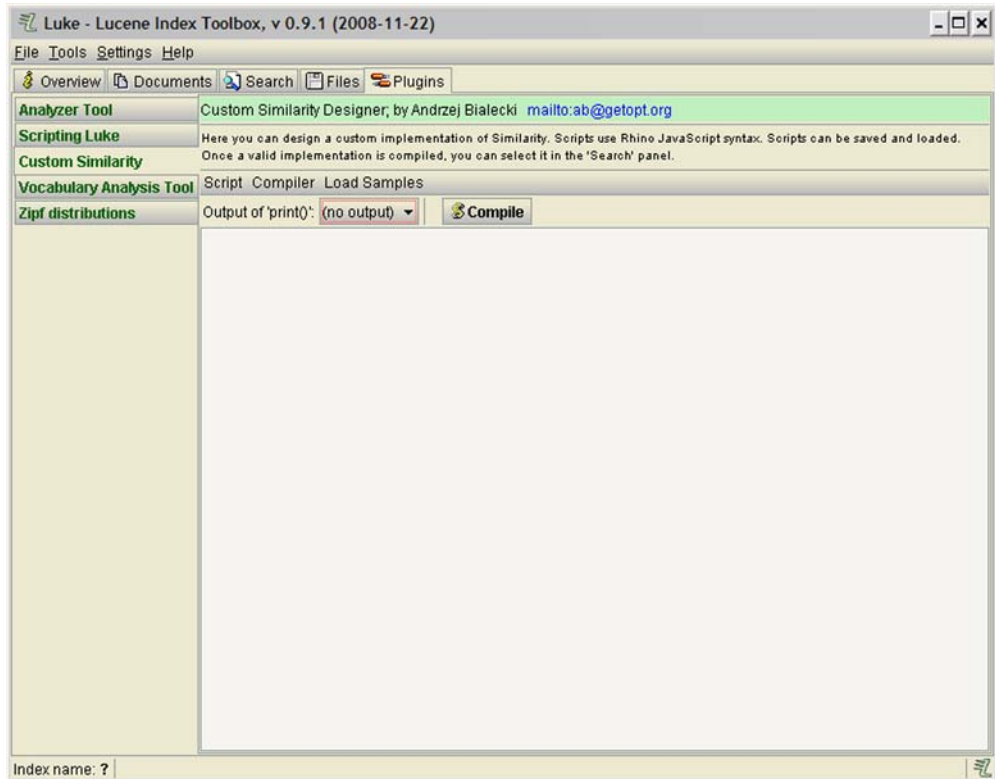
Filename	Size	Unit	Deletable?	In Use?
segments.ger	20	bytes	-	YES
segments_2x	281	bytes	-	YES
_17.cfs	175	kB	-	YES
_18.cfs	26	kB	-	YES
_19.cfs	15	kB	-	YES
_1a.cfs	20	kB	-	YES
_1b.cfs	21	kB	-	YES
_1c.cfs	20	kB	-	YES
_1d.cfs	21	kB	-	YES
_1e.cfs	21	kB	-	YES
_1f.cfs	17	kB	-	YES
_a.cfs	171	kB	-	YES
_l.cfs	188	kB	-	YES
_w.cfs	175	kB	-	YES

Index name: D:\jboss-4.2.0.GA\bin\dvdindexes\com.jboss.dvd.seam.Product

**Figure 2.10** This is a listing of all files associated with the currently open index

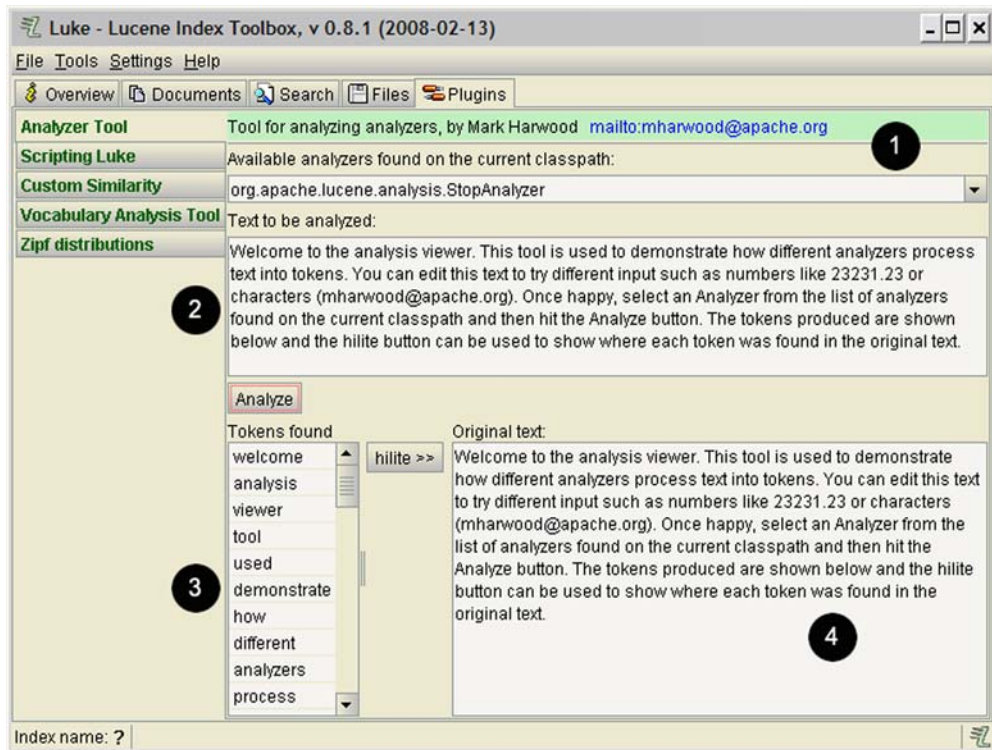
Remember, the greater the number of segment files (.cfs), the slower searches become, so be sure to optimize. How often you optimize depends on several factors, but this tab will help you determine how often you will need to do it. For more information about optimization, check out section 9.3.

Our last Luke tab is the Plugins tab. This is the developer's tab. Five items on this tab will help you accomplish several things. We're going to show only two of these tabs because they apply more to the discussions we're going to have later in the book. Figure 2.11 shows the Custom Similarity plug-in; this allows you to design and test your own Similarity object which enables you to implement your own document scoring mechanism. You will learn all about this in detail in chapter 12.



**Figure 2.11** The Custom Similarity Designer selection on the Plugins tab

Figure 2.12 shows the Analyzer Tool. This lets you examine exactly how a particular analyzer affects the text that's put into an index.



**Figure 2.12** The Analyzer Tool tab for examining how analyzers change text put into indexes

① is a drop-down list of all analyzers found in the classpath. Picking a particular analyzer and clicking the Analyze button will display a list of tokens ③ generated by that analyzer from the text you enter in the text window ②. This makes it quite easy to test an analyzer to see if it generates the tokens you thought it would. Clicking one of the tokens, then clicking the Hilite button will cause that token to be highlighted in the Original Text window ④.

**WARNING** Luke has several known issues when the Java Network Launching Protocol (JNLP) version is used. These are enumerated on the Luke download page. One of these issues is recognizing analyzers on the classpath. When you work with analyzers, the authors recommend that you download one of the standalone Luke versions and work with it.

The best recommendation we can give you for learning how these plug-ins work (and in case you want to write one yourself) is to study Luke's documentation and especially the various plug-ins' source code. Remember, Luke's source code is also available for download from the website.

That's all for Luke right now. Luke is your best friend in the full-text query jungle, so use it!

## 2.7 Summary

In chapter 1 you saw some of the issues that arise when you try to integrate domain model-centric applications and full-text search technologies (and in particular Lucene). These problems were threefold: a structural mismatch, a synchronization mismatch, and a retrieval mismatch. From what you've seen so far, does Hibernate Search address all these problems?

The first problem we faced was the structural mismatch. The structural mismatch comes in two flavors:

- Convert the rich object-type structure of a domain model into a set of strings.
- Express the relationship between objects at the index level.

Hibernate Search addresses the first problem by allowing you to annotate which property and which entity need to be indexed. From them, and thanks to sensitive defaults, Hibernate Search builds the appropriate Lucene indexes and converts the object model into an indexed model. The fine granularity is a plus because it helps the application developer to precisely define what Lucene needs to process and in which condition. This getting-started guide did not show how Hibernate Search solves the relationship issue, because we have to keep a few subjects for the rest of the book. Don't worry; this problem will be addressed.

The second mismatch involved synchronization: how to keep the database and the index synchronized with each other. Hibernate Search listens to changes executed by Hibernate Core on indexed entities and applies the same operation to the index. That way, the database and index are kept synchronized transparently for the application developer. Hibernate Search also provides explicit indexing APIs, which are very useful for filling the index initially from an existing data set.

The third mismatch was the retrieval mismatch. Hibernate Search provides a match between the Lucene index field names and the property names (out of the box), which helps you to write Lucene queries. The same namespace is used in the object world and the index world. The rest of the Lucene query is entirely up to the application developer. Hibernate Search doesn't hide the underlying Lucene API in order to keep intact all the flexibility of Lucene queries. However, Hibernate Search wraps the Lucene query into a Hibernate query, reusing the Hibernate or Java Persistence APIs to provide a smooth integration into the Hibernate query model. Hibernate Search queries return domain model objects rather than Lucene Document instances. Beyond the API alignment, the semantics of the retrieved objects are similar

between an HQL query and a full-text query. This makes the migration from one strategy to the other very simple and targeted.

Other than the fundamental mismatches, Hibernate Search doesn't require any specific configuration infrastructure as it integrates into the Hibernate Core configuration scheme and lifecycle. It doesn't require you to list all the indexed entities. We've only started our exploration of Hibernate Search, but you can already feel that this tool focuses on ease of use, has a deep integration with the persistence services, and addresses the mismatch problems of integrating a full-text solution like Lucene into a domain model-centric application.

Hopefully, you want to know more about Hibernate Search and explore more of its functionalities, and there's a lot more to explore. The next chapters of the book are all about making you an expert in Hibernate Search and helping you discover what it can solve for you!

# HIBERNATE SEARCH IN ACTION

Emmanuel Bernard and John Griffin

**L**ucene is an ideal tool for searching text but it approaches the task unaware of your application's data structures, as if your search space were flat. Hibernate Search, a full text search library, uses Hibernate's intimate knowledge of the data structures to inform and guide Lucene's indexing and searches. The result is much smarter searching. It also stays abreast of changes in your data, and lets you query using either full text or HQL.

**Hibernate Search in Action** introduces the subject of full-text search and helps readers master the Hibernate Search library. You'll start with the basics, like indexing your domain model and querying. Then, you'll learn to add human-friendly features like phonetic approximation, relevance ranking, and search by synonym. You'll also learn how to scale Lucene in a clustered environment and access Lucene natively to extend Hibernate Search. The book does not assume any previous knowledge of Hibernate Search.

## What's Inside

- Clear explanations of indexing, querying, and filtering
- Document scoring and access to native Lucene APIs
- Performance tuning and clustering

## About the Authors

**Emmanuel Bernard** is employed at JBoss where he leads the Hibernate Search as well as Hibernate Annotations and Hibernate EntityManager projects. He is also the JCP spec lead for Bean Validation, and a member of the Java Persistence 2.0 expert group. **John Griffin** is a software engineer and architect, an adjunct university professor, and an open source committer.

For online access to the authors, code samples, and a free ebook for owners of this book, go to [www.manning.com/HibernateSearchinAction](http://www.manning.com/HibernateSearchinAction)

**Free ebook**  
SEE INSERT

"A great resource for true database independent full text search."

—Aaron Walker, base2Services

"It has completely changed the way I do complex search. Awesome!"

—Ayende Rahien  
Author of *Building Domain Specific Languages in Boo*

"Love its vast coverage —the definitive source."

—Patrick Dennis  
Management Dynamics Inc.

"Covers it all... the only resource I need."

—Robert Hanson  
Author of *GWT in Action*

"A superb discussion of a complex topic."

—Spencer Stejskal  
SOS Staffing Services

ISBN-13: 978-1933988641  
ISBN-10: 1933988649



9 781933 988641



**MANNING**

US/CAN \$49.99 [INCLUDING EBOOK]