DZone Refcardz

Brought to you by... JBoss by Red Hat

CONTENTS INCLUDE:

Getting Started with Hibernate Search

By John Griffin

GOOGLE YOUR DATABASE!

Hibernate Search complements Hibernate Core by enabling full-text search queries on persistent domain models, and brings Lucene search features to the Hibernate world. Hibernate Search depends on Apache Lucene, a powerful full-text search engine library (and a de facto standard solution in Java) hosted at the Apache Software Foundation (http://www.apache.org/). This refcard explains installation and configuration, and covers Mapping entities, bridges, building indexes, querying them and examining their contents. Table 1 shows links to documentation.

Topic	URL
Lucene	http://lucene.apache.org/java/docs/
Hibernate Search	http://www.hibernate.org/410.html
Mailing lists	http://www.hibernate.org/20.html
JIRA	http://opensource.atlassian.com/projects/hibernate/secure/Dashboard.jspa

Table 1 Documentation Links

GETTING STARTED

In order to use Hibernate Search you should understand the basics of Hibernate, and be familiar with the object manipulation APIs from the Hibernate Session or the Java Persistence EntityManager as well as the query APIs. You should also be familiar with association mappings and the concept of bidirectional relationships.

Download Hibernate Search at http://www.hibernate.org or use the JBoss Maven repository (http://repository.jboss.org/maven2/org/hibernate/hibernate-search). It is interesting to download the Apache Lucene distribution as well, available at http://lucene.apache.org/java/. It contains both documentation and a contribution section containing various add-ons not included in Hibernate Search. Make sure to use the same Lucene version Hibernate Search is based on. You can find the correct version in the Hibernate Search distribution in lib/readme.txt.

Hibernate Search requires three JARs – all available in the Hibernate Search distribution:

- 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

You can also add the optional support for modular analyzers by adding: apache-solr-analyzer.jar. This JAR (available in the Hibernate Search distribution), is a subset of the SOLR distribution that contains various analyzers. While optional, it is recommended to add this JAR to your classpath as it greatly simplifies the use of analyzers.

Hot Tip The apache-solr-analyzer.jar capabilities are only available in Hibernate Search 3.1+.

Hibernate Search is not compatible with all versions of Hibernate Core and Hibernate Annotations. Refer to Table 2 for compatibility requirements. The latest version is available on the Hibernate download page at http://www.hibernate.org/6.html.

Package	Version	Core	Annotations	Entity Manager	Search
Hibernate Core	3.2.6 GA		3.2.x, 3.3.x	3.2.x, 3.3.x	3.0.x
	3.3.0 SP1		3.4.x	3.4.x	3.1.x
Hibernate Annotations	3.3.1 GA	3.2.x		3.3.x	3.0.x
	3.4.0 GA	3.3.x		3.4.x	3.1.x
Hibernate EntityManager	3.3.2 GA	3.2.x	3.3.x		3.0.x
	3.4.0 GA	3.3.x	3.4.x		3.1.x
Hibernate Validator	3.0.0 GA	3.2.x	3.3.x	3.3.x	3.0.x
	3.1.0 GA	3.3.x	3.4.x	3.4.x	3.1.x
Hibernate Search	3.0.1 GA	>= 3.2.2 (better if >= 3.2.6)	3.3.x (better if >= 3.3.1)	3.3.x	
	3.1.0 Beta1	3.3	3.4	3.4	-
Hibernate Shards	3.0.0 Beta2	3.2.x	3.3.x	Not compatible	3.0.x
Hibernate Tools	3.2.2	3.2.x	3.2.x and 3.3.x	3.2.x and 3.3.x	(3.2.0)

Table 2: Compatibility Matrix.

Hot Tip Dependencies needed to build and initially test Hibernate Search are included in the Hibernate Search distribution or can be found in the Maven dependency file (POM) which is included with the Hibernate Search download.

Configuration Parameters

Configuration parameters can be provided in three ways:

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

Listing 1: An example hibernate.cfg.xml file.

```
<?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 -->
   <property name="hibernate.search.default.indexBase">
      /users/application/indexes
   </property>

   <!-- mapping classes -->
   <mapping class="com.manning.dvdstore.model.Item"/>
   list additional entities
   </session-factory>
</hibernate-configuration>
```

Parameter	Description
`hibernate.search.autoregister_listeners`	Enable listeners auto registration in Hibernate Annotations and Entity-Manager. Default to *true.*
`hibernate.search.indexing_strategy`	Defines the indexing strategy, default to event. Other option is *manual.*
`hibernate.search.analyzer`	The default Lucene analyzer class.
`hibernate.search.similarity`	The default Lucene similarity class.
`hibernate.search.worker.batch_size`	Has been deprecated in favor of this explicit API
`hibernate.search.worker.backend`	Out of the box support for the Apache Lucene backend and the JMS back end. Defaults to *lucene*. Other option is *jms.*
`hibernate.search.worker.execution`	Supports synchronous and asynchronous execution. Defaults to sync. Other option is *async.*
`hibernate.search.worker.thread_pool.size`	Defines the number of threads in the pool. Useful only for asynchronous execution. Default to *1.*
`hibernate.search.worker.buffer_queue.max`	Defines the maximal number of work queue if the thread poll is starved. Useful only for asynchronous execution. Default to *infinite.* If the limit is reached, the work is done by the main thread.
`hibernate.search.worker.jndi.*`	Defines the JNDI properties to initiate the InitialContext (if needed). JNDI is only used by the JMS back end.
`hibernate.search.worker.jms.connection_factory`	Mandatory for the JMS back end. Defines the JNDI name to lookup the JMS con-nection factory from. (`java:/ConnectionFactory` by default in JBoss AS)
`hibernate.search.worker.jms.queue`	Mandatory for the JMS back end. Defines the JNDI name to lookup the JMS queue from. The queue will be used to post work messages.
`hibernate.search.reader.strategy`	Defines the reader strategy used. Defaults to shared. Other option is *not-shared.*

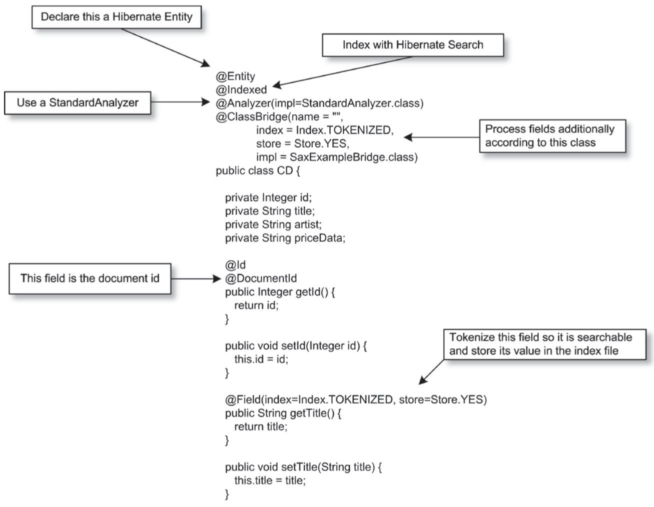
Table 3: Hibernate Search configuration parameters.

Table 3, continued

`hibernate.search.filter.cache_strategy`	The filter caching strategy class (must have a no-arg constructor and implement `FilterCachingStrategy).`
`hibernate.search.filter.cache_bit_results.size`	The hard ref count of our `Caching WrapperFilter.` Defaults to 5.

Table 3: Hibernate Search configuration parameters, continued

MAPPING ENTITIES


Declare this a Hibernate Entity
Index with Hibernate Search
Use a StandardAnalyzer
Process fields additionally according to this class
This field is the document id
Tokenize this field so it is searchable and store its value in the index file

```
@Entity
@Indexed
@Analyzer(impl=StandardAnalyzer.class)
@ClassBridge(name = "",
         index = Index.TOKENIZED,
         store = Store.YES,
         impl = SaxExampleBridge.class)
public class CD {

  private Integer id;
  private String title;
  private String artist;
  private String priceData;

  @Id
  @DocumentId
  public Integer getId() {
    return id;
  }

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

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

  public void setTitle(String title) {
    this.title = title;
  }
```

Figure 1: Basic entity mapping.

BRIDGES

Bridges fulfill several needs in the Hibernate Search architecture.

- Converts an object instance into a Lucene consumable representation (commonly a String) and adds it to a Lucene document.
- Reads information from the Lucene document and builds back the object representation.

Bridges that support both the conversion object to Lucene and Lucene to object are called two-way bridges. Table 4 lists all out-of-the-box Hibernate Search bridges.

Java Type	Build-in Bridge	Description
String	`StringBridge`	no-op
short / Short	`ShortBridge`	Use toString(), not comparable
int / Integer	`IntegerBridge`	Use toString(), not comparable
long / Long	`LongBridge`	Use toString(), not comparable
float / Float	`FloatBridge`	Use toString(), not comparable
double / Double	`DoubleBridge`	Use toString(), not comparable
BigDecimal	`BigDecimalBridge`	Use toString(), not comparable
BigInteger	`BigIntegerBridge`	Use toString(), not comparable
boolean / Boolean	`BooleanBridge`	String value: "true" / "false"
Class	`ClassBridge`	Allows manipulation of any combination of different fields.
Enum	`EnumBridge`	Use enum.name()
URL	`UrlBridge`	Converts to the String representation
URI	`UriBridge`	Converts to the String representation

Table 4: List of standard Hibernate Search bridges.

Bridges, continued

Date	`DateBridge`	The string representation depends on `@DateBridge`. Converting Date into string and back is not guaranteed to be idempotent

Table 4: List of standard Hibernate Search bridges, continued.

Custom bridges allow for converting unexpected data types. The `@FieldBridge` annotation is placed on a property (field or getter) that needs to be processed by a custom bridge. An example, including parameter passing, is given in Listing 2.

Listing 2: A custom bridge example with parameters.

```
@Entity
@Indexed
public class Item {
   @Field
   // property marked to use a custom bridge
   @FieldBridge(
      // declare the custom bridge implementation
      impl=PaddedRoundedPriceBridge.class,
      // optionally provide parameters
      params= {
         @Parameter(name="pad", value="3"),
         @Parameter(name="round", value="5") }
)
private double price;
...
}
```

Embeddable Objects

Embedded objects in Java Persistence (they are called components in Hibernate) are objects whose life cycle entirely depends on the owning entity. When the owning entity is deleted, the embedded object is deleted as well.

Listing 3: Embeddable Object example.

```
@Embeddable
public class Rating {
   // mark properties for indexing
   @Field(index=Index.UN_TOKENIZED) private Integer overall;
   @Field(index=Index.UN_TOKENIZED) private Integer scenario;
   @Field(index=Index.UN_TOKENIZED) private Integer soundtrack;
   @Field(index=Index.UN_TOKENIZED) private Integer picture;
...
}

@Entity
@Indexed
public class Item {
   // mark the association for indexing
   @IndexedEmbedded private Rating rating;
...
}
```

`@IndexedEmbedded` marks the association as embedded: the Lucene document contains `rating.overall, rating, scenario, rating.soundtrack, rating.picture`. When `Item` is deleted the embedded `Rating` object is also deleted.

Associations

Associations between objects are similar to embeddable objects except that an associated object's life time is not dependent on the owning entity. Below is an example association mapping.

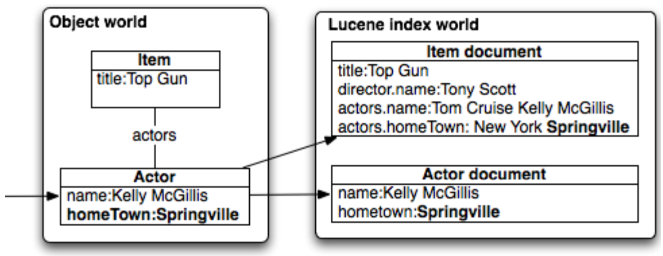


Figure 2: An example association.

Associations, continued

Listing 4 shows that `@ContainedIn` is paired to an `@IndexedEmbedded` annotation on the other side of the bi-directional relationship.

Listing 4: Figure 2 in code.

```
@Entity @Indexed
public class Item {
   @ManyToMany
   @IndexedEmbedded
   private Set<Actor> actors; // embed actors when indexing

   @ManyToOne
   @IndexedEmbedded
   private Director director; // embed director when indexing
...
}

@Entity @Indexed
public class Actor {
   @Field private String name;
   @ManyToMany(mappedBy="actors")
   @ContainedIn actor is contained in item index (1)
   private Set<Item> items;
...
}

@Entity @Indexed
public class Director {
   @Id @GeneratedValue @DocumentId private Integer id;
   @Field private String name;
   @OneToMany(mappedBy="director")
   @ContainedIn director is contained in item index
   private Set<Item> items;
...
}
```

Hot Tip The `@IndexEmbedded` depth setting (e.g. `@IndexEmbedded(depth=3)`) controls the maximum number of embeddings allowed per association.

ANALYZERS

Analyzers are responsible for taking text as input, chunking it into individual words (tokens) and optionally applying some operations (filters) on the tokens. A filter can alter the stream of tokens as it pleases. It can remove, change, and add words.

In addition to the SOLR analyzers mentioned previously, Lucene's `org.apache.lucene.analysis` package contains additional analyzers and many filters. Listing 5 is an example of defining an analyzer on an entity.

Listing 5

```
@Entity @Indexed
@AnalyzerDef(
   name="applicationanalyzer", // analyzer definition name
   tokenizer =
      // tokenizer factory
    @TokenizerDef(factory = StandardTokenizerFactory.
    class ),
      filters = {
         // list of filters to apply
         @TokenFilterDef(factory=LowerCaseFilterFactory.
         class),
         @TokenFilterDef(factory = StopFilterFactory.
         class,
         // parameters passed to the filter factory
         params = {
            @Parameter(name="words",
               value="com/manning/hsia/dvdstore/
               stopwords.txt"),
            @Parameter(name="ignoreCase", value="true")
        } )
} )
// Use the pre defined analyzer
@Analyzer(definition="applicationanalyzer")
public class Item {
...
}
```

INITIAL INDEXING OF ENTITIES

Manually

First you need an instance of either a `FullTextEntityManager` or a `FullTextSession` depending on whether or not you are using an `EntityManager`.

Listing 6: Manually indexing data.

```
SessionFactory factory =
   new AnnotationConfiguration().buildSessionFactory();
Session session = factory.openSession();

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

fts.getTransaction.begin()
for (Item item : items) {
   fts.index(item); //manually index an item instance
}
fts.getTransaction().commit(); //index is written at commit time
session.close();
or
EntityManagerFactory factory =
   Persistence.createEntityManagerFactory("…");
EntityManager em = factory.createEntityManager();
FullTextEntityManager ftem =
  org.hibernate.search.jpa.Search.getFullTextEntityManager(em);

ftem.getTransaction().begin();
for (Item item : items) {
   ftem.index(item); //manually index an item instance
}
//index is written at commit time
ftem.getTransaction().commit();
```

Hot Tip

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

From a dataset

Listing 7: Initial indexing of a dataset.

```
// disable flush operations
session.setFlushMode(FlushMode.MANUAL);
// disable 2nd level cache operations
session.setCacheMode(CacheMode.IGNORE);

Transaction tx = session.beginTransaction();
// read the data from the database
// scrollable results will avoid loading too many objects
// in memory
// ensure forward only result set
ScrollableResults results = session.createCriteria(Item.class)
   .scroll( ScrollMode.FORWARD_ONLY );

int index = 0;
while( results.next() ) {
   index++;
   session.index( results.get(0) ); index entities (4)
   if (index % BATCH_SIZE == 0) {
      session.flushToIndexes(); apply changes to the index (5)
      session.clear(); clear the session releasing memory
   }
}
tx.commit(); apply the remaining index changes
```

Updates, additions and deletions to indexes are handled automatically by Hibernate Search via entity listeners. If you are using Hibernate Annotations these listeners are automatically wired for you. If you are not using the annotations then you have to wire the listeners manually as shown in Listing 8.

From a dataset, continued

Listing 8: Wiring listeners when not using 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>
```

For versions of Hibernate Search prior to 3.1.x the configuration is slightly different as shown in Listing 9.

Listing 9: Wiring listeners prior to Hibernate Search version 3.1.x.

```
<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>
      <!-- collection listener is different -->
      <event type="post-collection-recreate">
         <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>
```

QUERYING INDEXES

Table 5 shows the three ways to obtain results.

Method Call	Description
`query.list()`	`List<Item> items = query.list();` All matching objects are loaded eagerly as opposed to lazily.

Table 5: Querying indexes

Querying indexes, continued

query.iterate()	`Iterator<Item> items = query.iterate(); while ( items.hasNext() ) { Item item = items.next(); }` All object identifiers are extracted from the Lucene index but objects are not loaded until `iterator.next()` is called
query.scroll()	`ScrollableResults items = query.scroll(); // process results Items.close();` `ScrollableResults` must be closed when processing is finished to free resources.

Table 5: Querying indexes, continued.

Listing 10: A FullTextQuery example.

```
SessionFactory factory =
   new AnnotationConfiguration().buildSessionFactory();
Session session = factory.openSession();

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

fts.getTransaction.begin()

// create a Term for the description field
Term term = new Term("description", "salesman");
TermQuery query = new TermQuery(term);

// generate a FullTextQuery and obtain a result list
org.hibernate.search.FullTextQuery hibQuery =
   s.createFullTextQuery(query, Dvd.class);
List<Dvd> results = hibQuery.list();
```

Basic Query Types

Table 6 presents the basic query types. Consult the Lucene API documentation at http://lucene.apache.org/java/2_4_0/api/index.html for a complete listing, specifically the `org.apache.lucene.search` package.

Query	Description
TermQuery	This is the basic building block of queries. It searches for a single term in a single field. Many other query types are reduced to one or more of these.
WildcardQuery	Queries with the help of two wildcard symbols '*' (multiple characters) and '?' (single character). These wildcard symbols allow queries to match any combination of characters.
PrefixQuery	A `WildcardQuery` that starts with characters and ends with the '*' symbol.
PhraseQuery	Also known as a proximity search, this queries for multiple terms enclosed by quotes.
FuzzyQuery	Queries using the Levenshtein distance between terms. Requires a minimum similarity float value that expands or contracts the distance.
RangeQuery	Allows you to search for results between two values. Values can be inclusive or exclusive but not mixed.
BooleanQuery	Holds every possible combination of any of the other query types including other `BooleanQuerys`. Boolean queries combine individual queries as SHOULD, MUST or MUST_NOT.
MatchAllDocsQuery	Returns all documents contained in a specified index.

Table 6: Basic query types.

HIBERNATE SEARCH ANNOTATIONS

Table 7 is a complete listing of all Hibernate Search annotations.

Annotation	Description
@Analyzer	Define an Analyzer for a given entity, method, attribute or Field. The order of precedence is: `@Field`, attribute/ method, entity, default. Able to reference an implementation or an `@AnalyzerDef` definition.
@AnalyzerDef	Reusable analyzer definition. An analyzer definition defines: one tokenizer and, optionally, some filters. Filters are applied in the order they are defined.

Table 7: Hibernate Search Annotations.

Hibernate Search Annotations, continued

@AnalyzerDefs	Reusable analyzer definitions. Allows multiple `@AnalyzerDef` declarations per element.
@Boost	Apply a boost factor to a field or an entire entity.
@ClassBridge	Allows a user to manipulate a Lucene document based on an entity change in any manner the user wishes.
@ClassBridges	Allows multiple `@ClassBridge` declarations per document.
@ContainedIn	Marks the owning entity as being part of the associated entity's index (to be more accurate, being part of the indexed object graph). This is only necessary when an entity is used as a `@IndexedEmbedded` target class. `@ContainedIn` must mark the property pointing back to the `@IndexedEmbedded` owning Entity. Not necessary if the class is an embeddable class.
@DateBridge	Defines the temporal resolution of a given property. Dates are stored as a String in GMT.
@DocumentId	Declare a property as the document id.
@Factory	Marks a method of a filter factory class as a Filter implementation provider. A factory method is called whenever a new instance of a filter is requested.
@Field	Marks a property as indexed. Contains field options for storage, tokenization, whether or not to store TermVector information, a specific analyzer and a Field-Bridge.
@FieldBridge	Specifies a field bridge implementation class. A field bridge converts (sometimes back and forth) a property value into a string representation or a representation stored in the Lucene Document.
@Fields	Marks a property as indexed into different fields. Useful if the field is used for sorting and searching or if different analyzers are used.
@FullTextFilterDef	Defines a full-text filter that can be optionally applied to full-text queries. While not related to a specific indexed entity, the annotation must be set on one of them.
@FullTextFilterDefs	Allows multiple `@FullTextFilterDef` per FullTextQuery.
@Indexed	Specifies that an entity is to be indexed. The index name defaulted to the fully qualified class name can be overridden using the name attribute.
@IndexedEmbedded	Specifies that an association (`@*To*`, `@Embedded`, `@CollectionOfEmbedded`) is to be indexed in the root entity index. It allows queries involving associated objects restrictions.
@Key	Marks a method of a filter factory class as a Filter key provider. A key is an object that uniquely identifies a filter instance associated with a given set of parameters. The key object must implement equals and hashcode so that 2 keys are equals if and only if the given target object types are the same and the set of parameters are the same. The key object is used in the filter cache implementation.
@Parameter	Basically a key/value descriptor. Used in `@ClassBridge`, `@FieldBridge`, `TokenFilterDef` and `@TokenizerDef`.
@ProvidedId	Objects whose identifier is provided externally, as opposed to being a part of the object state, should be marked with this annotation. This annotation should not be used in conjunction with `@DocumentId`. This annotation is primarily used in the JBoss Cache Searchable project. http://www.jboss.org/jbosscache and http://wiki.jboss.org/wiki/JBossCacheSearchable

@Similarity	Specifies a similarity implementation to use in scoring calculations.
	
	Ex. `@Entity`
	`@Indexed`
	`@Similarity(impl = BookSpecificSimilarity.public class Book {`
	`...`
	`}`

@TokenFilterDef	Specifies a `TokenFilterFactory` and its parameters inside a `@AnalyzerDef.`
@TokenizerDef	Defines a `TokenizerFactory` and its parameters inside a `@AnalyzerDef`

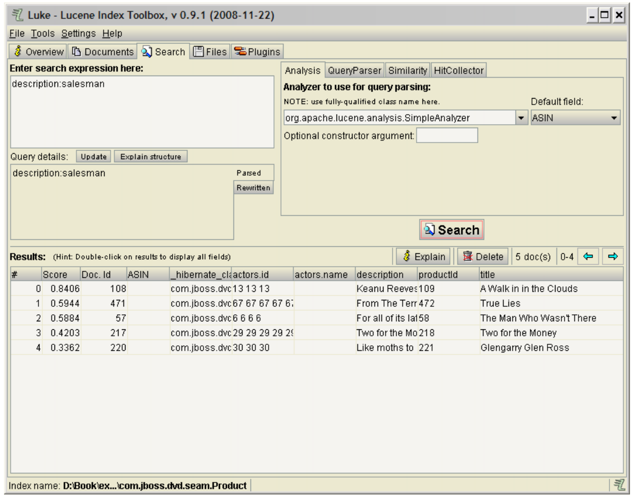
Table 7: Hibernate Search Annotations, continued

LUKE

The most indispensable utility you can have in your arsenal of index troubleshooting tools (in fact it may be the *only* one you really need) is Luke, shown in Figure 3. With Luke you can examine any facet of an index you can imagine. Some of its capabilities are:

- view individual documents
- execute a search, and browse the results
- selectively delete documents from the index
- examine term frequency, and many more...

The Luke author, Andrzej Bialecki, actively maintains Luke to keep up with the latest Lucene version. Luke is available for download, in several different formats, at http://www.getopt.org/luke/. The most current version of the Java WebStart JNLP direct download is the easiest to retrieve.


Luke - Lucene Index Toolbox, v 0.9.1 (2008-11-22)
File Tools Settings Help
Overview Documents Search Files Plugins
Analysis QueryParser Similarity HitCollector
Update Explain structure
Parsed
Rewritten
Search
Explain Delete

Enter search expression here: description:salesman

Analyzer to use for query parsing:
NOTE: use fully-qualified class name here.
org.apache.lucene.analysis.SimpleAnalyzer
Default field: ASIN
Optional constructor argument:

Query details: description:salesman

Results: (Hint: Double-click on results to display all fields) 5 doc(s) 0-4

#	Score	Doc. Id	ASIN	_hibernate_cl	actors.id	actors.name	description	productId	title
0	0.8406	108		com.jboss.dvc	13 13 13		Keanu Reeves	109	A Walk in in the Clouds
1	0.5944	471		com.jboss.dvc	67 67 67 67 67		From The Terr	472	True Lies
2	0.5884	57		com.jboss.dvc	6 6 6 6		For all of its lat	58	The Man Who Wasn't There
3	0.4203	217		com.jboss.dvc	29 29 29 29 29		Two for the Mo	218	Two for the Money
4	0.3362	220		com.jboss.dvc	30 30 30		Like moths to	221	Glengarry Glen Ross

Index name: D:\Book\ex...\com.jboss.dvd.seam.Product

Figure 3: The search window of the Luke utility for Lucene indexes.

ABOUT THE AUTHOR

John Griffin

John Griffin has been in the software and computer industry in one form or another since 1969. He remembers writing his first FORTRAN IV program on his way back from Woodstock. Currently, he is the software engineer/architect for SOS Staffing Services, Inc. He was formerly the lead e-commerce architect for Iomega Corporation and an independent consultant for the Dept. of the Interior, among many other callings. John is a member of the ACM. Currently, he resides in Layton, Utah with wife Judy and Australian Shepards Clancy and Molly.

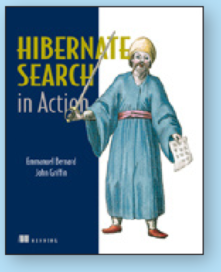
Publications

- *XML and SQL Server 2000*, New Riders Press
- *Hibernate Search in Action*, Manning Publications, co-authored with Emmanuel Bernard

Blog

http://thediningphilosopher.blogspot.com

RECOMMENDED BOOK



Hibernate Search In Action guides you through every step to set up full text search functionality in your Java applications, and provides a pragmatic, how-to exploration of more advanced topics such as Search clustering.

BUY NOW
books.dzone.com/books/hibernate-search

DZone, Inc.
1251 NW Maynard
Cary, NC 27513

888.678.0399
919.678.0300

Refcardz Feedback Welcome
refcardz@dzone.com

Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-34-9
ISBN-10: 1-934238-34-1
50795
9 781934 238349
$7.95

Version 1.0