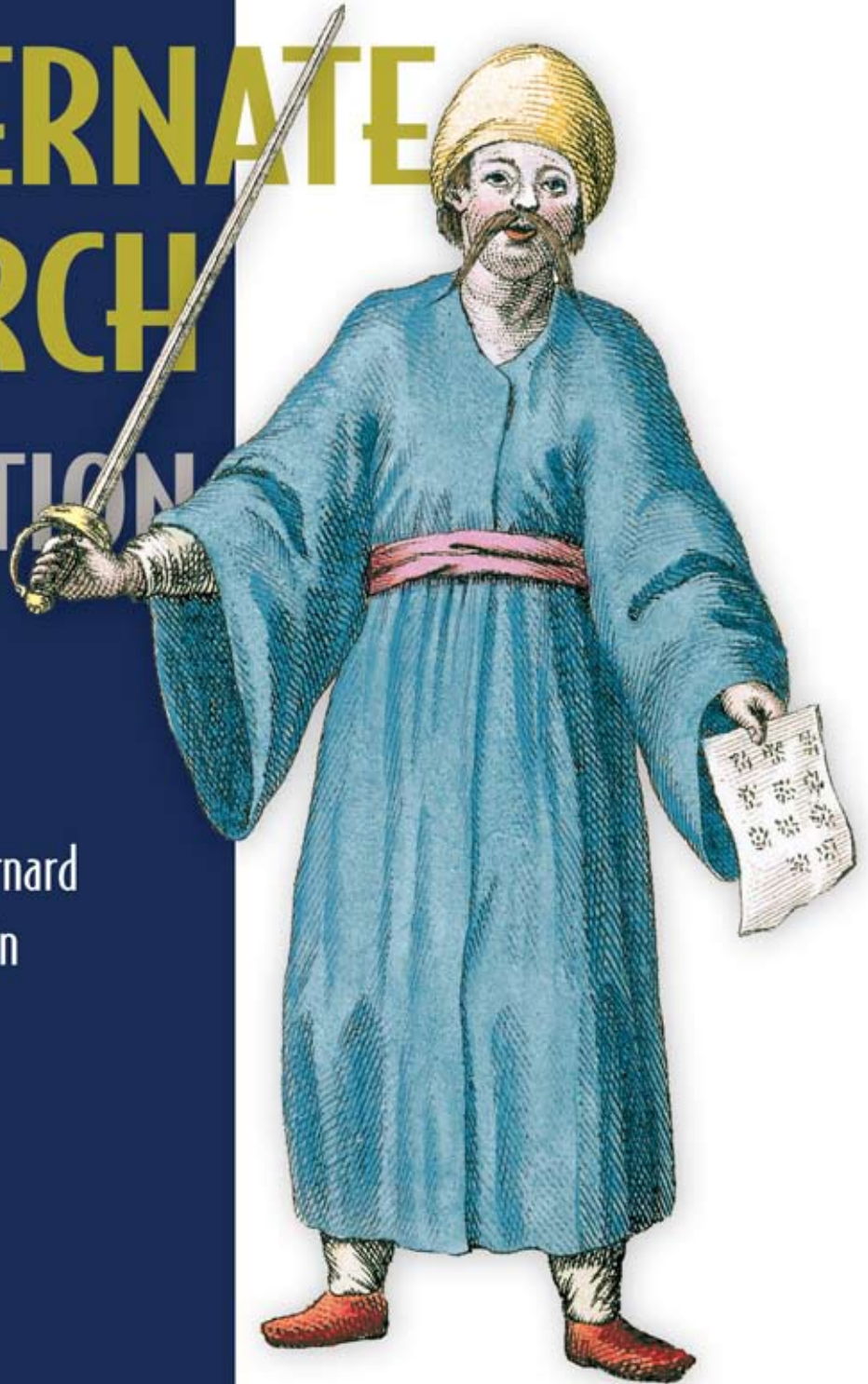


Sample Chapter

HIBERNATE SEARCH IN ACTION

Emmanuel Bernard
John Griffin





Hibernate Search in Action

by Emmanuel Bernard
and John Griffin

Chapter 11

Copyright 2009 Manning Publications

brief contents

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

BRIEF CONTENTS

PART 5 NATIVE LUCENE, SCORING, AND THE WHEEL..... 351

12 ■ Document ranking 353

13 ■ Don't reinvent the wheel 399

appendix Quick reference

11

Accessing Lucene natively

This chapter covers

- Utilizing the `SearchFactory`
- Accessing Lucene directories
- Working with
- `DirectoryProviders`
- Exploiting projections

If you have not realized it by now, or you just started reading this chapter first, Lucene is the driving force behind the Hibernate Search framework. The entire book up to this point has been dedicated to helping you understand how to implement Hibernate Search in your application. Eventually, everyone has questions related to working with Lucene directly. We hear questions similar to the following all the time:

- Hibernate Search's default constructs won't work for me. Now what do I do?
- I know Hibernate Search takes care of a lot of things for me, but I need to get at Lucene itself and work with its native constructs. How can I do that?

- Can I get an instance of a Lucene `Directory` object so I can work at a lower level?
- Hibernate Search has sharded my entity into three separate directories. Is it possible to work with them as a single object? Can I improve retrieval performance?

Do any of these sound familiar? We're going to answer these questions in this chapter. We'll start by looking at Hibernate Search's `SearchFactory`, which is the key entry point to Lucene. It allows access to `Directory` objects and `IndexReaders`. We'll look at the effects that sharding has on these classes, and along the way we'll also see what goes on when indexing multiple entities.

One of the things that developers must assess is whether or not the default `DirectoryProviders` included with Hibernate Search are sufficient for their purposes. In the event they are not, you can write a custom one. We'll discuss what you must take into account in this situation.

We'll also show you how to access the legacy Lucene document and several related values by utilizing a projection, and we'll demonstrate how it can affect performance.

We believe you'll be pleasantly surprised, because accessing native Lucene is much easier than you think.

11.1 *Getting to the bottom of Hibernate Search*

Getting under the covers to interact directly with native Lucene constructs is not difficult. The most important class here is Hibernate Search's `org.hibernate.search.SearchFactory` class. It's the gateway to native Lucene.

The `SearchFactory` keeps track of the underlying Lucene resources for Hibernate Search. The contract for `SearchFactoryImpl` is maintained by the `SearchFactory` and `SearchFactoryImplementor` interfaces.

You can access the `SearchFactory` from an `org.hibernate.search.FullTextSession` instance, which is obtained from a Hibernate session, as shown in the following code:

```
FullTextSession fullTextSession =  
    Search.createFullTextSession(SessionFactory.openSession());  
SearchFactory searchFactory = fullTextSession.getSearchFactory();
```

Once you have an instance of the `SearchFactory`, you have all you need to work directly with Lucene. From here you can obtain references to Lucene's `Directory` object and also Lucene's `IndexReader`. We'll look at these in the following sections.

11.1.1 *Accessing a Lucene directory*

Lucene has a notion of a `Directory`, which it uses to store its indexed information. A `Directory` is a list of flat files that may be written to once, when they are created. Once created, a file can be opened only for reading or deleting. Random access to the files is permitted at all times.

All file I/O goes through Lucene's API, so that it's nicely encapsulated, but you still retain all needed flexibility. This allows Lucene's indexes to be manipulated and stored several ways, such as these:

- A set of flat files on some type of persistent storage
- A RAM-based directory
- implementation
- A database index implementation, via JDBC

NOTE The authors do not recommend implementing the JDBC configuration. We'll discuss why in section 11.3.

You can always access the native Lucene directories through plain Lucene. The `Directory` structure is in no way different with or without Hibernate Search. However, there are more convenient ways to access a given `Directory`.

The `SearchFactory` we discussed previously keeps track of all of the `org.hibernate.search.store.DirectoryProviders` that an indexed class may utilize. You obtain access to directories via the `DirectoryProvider`. Notice the use of the plural of `DirectoryProvider` here. A given entity can have several `DirectoryProviders`, one per shard, if the index is sharded (see the discussion on index sharding in section 9.4). In the opposite vein, one `DirectoryProvider` can be shared among several indexed classes if the classes share the same underlying index directory. Section 11.1.5 provides an example of this index merging. The `DirectoryProvider` class's main aims are to:

- Set up a Lucene directory for an index
- Serve as an abstraction separating Hibernate Search from the Lucene directory implementation

This implementation could be in any form, even that of a server cluster and not just of a single file system directory.

Assuming we have an index built from `Order` information, here is a code example showing how to obtain an instance of a Lucene `org.apache.lucene.store.Directory`.

```
DirectoryProvider[] providers =
    searchFactory.getDirectoryProviders(Order.class);
org.apache.lucene.store.Directory directory =
    providers[0].getDirectory();
```

In this example code, `directory` points to the Lucene index storing `Order` information.

WARNING When utilizing the Hibernate Search framework to obtain an instance of a Lucene `Directory`, do not call `close()` on the obtained `Directory`. This is the responsibility of Hibernate Search. The one opening the resource has to ensure it gets closed; in this case you borrow a Hibernate Search managed instance.

Let's look at some examples so you'll better understand what to expect from the default `DirectoryProviders` that come bundled with Hibernate Search. We will accomplish this by the following steps:

- 1 Creating an index of a single entity and retrieving its `DirectoryProvider(s)` to see what we get
- 2 Creating a sharded index of a single entity and again retrieving the
- 3 `DirectoryProvider(s)` to compare with our first result
- 4 Creating a single index of two entities and examining how the
- 5 `DirectoryProvider(s)` have changed in this instance

11.1.2 *Obtaining DirectoryProviders from a non-sharded entity*

Listing 11.1 shows the simple `Animal` entity we'll use for the first example. This is nothing more than a simple JavaBean-style class and will work well for our example.

Listing 11.1 The simple `Animal` entity used in the examples

```
@Entity
@Indexed
@Analyzer(impl = StandardAnalyzer.class)
public class Animal {
    @Id
    @DocumentId
    private Integer id;

    @Field(index = Index.TOKENIZED, store = Store.YES)
    private String name;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

NOTE All of the examples in this chapter are based on the example tests included with the Hibernate Search source code. These tests are located in and around the `org.hibernate.search.test.shards` package.

We do not show all of the code in the examples that's necessary to make these examples work, although the complete code is included with the book's accompanying source files. For example, we provide scaffolding code that must be in place for the setup and so on. To examine this

code refer to the tests included with the Hibernate Search sources, specifically the code located in the `org.hibernate.search.test.shards.ShardsTest` class.

Listing 11.2 is the single non-sharded entity example. We create instances of our entity and build the index. Then we search for all records so we can clean up after ourselves by deleting everything we added. Finally, we retrieve the `DirectoryProviders`.

Listing 11.2 Indexing an entry to a non-sharded index

```
public class NonShardsTest extends SearchTestCase {
    Transaction tx;

    public void testNoShards() throws Exception {
        FullTextSession session = Search.getFullTextSession(openSession());
        buildIndex();

        tx = session.beginTransaction();
        QueryParser parser =
            new QueryParser("id", new StopAnalyzer());

        List results =
            session.createFullTextQuery(parser.
                parse("name:bear OR name:elephant")).list();

        assertEquals("Either insert or query failed", 2,
            results.size());

        SearchFactory searchFactory =
            session.getSearchFactory();
        DirectoryProvider[] providers =
            searchFactory.getDirectoryProviders(Animal
                .class);

        assertEquals("Wrong provider count", 1,
            providers.length);

        org.apache.lucene.store.Directory directory =
            providers[0].getDirectory();

        try {
            IndexReader reader =
                IndexReader.open(directory);
            assert reader.document(0).get("name").equals("Elephant")
                : "Incorrect document name";
            assert reader.document(1).get("name").equals("Bear")
                : "Incorrect document name";
            for (Object o : results) session.delete(o);
            tx.commit();
        }
        finally {
            if (reader != null)
                reader.close();
            session.close();
        }
    }

    private void buildIndex(FullTextSession session) {
        tx = session.beginTransaction();
    }
}
```

Extend for scaffolding code

Make sure everything worked properly

1 Get the DirectoryProvider for Animal

2 Check DirectoryProvider; should have only one

3 Retrieve the Lucene Directory object

Use a Directory to get the documents

4 Explicitly close the created reader

```

Animal a = new Animal();
a.setId(1);
a.setName("Elephant");
session.persist(a);
a = new Animal();

a.setId(2);
a.setName("Bear");
session.persist(a);
tx.commit();
session.clear();
}

@Override
protected void setUp() throws Exception {
    File sub = locateBaseDir();
    File[] files = sub.listFiles();
    if (files != null) {
        for (File file : files) {
            if (file.isDirectory()) {
                delete(file);
            }
        }
    }
    buildSessionFactory(getMappings(),
                       getAnnotatedPackages(),
                       getXmlFiles());
}

```

❶ shows that once we have an instance of `SearchFactory`, it is only one step to a `DirectoryProvider`. Because there's only one entity and no sharding, ❷ confirms that there's only one `DirectoryProvider`. Once we have an instance of a provider, we can easily obtain access to the Lucene `Directory` object ❸. Finally, ❹ demonstrates that we must close the readers we generated because these were created outside the Hibernate Search framework and it knows nothing about them. In section 11.2 we'll show you a situation where you *must not* call `close` on a reader.

11.1.3 And now for sharding one entity into two shards

In listing 11.3 we'll reuse the same code and make some simple changes. We'll add sharding to the `MergedAnimal` entity and divide one index into two different directories. The test code will confirm these changes. We're using the `MergedAnimal` entity here because it specifies the index name in the `@Indexed(index = "Animal")` annotation. The partial `MergedAnimal` entity showing the index name is given here:

```

@Entity
@Indexed(index = "Animal")
@Analyzer(impl = StandardAnalyzer.class)
public class MergedAnimal {
    @Id
    @DocumentId
    private Integer id;
    @Field(index = Index.TOKENIZED, store= Store.YES)
    private String name;
}

```

Listing 11.3 Indexing a single entity to a sharded index

```

public class TestShards extends SearchTestCase {
    Transaction tx;

    @Test
    public void testShards() throws Exception {
        FullTextSession session = Search.getFullTextSession(openSession());
        buildIndex();

        ReaderProvider readerProvider = null;

        IndexReader reader0 = null;
        IndexReader reader1 = null;
        List results;

        try {
            tx = session.beginTransaction();
            FullTextSession fullTextSession =
                Search.getFullTextSession(session);
            QueryParser parser = new QueryParser("id",
                new StandardAnalyzer());

            results = fullTextSession.createFullTextQuery(
                parser.parse("name:bear OR name:elephant")).list();
            assert results.size() == 2: "Either insert or query failed";

            SearchFactory searchFactory =
                fullTextSession.getSearchFactory();
            DirectoryProvider[] providers =
                searchFactory.getDirectoryProviders(MergedAnimal.class);
            assert providers.length == 2
                : "Wrong provider count";
            readerProvider =
                searchFactory.getReaderProvider();

            reader0 = readerProvider.openReader(providers[0]);
            reader1 = readerProvider.openReader(providers[1]);

            assert reader0.document(0).get("name").equals
                ("Bear"): "Incorrect document name";
            assert reader1.document(0).get("name").equals
                ("Elephant"): "Incorrect document name";
        }
        finally {
            for (Object o : results) session.delete(o);
            tx.commit();
        }

        finally {
            if (reader0 != null)
                readerProvider.closeReader(reader0);
            if (reader1 != null)
                readerProvider.closeReader(reader1);
            session.close();
        }
    }
}

```

 **1 Instantiate individual readers**

 **2 Check for splitting into shards**

 **3 Explicitly close the readers**

 **4 Check DirectoryProviders; better be two**

```

private void buildIndex() {
    tx = session.beginTransaction();

    MergedAnimal a = new MergedAnimal();
    a.setId(1);
    a.setName("Elephant");
    session.save(a);

    a = new MergedAnimal();
    a.setId(2);
    a.setName("Bear");
    session.save(a);
    tx.commit();
    session.clear();
}

@BeforeClass
protected void setUp() throws Exception {
    File sub = locateBaseDir();
    File[] files = sub.listFiles();
    if (files != null) {
        for (File file : files) {
            if (file.isDirectory()) {
                delete(file);
            }
        }
    }
    buildSessionFactory(getMappings(),
                       getAnnotatedPackages(),
                       getXmlFiles());
}

@Override
protected void configure(Configuration cfg) {
    super.configure(cfg);
    cfg.setProperty("hibernate.search.default
    ➤ .directory_provider", FSDirectoryProvider.class.getName());
    File sub = locateBaseDir();
    cfg.setProperty("hibernate.search.default.indexBase",
    ➤ sub.getAbsolutePath());
    cfg.setProperty("hibernate.search.Animal.
    ➤ sharding_strategy.nbr_of_shards", "2");
    4 Define two shards
    for entities

    cfg.setProperty("hibernate.search.Animal.0.indexName",
    "Animal00");
    5 Override the
    default shard name
}
}

```

We retrieve index readers for both of the indexes ❶. ❷ shows that our two entities were split into separate shards. We close the readers ❸, returning them to the `ReaderProvider`; in this case we have to avoid `IndexReader.close()`; we didn't open it directly but got it from the `ReaderProvider`, so this is necessary to manage its lifecycle.

To configure multiple shards we define the `nbr_of_shards` configuration quantity ❹ and decide to override the default shard name of the first shard ❺.

11.1.4 Indexing two non-sharded entities

In the final example, we'll index two different entities and not shard either of them. We'll use the `Animal` entity that we used in the previous two examples and add the `Furniture` entity that follows as the second entity. As with the `Animal` entity, this is nothing more than a simple JavaBean-style class.

```
@Entity
@Indexed
public class Furniture {
    @Id
    @GeneratedValue
    @DocumentId
    private Integer id;
    @Field(index= Index.TOKENIZED, store= Store.YES)
    private String color;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }
}
```

Listing 11.4 contains the code for this last example. The protected `void configure(Configuration cfg)` method in this example is identical to that shown in listing 11.2.

Listing 11.4 Indexing two entities to non-sharded indexes

```
public class TwoEntitiesTest extends SearchTestCase {
    FullTextSession session;
    Transaction tx;

    public void testTwoEntitiesNoShards() throws Exception {
        session = Search.getFullTextSession(openSession());
        buildIndex();

        tx = session.beginTransaction();
        FullTextSession fullTextSession =
            Search.getFullTextSession(session);
        QueryParser parser =
            new QueryParser("id", new StopAnalyzer());

        List results =
            fullTextSession.createFullTextQuery(parser.parse
            ➔ ("name:elephant OR color:blue")).list();
    }
}
```

```

assert results.size() == 2: "
    Either insert or query failed";

SearchFactory searchFactory =
    fullTextSession.getSearchFactory();
DirectoryProvider[] provider0 =
    searchFactory.getDirectoryProviders(
        Animal.class);
assert provider1.length == 1: "Wrong provider count";
org.apache.lucene.store.Directory directory0 =
    provider0[0].getDirectory();

DirectoryProvider[] provider1 = searchFactory.
    getDirectoryProviders(Furniture.class);
assert provider1.length == 1: "Wrong provider count";
org.apache.lucene.store.Directory directory1 =
    provider1[0].getDirectory();

IndexReader reader0 = IndexReader.open(directory0);
assert reader0.document(0).get("name")
    .equals("Elephant"): "Incorrect document name";
IndexReader reader1 = IndexReader.open(directory1);
assert reader1.document(0).get("color")
    .equals("dark blue"): "Incorrect color";
for (Object o : results) session.delete(o);
tx.commit();
}
Finally {
    session.close();
}

private void buildIndex() {
    tx = session.beginTransaction();

    Animal a = new Animal();
    a.setId(1);
    a.setName("Elephant");
    session.save(a);

    Furniture fur = new Furniture();
    fur.setColor("dark blue");
    session.save(fur);
    tx.commit();

    session.clear();
}
}

```

1 Ensure we have a result from each entity
2 Retrieve the DirectoryProvider
3 Retrieve a directory instance
4 Check for the correct Animal entity
5 Check for the correct Furniture entity
 Persist the second entity

After persisting one instance each of `Animal` and `Furniture`, we query for both of the instances **1**. We subsequently check for the correct number of results, getting the `DirectoryProviders` **2** for each entry, and from them we retrieve instances of the `Directory` objects **3**. Then we create `IndexReaders` and open them on the directories **4** and ensure that the entities were placed in the correct directories by examining their Lucene documents **5**.

We can draw several conclusions from the previous three examples. It seems that each entity is assigned its own directory and therefore its own `DirectoryProvider`. If

the entity is sharded, each of the shards also receives its own `DirectoryProvider`. That makes sense, and our examples prove it. So we're finished, right? As my old coach would say, "Not so fast, my friend!" Although this is the default behavior, Hibernate Search gives you the flexibility to combine entities into the same directory. We'll look at that next.

11.1.5 Shoehorning multiple entities into one index (merging)

Merge two entities into the same index? Why would we want to do this? Because we want to reuse the same `Directory`. Another example is to reuse an index that previously contained combined entities by completely deleting one of the entities it contains and replacing it with another.

NOTE The authors are presenting this technique so that you know the option is available. We do not feel that there's enough benefit in merging an index. The example we're presenting here will show that you have to do more work to accomplish the same thing you did in the previous examples.

There are actually two ways to accomplish merging an index:

- Configuring the property `hibernate.search.(fully qualified entity name).indexName=(relative directory from indexBase)`.
- Setting the `@Indexed` annotation's `index` property of the entity you want to merge to the directory you want the entity indexed into.

Look back at the partial `MergedAnimal` entity in 11.1.3. The `@Indexed` annotation shows the `index` property pointing to `Animal`. Now look at the `Furniture` entity shown just before listing 11.4. As is, it points to the `Furniture` directory, because that's the default value. If we wanted all `Furniture` instances to be indexed in the `Animal` index along with all instances of `Animal`, we would specify `@Indexed(index="Animal")`.

This does present a problem. Let's assume that the `id` value for the `Furniture` entities is actually a part number. What happens when we index both a piece of furniture and an animal with the same `id` value? We no longer get a single result back, and we must take into consideration the entity type we're searching for.

Let's look at an example where the additional concern of how to take entity type into consideration will become clear. We'll use an *entity filter* to make sure we work only the entity we need. We'll also use the `MergedAnimal` entity shown in section 11.1.3 as is, and we'll use the `Furniture` entity shown just before listing 11.4. The configuration will handle putting it in the same index as `Animal`. Listing 11.5 shows these two methods of merging entities into the same index.

Listing 11.5 Combining two entities into one index by setting configuration parameters

```
public class IndexMergeTest extends SearchTestCase {
    Transaction tx;

    public void testTwoEntitiesNoShards() throws Exception {
```

```

FullTextSession session = Search.getFullTextSession(openSession());
buildIndex(session);

tx = session.beginTransaction();
FullTextSession fullTextSession =
    Search.getFullTextSession(session);
QueryParser parser =
    new QueryParser("id", new StandardAnalyzer());

List results =
    fullTextSession.createFullTextQuery(parser
    ➤ .parse("id:1")).list();
    assert results.size() == 2: "Either insert or
    ➤ query failed";

    SearchFactory searchFactory =
        fullTextSession.getSearchFactory();
    DirectoryProvider[] provider =
        searchFactory.getDirectoryProviders(
    ➤ MergedAnimal.class);
    assert provider.length == 1
        : "Wrong provider count";
    org.apache.lucene.store.Directory directory =
        provider[0].getDirectory();

    BooleanQuery classFilter = new BooleanQuery();
    classFilter.setBoost(0);

    Term t = new Term(DocumentBuilder.CLASS_FIELDNAME,
    ➤ MergedFurniture.class.getName());
    TermQuery termQuery = new TermQuery(t);
    classFilter.add(termQuery, BooleanClause.Occur
    ➤ .SHOULD);

    Term luceneTerm = new Term("id", "1");
    Query luceneQuery = new TermQuery(luceneTerm);

    BooleanQuery filteredQuery = new BooleanQuery();
    filteredQuery.add(luceneQuery,
    ➤ BooleanClause.Occur.MUST);
    filteredQuery.add(classFilter,
    ➤ BooleanClause.Occur.MUST);

    IndexSearcher searcher = null;
    try {
        searcher = new IndexSearcher(directory);
        Hits hits = searcher.search(filteredQuery);
        assert hits.length() == 1: "Wrong hit count";

        Document doc = hits.doc(0);
        assert doc.get("color").equals("dark blue");

        for (Object o : results) session.delete(o);
        tx.commit();
    }
    finally {
        if (searcher != null)
            searcher.close();
        session.close();
    }

```

1 Returns two results from unfiltered query

2 Check only one provider exists

3 Remove CLASS_FIELDNAME scoring impact

4 Build the filter query boolean clause

5 Assemble the full query

6 Returns only one result from Filtered query

7 Check for correct color value

```

    }
}

private void buildIndex(FullTextSession session) {
    Transaction tx = session.beginTransaction();

    MergedAnimal a = new MergedAnimal();
    a.setId(1);
    a.setName("Elephant");
    session.save(a);

    MergedFurniture fur = new MergedFurniture();
    fur.setColor("dark blue");
    session.save(fur);

    tx.commit();
    session.clear();
}

@Override
protected void configure(Configuration cfg) {
    super.configure(cfg);

    cfg.setProperty("hibernate.search.default.
    ▶ directory_provider",
    ▶ FSDirectoryProvider.class.getName());
    File sub = getBaseIndexDir();

    cfg.setProperty("hibernate.search.default.
    ▶ indexBase", sub.getAbsolutePath());

    cfg.setProperty("hibernate.search.com.manning
    ▶ .hsia.ch11.Furniture.indexName", "Animal");
}
}
}

```

8 Put Furniture into the Animal index

❶ and ❷ are here solely to demonstrate that there are two results to a single query of `id=1` and there is only one `DirectoryProvider` for the merged index.

At ❸ we are canceling the effect that the `DocumentBuilder.CLASS_FIELDNAME` has on the scoring of the returned document. Remember that *Hibernate Search* adds a field by the name of `_hibernate_class` to all indexed documents. This field contains the class name of the indexed entity as its value.

NOTE As you will discover in chapter 12, having multiple documents in an index containing the same term (in this case the `id`) changes the score of a returned document. The scoring factor this affects is known as the *Inverse Document Frequency* (*idf*).

After this, ❹ builds our filter portion of the query by requiring that the `DocumentBuilder.CLASS_FIELDNAME` field should contain an instance of `org.hibernate.search.test.shard.Furniture`. This filter is combined with the other half of the query ❺ by specifying that the `id` should be 1. After the query is issued, ❻ confirms that, thanks to our restriction, there is now only one result instead of the two pointed out at ❶. ❼ corroborates that the color field does indeed contain the term

blue. ⑧ contains the configuration parameters that tell Hibernate Search to put the Furniture entity into the Animal index.

We hope you noticed that, as we said earlier, you need to do additional work here to obtain your answer. A `BooleanQuery` is now necessary because you are searching on two fields. The first is the `_hibernate_class` field, having a value of the entity you're searching for. The second is the actual field and value you're querying for, in this case `id` with a value of 1.

To find out how these two entities merged into a single index, let's turn to Luke and examine the two documents in the index. First look at figure 11.1, which shows document 0 in the index.

Several things should be apparent from this figure:

- Hibernate Search has automatically added the `<_hibernate_class>` field.
- The `<_hibernate_class>` value is `MergedAnimal`.
- The `<color>` field is marked as `<not available>` because `Animal` does not contain this field.
- This is not the document we're looking for.

Now let's look at figure 11.2, which shows the second document in the index shown in listing 11.2.

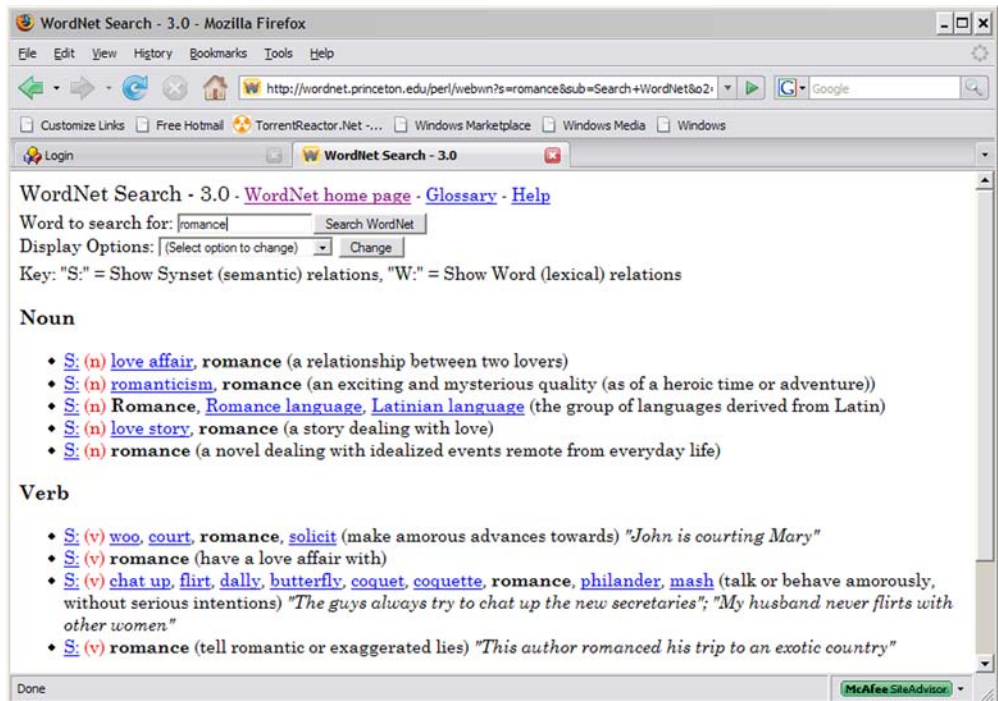


Figure 11.1 Examining document 0 of our example index. Even though this shows fields from both entities being present, if a particular entity doesn't contain one of the fields, it isn't available. Here `<color>` isn't available because `Animal` doesn't contain that field.

Several things should be apparent from this image also:

- The `<_hibernate_class>` value is Furniture.
- The `<name>` field is marked as `<not available>` since Animal does not contain this field.
- The `<color>` field is available in this document, and its value is dark blue.
- This is the document we are looking for because `<color>` contains the term blue.

These figures show that in merged indexes the individual entities are stored as is, and any fields that are not in a particular entity are ignored.

NOTE Future versions of Luke may not show the index contents exactly as shown here. The actual contents will, however, be as described.

That's enough of working with `DirectoryProviders` and `directories`. It's time to move on to our next topic concerning Hibernate Search's method of obtaining index readers. Do you remember what we talked about at the end of 11.1.2 concerning closing `IndexReaders`? We explained that because the `IndexReaders` that we instantiated in

Doc #: 1, document boost: 1.0

Flags: I - Indexed; T - Tokenized; S - Stored; V - Term Vector (o - offsets; p - positions)
O - Omit Norms; L - Lazy; B - Binary; C - Compressed

Field	ITSVop0LBC	Boost	String Value
<_hibernate_class>	I-S-----	1.0	com.manning.hsia.ch11.Furniture
<color>	ITS-----	1.0	dark blue
<id>	I-S-----	1.0	1
<name>	-----		<not available>

Figure 11.2 Examining document 1 of our example index. In this document `<name>` is not available since Furniture does not contain that field. The `<color>` field is available though, and it contains the value we're looking for, blue.

the examples were generated outside the Hibernate Search framework, we were required to explicitly close them. In the next section, we'll show you how to obtain an instance of an `IndexReader` within the confines of Hibernate Search so that the *framework* will take care of closing the reader for you.

11.2 Obtaining and using a Lucene `IndexReader` within the framework

Looking at listing 11.6 you can see that before you obtain an instance of an `IndexReader` you must first get a reference to a `ReaderProvider` instance.

Listing 11.6 Obtaining an instance of a Lucene `IndexReader`

```

DirectoryProvider orderProvider =
    searchFactory.getDirectoryProviders(Order.class)
↳ [0];
DirectoryProvider clientProvider =
    searchFactory.getDirectoryProviders(Client.class)
↳ [0];

ReaderProvider readerProvider =
    searchFactory.getReaderProvider();
IndexReader reader =
    readerProvider.openReader(orderProvider,
↳ clientProvider);
try {
    //do read-only operations on the reader
}
finally {
    readerProvider.closeReader(reader); ← Close in a
}                                       finally block

```

Obtain DirectoryProviders

Obtain a reader from a ReaderProvider

When `ReaderProviders` are created, they take into account the `ReaderStrategy` configuration property `hibernate.search.reader.strategy`, which can have any full classname of an implementation (see 9.2.2.1 for details), or use the `shared` and `not-shared` keywords, the default value being `shared`. This option is present because one of the most time-consuming operations in Lucene is opening an `IndexReader` on a particular index. If your code has many users, and the reader strategy is set to `not-shared`, this could result in performance problems depending on how often new index readers are needed. Using the `shared` `IndexReader` will make most queries much more efficient.

To solve this problem and maximize performance, Hibernate Search caches all instances of `IndexReader` when the `ReaderProvider` is set to `shared`. This means that there are some simple additional “good citizen” rules that you’ll have to follow:

- Never call `indexReader.close()` on a reader that was obtained via a `ReaderProvider`, but always call `readerProvider.closeReader(reader)`; a `finally` block is the best place to do this, as shown in listing 11.6.

- This `IndexReader` must not be used for modification operations (it's highly likely that in a future release we won't allow this to happen). If you want to use a read/write `IndexReader`, create an instance of one by calling the static `IndexReader.open(Directory directory)` method, like this:

```
DirectoryProvider[] provider =
    searchFactory.getDirectoryProviders(Order.class);
org.apache.lucene.store.Directory directory =
    provider[0].getDirectory();
IndexReader reader = IndexReader.open(directory);
```

WARNING Failure to follow these simple rules will result in clients not being able to access indexes. Worse, clients will fail at unpredictable times, making this a very difficult problem to track down. Also, and most important, if you create your own read/write `IndexReader`, you are responsible for closing it.

Aside from these rules, you can use the `IndexReader` freely.

On some occasions you may be required to furnish a custom `DirectoryProvider`. In the next section we will examine a couple of use cases from Emmanuel's customers who had specific storage requirements. This example shows what you must consider when writing a custom `DirectoryProvider`.

11.3 Writing a `DirectoryProvider` your way

It's impossible to cover even a small number of permutations with persistent storage configurations present in business today. First we'll discuss the `FSSlaveDirectoryProvider` classes' methods and what you must take into consideration for each of them. Then we'll talk about two situations that exemplify what you must take into account when writing your own `DirectoryProvider`. We won't go into detail with code, but we will discuss the use case requirements.

The `FSSlaveDirectoryProvider` is configured with these two directory settings:

- `hibernate.search.default.sourceBase` = *directory of master copy*
- `hibernate.search.default.indexBase` = *directory of local copy for querying*

These settings specify where the master index is located and also where the copies of the master index are to be placed.

Now let's investigate the `FSSlaveDirectoryProvider` class hierarchy. Examining figure 11.3 we see that there are four methods of the `DirectoryProvider` interface that it must implement:

- `void initialize(String directoryProviderName, Properties properties, SearchFactoryImplementor searchFactoryImplementor);` This method is a lightweight initialization that determines the location of the master index from the `sourceBase` setting and the slave index location(s) from the `indexBase` setting. It also determines the index name to allow the `equals` and `hashCode` to work as they must.

- `void start()` This method acquires the resources necessary for index copies. It determines the refresh period and creates the timed task for copying.
- `void stop()` This method is executed when the search factory is closed.
- `TDirectory getDirectory();` This method retrieves the appropriate directory based on the current copy operation being performed.

Figure 11.3 illustrates the class relationships.

In addition to these four method implementations, the comparator methods `equals(Object obj)` and `hashCode()` must also be implemented. These two methods are very important because they must guarantee equality between any two or more providers pointing to the same underlying Lucene store.

```
@Override
public boolean equals(Object obj) {
    if ( obj == this ) return true;
    if ( obj == null || !( obj instanceof FSSlaveDirectoryProvider ) ) return
false;
    return indexName.equals( ( (FSSlaveDirectoryProvider) obj ).indexName );
}

@Override
public int hashCode() {
    int hash = 11;
    return 37 * hash + indexName.hashCode();
}
```

Shown in figure 11.4 are the two inner classes of `FSSlaveDirectoryProvider`: `CopyDirectory` and `TriggerTask`.

It's possible that you may be required to write one or both of these classes to accomplish whatever is necessary for your implementation. The authors recommend that you examine the source code to see exactly what's going on in these classes.

Now let's take a look at the two use cases we told you about. In the first situation, Emmanuel had a client with a data storage problem, and it's quite possible that you may have experienced a similar difficulty. This customer had a Lucene index of several hundred million records. Planning estimates were that they should provide for build-out to one billion or so within the next three years.

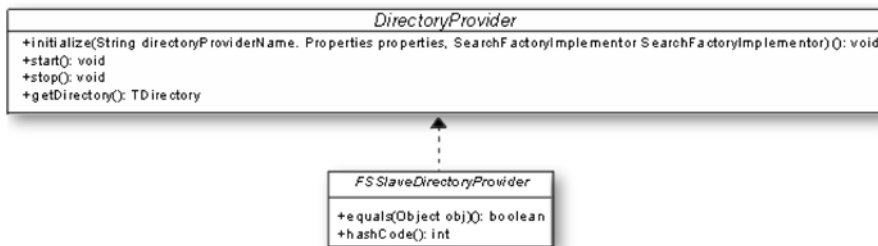


Figure 11.3 The class hierarchy diagram of `FSSlaveDirectoryProvider` shows the six methods of `DirectoryProvider` that must be implemented when writing a new implementation.

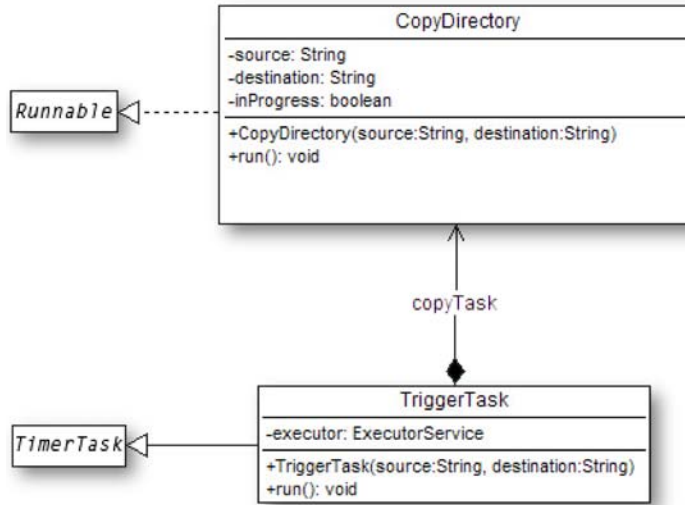


Figure 11.4 The two inner classes of `FSSlaveDirectoryProvider` showing their relationship

Problems arose when the customer revealed that the index was actually warehoused in a storage area network (SAN) that fed a 10-node server cluster. A typical SAN configuration is shown in figure 11.5. This cluster provided index query access for the user along with additional functionality. Their concern was that this would become an n -node problem. As the number of cluster nodes (n) grew, it would have been counterproductive to copy the index (`sourceBase`) from the SAN to each of the nodes of the cluster (`indexBase`). The SAN was faster, and the copying would generate much more network traffic than desired. As n increased, so would the traffic and maintenance of the copies.

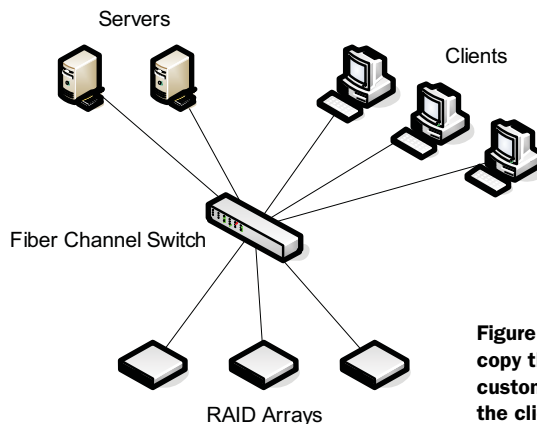


Figure 11.5 A typical SAN configuration. Rather than copy the Lucene indexes out to multiple servers, the customer wanted to keep them on the SAN and have the clients access them there.

NOTE For the following discussion, if necessary, refer to chapter 10 and specifically figure 10.2.

The customer wanted to know if it were possible to configure the `FSSlaveDirectoryProvider` to not have an `indexBase` (slave copies of the master index from the index source located on the server cluster) and configure only a `sourceBase` (index source location) to prevent the copying cycle from reaching the outlying servers. This would make the index source copy of the master index in figure 10.2 the slave index. This is possible but not out of the box. The customer would have to write its own `DirectoryProvider`. Emmanuel's recommendations were:

- Remove the actual copy to the cluster by modifying the `run` method of the inner `CopyDirectory` class and copy it to one slave index on the SAN.
- Leave the rest of the code alone: `TriggerTask`, marker determination, and so on.
- Set the refresh period to a small value to keep the slave index (the index source) up to date. This was now local I/O to the SAN and would be very fast.

Since the slave index was read-only and therefore would not have any Lucene locking issues, the slave directory could be easily shared. This prevents future scaling issues, and SANs are fast enough that they should not become bottlenecks. In the event they should, they also can be easily scaled up. We're sure that you'll probably come up with a different way of doing this—there always is—but that's up to you.

The second use case involved a user who wrote a custom `DirectoryProvider` that stores an index in a database.

The authors recommend against doing this for several reasons:

- *Lucene index segments are usually stored as blobs in a database.* Those of you who have stored data this way know that this isn't an efficient manner of storage, especially for frequent access.
- *Because of the way Lucene indexes work, a database would still need a global pessimistic lock for writes.* This does not scale well.
- *Indexes that mirror database tables and are used to increase search speed instead of using slow SQL queries are somewhat "disposable" for the following reason:* There is no strong incentive for them to be transactional because they can be fairly easily rebuilt. (No one wants their order-entry system to fail, for example, because they cannot update their order index synchronously with the database.)

Emmanuel discusses these issues with the user on the Hibernate forum at <http://forum.hibernate.org/viewtopic.php?t=980778&highlight=lucene+directory+database>. This is an informative discussion, and we recommend that you read it.

By now you have probably realized that supplying your own version of a `DirectoryProvider` is not as simple as in some situations where you provide your own implementation of a method. You may have to change or write one of the inner classes or rewrite several methods. It's not that difficult a task, but it is more than a simple method change.

As one last example of how you can access Lucene natively from Hibernate Search, we're going to work with projection in the next section. This concept comes from Hibernate and allows access to several entity-related values.

11.4 Projecting your will on indexes

Hibernate Search (and Hibernate itself for that matter) has a concept called *projection*. Utilizing a projection allows access to several underlying Lucene entity values, which are enumerated in the interface `ProjectionConstants`:

- `public String THIS` Represents the Hibernate entity itself returned in a search.
- `public String DOCUMENT` Represents the Lucene (legacy) document returned by a search.
- `public String SCORE` Represents the legacy document's score from a search.
- `public String BOOST` Represents the boost value of the legacy document.
- `public String ID` Represents the entity's id property.
- `public String DOCUMENT_ID` Represents the Lucene document id, which can change over time between two different `IndexReader` openings.
- `public String EXPLANATION` Represents Lucene's `org.apache.lucene.search.Explanation` object describing the score computation for the object/document. This feature is relatively expensive, and we recommend that you not use it unless you return a limited number of objects (using pagination). To retrieve an explanation of a single result, we recommend retrieving the explanation by using `fullTextQuery.explain(int)`. If you need to, refer to chapter 12 for a review of the `explain` method.

In addition to the items listed in `ProjectionConstants`, any of the fields contained in the indexed entity can be requested by the projection. The following code shows the `Employee` entity that we will use in the example:

```
@Entity
@Indexed
public class Employee {
    private Integer id;
    private String lastname;
    private String dept;

    public Employee() {
    }

    public Employee(Integer id, String lastname, String dept) {
        this.id = id;
        this.lastname = lastname;
        this.dept = dept;
    }

    @Id
    @DocumentId
    public Integer getId() {
```

```

        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    @Field( index = Index.NO, store = Store.YES )
    public String getLastname() {
        return lastname;
    }

    public void setLastname(String lastname) {
        this.lastname = lastname;
    }

    @Field( index = Index.TOKENIZED, store = Store.YES )
    public String getDept() {
        return dept;
    }

    public void setDept(String dept) {
        this.dept = dept;
    }
}

```

Listing 11.7 demonstrates the use of a Hibernate Search projection. As explained in the note at the beginning of this chapter, the scaffolding code principle still holds for this example. In this case, we examine the tests in the `org.hibernate.search.test.query` package.

Listing 11.7 Utilizing a Hibernate Search projection to retrieve values

```

public class ProjectionQueryTest extends SearchTestCase {
    FullTextSession session;
    Transaction tx;

    public void testLuceneObjectsProjectionWithScroll() throws Exception {
        session = Search.getFullTextSession(openSession());
        buildIndex();

        tx = session.beginTransaction();
        QueryParser parser =
            new QueryParser("dept", new StandardAnalyzer());

        Query query = parser.parse("dept:ITech");
        org.hibernate.search.FullTextQuery hibQuery =
            session.createFullTextQuery(query, Employee.class);
        hibQuery.setProjection("id", "lastname", "dept",
            FullTextQuery.THIS, FullTextQuery.SCORE,
            FullTextQuery.DOCUMENT, FullTextQuery.ID);

        ScrollableResults projections = hibQuery.scroll();
        projections.beforeFirst();
        projections.next();
        Object[] projection = projections.get();
    }
}

```

1 Applying a projection to the query

```

assert (Integer)projection[0] == 1000
    : "id incorrect";
assert ((String) projection[1]).equals("Griffin")
    : "lastname incorrect";
assert ((String)projection[2]).equals("ITech")
    : "dept incorrect";

assert session.get(Employee.class,
    (Serializable) projection[0])
    .equals(projection[3]): "THIS incorrect";

assertEquals("SCORE incorrect", 1.0F, projection[4]);
assert (Float)projection[4] == 1.0F
    : "SCORE incorrect";
assert projection[5] instanceof Document
    : "DOCUMENT incorrect";
assert ((Document) projection[5]).getFields()
    .size() == 4: "DOCUMENT size incorrect";
assert (Integer)projection[6] == 1000
    : "legacy ID incorrect";
assert projections.isFirst();

assert ((Employee) projection[3])
    .getId() == 1000: "Incorrect entity returned";

for (Object element : session.createQuery("from "
    + Employee.class.getName()).list())
    session.delete(element);
tx.commit();
}
finally {
    session.close();
}

private void buildIndex() {
    Transaction tx = session.beginTransaction();
    Employee e1 =
        new Employee(1000, "Griffin", "ITech");
    session.save(e1);

    Employee e2 =
        new Employee(1001, "Jackson", "Accounting");
    session.save(e2);
    tx.commit();

    session.clear();
}
}

```

2 Assertions on entity fields

3 Assertions on ProjectionConstants values

4 Checking a value of the returned entity

① applies a projection to our "dept:ITech" query. The results of the test query are then checked: ② checks the id, lastname, and dept fields from the `Employee` entity, ③ checks the validity of the requested `ProjectionConstants` values, and ④ retrieves the `FullTextQuery.THIS` entity of `ProjectionConstants` and validates that the lastname field corresponds to the correct value.

Projections are one of the performance improvements that we recommend. You should not make a round-trip to the database to retrieve data if you can retrieve it from the index. Lucene is incredibly fast at locating data, and that's where a projection comes in. It allows you to specify exactly the data you want to examine and skip loading any of the data from the database you're not interested in. This can increase performance, but as usual, you need to test to be sure.

Native Lucene emulates this process through a combination of the `org.apache.lucene.document.MapFieldSelector` and the `org.apache.lucene.document.FieldSelectorResult` classes, to prevent loading data that you're not interested in.

11.5 Summary

You've seen that accessing Lucene natively is a straightforward task as long as you have an instance of the `SearchFactory` object to start things off. Once you have an instance of this class, you can get references to `Lucene Directorys` and `DirectoryProvider(s)`.

These classes make creating `IndexReaders` and `ReaderProviders` a straightforward operation, although section 11.2 lists some rules you will have to live by to utilize them safely. Sometimes, but not often, it's necessary to write your own `DirectoryProvider`. We examined that process and found that it may be necessary to provide overriding implementations of the following:

- `void initialize(String directoryProviderName, Properties properties, SearchFactoryImplementor searchFactoryImplementor);`
- `void start();`
- `void stop();`
- `TDirectory getDirectory();`

The `equals` and `hashCode` methods must also be overridden, but these are certainly not insurmountable tasks.

Remember, implementing projections can increase result-retrieval performance. We recommend you test for performance gains and use them as you see fit.

Next we're going to talk about a subject that's a little more difficult to grasp but not overly so: document scoring and changing how it is calculated so that it fits our needs. This is a lengthy topic, but we're sure that when you finish it you'll understand a lot more about Lucene and information retrieval in general.

HIBERNATE SEARCH IN ACTION

Emmanuel Bernard and John Griffin

Lucene 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

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: 978-1-93398864-1
ISBN-10: 1-93398864-9



9 781933 988641



MANNING

US/CAN \$49.99 [INCLUDING EBOOK]