Hibernate Search

Apache Lucene™ Integration

Reference Guide

3.1.1.GA

Hibernate Search

Preface

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 model. Amongst other things indexes have to be kept up to date and mismatches between index structure and domain model as well as query mismatches have to 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 [http://www.hibernate.org] and Apache Lucene [http://lucene.apache.org].

Chapter 1. Getting started

Welcome to Hibernate Search! The following chapter will guide you through the initial steps required to integrate Hibernate Search into an existing Hibernate enabled application. In case you are a Hibernate new timer we recommend you start here [http://hibernate.org/152.html].

1.1. System Requirements

Table 1.1. System requirements

Java Runtime	A JDK or JRE version *5* or greater. You can download a Java Runtime for Windows/Linux/Solaris here [http://java.sun.com/javase/downloads/].
Hibernate Search	`hibernate-search.jar` and all runtime dependencies from the `lib` directory of the Hibernate Search distribution. Please refer to `README.txt` in the lib directory to understand which dependencies are required.
Hibernate Core	This instructions have been tested against Hibernate 3.3.x. You will need `hibernate-core.jar` and its transitive dependencies from the `lib` directory of the distribution. Refer to `README.txt` in the `lib` directory of the distribution to determine the minimum runtime requirements.
Hibernate Annotations	Even though Hibernate Search can be used without Hibernate Annotations the following instructions will use them for basic entity configuration (*@Entity, @Id, @OneToMany,...*). This part of the configuration could also be expressed in xml or code. However, Hibernate Search itself has its own set of annotations (*@Indexed, @DocumentId, @Field,...*) for which there exists so far no alternative configuration. The tutorial is tested against version 3.4.x of Hibernate Annotations.

You can download all dependencies from the Hibernate download site [http://www.hibernate.org/6.html]. You can also verify the dependency versions against the Hibernate Compatibility Matrix [http://www.hibernate.org/6.html#A3].

1.2. Using Maven

Instead of managing all dependencies manually, maven
users have the possibility to use the JBoss maven repository
[http://repository.jboss.com/maven2]. Just add the JBoss repository url to the
repositories section of your `pom.xml` or `settings.xml`:

Example 1.1. Adding the JBoss maven repository to `settings.xml`

```
<repository>
  <id>repository.jboss.org</id>
  <name>JBoss Maven Repository</name>
  <url>http://repository.jboss.org/maven2</url>
  <layout>default</layout>
</repository>
```

Then add the following dependencies to your pom.xml:

Example 1.2. Maven dependencies for Hibernate Search

```
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-search</artifactId>
   <version>3.1.1.GA</version>
</dependency>
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-annotations</artifactId>
   <version>3.4.0.GA</version>
</dependency>
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-entitymanager</artifactId>
   <version>3.4.0.GA</version>
</dependency>
<dependency>
   <groupId>org.apache.solr</groupId>
   <artifactId>solr-common</artifactId>
   <version>1.3.0</version>
</dependency>
<dependency>
   <groupId>org.apache.solr</groupId>
   <artifactId>solr-core</artifactId>
   <version>1.3.0</version>
</dependency>
<dependency>
   <groupId>org.apache.lucene</groupId>
   <artifactId>lucene-snowball</artifactId>
   <version>2.4.1</version>
</dependency>
```

Not all dependencies are required. Only the *hibernate-search* dependency is mandatory. This dependency, together with its required transitive dependencies, contain all required classes needed to use Hibernate Search. *hibernate-annotations* is only needed if you want to use annotations to configure your domain model as we do in this tutorial. However, even if you choose not to use Hibernate Annotations you still have to use the Hibernate Search specific annotations, which are bundled with the hibernate-search jar file, to configure your Lucene index. Currently there is no XML configuration available for Hibernate Search. *hibernate-entitymanager* is required if you want to use Hibernate Search in conjunction with JPA. The Solr dependencies are needed if you want to utilize Solr's analyzer framework. More about this later. And finally, the `lucene-snowball` dependency is needed if you want to use Lucene's snowball stemmer.

1.3. Configuration

Once you have downloaded and added all required dependencies to your application you have to add a couple of properties to your hibernate configuration file. If you are using Hibernate directly this can be done in `hibernate.properties` or `hibernate.cfg.xml`. If you are using Hibernate via JPA you can also add the properties to `persistence.xml`. The good news is that for standard use most properties offer a sensible default. An example `persistence.xml` configuration could look like this:

Example 1.3. Basic configuration options to be added to `hibernate.properties`, `hibernate.cfg.xml` or `persistence.xml`

```
...
<property name="hibernate.search.default.directory_provider"
   value="org.hibernate.search.store.FSDirectoryProvider"/>

<property name="hibernate.search.default.indexBase"
 value="/var/lucene/indexes"/>
...
```

First you have to tell Hibernate Search which `DirectoryProvider` to use. This can be achieved by setting the `hibernate.search.default.directory_provider` property. Apache Lucene has the notion of a `Directory` to store the index files. Hibernate Search handles the initialization and configuration of a Lucene `Directory` instance via a `DirectoryProvider`. In this tutorial we will use a subclass of `DirectoryProvider` called `FSDirectoryProvider`. This will give us the ability to physically inspect the Lucene indexes created by Hibernate Search (eg via Luke [http://www.getopt.org/luke/]). Once you have a working configuration you can start experimenting with other directory providers (see Section 3.1, “Directory configuration”). Next to the directory provider you also have to specify the default root directory for all indexes via `hibernate.search.default.indexBase`.

Lets assume that your application contains the Hibernate managed classes `example.Book` and `example.Author` and you want to add free text search capabilities to your application in order to search the books contained in your database.

Example 1.4. Example entities Book and Author before adding Hibernate Search specific annotations

```
package example;
...
@Entity
public class Book {

  @Id
  @GeneratedValue
  private Integer id;

  private String title;

  private String subtitle;

  @ManyToMany
  private Set<Author> authors = new HashSet<Author>();

  private Date publicationDate;

  public Book() {
  }

  // standard getters/setters follow here
  ...
}
```

```
package example;
...
@Entity
public class Author {

  @Id
  @GeneratedValue
  private Integer id;

  private String name;

  public Author() {
  }

  // standard getters/setters follow here
  ...
}
```

To achieve this you have to add a few annotations to the `Book` and `Author` class. The first annotation `@Indexed` marks `Book` as indexable. By design

Hibernate Search needs to store an untokenized id in the index to ensure index unicity for a given entity. `@DocumentId` marks the property to use for this purpose and is in most cases the same as the database primary key. In fact since the 3.1.0 release of Hibernate Search `@DocumentId` is optional in the case where an `@Id` annotation exists.

Next you have to mark the fields you want to make searchable. Let's start with `title` and `subtitle` and annotate both with `@Field`. The parameter `index=Index.TOKENIZED` will ensure that the text will be tokenized using the default Lucene analyzer. Usually, tokenizing means chunking a sentence into individual words and potentially excluding common words like `'a'` or '`the`'. We will talk more about analyzers a little later on. The second parameter we specify within `@Field`, `store=Store.NO`, ensures that the actual data will not be stored in the index. Whether this data is stored in the index or not has nothing to do with the ability to search for it. From Lucene's perspective it is not necessary to keep the data once the index is created. The benefit of storing it is the ability to retrieve it via projections (Section 5.1.2.5, “Projection”).

Without projections, Hibernate Search will per default execute a Lucene query in order to find the database identifiers of the entities matching the query critera and use these identifiers to retrieve managed objects from the database. The decision for or against projection has to be made on a case to case basis. The default behaviour - `Store.NO` - is recommended since it returns managed objects whereas projections only return object arrays.

After this short look under the hood let's go back to annotating the `Book` class. Another annotation we have not yet discussed is `@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 more details see Section 4.2, “Property/Field Bridge”.

This leaves us with `@IndexedEmbedded`. This annotation is used to index associated entities (`@ManyToMany`, `@*ToOne` and `@Embedded`) as part of the owning entity. This is needed since a Lucene index document is a flat data structure which does not know anything about object relations. To ensure that the authors' name wil be searchable you have to make sure that the names are indexed as part of the book itself. On top of `@IndexedEmbedded` you will also have to mark all fields of the associated entity you want to have included in the index with `@Indexed`. For more details see Section 4.1.3, “Embedded and associated objects”.

These settings should be sufficient for now. For more details on entity mapping refer to Section 4.1, “Mapping an entity”.

Example 1.5. Example entities after adding Hibernate Search annotations

```
package example;
...
@Entity
@Indexed
public class Book {

  @Id
  @GeneratedValue
  @DocumentId
  private Integer id;

  @Field(index=Index.TOKENIZED, store=Store.NO)
  private String title;

  @Field(index=Index.TOKENIZED, store=Store.NO)
  private String subtitle;

  @IndexedEmbedded
  @ManyToMany
  private Set<Author> authors = new HashSet<Author>();

  @Field(index = Index.UN_TOKENIZED, store = Store.YES)
  @DateBridge(resolution = Resolution.DAY)
  private Date publicationDate;

  public Book() {
  }

  // standard getters/setters follow here
  ...
}
```

```
package example;
...
@Entity
public class Author {

  @Id
  @GeneratedValue
  private Integer id;

  @Field(index=Index.TOKENIZED, store=Store.NO)
  private String name;

  public Author() {
  }

  // standard getters/setters follow here
  ...
}
```

1.4. Indexing

Hibernate Search will transparently index every entity persisted, updated or removed through Hibernate Core. However, you have to trigger an initial indexing to populate the Lucene index with 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 (see also Chapter 6, *Manual indexing*):

Example 1.6. Using Hibernate Session to index data

```
FullTextSession fullTextSession =
 Search.getFullTextSession(session);
Transaction tx = fullTextSession.beginTransaction();

List books = session.createQuery("from Book as book").list();
for (Book book : books) {
    fullTextSession.index(book);
}

tx.commit(); //index is written at commit time
```

Example 1.7. Using JPA to index data

```
EntityManager em = entityManagerFactory.createEntityManager();
FullTextEntityManager fullTextEntityManager =
 Search.getFullTextEntityManager(em);
em.getTransaction().begin();

List books = em.createQuery("select book from Book as
 book").getResultList();
for (Book book : books) {
    fullTextEntityManager.index(book);
}

em.getTransaction().commit();
em.close();
```

After executing the above code, you should be able to see a Lucene index under `/var/lucene/indexes/example.Book`. Go ahead an inspect this index with Luke [http://www.getopt.org/luke/]. It will help you to understand how Hibernate Search works.

1.5. Searching

Now it is time to execute a first search. The general approach is to create a native Lucene query 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 `Book`s.

Example 1.8. Using Hibernate Session to create and execute a search

```
FullTextSession fullTextSession =
 Search.getFullTextSession(session);
Transaction tx = fullTextSession.beginTransaction();

// create native Lucene query
String[] fields = new String[]{"title", "subtitle", "authors.name",
 "publicationDate"};
MultiFieldQueryParser parser = new MultiFieldQueryParser(fields, new
 StandardAnalyzer());
org.apache.lucene.search.Query query = parser.parse( "Java rocks!"
 );

// wrap Lucene query in a org.hibernate.Query
org.hibernate.Query hibQuery =
 fullTextSession.createFullTextQuery(query, Book.class);

// execute search
List result = hibQuery.list();

tx.commit();
session.close();
```

Example 1.9. Using JPA to create and execute a search

```
EntityManager em = entityManagerFactory.createEntityManager();
FullTextEntityManager fullTextEntityManager =

 org.hibernate.hibernate.search.jpa.Search.getFullTextEntityManager(em);
em.getTransaction().begin();

// create native Lucene query
String[] fields = new String[]{"title", "subtitle", "authors.name",
 "publicationDate"};
MultiFieldQueryParser parser = new MultiFieldQueryParser(fields, new
 StandardAnalyzer());
org.apache.lucene.search.Query query = parser.parse( "Java rocks!"
 );

// wrap Lucene query in a javax.persistence.Query
javax.persistence.Query persistenceQuery =
 fullTextEntityManager.createFullTextQuery(query, Book.class);

// execute search
List result = persistenceQuery.getResultList();

em.getTransaction().commit();
em.close();
```

1.6. Analyzer

Let's make things a little more interesting now. Assume that one of your indexed book entities 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** search process. Hibernate Search offers several ways to configure the analyzer to use (see Section 4.1.5, “Analyzer”):

- 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

Solr analyzer framework with its factories approach is utilized. To find out more about the factory classes available you can either browse the Solr JavaDoc or read the corresponding section on the Solr Wiki. [http://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters] Note that depending on the chosen factory class additional libraries on top of the Solr dependencies might be required. For example, the `PhoneticFilterFactory` depends on commons-codec [http://commons.apache.org/codec].

In the example below a `StandardTokenizerFactory` is used followed by two filter factories, `LowerCaseFilterFactory` and `SnowballPorterFilterFactory`. The standard tokenizer splits words at punctuation characters and hyphens while keeping email addresses and internet hostnames intact. It is a good general purpose tokenizer. The lowercase filter lowercases the letters in each token whereas the snowball filter finally applies language specific stemming.

Generally, when using the Solr framework you have to start with a tokenizer followed by an arbitrary number of filters.

Example 1.10. Using `@AnalyzerDef` and the Solr framework to define and use an analyzer

```
package example;
...
@Entity
@Indexed
@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 Book {

  @Id
  @GeneratedValue
  @DocumentId
  private Integer id;

  @Field(index=Index.TOKENIZED, store=Store.NO)
  @Analyzer(definition = "customanalyzer")
  private String title;

  @Field(index=Index.TOKENIZED, store=Store.NO)
  @Analyzer(definition = "customanalyzer")
  private String subtitle;

  @IndexedEmbedded
  @ManyToMany
  private Set<Author> authors = new HashSet<Author>();

  @Field(index = Index.UN_TOKENIZED, store = Store.YES)
  @DateBridge(resolution = Resolution.DAY)
  private Date publicationDate;

  public Book() {
  }

  // standard getters/setters follow here
  ...
}
```

1.7. What's next

The above paragraphs hopefully helped you getting an overview of Hibernate Search. Using the maven archetype plugin and the following command you can create an initial runnable maven project structure populated with the example code of this tutorial.

Example 1.11. Using the Maven archetype to create tutorial sources

```
mvn archetype:create \
    -DarchetypeGroupId=org.hibernate \
    -DarchetypeArtifactId=hibernate-search-quickstart \
    -DarchetypeVersion=3.1.1.GA \
    -DgroupId=my.company -DartifactId=quickstart
```

Using the maven project you can execute the examples, inspect the file system based index and search and retrieve a list of managed objects. Just run *mvn package* to compile the sources and run the unit tests.

The next step after this tutorial is to get more familiar with the overall architecture of Hibernate Search (Chapter 2, *Architecture*) and explore the basic features in more detail. Two topics which were only briefly touched in this tutorial were analyzer configuration (Section 4.1.5, “Analyzer”) and field bridges (Section 4.2, “Property/Field Bridge”), both important features required for more fine-grained indexing. More advanced topics cover clustering (Section 3.5, “JMS Master/Slave configuration”) and large indexes handling (Section 3.2, “Sharding indexes”).

Chapter 2. Architecture

2.1. Overview

Hibernate Search consists of an indexing component and an index search component. Both are backed by Apache Lucene.

Each time an entity is inserted, updated or removed in/from the database, Hibernate Search keeps track of this event (through the Hibernate event system) and schedules an index update. All the index updates are handled without you having to use the Apache Lucene APIs (see Section 3.7, “Enabling Hibernate Search and automatic indexing”).

To interact with Apache Lucene indexes, Hibernate Search has the notion of `DirectoryProvider`s. A directory provider will manage a given Lucene `Directory` type. You can configure directory providers to adjust the directory target (see Section 3.1, “Directory configuration”).

Hibernate Search uses the Lucene index to search an entity and return a list of managed entities saving you the tedious object to Lucene document mapping. The same persistence context is shared between Hibernate and Hibernate Search. As a matter of fact, the `FullTextSession` is built on top of the Hibernate Session. so that the application code can use the unified `org.hibernate.Query` or `javax.persistence.Query` APIs exactly the way a HQL, JPA-QL or native queries would do.

To be more efficient, Hibernate Search batches the write interactions with the Lucene index. There is currently two types of batching depending on the expected scope. Outside a transaction, the index update operation is executed right after the actual database operation. This scope is really a no scoping setup and no batching is performed. However, it is recommended - for both your database and Hibernate Search - to execute your operation in a transaction be it JDBC or JTA. When in a transaction, the index update operation is scheduled for the transaction commit phase and discarded in case of transaction rollback. The batching scope is the transaction. There are two immediate benefits:

- Performance: Lucene indexing works better when operation are executed in batch.
- ACIDity: The work executed has the same scoping as the one executed by the database transaction and is executed if and only if the transaction is committed. This is not ACID in the strict sense of it, but ACID behavior is

rarely useful for full text search indexes since they can be rebuilt from the source at any time.

You can think of those two scopes (no scope vs transactional) as the equivalent of the (infamous) autocommit vs transactional behavior. From a performance perspective, the *in transaction* mode is recommended. The scoping choice is made transparently. Hibernate Search detects the presence of a transaction and adjust the scoping.

Note

Hibernate Search works perfectly fine in the Hibernate / EntityManager long conversation pattern aka. atomic conversation.

Note

Depending on user demand, additional scoping will be considered, the pluggability mechanism being already in place.

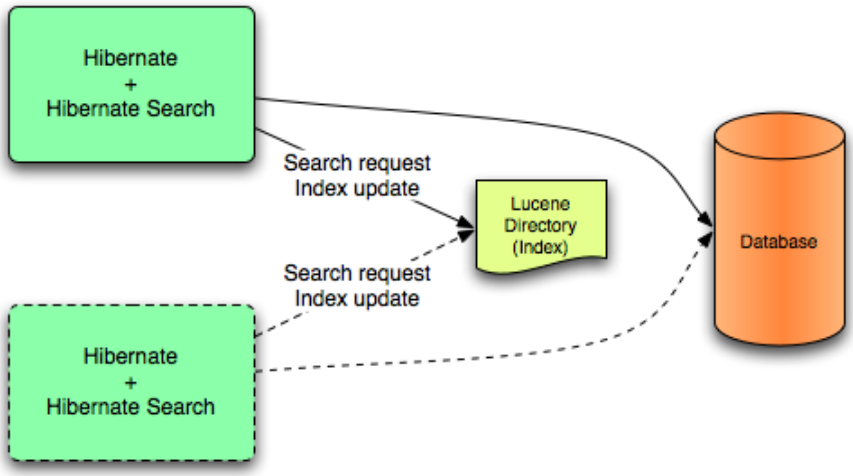
2.2. Back end

Hibernate Search offers the ability to let the scoped work being processed by different back ends. Two back ends are provided out of the box for two different scenarios.

2.2.1. Back end types

2.2.1.1. Lucene

In this mode, all index update operations applied on a given node (JVM) will be executed to the Lucene directories (through the directory providers) by the same node. This mode is typically used in non clustered environment or in clustered environments where the directory store is shared.



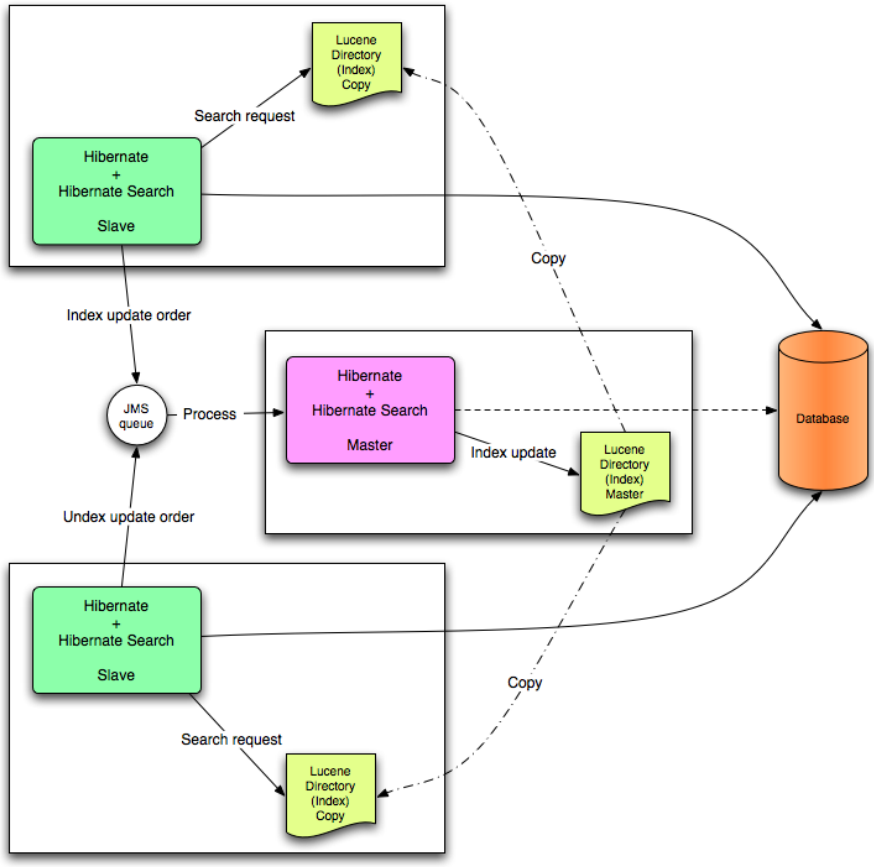
Lucene back end configuration.

This mode targets non clustered applications, or clustered applications where the Directory is taking care of the locking strategy.

The main advantage is simplicity and immediate visibility of the changes in Lucene queries (a requirement in some applications).

2.2.1.2. JMS

All index update operations applied on a given node are sent to a JMS queue. A unique reader will then process the queue and update the master index. The master index is then replicated on a regular basis to the slave copies. This is known as the master/slaves pattern. The master is the sole responsible for updating the Lucene index. The slaves can accept read as well as write operations. However, they only process the read operation on their local index copy and delegate the update operations to the master.



JMS back end configuration.

This mode targets clustered environments where throughput is critical, and index update delays are affordable. Reliability is ensured by the JMS provider and by having the slaves working on a local copy of the index.

Note

Hibernate Search is an extensible architecture. Feel free to drop ideas for other third party back ends to `hibernate-dev@lists.jboss.org`.

2.2.2. Work execution

The indexing work (done by the back end) can be executed synchronously with the transaction commit (or update operation if out of transaction), or asynchronously.

2.2.2.1. Synchronous

This is the safe mode where the back end work is executed in concert with the transaction commit. Under highly concurrent environment, this can lead to throughput limitations (due to the Apache Lucene lock mechanism) and it can increase the system response time if the backend is significantly slower than the transactional process and if a lot of IO operations are involved.

2.2.2.2. Asynchronous

This mode delegates the work done by the back end to a different thread. That way, throughput and response time are (to a certain extend) decorrelated from the back end performance. The drawback is that a small delay appears between the transaction commit and the index update and a small overhead is introduced to deal with thread management.

It is recommended to use synchronous execution first and evaluate asynchronous execution if performance problems occur and after having set up a proper benchmark (ie not a lonely cowboy hitting the system in a completely unrealistic way).

2.3. Reader strategy

When executing a query, Hibernate Search interacts with the Apache Lucene indexes through a reader strategy. Choosing a reader strategy will depend on the profile of the application (frequent updates, read mostly, asynchronous index update etc). See also Section 3.6, “Reader strategy configuration”

2.3.1. Shared

With this strategy, Hibernate Search will share the same `IndexReader`, for a given Lucene index, across multiple queries and threads provided that the `IndexReader` is still up-to-date. If the `IndexReader` is not up-to-date, a new one is opened and provided. Each `IndexReader` is made of several

`SegmentReader`s. This strategy only reopens segments that have been modified or created after last opening and shares the already loaded segments from the previous instance. This strategy is the default.

The name of this strategy is `shared`.

2.3.2. Not-shared

Every time a query is executed, a Lucene `IndexReader` is opened. This strategy is not the most efficient since opening and warming up an `IndexReader` can be a relatively expensive operation.

The name of this strategy is `not-shared`.

2.3.3. Custom

You can write your own reader strategy that suits your application needs by implementing `org.hibernate.search.reader.ReaderProvider`. The implementation must be thread safe.

Chapter 3. Configuration

3.1. Directory configuration

Apache Lucene has a notion of `Directory` to store the index files. The `Directory` implementation can be customized, but Lucene comes bundled with a file system (`FSDirectoryProvider`) and an in memory (`RAMDirectoryProvider`) implementation. `DirectoryProvider`s are the Hibernate Search abstraction around a Lucene `Directory` and handle the configuration and the initialization of the underlying Lucene resources. Table 3.1, “List of built-in Directory Providers” shows the list of the directory providers bundled with Hibernate Search.

	your operating system and available RAM; most people reported good results using values between 16 and 64MB.	

Table 3.1. List of built-in Directory Providers

org.hibernate.search.store.FSSlaveDirectoryProvider	Filesystem based directory. Like FSDirectoryProvider, but retrieves a master version (source) on a regular basis. To avoid locking and inconsistent search results, 2 local copies are kept. The recommended value for the refresh period is (at least) 50% higher that the time to copy the information (default 3600 seconds - 60 minutes). Note that the copy is based on an incremental copy mechanism reducing the average copy time. DirectoryProvider typically used on slave nodes using a JMS back end. The `buffer_size_on_copy` optimum depends on your operating system and available RAM; most people reported good results using values between 16 and 64MB.	`indexBase`: Base directory `indexName`: override @Indexed.index (useful for sharded indexes) `sourceBase`: Source (copy) base directory. `source`: Source directory suffix (default to `@Indexed.index`). The actual source directory name being `<sourceBase>/<source>` `refresh`: refresh period in second (the copy will take place every refresh seconds). `buffer_size_on_copy`: The amount of MegaBytes to move in a single low level copy instruction; defaults to 16MB.
org.hibernate.search.store.RAMDirectoryProvider	Memory based directory, the directory will be uniquely identified (in the same deployment unit) by the `@Indexed.index` element	none

If the built-in directory providers do not fit your needs, you can write your own directory provider by implementing the `org.hibernate.store.DirectoryProvider` interface.

Each indexed entity is associated to a Lucene index (an index can be shared by several entities but this is not usually the case). You can configure the index through properties prefixed by `hibernate.search.`*`indexname`*. Default properties inherited to all indexes can be defined using the prefix `hibernate.search.default.`

To define the directory provider of a given index, you use the `hibernate.search.`*`indexname`*`.directory_provider`

Example 3.1. Configuring directory providers

```
hibernate.search.default.directory_provider
 org.hibernate.search.store.FSDirectoryProvider
hibernate.search.default.indexBase=/usr/lucene/indexes
hibernate.search.Rules.directory_provider
 org.hibernate.search.store.RAMDirectoryProvider
```

applied on

Example 3.2. Specifying the index name using the `index` parameter of `@Indexed`

```
@Indexed(index="Status")
public class Status { ... }

@Indexed(index="Rules")
public class Rule { ... }
```

will create a file system directory in `/usr/lucene/indexes/Status` where the Status entities will be indexed, and use an in memory directory named `Rules` where Rule entities will be indexed.

You can easily define common rules like the directory provider and base directory, and override those defaults later on on a per index basis.

Writing your own `DirectoryProvider`, you can utilize this configuration mechanism as well.

3.2. Sharding indexes

In some extreme cases involving huge indexes (in size), it is necessary to split (shard) the indexing data of a given entity type into several Lucene indexes. This solution is not recommended until you reach significant index

sizes and index update times are slowing the application down. The main drawback of index sharding is that searches will end up being slower since more files have to be opened for a single search. In other words don't do it until you have problems :)

Despite this strong warning, Hibernate Search allows you to index a given entity type into several sub indexes. Data is sharded into the different sub indexes thanks to an `IndexShardingStrategy`. By default, no sharding strategy is enabled, unless the number of shards is configured. To configure the number of shards use the following property

Example 3.3. Enabling index sharding by specifying nbr_of_shards for a specific index

```
hibernate.search.<indexName>.sharding_strategy.nbr_of_shards 5
```

This will use 5 different shards.

The default sharding strategy, when shards are set up, splits the data according to the hash value of the id string representation (generated by the Field Bridge). This ensures a fairly balanced sharding. You can replace the strategy by implementing `IndexShardingStrategy` and by setting the following property

Example 3.4. Specifying a custom sharding strategy

```
hibernate.search.<indexName>.sharding_strategy
 my.shardingstrategy.Implementation
```

Each shard has an independent directory provider configuration as described in Section 3.1, “Directory configuration”. The DirectoryProvider default name for the previous example are `<indexName>.0` to `<indexName>.4`. In other words, each shard has the name of it's owning index followed by . (dot) and its index number.

Example 3.5. Configuring the sharding configuration for an example entity `Animal`

```
hibernate.search.default.indexBase /usr/lucene/indexes

hibernate.search.Animal.sharding_strategy.nbr_of_shards 5
hibernate.search.Animal.directory_provider
 org.hibernate.search.store.FSDirectoryProvider
hibernate.search.Animal.0.indexName Animal00
hibernate.search.Animal.3.indexBase /usr/lucene/sharded
hibernate.search.Animal.3.indexName Animal03
```

This configuration uses the default id string hashing strategy and shards the Animal index into 5 subindexes. All subindexes are `FSDirectoryProvider` instances and the directory where each subindex is stored is as followed:

- for subindex 0: /usr/lucene/indexes/Animal00 (shared indexBase but overridden indexName)
- for subindex 1: /usr/lucene/indexes/Animal.1 (shared indexBase, default indexName)
- for subindex 2: /usr/lucene/indexes/Animal.2 (shared indexBase, default indexName)
- for subindex 3: /usr/lucene/shared/Animal03 (overridden indexBase, overridden indexName)
- for subindex 4: /usr/lucene/indexes/Animal.4 (shared indexBase, default indexName)

3.3. Sharing indexes (two entities into the same directory)

Note

This is only presented here so that you know the option is available. There is really not much benefit in sharing indexes.

It is technically possible to store the information of more than one entity into a single Lucene index. There are two ways to accomplish this:

- Configuring the underlying directory providers to point to the same physical index directory. In practice, you set the property `hibernate.search.[fully qualified entity name].indexName` to the same value. As an example let's use the same index (directory) for the `Furniture` and `Animal` entity. We just set `indexName` for both entities to for example "Animal". Both entities will then be stored in the Animal directory

```
hibernate.search.org.hibernate.search.test.shards.Furniture.indexName = Animal
hibernate.search.org.hibernate.search.test.shards.Animal.indexName = Animal
```

- Setting the `@Indexed` annotation's `index` attribute of the entities you want to merge to the same value. If we again wanted all `Furniture` instances to be indexed in the `Animal` index along with all instances of `Animal` we would specify `@Indexed(index="Animal")` on both `Animal` and `Furniture` classes.

3.4. Worker configuration

It is possible to refine how Hibernate Search interacts with Lucene through the worker configuration. The work can be executed to the Lucene directory or sent to a JMS queue for later processing. When processed to the Lucene directory, the work can be processed synchronously or asynchronously to the transaction commit.

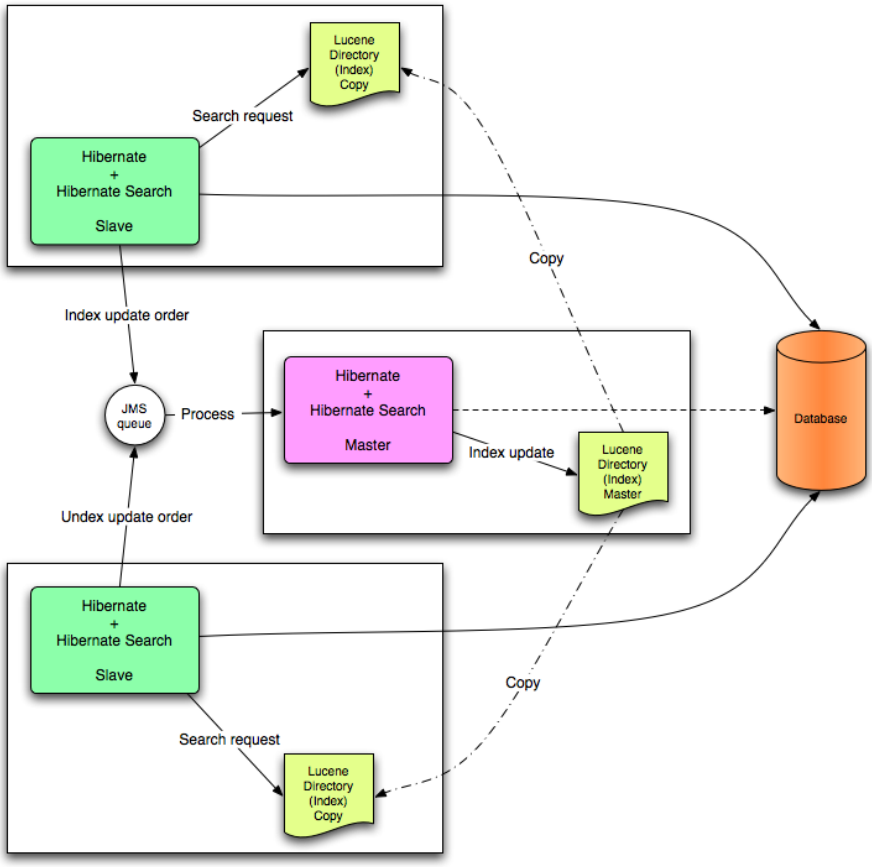
You can define the worker configuration using the following properties

Table 3.2. worker configuration

Property	Description
`hibernate.search.worker.backend`	Out of the box support for the Apache Lucene back end and the JMS back end. Default to `lucene`. Supports also `jms`.
`hibernate.search.worker.execution`	Supports synchronous and asynchronous execution. Default to `sync`. Supports also `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 connection factory from (`/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.

3.5. JMS Master/Slave configuration

This section describes in greater detail how to configure the Master / Slaves Hibernate Search architecture.



JMS Master/Slave architecture overview.

3.5.1. Slave nodes

Every index update operation is sent to a JMS queue. Index querying operations are executed on a local index copy.

Example 3.6. JMS Slave configuration

```
### slave configuration

## DirectoryProvider
# (remote) master location
hibernate.search.default.sourceBase =
 /mnt/mastervolume/lucenedirs/mastercopy

# local copy location
hibernate.search.default.indexBase = /Users/prod/lucenedirs

# refresh every half hour
hibernate.search.default.refresh = 1800

# appropriate directory provider
hibernate.search.default.directory_provider =
 org.hibernate.search.store.FSSlaveDirectoryProvider

## Backend configuration
hibernate.search.worker.backend = jms
hibernate.search.worker.jms.connection_factory = /ConnectionFactory
hibernate.search.worker.jms.queue = queue/hibernatesearch
#optional jndi configuration (check your JMS provider for more
 information)

## Optional asynchronous execution strategy
# hibernate.search.worker.execution = async
# hibernate.search.worker.thread_pool.size = 2
# hibernate.search.worker.buffer_queue.max = 50
```

A file system local copy is recommended for faster search results.

The refresh period should be higher that the expected time copy.

3.5.2. Master node

Every index update operation is taken from a JMS queue and executed. The master index is copied on a regular basis.

Example 3.7. JMS Master configuration

```
### master configuration

## DirectoryProvider
# (remote) master location where information is copied to
hibernate.search.default.sourceBase =
 /mnt/mastervolume/lucenedirs/mastercopy

# local master location
hibernate.search.default.indexBase = /Users/prod/lucenedirs

# refresh every half hour
hibernate.search.default.refresh = 1800

# appropriate directory provider
hibernate.search.default.directory_provider =
 org.hibernate.search.store.FSMasterDirectoryProvider

## Backend configuration
#Backend is the default lucene one
```

The refresh period should be higher that the expected time copy.

In addition to the Hibernate Search framework configuration, a Message Driven Bean should be written and set up to process the index works queue through JMS.

Example 3.8. Message Driven Bean processing the indexing queue

```
@MessageDriven(activationConfig = {
      @ActivationConfigProperty(propertyName="destinationType",
 propertyValue="javax.jms.Queue"),
      @ActivationConfigProperty(propertyName="destination",
 propertyValue="queue/hibernatesearch"),
      @ActivationConfigProperty(propertyName="DLQMaxResent",
 propertyValue="1")
   } )
public class MDBSearchController extends
 AbstractJMSHibernateSearchController implements MessageListener {
    @PersistenceContext EntityManager em;

    //method retrieving the appropriate session
    protected Session getSession() {
        return (Session) em.getDelegate();
    }

    //potentially close the session opened in #getSession(), not
 needed here
    protected void cleanSessionIfNeeded(Session session)
    }
}
```

This example inherits from the abstract JMS controller class available in the Hibernate Search source code and implements a JavaEE 5 MDB. This implementation is given as an example and, while most likely be more complex, can be adjusted to make use of non Java EE Message Driven Beans. For more information about the `getSession()` and `cleanSessionIfNeeded()`, please check `AbstractJMSHibernateSearchController`'s javadoc.

3.6. Reader strategy configuration

The different reader strategies are described in Reader strategy. Out of the box strategies are:

- `shared`: share index readers across several queries. This strategy is the most efficient.
- `not-shared`: create an index reader for each individual query

The default reader strategy is `shared`. This can be adjusted:

```
hibernate.search.reader.strategy = not-shared
```

Adding this property switches to the `not-shared` strategy.

Or if you have a custom reader strategy:

```
hibernate.search.reader.strategy =
 my.corp.myapp.CustomReaderProvider
```

where `my.corp.myapp.CustomReaderProvider` is the custom strategy implementation.

3.7. Enabling Hibernate Search and automatic indexing

3.7.1. Enabling Hibernate Search

Hibernate Search is enabled out of the box when using Hibernate Annotations or Hibernate EntityManager. If, for some reason you need to disable it, set `hibernate.search.autoregister_listeners` to false. Note that there is no performance penalty when the listeners are enabled even though no entities are indexed.

To enable Hibernate Search in Hibernate Core (ie. if you don't use Hibernate Annotations), add the `FullTextIndexEventListener` for the following six Hibernate events and also add it after the default `DefaultFlushEventListener`, as in the following example.

Example 3.9. Explicitly enabling Hibernate Search by configuring the `FullTextIndexEventListener`

```
<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>
        <event type="flush">
            <listener
 class="org.hibernate.event.def.DefaultFlushEventListener"/>
            <listener
 class="org.hibernate.search.event.FullTextIndexEventListener"/>
        </event>
    </session-factory>
</hibernate-configuration>
```

3.7.2. 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 (see Chapter 6, *Manual indexing*).

To disable event based indexing, set

```
hibernate.search.indexing_strategy manual
```

Note

In most case, the JMS backend provides the best of both world, a lightweight event based system keeps track of all changes in the system, and the heavyweight indexing process is done by a separate process or machine.

3.8. Tuning Lucene indexing performance

Hibernate Search allows you to tune the Lucene indexing performance by specifying a set of parameters which are passed through to underlying Lucene `IndexWriter` such as `mergeFactor`, `maxMergeDocs` and `maxBufferedDocs`. You can specify these parameters either as default values applying for all indexes, on a per index basis, or even per shard.

There are two sets of parameters allowing for different performance settings depending on the use case. During indexing operations triggered by database modifications, the parameters are grouped by the `transaction` keyword:

```
hibernate.search.[default|<indexname>].indexwriter.transaction.<parameter_name>
```

When indexing occurs via `FullTextSession.index()` (see Chapter 6, *Manual indexing*), the used properties are those grouped under the `batch` keyword:

```
hibernate.search.[default|<indexname>].indexwriter.batch.<parameter_name>
```

Unless the corresponding `.batch` property is explicitly set, the value will default to the `.transaction` property. If no value is set for a `.batch` value in a specific shard configuration, Hibernate Search will look at the index section, then at the default section and after that it will look for a `.transaction` in the same order:

```
hibernate.search.Animals.2.indexwriter.transaction.max_merge_docs 10
hibernate.search.Animals.2.indexwriter.transaction.merge_factor 20
hibernate.search.default.indexwriter.batch.max_merge_docs 100
```

This configuration will result in these settings applied to the second shard of Animals index:

- `transaction.max_merge_docs` = 10
- `batch.max_merge_docs` = 100
- `transaction.merge_factor` = 20
- `batch.merge_factor` = 20

All other values will use the defaults defined in Lucene.

The default for all values is to leave them at Lucene's own default, so the listed values in the following table actually depend on the version of Lucene you are using; values shown are relative to version `2.4`. For more information about Lucene indexing performances, please refer to the Lucene documentation.

Table 3.3. List of indexing performance and behavior properties

	When used together max_buffered_docs a flush occurs for whichever event happens first. Generally for faster indexing performance it's best to flush by RAM usage instead of document count and use as large a RAM buffer as you can.	
`hibernate.search.[default|<indexname>].indexwriter.[transaction|batch].term_index_inter`	Expert: Set the interval between indexed terms. Large values cause less memory to be used by IndexReader, but slow random-access to terms. Small values cause more memory to be used by an IndexReader, and speed random-access to terms. See Lucene documentation for more details.	128
`hibernate.search.[default|<indexname>].indexwriter.[transaction|batch].use_compound_fil`	The advantage of using the compound file format is that less file descriptors are used. The disadvantage is that indexing takes more time and temporary disk space. You can set this parameter to `false` in an attempt to improve the indexing time, but you could run out of file descriptors if `mergeFactor` is also large. Boolean parameter, use "`true`" or "`false`". The default value for this option is `true`.	true

Chapter 4. Mapping entities to the index structure

All the metadata information needed to index entities is described through annotations. There is no need for xml mapping files. In fact there is currently no xml configuration option available (see HSEARCH-210 [http://opensource.atlassian.com/projects/hibernate/browse/HSEARCH-210]). You can still use hibernate mapping files for the basic Hibernate configuration, but the Search specific configuration has to be expressed via annotations.

4.1. Mapping an entity

4.1.1. Basic mapping

First, we must declare a persistent class as indexable. This is done by annotating the class with `@Indexed` (all entities not annotated with `@Indexed` will be ignored by the indexing process):

Example 4.1. Making a class indexable using the `@Indexed` annotation

```
@Entity
@Indexed(index="indexes/essays")
public class Essay {
    ...
}
```

The `index` attribute tells Hibernate what the Lucene directory name is (usually a directory on your file system). It is recommended to define a base directory for all Lucene indexes using the `hibernate.search.default.indexBase` property in your configuration file. Alternatively you can specify a base directory per indexed entity by specifying `hibernate.search.<index>.indexBase,` where `<index>` is the fully qualified classname of the indexed entity. Each entity instance will be represented by a Lucene `Document` inside the given index (aka Directory).

For each property (or attribute) of your entity, you have the ability to describe how it will be indexed. The default (no annotation present) means that the property is completely ignored by the indexing process. `@Field` does declare a property as indexed. When indexing an element to a Lucene document you can specify how it is indexed:

- `name` : describe under which name, the property should be stored in the Lucene Document. The default value is the property name (following the JavaBeans convention)

- `store` : describe whether or not the property is stored in the Lucene index. You can store the value `Store.YES` (consuming more space in the index but allowing projection, see Section 5.1.2.5, “Projection” for more information), store it in a compressed way `Store.COMPRESS` (this does consume more CPU), or avoid any storage `Store.NO` (this is the default value). When a property is stored, you can retrieve its original value from the Lucene Document. This is not related to whether the element is indexed or not.

- index: describe how the element is indexed and the type of information store. The different values are `Index.NO` (no indexing, ie cannot be found by a query), `Index.TOKENIZED` (use an analyzer to process the property), `Index.UN_TOKENIZED` (no analyzer pre-processing), `Index.NO_NORMS` (do not store the normalization data). The default value is `TOKENIZED`.

- termVector: describes collections of term-frequency pairs. This attribute enables term vectors being stored during indexing so they are available within documents. The default value is TermVector.NO.

  The different values of this attribute are:

Value	Definition
TermVector.YES	Store the term vectors of each document. This produces two synchronized arrays, one contains document terms and the other contains the term's frequency.
TermVector.NO	Do not store term vectors.
TermVector.WITH_OFFSETS	Store the term vector and token offset information. This is the same as TermVector.YES plus it contains the starting and ending offset position information for the terms.
TermVector.WITH_POSITIONS	Store the term vector and token position information. This is the same as TermVector.YES plus it contains the ordinal positions of each occurrence of a term in a document.
TermVector.WITH_POSITION_OFFSETS	

Value	Definition
	Store the term vector, token position and offset information. This is a combination of the YES, WITH_OFFSETS and WITH_POSITIONS.

Whether or not you want to store the original data in the index depends on how you wish to use the index query result. For a regular Hibernate Search usage storing is not necessary. However you might want to store some fields to subsequently project them (see Section 5.1.2.5, “Projection” for more information).

Whether or not you want to tokenize a property depends on whether you wish to search the element as is, or by the words it contains. It make sense to tokenize a text field, but tokenizing a date field probably not. Note that fields used for sorting must not be tokenized.

Finally, the id property of an entity is a special property used by Hibernate Search to ensure index unicity of a given entity. By design, an id has to be stored and must not be tokenized. To mark a property as index id, use the `@DocumentId` annotation. If you are using Hibernate Annotations and you have specified @Id you can omit @DocumentId. The chosen entity id will also be used as document id.

Example 4.2. Adding `@DocumentId` ad `@Field` annotations to an indexed entity

```
@Entity
@Indexed(index="indexes/essays")
public class Essay {
    ...

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

    @Field(name="Abstract", index=Index.TOKENIZED, store=Store.YES)
    public String getSummary() { return summary; }

    @Lob
    @Field(index=Index.TOKENIZED)
    public String getText() { return text; }
}
```

The above annotations define an index with three fields: `id` , `Abstract` and `text` . Note that by default the field name is decapitalized, following the JavaBean specification

4.1.2. Mapping properties multiple times

Sometimes one has to map a property multiple times per index, with slightly different indexing strategies. For example, sorting a query by field requires the field to be `UN_TOKENIZED`. If one wants to search by words in this property and still sort it, one need to index it twice - once tokenized and once untokenized. @Fields allows to achieve this goal.

Example 4.3. Using @Fields to map a property multiple times

```
@Entity
@Indexed(index = "Book" )
public class Book {
    @Fields( {
            @Field(index = Index.TOKENIZED),
            @Field(name = "summary_forSort", index =
 Index.UN_TOKENIZED, store = Store.YES)
            } )
    public String getSummary() {
        return summary;
    }

    ...
}
```

The field `summary` is indexed twice, once as `summary` in a tokenized way, and once as `summary_forSort` in an untokenized way. @Field supports 2 attributes useful when @Fields is used:

- analyzer: defines a @Analyzer annotation per field rather than per property
- bridge: defines a @FieldBridge annotation per field rather than per property

See below for more information about analyzers and field bridges.

4.1.3. Embedded and associated objects

Associated objects as well as embedded objects can be indexed as part of the root entity index. This is useful if you expect to search a given entity based on properties of associated objects. In the following example the aim is to return places where the associated city is Atlanta (In the Lucene query parser language, it would translate into `address.city:Atlanta`).

Example 4.4. Using @IndexedEmbedded to index associations

```
@Entity
@Indexed
public class Place {
    @Id
    @GeneratedValue
    @DocumentId
    private Long id;

    @Field( index = Index.TOKENIZED )
    private String name;

    @OneToOne( cascade = { CascadeType.PERSIST, CascadeType.REMOVE }
 )
    @IndexedEmbedded
    private Address address;
    ....
}

@Entity
public class Address {
    @Id
    @GeneratedValue
    private Long id;

    @Field(index=Index.TOKENIZED)
    private String street;

    @Field(index=Index.TOKENIZED)
    private String city;

    @ContainedIn
    @OneToMany(mappedBy="address")
    private Set<Place> places;
    ...
}
```

In this example, the place fields will be indexed in the `Place` index. The `Place` index documents will also contain the fields `address.id`, `address.street`, and `address.city` which you will be able to query. This is enabled by the `@IndexedEmbedded` annotation.

Be careful. Because the data is denormalized in the Lucene index when using the `@IndexedEmbedded` technique, Hibernate Search needs to be aware of any change in the `Place` object and any change in the `Address` object to keep the index up to date. To make sure the `Place` Lucene document is updated when it's `Address` changes, you need to mark the other side of the bidirectional relationship with `@ContainedIn`.

`@ContainedIn` is only useful on associations pointing to entities as opposed to embedded (collection of) objects.

Let's make our example a bit more complex:

Example 4.5. Nested usage of `@IndexedEmbedded` and `@ContainedIn`

```
@Entity
@Indexed
public class Place {
    @Id
    @GeneratedValue
    @DocumentId
    private Long id;

    @Field( index = Index.TOKENIZED )
    private String name;

    @OneToOne( cascade = { CascadeType.PERSIST, CascadeType.REMOVE }
 )
    @IndexedEmbedded
    private Address address;
    ....
}

@Entity
public class Address {
    @Id
    @GeneratedValue
    private Long id;

    @Field(index=Index.TOKENIZED)
    private String street;

    @Field(index=Index.TOKENIZED)
    private String city;

    @IndexedEmbedded(depth = 1, prefix = "ownedBy_")
    private Owner ownedBy;

    @ContainedIn
    @OneToMany(mappedBy="address")
    private Set<Place> places;
    ...
}

@Embeddable
public class Owner {
    @Field(index = Index.TOKENIZED)
    private String name;
   ...
}
```

Any `@*ToMany, @*ToOne` and `@Embedded` attribute can be annotated with `@IndexedEmbedded`. The attributes of the associated class will then be added

to the main entity index. In the previous example, the index will contain the following fields

- id
- name
- address.street
- address.city
- address.ownedBy_name

The default prefix is `propertyName.`, following the traditional object navigation convention. You can override it using the `prefix` attribute as it is shown on the `ownedBy` property.

Note

The prefix cannot be set to the empty string.

The `depth` property is necessary when the object graph contains a cyclic dependency of classes (not instances). For example, if `Owner` points to `Place`. Hibernate Search will stop including Indexed embedded attributes after reaching the expected depth (or the object graph boundaries are reached). A class having a self reference is an example of cyclic dependency. In our example, because `depth` is set to 1, any `@IndexedEmbedded` attribute in Owner (if any) will be ignored.

Using `@IndexedEmbedded` for object associations allows you to express queries such as:

- Return places where name contains JBoss and where address city is Atlanta. In Lucene query this would be

```
+name:jboss +address.city:atlanta
```

- Return places where name contains JBoss and where owner's name contain Joe. In Lucene query this would be

```
+name:jboss +address.orderBy_name:joe
```

In a way it mimics the relational join operation in a more efficient way (at the cost of data duplication). Remember that, out of the box, Lucene indexes have no notion of association, the join operation is simply non-existent. It might help to keep the relational model normalized while benefiting from the full text index speed and feature richness.

Note

An associated object can itself (but does not have to) be `@Indexed`

When @IndexedEmbedded points to an entity, the association has to be directional and the other side has to be annotated `@ContainedIn` (as seen in the previous example). If not, Hibernate Search has no way to update the root index when the associated entity is updated (in our example, a `Place` index document has to be updated when the associated `Address` instance is updated).

Sometimes, the object type annotated by `@IndexedEmbedded` is not the object type targeted by Hibernate and Hibernate Search. This is especially the case when interfaces are used in lieu of their implementation. For this reason you can override the object type targeted by Hibernate Search using the `targetElement` parameter.

Example 4.6. Using the `targetElement` property of `@IndexedEmbedded`

```
@Entity
@Indexed
public class Address {
    @Id
    @GeneratedValue
    @DocumentId
    private Long id;

    @Field(index= Index.TOKENIZED)
    private String street;

    @IndexedEmbedded(depth = 1, prefix = "ownedBy_", targetElement =
 Owner.class)
    @Target(Owner.class)
    private Person ownedBy;


    ...
}

@Embeddable
public class Owner implements Person { ... }
```

4.1.4. Boost factor

Lucene has the notion of *boost factor*. It's a way to give more weight to a field or to an indexed element over others during the indexation process. You can use `@Boost` at the @Field, method or class level.

Example 4.7. Using different ways of increasing the weight of an indexed element using a boost factor

```
@Entity
@Indexed(index="indexes/essays")
@Boost(1.7f)
public class Essay {
    ...

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

    @Field(name="Abstract", index=Index.TOKENIZED, store=Store.YES,
 boost=@Boost(2f))
    @Boost(1.5f)
    public String getSummary() { return summary; }

    @Lob
    @Field(index=Index.TOKENIZED, boost=@Boost(1.2f))
    public String getText() { return text; }

    @Field
    public String getISBN() { return isbn; }

}
```

In our example, `Essay`'s probability to reach the top of the search list will be multiplied by 1.7. The `summary` field will be 3.0 (2 * 1.5 - `@Field.boost` and `@Boost` on a property are cumulative) more important than the `isbn` field. The `text` field will be 1.2 times more important than the `isbn` field. Note that this explanation in strictest terms is actually wrong, but it is simple and close enough to reality for all practical purposes. Please check the Lucene documentation or the excellent *Lucene In Action* from Otis Gospodnetic and Erik Hatcher.

4.1.5. Analyzer

The default analyzer class used to index tokenized fields is configurable through the `hibernate.search.analyzer` property. The default value for this property is `org.apache.lucene.analysis.standard.StandardAnalyzer`.

You can also define the analyzer class per entity, property and even per @Field (useful when multiple fields are indexed from a single property).

Example 4.8. Different ways of specifying an analyzer

```
@Entity
@Indexed
@Analyzer(impl = EntityAnalyzer.class)
public class MyEntity {
    @Id
    @GeneratedValue
    @DocumentId
    private Integer id;

    @Field(index = Index.TOKENIZED)
    private String name;

    @Field(index = Index.TOKENIZED)
    @Analyzer(impl = PropertyAnalyzer.class)
    private String summary;

    @Field(index = Index.TOKENIZED, analyzer = @Analyzer(impl =
 FieldAnalyzer.class)
    private String body;

    ...
}
```

In this example, `EntityAnalyzer` is used to index all tokenized properties (eg. `name`), except `summary` and `body` which are indexed with `PropertyAnalyzer` and `FieldAnalyzer` respectively.

Caution

Mixing different analyzers in the same entity is most of the time a bad practice. It makes query building more complex and results less predictable (for the novice), especially if you are using a QueryParser (which uses the same analyzer for the whole query). As a rule of thumb, for any given field the same analyzer should be used for indexing and querying.

4.1.5.1. Analyzer definitions

Analyzers can become quite complex to deal with for which reason Hibernate Search introduces the notion of analyzer definitions. An analyzer definition can be reused by many `@Analyzer` declarations. An analyzer definition is composed of:

- a name: the unique string used to refer to the definition
- a tokenizer: responsible for tokenizing the input stream into individual words

- a list of filters: each filter is responsible to remove, modify or sometimes even add words into the stream provided by the tokenizer

This separation of tasks - a tokenizer followed by a list of filters - allows for easy reuse of each individual component and let you build your customized analyzer in a very flexible way (just like Lego). Generally speaking the `Tokenizer` starts the analysis process by turning the character input into tokens which are then further processed by the `TokenFilter`s. Hibernate Search supports this infrastructure by utilizing the Solr analyzer framework. Make sure to add `solr-core.jar and solr-common.jar` to your classpath to use analyzer definitions. In case you also want to utilizing a snowball stemmer also include the `lucene-snowball.jar`. Other Solr analyzers might depend on more libraries. For example, the `PhoneticFilterFactory` depends on commons-codec [http://commons.apache.org/codec]. Your distribution of Hibernate Search provides these dependencies in its `lib` directory.

Example 4.9. `@AnalyzerDef` and the Solr framework

```
@AnalyzerDef(name="customanalyzer",
        tokenizer = @TokenizerDef(factory =
 StandardTokenizerFactory.class),
        filters = {
                @TokenFilterDef(factory =
 ISOLatin1AccentFilterFactory.class),
                @TokenFilterDef(factory =
 LowerCaseFilterFactory.class),
                @TokenFilterDef(factory = StopFilterFactory.class,
 params = {
                    @Parameter(name="words", value=
 "org/hibernate/search/test/analyzer/solr/stoplist.properties" ),
                    @Parameter(name="ignoreCase", value="true")
                })
})
public class Team {
    ...
}
```

A tokenizer is defined by its factory which is responsible for building the tokenizer and using the optional list of parameters. This example use the standard tokenizer. A filter is defined by its factory which is responsible for creating the filter instance using the optional parameters. In our example, the StopFilter filter is built reading the dedicated words property file and is expected to ignore case. The list of parameters is dependent on the tokenizer or filter factory.

Warning

Filters are applied in the order they are defined in the `@AnalyzerDef` annotation. Make sure to think twice about this order.

Once defined, an analyzer definition can be reused by an `@Analyzer` declaration using the definition name rather than declaring an implementation class.

Example 4.10. Referencing an analyzer by name

```
@Entity
@Indexed
@AnalyzerDef(name="customanalyzer", ... )
public class Team {
    @Id
    @DocumentId
    @GeneratedValue
    private Integer id;

    @Field
    private String name;

    @Field
    private String location;

    @Field @Analyzer(definition = "customanalyzer")
    private String description;
}
```

Analyzer instances declared by `@AnalyzerDef` are available by their name in the `SearchFactory`.

```
Analyzer analyzer =
 fullTextSession.getSearchFactory().getAnalyzer("customanalyzer");
```

This is quite useful wen building queries. Fields in queries should be analyzed with the same analyzer used to index the field so that they speak a common "language": the same tokens are reused between the query and the indexing process. This rule has some exceptions but is true most of the time. Respect it unless you know what you are doing.

4.1.5.2. Available analyzers

Solr and Lucene come with a lot of useful default tokenizers and filters.
You can find a complete list of tokenizer factories and filter factories at http://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters. Let check a few of them.

Table 4.1. Some of the available tokenizers

Factory	Description	parameters
StandardTokenizerFactory	Use the Lucene StandardTokenizer	none
HTMLStripStandardTokenizerFactory	Remove HTML tags, keep the text and pass it to a StandardTokenizer	none

Table 4.2. Some of the available filters

Factory	Description	parameters
StandardFilterFactory	Remove dots from acronyms and 's from words	none
LowerCaseFilterFactory	Lowercase words	none
StopFilterFactory	remove words (tokens) matching a list of stop words	`words`: points to a resource file containing the stop words ignoreCase: true if `case` should be ignore when comparing stop words, `false` otherwise
SnowballPorterFilterFactory	Reduces a word to it's root in a given language. (eg. protect, protects, protection share the same root). Using such a filter allows searches matching related words.	`language`: Danish, Dutch, English, Finnish, French, German, Italian, Norwegian, Portuguese, Russian, Spanish, Swedish and a few more
ISOLatin1AccentFilterFactory	remove accents for languages like French	none

We recommend to check all the implementations of `org.apache.solr.analysis.TokenizerFactory` and `org.apache.solr.analysis.TokenFilterFactory` in your IDE to see the implementations available.

4.1.5.3. Analyzer discriminator (experimental)

So far all the introduced ways to specify an analyzer were static. However, there are use cases where it is useful to select an analyzer depending on

the current state of the entity to be indexed, for example in multilingual application. For an `BlogEntry` class for example the analyzer could depend on the language property of the entry. Depending on this property the correct language specific stemmer should be chosen to index the actual text.

To enable this dynamic analyzer selection Hibernate Search introduces the `AnalyzerDiscriminator` annotation. The following example demonstrates the usage of this annotation:

Example 4.11. Usage of @AnalyzerDiscriminator in order to select an analyzer depending on the entity state

```
@Entity
@Indexed
@AnalyzerDefs({
  @AnalyzerDef(name = "en",
    tokenizer = @TokenizerDef(factory =
 StandardTokenizerFactory.class),
    filters = {
      @TokenFilterDef(factory = LowerCaseFilterFactory.class),
      @TokenFilterDef(factory = EnglishPorterFilterFactory.class
      )
    }),
  @AnalyzerDef(name = "de",
    tokenizer = @TokenizerDef(factory =
 StandardTokenizerFactory.class),
    filters = {
      @TokenFilterDef(factory = LowerCaseFilterFactory.class),
      @TokenFilterDef(factory = GermanStemFilterFactory.class)
    })
})
public class BlogEntry {

    @Id
    @GeneratedValue
    @DocumentId
    private Integer id;

    @Field
    @AnalyzerDiscriminator(impl = LanguageDiscriminator.class)
    private String language;

    @Field
    private String text;

    private Set<BlogEntry> references;

    // standard getter/setter
    ...
}
```

```
public class LanguageDiscriminator implements Discriminator {

    public String getAnanyzerDefinitionName(Object value, Object
 entity, String field) {
        if ( value == null || !( entity instanceof Article ) ) {
            return null;
        }
        return (String) value;
    }
}
```

The prerequisite for using `@AnalyzerDiscriminator` is that all analyzers which are going to be used are predefined via `@AnalyzerDef` definitions. If this is the case one can place the `@AnalyzerDiscriminator` annotation either on the class or on a specific property of the entity for which to dynamically select an analyzer. Via the `impl` parameter of the `AnalyzerDiscriminator` you specify a concrete implementation of the `Discriminator` interface. It is up to you to provide an implementation for this interface. The only method you have to implement is `getAnanyzerDefinitionName()` which gets called for each field added to the Lucene document. The entity which is getting indexed is also passed to the interface method. The `value` parameter is only set if the `AnalyzerDiscriminator` is placed on property level instead of class level. In this case the value represents the current value of this property.

An implemention of the `Discriminator` interface has to return the name of an existing analyzer definition if the analyzer should be set dynamically or `null` if the default analyzer should not be overridden. The given example assumes that the language parameter is either 'de' or 'en' which matches the specified names in the `@AnalyzerDef`s.

Note

The `@AnalyzerDiscriminator` is currently still experimental and the API might still change. We are hoping for some feedback from the community about the usefulness and usability of this feature.

4.1.5.4. Retrieving an analyzer

During indexing time, Hibernate Search is using analyzers under the hood for you. In some situations, retrieving analyzers can be handy. If your domain model makes use of multiple analyzers (maybe to benefit from stemming, use phonetic approximation and so on), you need to make sure to use the same analyzers when you build your query.

Note

This rule can be broken but you need a good reason for it. If you are unsure, use the same analyzers.

You can retrieve the scoped analyzer for a given entity used at indexing time by Hibernate Search. A scoped analyzer is an analyzer which applies the right analyzers depending on the field indexed: multiple analyzers can be defined on a given entity each one working on an individual field, a scoped analyzer unify all these analyzers into a context-aware analyzer. While the theory seems a bit complex, using the right analyzer in a query is very easy.

Example 4.12. Using the scoped analyzer when building a full-text query

```
org.apache.lucene.queryParser.QueryParser parser = new QueryParser(
    "title",
    fullTextSession.getSearchFactory().getAnalyzer( Song.class )
);

org.apache.lucene.search.Query luceneQuery =
    parser.parse( "title:sky Or title_stemmed:diamond" );

org.hibernate.Query fullTextQuery =
    fullTextSession.createFullTextQuery( luceneQuery, Song.class );

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

In the example above, the song title is indexed in two fields: the standard analyzer is used in the field `title` and a stemming analyzer is used in the field `title_stemmed`. By using the analyzer provided by the search factory, the query uses the appropriate analyzer depending on the field targeted.

If your query targets more that one query and you wish to use your standard analyzer, make sure to describe it using an analyzer definition. You can retrieve analyzers by their definition name using `searchFactory.getAnalyzer(String)`.

4.2. Property/Field Bridge

In Lucene all index fields have to be represented as Strings. For this reason all entity properties annotated with `@Field` have to be indexed in a String form. For most of your properties, Hibernate Search does the translation job for you thanks to a built-in set of bridges. In some cases, though you need a more fine grain control over the translation process.

4.2.1. Built-in bridges

Hibernate Search comes bundled with a set of built-in bridges between a Java property type and its full text representation.

null
: null elements are not indexed. Lucene does not support null elements and this does not make much sense either.

java.lang.String
: String are indexed as is

short, Short, integer, Integer, long, Long, float, Float, double, Double, BigInteger, BigDecimal
: Numbers are converted in their String representation. Note that numbers cannot be compared by Lucene (ie used in ranged queries) out of the box: they have to be padded

> **Note**
>
> Using a Range query is debatable and has drawbacks, an alternative approach is to use a Filter query which will filter the result query to the appropriate range.
>
> Hibernate Search will support a padding mechanism

java.util.Date
: Dates are stored as yyyyMMddHHmmssSSS in GMT time (200611072203012 for Nov 7th of 2006 4:03PM and 12ms EST). You shouldn't really bother with the internal format. What is important is that when using a DateRange Query, you should know that the dates have to be expressed in GMT time.

  Usually, storing the date up to the millisecond is not necessary. `@DateBridge` defines the appropriate resolution you are willing to store in the index (`@DateBridge(resolution=Resolution.DAY)`). The date pattern will then be truncated accordingly.

```
@Entity
@Indexed
public class Meeting {
    @Field(index=Index.UN_TOKENIZED)
    @DateBridge(resolution=Resolution.MINUTE)
    private Date date;
    ...
```

> **Warning**
>
> A Date whose resolution is lower than `MILLISECOND` cannot be a `@DocumentId`

java.net.URI, java.net.URL
: URI and URL are converted to their string representation

java.lang.Class
: Class are converted to their fully qualified class name. The thread context classloader is used when the class is rehydrated

4.2.2. Custom Bridge

Sometimes, the built-in bridges of Hibernate Search do not cover some of your property types, or the String representation used by the bridge does not meet your requirements. The following paragraphs describe several solutions to this problem.

4.2.2.1. StringBridge

The simplest custom solution is to give Hibernate Search an implementation of your expected `Object` to `String` bridge. To do so you need to implements the `org.hibernate.search.bridge.StringBridge` interface. All implementations have to be thread-safe as they are used concurrently.

Example 4.13. Implementing your own `StringBridge`

```
/**
 * Padding Integer bridge.
 * All numbers will be padded with 0 to match 5 digits
 *
 * @author Emmanuel Bernard
 */
public class PaddedIntegerBridge implements StringBridge {

    private int PADDING = 5;

    public String objectToString(Object object) {
        String rawInteger = ( (Integer) object ).toString();
        if (rawInteger.length() > PADDING)
            throw new IllegalArgumentException( "Try to pad on a
 number too big" );
        StringBuilder paddedInteger = new StringBuilder( );
        for ( int padIndex = rawInteger.length() ; padIndex <
 PADDING ; padIndex++ ) {
            paddedInteger.append('0');
        }
        return paddedInteger.append( rawInteger ).toString();
    }
}
```

Then any property or field can use this bridge thanks to the `@FieldBridge` annotation

```
@FieldBridge(impl = PaddedIntegerBridge.class)
private Integer length;
```

Parameters can be passed to the Bridge implementation making it more flexible. The Bridge implementation implements a `ParameterizedBridge`

interface, and the parameters are passed through the `@FieldBridge` annotation.

Example 4.14. Passing parameters to your bridge implementation

```
public class PaddedIntegerBridge implements StringBridge,
 ParameterizedBridge {

    public static String PADDING_PROPERTY = "padding";
    private int padding = 5; //default

    public void setParameterValues(Map parameters) {
        Object padding = parameters.get( PADDING_PROPERTY );
        if (padding != null) this.padding = (Integer) padding;
    }

    public String objectToString(Object object) {
        String rawInteger = ( (Integer) object ).toString();
        if (rawInteger.length() > padding)
            throw new IllegalArgumentException( "Try to pad on a
 number too big" );
        StringBuilder paddedInteger = new StringBuilder( );
        for ( int padIndex = rawInteger.length() ; padIndex <
 padding ; padIndex++ ) {
            paddedInteger.append('0');
        }
        return paddedInteger.append( rawInteger ).toString();
    }
}


//property
@FieldBridge(impl = PaddedIntegerBridge.class,
             params = @Parameter(name="padding", value="10")
            )
private Integer length;
```

The `ParameterizedBridge` interface can be implemented by `StringBridge`, `TwoWayStringBridge`, `FieldBridge` implementations.

All implementations have to be thread-safe, but the parameters are set during initialization and no special care is required at this stage.

If you expect to use your bridge implementation on an id property (ie annotated with `@DocumentId`), you need to use a slightly extended version of `StringBridge` named `TwoWayStringBridge`. Hibernate Search needs to read the string representation of the identifier and generate the object out of it. There is not difference in the way the `@FieldBridge` annotation is used.

Example 4.15. Implementing a TwoWayStringBridge which can for example be used for id properties

```
public class PaddedIntegerBridge implements TwoWayStringBridge,
 ParameterizedBridge {

    public static String PADDING_PROPERTY = "padding";
    private int padding = 5; //default

    public void setParameterValues(Map parameters) {
        Object padding = parameters.get( PADDING_PROPERTY );
        if (padding != null) this.padding = (Integer) padding;
    }

    public String objectToString(Object object) {
        String rawInteger = ( (Integer) object ).toString();
        if (rawInteger.length() > padding)
            throw new IllegalArgumentException( "Try to pad on a
 number too big" );
        StringBuilder paddedInteger = new StringBuilder( );
        for ( int padIndex = rawInteger.length() ; padIndex <
 padding ; padIndex++ ) {
            paddedInteger.append('0');
        }
        return paddedInteger.append( rawInteger ).toString();
    }

    public Object stringToObject(String stringValue) {
        return new Integer(stringValue);
    }
}


//id property
@DocumentId
@FieldBridge(impl = PaddedIntegerBridge.class,
             params = @Parameter(name="padding", value="10")
private Integer id;
```

It is critically important for the two-way process to be idempotent (ie object = stringToObject(objectToString(object))).

4.2.2.2. FieldBridge

Some use cases require more than a simple object to string translation when mapping a property to a Lucene index. To give you the greatest possible flexibility you can also implement a bridge as a `FieldBridge`. This interface gives you a property value and let you map it the way you want in your Lucene `Document`.The interface is very similar in its concept to the Hibernate `UserTypeS`.

You can for example store a given property in two different document fields:

Example 4.16. Implementing the FieldBridge interface in order to a given property into multiple document fields

```
/**
 * Store the date in 3 different fields - year, month, day - to ease
 Range Query per
 * year, month or day (eg get all the elements of December for the
 last 5 years).
 *
 * @author Emmanuel Bernard
 */
public class DateSplitBridge implements FieldBridge {
    private final static TimeZone GMT = TimeZone.getTimeZone("GMT");

    public void set(String name, Object value, Document document,
                    LuceneOptions luceneOptions) {
        Date date = (Date) value;
        Calendar cal = GregorianCalendar.getInstance(GMT);
        cal.setTime(date);
        int year = cal.get(Calendar.YEAR);
        int month = cal.get(Calendar.MONTH) + 1;
        int day = cal.get(Calendar.DAY_OF_MONTH);

        // set year
        Field field = new Field(name + ".year",
 String.valueOf(year),
            luceneOptions.getStore(), luceneOptions.getIndex(),
            luceneOptions.getTermVector());
        field.setBoost(luceneOptions.getBoost());
        document.add(field);

        // set month and pad it if needed
        field = new Field(name + ".month", month < 10 ? "0" : ""
            + String.valueOf(month), luceneOptions.getStore(),
            luceneOptions.getIndex(),
 luceneOptions.getTermVector());
        field.setBoost(luceneOptions.getBoost());
        document.add(field);

        // set day and pad it if needed
        field = new Field(name + ".day", day < 10 ? "0" : ""
            + String.valueOf(day), luceneOptions.getStore(),
            luceneOptions.getIndex(),
 luceneOptions.getTermVector());
        field.setBoost(luceneOptions.getBoost());
        document.add(field);
    }
}

//property
@FieldBridge(impl = DateSplitBridge.class)
private Date date;
```

4.2.2.3. ClassBridge

It is sometimes useful to combine more than one property of a given entity and index this combination in a specific way into the Lucene index. The `@ClassBridge` and `@ClassBridge` annotations can be defined at the class level (as opposed to the property level). In this case the custom field bridge implementation receives the entity instance as the value parameter instead of a particular property. Though not shown in this example, `@ClassBridge` supports the `termVector` attribute discussed in section Section 4.1.1, “Basic mapping”.

Example 4.17. Implementing a class bridge

```
@Entity
@Indexed
@ClassBridge(name="branchnetwork",
             index=Index.TOKENIZED,
             store=Store.YES,
             impl = CatFieldsClassBridge.class,
             params = @Parameter( name="sepChar", value=" " ) )
public class Department {
    private int id;
    private String network;
    private String branchHead;
    private String branch;
    private Integer maxEmployees
    ...
}


public class CatFieldsClassBridge implements FieldBridge,
 ParameterizedBridge {
    private String sepChar;

    public void setParameterValues(Map parameters) {
        this.sepChar = (String) parameters.get( "sepChar" );
    }

    public void set(String name, Object value, Document document,
 LuceneOptions luceneOptions) {
        // In this particular class the name of the new field was
 passed
        // from the name field of the ClassBridge Annotation. This
 is not
        // a requirement. It just works that way in this instance.
 The
        // actual name could be supplied by hard coding it below.
        Department dep = (Department) value;
        String fieldValue1 = dep.getBranch();
        if ( fieldValue1 == null ) {
            fieldValue1 = "";
        }
        String fieldValue2 = dep.getNetwork();
        if ( fieldValue2 == null ) {
            fieldValue2 = "";
        }
        String fieldValue = fieldValue1 + sepChar + fieldValue2;
        Field field = new Field( name, fieldValue,
 luceneOptions.getStore(), luceneOptions.getIndex(),
 luceneOptions.getTermVector() );
        field.setBoost( luceneOptions.getBoost() );
        document.add( field );
   }
}
```

In this example, the particular `CatFieldsClassBridge` is applied to the `department` instance, the field bridge then concatenate both branch and network and index the concatenation.

4.3. Providing your own id

Warning

This part of the documentation is a work in progress.

You can provide your own id for Hibernate Search if you are extending the internals. You will have to generate a unique value so it can be given to Lucene to be indexed. This will have to be given to Hibernate Search when you create an org.hibernate.search.Work object - the document id is required in the constructor.

4.3.1. The ProvidedId annotation

Unlike conventional Hibernate Search API and @DocumentId, this annotation is used on the class and not a field. You also can provide your own bridge implementation when you put in this annotation by calling the bridge() which is on @ProvidedId. Also, if you annotate a class with @ProvidedId, your subclasses will also get the annotation - but it is not done by using the java.lang.annotations.@Inherited. Be sure however, to *not* use this annotation with @DocumentId as your system will break.

Example 4.18. Providing your own id

```
@ProvidedId (bridge = org.my.own.package.MyCustomBridge)
@Indexed
public class MyClass{
    @Field
    String MyString;
    ...
}
```

Chapter 5. Querying

The second most important capability of Hibernate Search is the ability to execute a Lucene query and retrieve entities managed by an Hibernate session, providing the power of Lucene without leaving the Hibernate paradigm, and giving another dimension to the Hibernate classic search mechanisms (HQL, Criteria query, native SQL query). Preparing and executing a query consists of four simple steps:

- Creating a `FullTextSession`
- Creating a Lucene query
- Wrapping the Lucene query using a `org.hibernate.Query`
- Executing the search by calling for example `list()` or `scroll()`

To access the querying facilities, you have to use an `FullTextSession`. This Search specific session wraps a regular `org.hibernate.Session` to provide query and indexing capabilities.

Example 5.1. Creating a FullTextSession

```
Session session = sessionFactory.openSession();
...
FullTextSession fullTextSession =
 Search.getFullTextSession(session);
```

The actual search facility is built on native Lucene queries which the following example illustrates.

Example 5.2. Creating a Lucene query

```
org.apache.lucene.queryParser.QueryParser parser =
    new QueryParser("title", new StopAnalyzer() );

org.apache.lucene.search.Query luceneQuery = parser.parse(
 "summary:Festina Or brand:Seiko" );
org.hibernate.Query fullTextQuery =
 fullTextSession.createFullTextQuery( luceneQuery );

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

The Hibernate query built on top of the Lucene query is a regular `org.hibernate.Query`, which means you are in the same paradigm as the other Hibernate query facilities (HQL, Native or Criteria). The regular `list()` , `uniqueResult()`, `iterate()` and `scroll()` methods can be used.

In case you are using the Java Persistence APIs of Hibernate (aka EJB 3.0 Persistence), the same extensions exist:

Example 5.3. Creating a Search query using the JPA API

```
EntityManager em = entityManagerFactory.createEntityManager();

FullTextEntityManager fullTextEntityManager =

 org.hibernate.hibernate.search.jpa.Search.getFullTextEntityManager(em);

...
org.apache.lucene.queryParser.QueryParser parser =
    new QueryParser("title", new StopAnalyzer() );

org.apache.lucene.search.Query luceneQuery = parser.parse(
 "summary:Festina Or brand:Seiko" );
javax.persistence.Query fullTextQuery =
 fullTextEntityManager.createFullTextQuery( luceneQuery );

List result = fullTextQuery.getResultList(); //return a list of
 managed objects
```

The following examples we will use the Hibernate APIs but the same example can be easily rewritten with the Java Persistence API by just adjusting the way the `FullTextQuery` is retrieved.

5.1. Building queries

Hibernate Search queries are built on top of Lucene queries which gives you total freedom on the type of Lucene query you want to execute. However, once built, Hibernate Search wraps further query processing using `org.hibernate.Query` as your primary query manipulation API.

5.1.1. Building a Lucene query

It is out of the scope of this documentation on how to exactly build a Lucene query. Please refer to the online Lucene documentation or get hold of a copy of either Lucene In Action or Hibernate Search in Action.

5.1.2. Building a Hibernate Search query

5.1.2.1. Generality

Once the Lucene query is built, it needs to be wrapped into an Hibernate Query.

Example 5.4. Wrapping a Lucene query into a Hibernate Query

```
FullTextSession fullTextSession = Search.getFullTextSession( session
 );
org.hibernate.Query fullTextQuery =
 fullTextSession.createFullTextQuery( luceneQuery );
```

If not specified otherwise, the query will be executed against all indexed entities, potentially returning all types of indexed classes. It is advised, from a performance point of view, to restrict the returned types:

Example 5.5. Filtering the search result by entity type

```
org.hibernate.Query fullTextQuery =
 fullTextSession.createFullTextQuery( luceneQuery, Customer.class );
// or
fullTextQuery = fullTextSession.createFullTextQuery( luceneQuery,
 Item.class, Actor.class );
```

The first example returns only matching `Customer`s, the second returns matching `Actor`s and `Item`s. The type restriction is fully polymorphic which means that if there are two indexed subclasses `Salesman` and `Customer` of the baseclass `Person`, it is possible to just specify `Person.class` in order to filter on result types.

5.1.2.2. Pagination

Out of performance reasons it is recommended to restrict the number of returned objects per query. In fact is a very common use case anyway that the user navigates from one page to an other. The way to define pagination is exactly the way you would define pagination in a plain HQL or Criteria query.

Example 5.6. Defining pagination for a search query

```
org.hibernate.Query fullTextQuery =
 fullTextSession.createFullTextQuery( luceneQuery, Customer.class );
fullTextQuery.setFirstResult(15); //start from the 15th element
fullTextQuery.setMaxResults(10); //return 10 elements
```

Note

It is still possible to get the total number of matching elements regardless of the pagination via `fulltextQuery.getResultSize()`

5.1.2.3. Sorting

Apache Lucene provides a very flexible and powerful way to sort results. While the default sorting (by relevance) is appropriate most of the time, it can be interesting to sort by one or several other properties. In order to do so set the Lucene Sort object to apply a Lucene sorting strategy.

Example 5.7. Specifying a Lucene `Sort` in order to sort the results

```
org.hibernate.search.FullTextQuery query = s.createFullTextQuery(
 query, Book.class );
org.apache.lucene.search.Sort sort = new Sort(new
 SortField("title"));
query.setSort(sort);
List results = query.list();
```

One can notice the `FullTextQuery` interface which is a sub interface of `org.hibernate.Query`. Be aware that fields used for sorting must not be tokenized.

5.1.2.4. Fetching strategy

When you restrict the return types to one class, Hibernate Search loads the objects using a single query. It also respects the static fetching strategy defined in your domain model.

It is often useful, however, to refine the fetching strategy for a specific use case.

Example 5.8. Specifying `FetchMode` on a query

```
Criteria criteria = s.createCriteria( Book.class ).setFetchMode(
 "authors", FetchMode.JOIN );
s.createFullTextQuery( luceneQuery ).setCriteriaQuery( criteria );
```

In this example, the query will return all Books matching the luceneQuery. The authors collection will be loaded from the same query using an SQL outer join.

When defining a criteria query, it is not needed to restrict the entity types returned while creating the Hibernate Search query from the full text session: the type is guessed from the criteria query itself. Only fetch mode can be adjusted, refrain from applying any other restriction.

One cannot use `setCriteriaQuery` if more than one entity type is expected to be returned.

5.1.2.5. Projection

For some use cases, returning the domain object (graph) is overkill. Only a small subset of the properties is necessary. Hibernate Search allows you to return a subset of properties:

Example 5.9. Using projection instead of returning the full domain object

```
org.hibernate.search.FullTextQuery query = s.createFullTextQuery(
 luceneQuery, Book.class );
query.setProjection( "id", "summary", "body", "mainAuthor.name" );
List results = query.list();
Object[] firstResult = (Object[]) results.get(0);
Integer id = firstResult[0];
String summary = firstResult[1];
String body = firstResult[2];
String authorName = firstResult[3];
```

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), but has some constraints:

- the properties projected must be stored in the index (`@Field(store=Store.YES)`), which increase the index size
- the properties projected must use a `FieldBridge` implementing `org.hibernate.search.bridge.TwoWayFieldBridge` or `org.hibernate.search.bridge.TwoWayStringBridge`, the latter being the simpler version. All Hibernate Search built-in types are two-way.
- you can only project simple properties of the indexed entity or its embedded associations. This means you cannot project a whole embedded entity.
- projection does not work on collections or maps which are indexed via `@IndexedEmbedded`

Projection is useful for another kind of use cases. Lucene provides some metadata information to the user about the results. By using some special placeholders, the projection mechanism can retrieve them:

Example 5.10. Using projection in order to retrieve meta data

```
org.hibernate.search.FullTextQuery query = s.createFullTextQuery(
 luceneQuery, Book.class );
query.setProjection( FullTextQuery.SCORE, FullTextQuery.THIS,
 "mainAuthor.name" );
List results = query.list();
Object[] firstResult = (Object[]) results.get(0);
float score = firstResult[0];
Book book = firstResult[1];
String authorName = firstResult[2];
```

You can mix and match regular fields and special placeholders. Here is the list of available placeholders:

- FullTextQuery.THIS: returns the initialized and managed entity (as a non projected query would have done).
- FullTextQuery.DOCUMENT: returns the Lucene Document related to the object projected.
- FullTextQuery.OBJECT_CLASS: returns the class of the indexed entity.
- FullTextQuery.SCORE: returns the document score in the query. Scores are handy to compare one result against an other for a given query but are useless when comparing the result of different queries.
- FullTextQuery.ID: the id property value of the projected object.
- FullTextQuery.DOCUMENT_ID: the Lucene document id. Careful, Lucene document id can change overtime between two different IndexReader opening (this feature is experimental).
- FullTextQuery.EXPLANATION: returns the Lucene Explanation object for the matching object/document in the given query. Do not use if you retrieve a lot of data. Running explanation typically is as costly as running the whole Lucene query per matching element. Make sure you use projection!

5.2. Retrieving the results

Once the Hibernate Search query is built, executing it is in no way different than executing a HQL or Criteria query. The same paradigm and object semantic applies. All the common operations are available: `list()`, `uniqueResult()`, `iterate()`, `scroll()`.

5.2.1. Performance considerations

If you expect a reasonable number of results (for example using pagination) and expect to work on all of them, `list()` or `uniqueResult()` are recommended. `list()` work best if the entity `batch-size` is set up properly. Note that Hibernate Search has to process all Lucene Hits elements (within the pagination) when using `list()` , `uniqueResult()` and `iterate()`.

If you wish to minimize Lucene document loading, `scroll()` is more appropriate. Don't forget to close the `ScrollableResults` object when you're done, since it keeps Lucene resources. If you expect to use `scroll`, but wish to load objects in batch, you can use `query.setFetchSize()`. When an object is accessed, and if not already loaded, Hibernate Search will load the next `fetchSize` objects in one pass.

Pagination is a preferred method over scrolling though.

5.2.2. Result size

It is sometime useful to know the total number of matching documents:

- for the Google-like feature 1-10 of about 888,000,000
- to implement a fast pagination navigation
- to implement a multi step search engine (adding approximation if the restricted query return no or not enough results)

Of course it would be too costly to retrieve all the matching documents. Hibernate Search allows you to retrieve the total number of matching documents regardless of the pagination parameters. Even more interesting, you can retrieve the number of matching elements without triggering a single object load.

Example 5.11. Determining the result size of a query

```
org.hibernate.search.FullTextQuery query = s.createFullTextQuery(
 luceneQuery, Book.class );
assert 3245 == query.getResultSize(); //return the number of
 matching books without loading a single one

org.hibernate.search.FullTextQuery query = s.createFullTextQuery(
 luceneQuery, Book.class );
query.setMaxResult(10);
List results = query.list();
assert 3245 == query.getResultSize(); //return the total number of
 matching books regardless of pagination
```

Note

Like Google, the number of results is approximative if the index is not fully up-to-date with the database (asynchronous cluster for example).

5.2.3. ResultTransformer

Especially when using projection, the data structure returned by a query (an object array in this case), is not always matching the application needs. It is possible to apply a `ResultTransformer` operation post query to match the targeted data structure:

Example 5.12. Using ResultTransformer in conjunction with projections

```
org.hibernate.search.FullTextQuery query = s.createFullTextQuery(
 luceneQuery, Book.class );
query.setProjection( "title", "mainAuthor.name" );

query.setResultTransformer(
    new StaticAliasToBeanResultTransformer( BookView.class, "title",
 "author" )
);
List<BookView> results = (List<BookView>) query.list();
for(BookView view : results) {
    log.info( "Book: " + view.getTitle() + ", " + view.getAuthor()
 );
}
```

Examples of `ResultTransformer` implementations can be found in the Hibernate Core codebase.

5.2.4. Understanding results

You will find yourself sometimes puzzled by a result showing up in a query or a result not showing up in a query. Luke is a great tool to understand those mysteries. However, Hibernate Search also gives you access to the Lucene `Explanation` object for a given result (in a given query). This class is considered fairly advanced to Lucene users but can provide a good understanding of the scoring of an object. You have two ways to access the Explanation object for a given result:

- Use the `fullTextQuery.explain(int)` method
- Use projection

The first approach takes a document id as a parameter and return the Explanation object. The document id can be retrieved using projection and the `FullTextQuery.DOCUMENT_ID` constant.

Warning

The Document id has nothing to do with the entity id. Do not mess up these two notions.

The second approach let's you project the `Explanation` object using the `FullTextQuery.EXPLANATION` constant.

Example 5.13. Retrieving the Lucene Explanation object using projection

```
FullTextQuery ftQuery = s.createFullTextQuery( luceneQuery,
 Dvd.class )
        .setProjection( FullTextQuery.DOCUMENT_ID,
 FullTextQuery.EXPLANATION, FullTextQuery.THIS );
@SuppressWarnings("unchecked") List<Object[]> results =
 ftQuery.list();
for (Object[] result : results) {
    Explanation e = (Explanation) result[1];
    display( e.toString() );
}
```

Be careful, building the explanation object is quite expensive, it is roughly as expensive as running the Lucene query again. Don't do it if you don't need the object

5.3. Filters

Apache Lucene has a powerful feature that allows to filter query results according to a custom filtering process. This is a very powerful way to apply additional data restrictions, especially since filters can be cached and reused. Some interesting use cases are:

- security
- temporal data (eg. view only last month's data)
- population filter (eg. search limited to a given category)
- and many more

Hibernate Search pushes the concept further by introducing the notion of parameterizable named filters which are transparently cached. For people familiar with the notion of Hibernate Core filters, the API is very similar:

Example 5.14. Enabling fulltext filters for a given query

```
fullTextQuery = s.createFullTextQuery( query, Driver.class );
fullTextQuery.enableFullTextFilter("bestDriver");
fullTextQuery.enableFullTextFilter("security").setParameter(
 "login", "andre" );
fullTextQuery.list(); //returns only best drivers where andre has
 credentials
```

In this example we enabled two filters on top of the query. You can enable (or disable) as many filters as you like.

Declaring filters is done through the `@FullTextFilterDef` annotation. This annotation can be on any `@Indexed` entity regardless of the query the filter is later applied to. This implies that filter definitions are global and their names must be unique. A `SearchException` is thrown in case two different `@FullTextFilterDef` annotations with the same name are defined. Each named filter has to specify its actual filter implementation.

Example 5.15. Defining and implementing a Filter

```
@Entity
@Indexed
@FullTextFilterDefs( {
    @FullTextFilterDef(name = "bestDriver", impl =
 BestDriversFilter.class),
    @FullTextFilterDef(name = "security", impl =
 SecurityFilterFactory.class)
})
public class Driver { ... }
```

```
public class BestDriversFilter extends
 org.apache.lucene.search.Filter {

    public DocIdSet getDocIdSet(IndexReader reader) throws
 IOException {
        OpenBitSet bitSet = new OpenBitSet( reader.maxDoc() );
        TermDocs termDocs = reader.termDocs( new Term( "score", "5"
 ) );
        while ( termDocs.next() ) {
            bitSet.set( termDocs.doc() );
        }
        return bitSet;
    }
}
```

`BestDriversFilter` is an example of a simple Lucene filter which reduces the result set to drivers whose score is 5. In this example the specified filter implements the `org.apache.lucene.search.Filter` directly and contains a no-arg constructor.

If your Filter creation requires additional steps or if the filter you want to use does not have a no-arg constructor, you can use the factory pattern:

Example 5.16. Creating a filter using the factory pattern

```
@Entity
@Indexed
@FullTextFilterDef(name = "bestDriver", impl =
 BestDriversFilterFactory.class)
public class Driver { ... }

public class BestDriversFilterFactory {

    @Factory
    public Filter getFilter() {
        //some additional steps to cache the filter results per
 IndexReader
        Filter bestDriversFilter = new BestDriversFilter();
        return new CachingWrapperFilter(bestDriversFilter);
    }
}
```

Hibernate Search will look for a `@Factory` annotated method and use it to build the filter instance. The factory must have a no-arg constructor. For people familiar with JBoss Seam, this is similar to the component factory pattern, but the annotation is different!

Named filters come in handy where parameters have to be passed to the filter. For example a security filter might want to know which security level you want to apply:

Example 5.17. Passing parameters to a defined filter

```
fullTextQuery = s.createFullTextQuery( query, Driver.class );
fullTextQuery.enableFullTextFilter("security").setParameter(
 "level", 5 );
```

Each parameter name should have an associated setter on either the filter or filter factory of the targeted named filter definition.

Example 5.18. Using parameters in the actual filter implementation

```
public class SecurityFilterFactory {
    private Integer level;

    /**
     * injected parameter
     */
    public void setLevel(Integer level) {
        this.level = level;
    }

    @Key
    public FilterKey getKey() {
        StandardFilterKey key = new StandardFilterKey();
        key.addParameter( level );
        return key;
    }

    @Factory
    public Filter getFilter() {
        Query query = new TermQuery( new Term("level",
 level.toString() ) );
        return new CachingWrapperFilter( new
 QueryWrapperFilter(query) );
    }
}
```

Note the method annotated `@Key` returning a `FilterKey` object. The returned object has a special contract: the key object must implement `equals()` / `hashCode()` so that 2 keys are equal if and only if the given `Filter` types are the same and the set of parameters are the same. In other words, 2 filter keys are equal if and only if the filters from which the keys are generated can be interchanged. The key object is used as a key in the cache mechanism.

`@Key` methods are needed only if:

- you enabled the filter caching system (enabled by default)
- your filter has parameters

In most cases, using the `StandardFilterKey` implementation will be good enough. It delegates the `equals()` / `hashCode()` implementation to each of the parameters equals and hashcode methods.

As mentioned before the defined filters are per default cached and the cache uses a combination of hard and soft references to allow disposal of memory when needed. The hard reference cache keeps track of the most recently used filters and transforms the ones least used to `SoftReferences`

when needed. Once the limit of the hard reference cache is reached additional filters are cached as `SoftReferences`. To adjust the size of the hard reference cache, use `hibernate.search.filter.cache_strategy.size` (defaults to 128). For advanced use of filter caching, you can implement your own `FilterCachingStrategy`. The classname is defined by `hibernate.search.filter.cache_strategy`.

This filter caching mechanism should not be confused with caching the actual filter results. In Lucene it is common practice to wrap filters using the `IndexReader` around a `CachingWrapperFilter`. The wrapper will cache the `DocIdSet` returned from the `getDocIdSet(IndexReader reader)` method to avoid expensive recomputation. It is important to mention that the computed `DocIdSet` is only cachable for the same `IndexReader` instance, because the reader effectively represents the state of the index at the moment it was opened. The document list cannot change within an opened `IndexReader`. A different/new `IndexReader` instance, however, works potentially on a different set of `Document`s (either from a different index or simply because the index has changed), hence the cached `DocIdSet` has to be recomputed.

Hibernate Search also helps with this aspect of caching.
Per default the `cache` flag of `@FullTextFilterDef` is set to `FilterCacheModeType.INSTANCE_AND_DOCIDSETRESULTS` which will automatically cache the filter instance as well as wrap the specified filter around a Hibernate specific implementation of `CachingWrapperFilter` (`org.hibernate.search.filter.CachingWrapperFilter`). In contrast to Lucene's version of this class `SoftReference`s are used together with a hard reference count (see discussion about filter cache). The hard reference count can be adjusted using `hibernate.search.filter.cache_docidresults.size` (defaults to 5). The wrapping behaviour can be controlled using the `@FullTextFilterDef.cache` parameter. There are three different values for this parameter:

Value	Definition
FilterCacheModeType.NONE	No filter instance and no result is cached by Hibernate Search. For every filter call, a new filter instance is created. This setting might be useful for rapidly changing data sets or heavily memory constrained environments.
FilterCacheModeType.INSTANCE_ONLY	The filter instance is cached and reused across concurrent `Filter.getDocIdSet()` calls. `DocIdSet` results are not cached. This setting

Value	Definition
	is useful when a filter uses its own specific caching mechanism or the filter results change dynamically due to application specific events making `DocIdSet` caching in both cases unnecessary.
FilterCacheModeType.INSTANCE_AND_DOCIDSETRESULTS	Both the filter instance and the `DocIdSet` results are cached. This is the default value.

Last but not least - why should filters be cached? There are two areas where filter caching shines:

- the system does not update the targeted entity index often (in other words, the IndexReader is reused a lot)
- the Filter's DocIdSet is expensive to compute (compared to the time spent to execute the query)

5.4. Optimizing the query process

Query performance depends on several criteria:

- the Lucene query itself: read the literature on this subject
- the number of object loaded: use pagination (always ;-)) or index projection (if needed)
- the way Hibernate Search interacts with the Lucene readers: defines the appropriate Reader strategy.

5.5. Native Lucene Queries

If you wish to use some specific features of Lucene, you can always run Lucene specific queries. Check Chapter 8, *Advanced features* for more information.

Chapter 6. Manual indexing

6.1. Indexing

It is sometimes useful to index an entity even if this entity is not inserted or updated to the database. This is for example the case when you want to build your index for the first time. `FullTextSession.index()` allows you to do so.

Example 6.1. Indexing an entity via `FullTextSession.index()`

```
FullTextSession fullTextSession =
 Search.getFullTextSession(session);
Transaction tx = fullTextSession.beginTransaction();
for (Customer customer : customers) {
    fullTextSession.index(customer);
}
tx.commit(); //index are written at commit time
```

For maximum efficiency, Hibernate Search batches index operations and executes them at commit time. If you expect to index a lot of data, however, you need to be careful about memory consumption since all documents are kept in a queue until the transaction commit. You can potentially face an `OutOfMemoryException`. To avoid this exception, you can use `fullTextSession.flushToIndexes()`. Every time `fullTextSession.flushToIndexes()` is called (or if the transaction is committed), the batch queue is processed (freeing memory) applying all index changes. Be aware that once flushed changes cannot be rolled back.

Note

`hibernate.search.worker.batch_size` has been deprecated in favor of this explicit API which provides better control

Other parameters which also can affect indexing time and memory consumption are:

- `hibernate.search.[default|<indexname>].indexwriter.[batch|transaction].max_buffered_do`
- `hibernate.search.[default|<indexname>].indexwriter.[batch|transaction].max_field_lengt`
- `hibernate.search.[default|<indexname>].indexwriter.[batch|transaction].max_merge_docs`
- `hibernate.search.[default|<indexname>].indexwriter.[batch|transaction].merge_factor`
- `hibernate.search.[default|<indexname>].indexwriter.[batch|transaction].ram_buffer_size`

- `hibernate.search.[default|<indexname>].indexwriter.[batch|transaction].term_index_inte`

These parameters are Lucene specific and Hibernate Search is just passing these parameters through - see Section 3.8, “Tuning Lucene indexing performance” for more details.

Example 6.2. Efficiently indexing a given class (useful for index (re)initialization)

```
fullTextSession.setFlushMode(FlushMode.MANUAL);
fullTextSession.setCacheMode(CacheMode.IGNORE);
transaction = fullTextSession.beginTransaction();
//Scrollable results will avoid loading too many objects in memory
ScrollableResults results = fullTextSession.createCriteria(
 Email.class )
    .setFetchSize(BATCH_SIZE)
    .scroll( ScrollMode.FORWARD_ONLY );
int index = 0;
while( results.next() ) {
    index++;
    fullTextSession.index( results.get(0) ); //index each element
    if (index % BATCH_SIZE == 0) {
        fullTextSession.flushToIndexes(); //apply changes to indexes
        fullTextSession.clear(); //clear since the queue is
 processed
    }
}
transaction.commit();
```

Try to use a batch size that guarantees that your application will not run out of memory.

6.2. Purging

It is equally possible to remove an entity or all entities of a given type from a Lucene index without the need to physically remove them from the database. This operation is named purging and is also done through the `FullTextSession`.

Example 6.3. Purging a specific instance of an entity from the index

```
FullTextSession fullTextSession =
 Search.getFullTextSession(session);
Transaction tx = fullTextSession.beginTransaction();
for (Customer customer : customers) {
    fullTextSession.purge( Customer.class, customer.getId() );
}
tx.commit(); //index are written at commit time
```

Purging will remove the entity with the given id from the Lucene index but will not touch the database.

If you need to remove all entities of a given type, you can use the `purgeAll` method. This operation remove all entities of the type passed as a parameter as well as all its subtypes.

Example 6.4. Purging all instances of an entity from the index

```
FullTextSession fullTextSession =
 Search.getFullTextSession(session);
Transaction tx = fullTextSession.beginTransaction();
fullTextSession.purgeAll( Customer.class );
//optionally optimize the index
//fullTextSession.getSearchFactory().optimize( Customer.class );
tx.commit(); //index are written at commit time
```

It is recommended to optimize the index after such an operation.

Note

Methods `index`, `purge` and `purgeAll` are available on `FullTextEntityManager` as well.

Chapter 7. Index Optimization

From time to time, the Lucene index needs to be optimized. The process is essentially a defragmentation. Until an optimization is triggered Lucene only marks deleted documents as such, no physical deletions are applied. During the optimization process the deletions will be applied which also effects the number of files in the Lucene Directory.

Optimizing the Lucene index speeds up searches but has no effect on the indexation (update) performance. During an optimization, searches can be performed, but will most likely be slowed down. All index updates will be stopped. It is recommended to schedule optimization:

- on an idle system or when the searches are less frequent
- after a lot of index modifications

7.1. Automatic optimization

Hibernate Search can automatically optimize an index after:

- a certain amount of operations (insertion, deletion)
- or a certain amount of transactions

The configuration for automatic index optimization can be defined on a global level or per index:

Example 7.1. Defining automatic optimization parameters

```
hibernate.search.default.optimizer.operation_limit.max = 1000
hibernate.search.default.optimizer.transaction_limit.max = 100
hibernate.search.Animal.optimizer.transaction_limit.max = 50
```

An optimization will be triggered to the `Animal` index as soon as either:

- the number of additions and deletions reaches 1000
- the number of transactions reaches 50 (`hibernate.search.Animal.optimizer.transaction_limit.max` having priority over `hibernate.search.default.optimizer.transaction_limit.max`)

If none of these parameters are defined, no optimization is processed automatically.

7.2. Manual optimization

You can programmatically optimize (defragment) a Lucene index from Hibernate Search through the `SearchFactory`:

Example 7.2. Programmatic index optimization

```
FullTextSession fullTextSession =
 Search.getFullTextSession(regularSession);
SearchFactory searchFactory = fullTextSession.getSearchFactory();

searchFactory.optimize(Order.class);
// or
searchFactory.optimize();
```

The first example optimizes the Lucene index holding `Order`s; the second, optimizes all indexes.

Note

`searchFactory.optimize()` has no effect on a JMS backend. You must apply the optimize operation on the Master node.

7.3. Adjusting optimization

Apache Lucene has a few parameters to influence how optimization is performed. Hibernate Search exposes those parameters.

Further index optimization parameters include:

- `hibernate.search.[default|<indexname>].indexwriter.[batch|transaction].max_buffered_do`
- `hibernate.search.[default|<indexname>].indexwriter.[batch|transaction].max_field_lengt`
- `hibernate.search.[default|<indexname>].indexwriter.[batch|transaction].max_merge_docs`
- `hibernate.search.[default|<indexname>].indexwriter.[batch|transaction].merge_factor`
- `hibernate.search.[default|<indexname>].indexwriter.[batch|transaction].ram_buffer_size`
- `hibernate.search.[default|<indexname>].indexwriter.[batch|transaction].term_index_inte`

See Section 3.8, “Tuning Lucene indexing performance” for more details.

Chapter 8. Advanced features

8.1. SearchFactory

The `SearchFactory` object keeps track of the underlying Lucene resources for Hibernate Search, it's also a convenient way to access Lucene natively. The `SearchFactory` can be accessed from a `FullTextSession`:

Example 8.1. Accessing the `SearchFactory`

```
FullTextSession fullTextSession =
 Search.getFullTextSession(regularSession);
SearchFactory searchFactory = fullTextSession.getSearchFactory();
```

8.2. Accessing a Lucene Directory

You can always access the Lucene directories through plain Lucene, the Directory structure is in no way different with or without Hibernate Search. However there are some more convenient ways to access a given Directory. The `SearchFactory` keeps track of the `DirectoryProvider`s per indexed class. One directory provider can be shared amongst several indexed classes if the classes share the same underlying index directory. While usually not the case, a given entity can have several `DirectoryProvider`s if the index is sharded (see Section 3.2, “Sharding indexes”).

Example 8.2. Accessing the Lucene `Directory`

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

In this example, directory points to the lucene index storing `Order`s information. Note that the obtained Lucene directory must not be closed (this is Hibernate Search responsibility).

8.3. Using an IndexReader

Queries in Lucene are executed on an `IndexReader`. Hibernate Search caches all index readers to maximize performance. Your code can access this cached resources, but you have to follow some "good citizen" rules.

Example 8.3. Accessing an `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);
}
```

The ReaderProvider (described in Reader strategy), will open an IndexReader on top of the index(es) referenced by the directory providers. Because this `IndexReader` is shared amongst several clients, you must adhere to the following rules:

- Never call indexReader.close(), but always call readerProvider.closeReader(reader), preferably in a finally block.
- Don't use this `IndexReader` for modification operations (you would get an exception). If you want to use a read/write index reader, open one from the Lucene Directory object.

Aside from those rules, you can use the IndexReader freely, especially to do native queries. Using the shared `IndexReader`s will make most queries more efficient.

8.4. Customizing Lucene's scoring formula

Lucene allows the user to customize its scoring formula by extending `org.apache.lucene.search.Similarity`. The abstract methods defined in this class match the factors of the following formula calculating the score of query q for document d:

score(q,d) = coord(q,d) · queryNorm(q) · $\#_{t \text{ in } q}$ (tf(t in d) · idf(t)^2 · t.getBoost() · norm(t,d))

Factor	Description
tf(t ind)	Term frequency factor for the term (t) in the document (d).

Factor	Description
idf(t)	Inverse document frequency of the term.
coord(q,d)	Score factor based on how many of the query terms are found in the specified document.
queryNorm(q)	Normalizing factor used to make scores between queries comparable.
t.getBoost()	Field boost.
norm(t,d)	Encapsulates a few (indexing time) boost and length factors.

It is beyond the scope of this manual to explain this formula in more detail. Please refer to `Similarity`'s Javadocs for more information.

Hibernate Search provides two ways to modify Lucene's similarity calculation. First you can set the default similarity by specifying the fully specified classname of your `Similarity` implementation using the property `hibernate.search.similarity`. The default value is `org.apache.lucene.search.DefaultSimilarity`. Additionally you can override the default similarity on class level using the `@Similarity` annotation.

```
@Entity
@Indexed
@Similarity(impl = DummySimilarity.class)
public class Book {
   ...
}
```

As an example, let's assume it is not important how often a term appears in a document. Documents with a single occurrence of the term should be scored the same as documents with multiple occurrences. In this case your custom implementation of the method `tf(float freq)` should return 1.0.