Hibernate Search with Analyzer

**What is a Hibernate Search?**

Full text search engines like Apache Lucene are very powerful technologies to add efficient free text search capabilities to applications. However, Lucene suffers several mismatches when dealing with object domain models. Amongst other things indexes must be kept up to date and mismatches between index structure and domain model as well as query mismatches must be avoided.

Hibernate Search addresses these shortcomings - it indexes your domain model with the help of a few annotations, takes care of database/index synchronization and brings back regular managed objects from free text queries. To achieve this, Hibernate Search is combining the power of Hibernate and Apache Lucene.

**Why, Search?**

Apache Lucene is a high-performance, extensible full-text search-engine library written in Java. Despite these similarities, the differences between Hibernate/JPA and Lucene are also obvious.

Hibernate/JPA promotes domain model-oriented programming by encouraging developers to work out a rich domain-object graph that naturally represents the complexities of the real-world business through object association, inheritance, polymorphism, composition, and collections.

Hibernate Search comes up with handling POJOs. AS O-R mapping solution, hibernate is one of the best ORM tools available. Along with its Search, making use of Apache Lucene with managing indexing automatically & updating the same is taken care by hibernate search only.

**Hibernate Search?**

These are important areas should be taken care before using Hibernate Search:

1. **Required Libraries**
2. **Hibernate Search Properties**
3. **Hibernate Search Annotations**
4. **Indexing**
5. **Searching**
6. **Analyzer**
7. **Required Libraries**

The Hibernate Search library is split in several modules to allow you to pick the minimal set of dependencies you need. It requires Apache Lucene, Hibernate ORM and some standard APIs such as the Java Persistence API and the Java Transactions API.

* Hibernate Search 5.5.5.Final
* Hibernate ORM (and EntityManager) 5.0.11.Final
* Apache Lucene 5.3.1
* Hibernate Common Annotations 5.0.1.Final

1. **Hibernate Search Properties**

No configuration properties are required to get started: Hibernate Search will automatically integrate with your Hibernate ORM Session(s) or EntityManager(s) when it is found on the classpath. However, these are few important properties:

hibernate.search.default.directory\_provider **filesystem**; the default index manager uses Lucene's notion of a Directory to store the index files.

hibernate.search.default.indexBase **var/hiber/index**; specify the default base directory for all indexes, This defines the path where indexes are stored.

hibernate.search.error\_handler Set to the fully qualified classname of a custom “**ErrorHandler**” implementation to handle all indexing errors.

hibernate.search.default.worker.execution “**sync**” is the default. Set to “**async**” to perform the index write operations in background.

hibernate.search.default.index\_flush\_interval The interval (in ms) between flushes of write operations to the index. Only applies if worker.execution is configured as async.

1. **Hibernate Search Annotations**

To make hibernate full text search working, Entities need to be indexed, as above properties or behavior. To make an entity indexable and fields or column searchable, hibernate search has provided few annotations, which can be applied to entity classes as well. These are:

@Indexed - This annotation marks an Entity as indexable for hibernate search. By design Hibernate Search needs to store an untokenized id in the index to ensure index unicity for a given entity.

@Field - Next you have to mark the fields you want to make searchable. Let’s start with title and subtitle and annotate both with @Field. Annotation used for marking a property as indexable. The default parameters & its value are :

@Field( index=Index.YES, analyze=Analyze.YES and store=Store.NO)

index=Index.YES : will ensure that the text will be indexed, while analyze=Analyze.YES : ensures that the text will be analyzed using the default Lucene analyzer. Usually, analyzing or tokenizing means chunking a sentence into individual words and potentially excluding common words like "a" or "the". store=Store.NO : which ensures that the actual data will not be stored in the index. Whether data is stored in the index or not has nothing to do with the ability to search for it.

@DocumentId - marks the property to use for this purpose and is in most cases the same as the database primary key. The @DocumentId annotation is optional in the case where an @Id annotation exists.

@DateBridge - This annotation is one of the built-in field bridges in Hibernate Search. The Lucene index is purely string based. For this reason Hibernate Search must convert the data types of the indexed fields to strings and vice versa. A range of predefined bridges are provided, including the DateBridge which will convert a java.util.Date into a String with the specified resolution.

For Example, we can consider the following BOOK Entity which will carry these annotations to make hibernate search enable & searchable the properties:

@Entity

@Indexed

public class Book {

@Id

@GeneratedValue

private Integer id;

@Field(index = Index.YES, analyze = Analyze.YES, store = Store.NO)

private String title;

@Field(index = Index.YES, analyze = Analyze.YES, store = Store.NO)

private String subtitle;

@Field(index = Index.YES, analyze = Analyze.NO, store = Store.YES)

@DateBridge(resolution = Resolution.DAY)

private Date publicationDate;

//setters & getters

}

1. **Indexing**

Hibernate Search will transparently index every entity persisted, updated or removed through Hibernate ORM. However, you must create an initial Lucene index for the data already present in your database. Once you have added the above properties and annotations it is time to trigger an initial batch index of your books. You can achieve this by using one of the following code snippets:

Using *Hibernate Session* to index data

FullTextSession fullTextSession = Search.getFullTextSession(session);

fullTextSession.createIndexer().startAndWait();

Using *JPA* to index data

EntityManager em = entityManagerFactory.createEntityManager();

FullTextEntityManager fullTextEntityManager = Search.getFullTextEntityManager(em);

fullTextEntityManager.createIndexer().startAndWait();

**A close up of a device

Description generated with high confidence**

**Automatic indexing**

By default, every time an object is inserted, updated or deleted through Hibernate, Hibernate Search updates the according Lucene index. It is sometimes desirable to disable that features if either your index is read-only or if index updates are done in a batch way.

To disable event based indexing, set hibernate.search.indexing\_strategy = manual

**Directory configuration**

Apache Lucene has a notion of a Directory to store the index files. After executing the above code, you should be able to see a Lucene index under based on path depending how you configured the property hibernate.search.default.directory\_provider under root project directory.  
hibernate.search.default.indexBase will specify the base directory to store the index files. Some built-in directory providers are: *ram, filesystem, filesystem-master, filesystem-slave, infinispan*.

Infinispan is a distributed, scalable, highly available data grid platform which supports autodiscovery of peer nodes. Using Infinispan and Hibernate Search in combination, it is possible to store the Lucene index in a distributed environment where index updates are quickly available on all nodes.

**Exception Handling Configuration**

Hibernate Search allows you to configure how exceptions are handled during the indexing process. If no configuration is provided then exceptions are logged to the log output by default. It is possible to explicitly declare the exception logging mechanism as seen below:

hibernate.search.error\_handler CustomerErrorHandler [fully qualified class name]

In order to provide your own implementation you must implement the ErrorHandler interface, which provides the handle(ErrorContext context) method.

1. **Searching**

Preparing and executing a query consists of four simple steps:

* Creating a FullTextSession
* Creating a Lucene query either via the Hibernate Search query DSL (recommended) or by utilizing the Lucene query API
* Wrapping the Lucene query using an org.hibernate.Query
* Executing the search by calling for example list() or scroll()

To execute a first search, the general approach is to create a Lucene query, either via the Lucene API (Building a Lucene query using the Lucene API) or via the Hibernate Search query DSL (Building a Lucene query with the Hibernate Search query DSL), and then wrap this query into a org.hibernate.Query in order to get all the functionality one is used to from the Hibernate API. The following code will prepare a query against the indexed fields, execute it and return a list of TestBO instances.

**//Creating Lucene Query via Hibernate Search Query DSL**

QueryBuilder qb = fullTextSession

.getSearchFactory()

.buildQueryBuilder()

.forEntity(TestBO.**class**)

.get();

org.apache.lucene.search.Query query = qb

.keyword()

.onFields("username")

.matching("zh17")

.createQuery();

// wrap Lucene query in a org.hibernate.Query

org.hibernate.Query hibQuery = fullTextSession.createFullTextQuery(query, TestBO.**class**);

// execute search

@SuppressWarnings("unchecked")

List<TestBO> result = hibQuery.list();

The above code snippets will ensure to get list of qualified entities against the search criteria.

**// Creating a Lucene query via the QueryParser**

SearchFactory searchFactory = fullTextSession.getSearchFactory();

QueryParser parser = **new** QueryParser(Version.***LUCENE\_36***, "customername", searchFactory.getAnalyzer(CustomerBO.**class**));

**try** {

//creating lucene query for search using default column passed in QueryParser

org.apache.lucene.search.Query luceneQuery = parser.parse( "aks" );

//can use other column also as "column/fieldname : value-to-search", f.e.

//org.apache.lucene.search.Query newQuery = parser.parse( "customerid:4" );

org.hibernate.Query fullQuery = fullTextSession.createFullTextQuery(luceneQuery);

List result = fullQuery.list(); //return a list of managed objects

***logger***.info("----No of matching result : " + result.size());

} **catch** (ParseException e) {

//handle parsing failure

}

**Building a Hibernate Search query**

**Pagination :-** exactly the way you would define pagination in a plain HQL or Criteria query.

org.hibernate.Query hibQuery = fullTextSession .createFullTextQuery(query,CustomerBO.class)

.setFirstResult(4);

.setMaxResults(30);

**Range :-** A range query searches for a value in between given boundaries (included or not) or for a value below or above a given boundary (included or not).

org.apache.lucene.search.Query query = qb

.range()

.onField("customerid")

.above(4)

.createQuery();

**Projections :-** Hibernate Search extracts the properties from the Lucene index and convert them back to their object representation, returning a list of Object[]. Projections avoid a potential database round trip (useful if the query response time is critical).

The properties projected must be stored in the index (@Field(store=Store.YES)), which increases the index size.

org.hibernate.Query hibQuery = fullTextSession

.createFullTextQuery(query, CustomerBO.class)

.setProjection("customername", "mobileno");

**Phrase Search :-** can also search exact or approximate sentences. You can search approximate sentences by adding a slop factor. The slop factor represents the number of other words permitted in the sentence: this works like a within or near operator.

org.apache.lucene.search.Query query = qb

.phrase().withSlop(3)

.onField("customername")

.sentence("You can queries")

.createQuery();  
The above query will search for sentence having exactly 3 words among these tokens.

**Wildcard Search:-** wildcard queries (queries where some of parts of the word are unknown). ? represents a single character and \* represents any character sequence.

org.apache.lucene.search.Query query = qb

.keyword()

.wildcard()

.onField("customername")

.matching("aks\*")

.createQuery();

1. **Analyzer**

Now let’s this all this searching very interesting as making these columns or properties searchable with search query/text as not exactly same.

Assume that one of your indexed Book entities (as we have taken into consideration above also) has the title "Refactoring: Improving the Design of Existing Code" and you want to get hits for all of the following queries: "refactor", "refactors", "refactored" and "refactoring".

In Lucene, this can be achieved by choosing an analyzer class which applies word stemming during the indexing as well as the search process. Hibernate Search offers several ways to configure the analyzer to be used:

* Setting the hibernate.search.analyzer property in the configuration file. The specified class will then be the default analyzer.
* Setting the @Analyzer annotation at the entity level.
* Setting the @Analyzer annotation at the field level.

When using the @Analyzer annotation one can either specify the fully qualified classname of the analyzer to use or one can refer to an analyzer definition defined by the @AnalyzerDef annotation. In the latter case the analyzer framework with its factories approach is utilized.

@Entity

@Table(name = "test")

@AnalyzerDef(name = "customanalyzer",

tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.**class**),

filters = { @TokenFilterDef(factory = LowerCaseFilterFactory.**class**),

@TokenFilterDef(factory = SnowballPorterFilterFactory.**class**, params = {

@Parameter(name = "language", value = "English")

})

})

**public** **class** TestBO {

@Id

@GeneratedValue

//@Field

**private** **int** id;

@Field

@Analyzer(definition= "customanalyzer")

**private** String name;

…

}

In the above entity, we have used tokenizer along with tokenizer factory & tokenizer filters. These uses details are as follows:

@AnalyzerDef annotation takes as parameter name of this defined analyzer & tokenizer to be used for search analyzer. Assume we are making any search query which next will send to analyzer if any defined on that property. And then subsequent filters defined. If more than one filters defined then they process in order they are defined.

StandardTokenizerFactory - This tokenizer splits the text field into tokens, treating whitespace, hyphens and punctuation as delimiters. It is a good general purpose tokenizer.

There are lots of it which can be found [here](https://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters) https://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters

LowerCaseFilterFactory - Converts any uppercase letters in a token to the equivalent lowercase token. All other characters are left unchanged.

Note: Required gradle dependencies are:

compile (“org.hibernate: hibernate-core : 4.3.0.Final”)

compile (“org.hibernate: hibernate-search-orm : 4.5.1.Final”)

***For References:***

<http://www.javaworld.com/article/2077880/data-storage/data-storage-introduction-to-hibernate-search.html>

<https://docs.jboss.org/hibernate/search/4.2/reference/en-US/html/search-configuration.html#infinispan-directories>

<https://dzone.com/refcardz/getting-started-with-hibernate>