

Hibernate Search

Apache Lucene™ Integration

Reference Guide

3.4.0.Final

Preface	vii
1. Getting started	1
1.1. System Requirements	1
1.2. Using Maven	2
1.3. Configuration	3
1.4. Indexing	7
1.5. Searching	7
1.6. Analyzer	8
1.7. What's next	10
2. Architecture	11
2.1. Overview	11
2.2. Back end	12
2.2.1. Back end types	12
2.2.2. Work execution	14
2.3. Reader strategy	14
2.3.1. Shared	14
2.3.2. Not-shared	15
2.3.3. Custom	15
3. Configuration	17
3.1. Enabling Hibernate Search and automatic indexing	17
3.1.1. Enabling Hibernate Search	17
3.1.2. Automatic indexing	17
3.2. Directory configuration	17
3.3. Sharding indexes	23
3.4. Sharing indexes	25
3.5. Worker configuration	25
3.6. JMS Master/Slave configuration	28
3.6.1. Slave nodes	29
3.6.2. Master node	30
3.7. JGroups Master/Slave configuration	31
3.7.1. Slave nodes	31
3.7.2. Master node	32
3.7.3. JGroups channel configuration	32
3.8. Infinispan Directory configuration	33
3.8.1. Requirements	34
3.8.2. Architecture	34
3.8.3. Infinispan Configuration	35
3.9. Reader strategy configuration	36
3.10. Tuning Lucene indexing performance	36
3.11. LockFactory configuration	41
3.12. Exception Handling Configuration	42
4. Mapping entities to the index structure	45
4.1. Mapping an entity	45
4.1.1. Basic mapping	45

4.1.2. Mapping properties multiple times	49
4.1.3. Embedded and associated objects	49
4.2. Boosting	53
4.2.1. Static index time boosting	53
4.2.2. Dynamic index time boosting	54
4.3. Analysis	55
4.3.1. Default analyzer and analyzer by class	55
4.3.2. Named analyzers	56
4.3.3. Dynamic analyzer selection (experimental)	62
4.3.4. Retrieving an analyzer	63
4.4. Bridges	64
4.4.1. Built-in bridges	64
4.4.2. Custom bridges	66
4.5. Providing your own id	71
4.5.1. The ProvidedId annotation	71
4.6. Programmatic API	72
4.6.1. Mapping an entity as indexable	73
4.6.2. Adding DocumentId to indexed entity	74
4.6.3. Defining analyzers	74
4.6.4. Defining full text filter definitions	76
4.6.5. Defining fields for indexing	77
4.6.6. Programmatically defining embedded entities	78
4.6.7. Contained In definition	79
4.6.8. Date/Calendar Bridge	81
4.6.9. Defining bridges	82
4.6.10. Mapping class bridge	83
4.6.11. Mapping dynamic boost	84
5. Querying	85
5.1. Building queries	87
5.1.1. Building a Lucene query using the Lucene API	87
5.1.2. Building a Lucene query with the Hibernate Search query DSL	87
5.1.3. Building a Hibernate Search query	94
5.2. Retrieving the results	101
5.2.1. Performance considerations	102
5.2.2. Result size	102
5.2.3. ResultTransformer	103
5.2.4. Understanding results	103
5.3. Filters	104
5.3.1. Using filters in a sharded environment	108
5.4. Faceting	110
5.4.1. Creating a faceting request	112
5.4.2. Applying a faceting request	113
5.4.3. Restricting query results	114
5.5. Optimizing the query process	115

5.5.1. Caching index values: FieldCache	115
6. Manual index changes	117
6.1. Adding instances to the index	117
6.2. Deleting instances from the index	117
6.3. Rebuilding the whole index	118
6.3.1. Using flushToIndexes()	118
6.3.2. Using a MassIndexer	119
7. Index Optimization	123
7.1. Automatic optimization	123
7.2. Manual optimization	124
7.3. Adjusting optimization	124
8. Monitoring	125
8.1. JMX	125
8.1.1. StatisticsInfoMBean	125
8.1.2. IndexControlMBean	125
8.1.3. IndexingProgressMonitorMBean	125
9. Advanced features	127
9.1. Accessing the SearchFactory	127
9.2. Accessing a Lucene Directory	127
9.3. Using an IndexReader	127
9.4. Use external services in Hibernate Search components (experimental)	128
9.4.1. Exposing a service	129
9.4.2. Using a service	130
9.5. Customizing Lucene's scoring formula	131
10. Further reading	133

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 models. 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) [http://www.hibernate.org] and [Apache Lucene](http://lucene.apache.org) [http://lucene.apache.org].

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/quick-start.html) [http://hibernate.org/quick-start.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://www.oracle.com/technetwork/java/javase/downloads/index.html].
Hibernate Search	hibernate-search-3.4.0.Final.jar and all runtime dependencies. You can get the jar artifacts either from the dist/lib directory of the Hibernate Search distribution [http://sourceforge.net/projects/hibernate/files/hibernate-search/] or you can download them from the JBoss maven repository [http://repository.jboss.org/nexus/content/groups/public-jboss/].
Hibernate Core	This instructions have been tested against Hibernate 3.6. You will need hibernate-core-3.6.3.Final.jar and its transitive dependencies (either from the distribution bundle [http://sourceforge.net/projects/hibernate/files/hibernate3/] or the maven repository).
JPA 2	<p>Even though Hibernate Search can be used without JPA annotations the following instructions will use them for basic entity configuration (<i>@Entity</i>, <i>@Id</i>, <i>@OneToMany</i>,...). This part of the configuration could also be expressed in xml or code.</p> <p>Hibernate Search, however, has itself its own set of annotations (<i>@Indexed</i>, <i>@DocumentId</i>, <i>@Field</i>,...) for which there exists so far no alternative configuration.</p>

1.2. Using Maven

Instead of managing all dependencies manually, maven users have the possibility to use the *JBoss maven repository* [<http://repository.jboss.org/nexus/content/groups/public-jboss/>]. Add the following to your Maven `settings.xml` file (see also *Maven Getting Started* [<http://community.jboss.org/wiki/MavenGettingStarted-Users>]):

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

```
<settings>
...
<profiles>
...
  <profile>
    <id>jboss-public-repository</id>
    <repositories>
      <repository>
        <id>jboss-public-repository-group</id>
        <name>JBoss Public Maven Repository Group</name>
        <url>https://repository.jboss.org/nexus/content/groups/public-jboss/</url>
        <layout>default</layout>
        <releases>
          <enabled>true</enabled>
          <updatePolicy>never</updatePolicy>
        </releases>
        <snapshots>
          <enabled>true</enabled>
          <updatePolicy>never</updatePolicy>
        </snapshots>
      </repository>
    </repositories>
    <pluginRepositories>
      <pluginRepository>
        <id>jboss-public-repository-group</id>
        <name>JBoss Public Maven Repository Group</name>
        <url>https://repository.jboss.org/nexus/content/groups/public-jboss/</url>
        <layout>default</layout>
        <releases>
          <enabled>true</enabled>
          <updatePolicy>never</updatePolicy>
        </releases>
        <snapshots>
          <enabled>true</enabled>
          <updatePolicy>never</updatePolicy>
        </snapshots>
      </pluginRepository>
    </pluginRepositories>
  </profile>
</profiles>

<activeProfiles>
  <activeProfile>jboss-public-repository</activeProfile>
</activeProfiles>
...
```

```
</settings>
```

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>3.4.0.Final</version>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-entitymanager</artifactId>
  <version>3.6.3.Final</version>
</dependency>
```

Only the *hibernate-search* dependency is mandatory, because it contains together with its required transitive dependencies all required classes needed to use Hibernate Search. *hibernate-entitymanager* is only required if you want to use Hibernate Search in conjunction with JPA.



Note

There is no XML configuration available for Hibernate Search but we provide a powerful programmatic mapping API that elegantly replace this kind of deployment form (see [Section 4.6, "Programmatic API"](#) for more information).

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="filesystem"/>

<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 a directory provider storing the index in the file system. This will give us the ability to physically inspect the Lucene indexes created by Hibernate Search (eg via [Luke](http://code.google.com/p/luke/) [http://code.google.com/p/luke/]). Once you have a working configuration you can start experimenting with other directory providers (see [Section 3.2, "Directory configuration"](#)). Next to the directory provider you also have to specify the default base 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. The `@DocumentId` annotation 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 (see [Section 5.1.3.5, "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 criteria 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 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 4.4, "Bridges"](#).

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 will 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
    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 create an initial Lucene index for the data already present in your database. Once you have added the above properties and annotations it is time to trigger an initial batch index of your books. You can achieve this by using one of the following code snippets (see also [Section 6.3, “Rebuilding the whole index”](#)):

Example 1.6. Using Hibernate Session to index data

```
FullTextSession fullTextSession = Search.getFullTextSession(session);
fullTextSession.createIndexer().startAndWait();
```

Example 1.7. Using JPA to index data

```
EntityManager em = entityManagerFactory.createEntityManager();
FullTextEntityManager fullTextEntityManager = Search.getFullTextEntityManager(em);
fullTextEntityManager.createIndexer().startAndWait();
```

After executing the above code, you should be able to see a Lucene index under `/var/lucene/indexes/example.Book`. Go ahead and inspect this index with [Luke](http://code.google.com/p/luke/) [http://code.google.com/p/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 Lucene query (either via the Lucene API ([Section 5.1.1, “Building a Lucene query using the Lucene API”](#)) or via the Hibernate Search query DSL ([Section 5.1.2, “Building a Lucene query with the Hibernate Search query DSL”](#))) and then wrap this query into a `org.hibernate.Query` in order to get all the functionality one is used to from the Hibernate API. The following code will prepare a query against the indexed fields, execute it and return a list of `Books`.

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 using the query DSL
// alternatively you can write the Lucene query using the Lucene query parser
// or the Lucene programmatic API. The Hibernate Search DSL is recommended though
QueryBuilder qb = fullTextSession.getSearchFactory()
    .buildQueryBuilder().forEntity( Book.class ).get();
org.apache.lucene.search.Query query = qb
    .keyword()
    .onFields("title", "subtitle", "authors.name", "publicationDate")
    .matching("Java rocks!");
```

```
.createQuery();

// 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.search.jpa.Search.getFullTextEntityManager(em);
em.getTransaction().begin();

// create native Lucene query unsing the query DSL
// alternatively you can write the Lucene query using the Lucene query parser
// or the Lucene programmatic API. The Hibernate Search DSL is recommended though
QueryBuilder qb = fullTextSession.getSearchFactory()
    .buildQueryBuilder().forEntity( Book.class ).get();
org.apache.lucene.search.Query query = qb
    .keyword()
    .onFields("title", "subtitle", "authors.name", "publicationDate")
    .matching("Java rocks!");
.createQuery();

// 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** the search process. Hibernate Search offers several ways to configure the analyzer to be used (see [Section 4.3.1, "Default analyzer and analyzer by class"](#)):

- 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). [http://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters]

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

```
@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 helped you getting an overview of Hibernate Search. 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 4.3.1, “Default analyzer and analyzer by class”](#)) and field bridges ([Section 4.4, “Bridges”](#)). Both are important features required for more fine-grained indexing. More advanced topics cover clustering ([Section 3.6, “JMS Master/Slave configuration”](#), [Section 3.8, “Infinispan Directory configuration”](#)) and large index handling ([Section 3.3, “Sharding indexes”](#)).

Architecture

2.1. Overview

Hibernate Search consists of an indexing 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 3.1, “Enabling Hibernate Search and automatic indexing”](#)).

To interact with Apache Lucene indexes, Hibernate Search has the notion of `DirectoryProviders`. A directory provider will manage a given Lucene `Directory` type. You can configure directory providers to adjust the directory target (see [Section 3.2, “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 same way a HQL, JPA-QL or native query would do.

To be more efficient Hibernate Search batches the write interactions with the Lucene index. There are currently two types of batching. Outside a transaction, the index update operation is executed right after the actual database operation. This is really a no batching setup. In the case of an ongoing 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 batch modes (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.



Tip

It is recommended - for both your database and Hibernate Search - to execute your operations in a transaction, be it JDBC or JTA.



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.

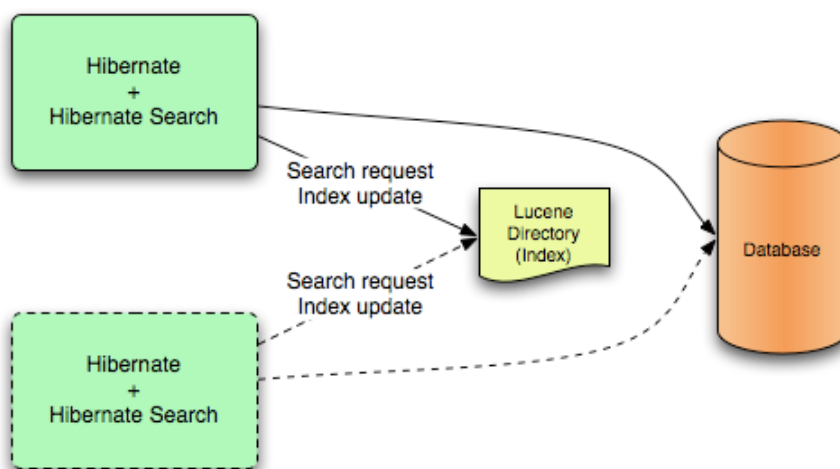
2.2. Back end

Hibernate Search offers the ability to let the batched work being processed by different back ends. Three back ends are provided out of the box and you have the option to plugin in your own implementation.

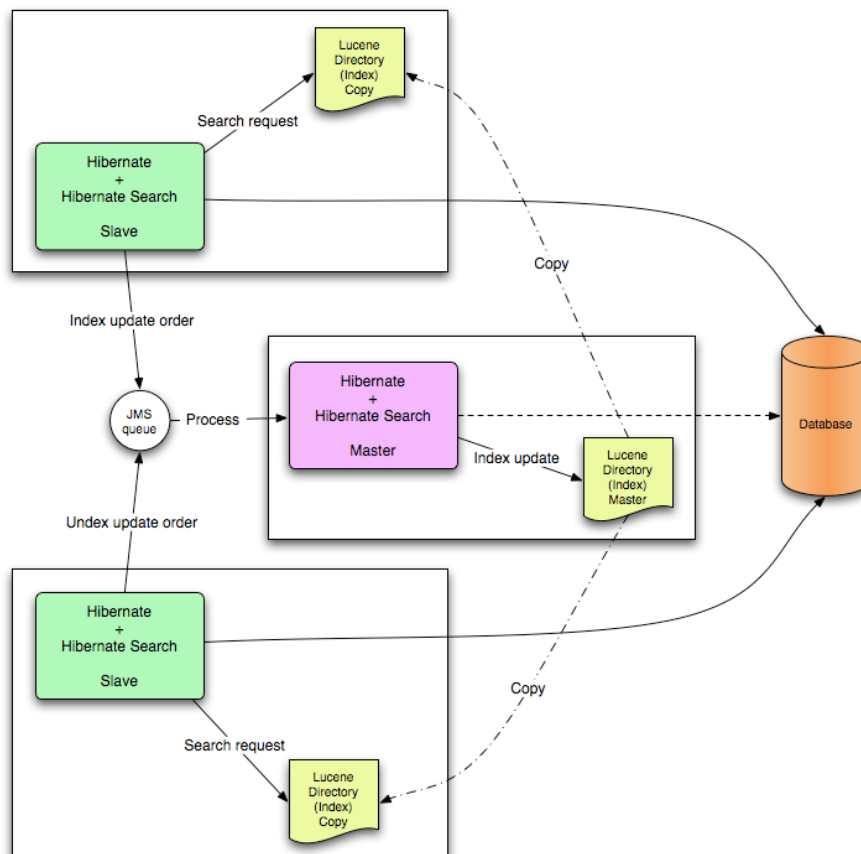
2.2.1. Back end types

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.



2.2.1.3. JGroups

The JGroups based back end works similar to the JMS one and is designed after the same master/slave pattern. However, instead of JMS the JGroups toolkit is used as a replication mechanism. This back end can be used as an alternative to JMS when response time is critical, but i.e. JNDI service is not available.



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.

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 3.9, “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 `SegmentReaders`. 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.

Configuration

3.1. Enabling Hibernate Search and automatic indexing

Let's start with the most basic configuration question - how to enable Hibernate Search in your system.

3.1.1. Enabling Hibernate Search

The good news is that Hibernate Search is enabled out of the box when detected on the classpath by Hibernate Core. 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 but no entities are annotated as indexed.

3.1.2. 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 [Section 6.3, "Rebuilding the whole index"](#)).

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.

3.2. Directory configuration

Apache Lucene has a notion of a `Directory` to store the index files. The `Directory` implementation can be customized and Lucene comes bundled with a file system and an in-memory implementation. `DirectoryProvider` is the Hibernate Search abstraction around a Lucene `Directory` and handles the configuration and the initialization of the underlying Lucene resources. [Table 3.1, "List of built-in DirectoryProviders"](#) shows the list of the directory providers available in Hibernate Search together with their corresponding options.

To configure your `DirectoryProvider` you have to understand that each indexed entity is associated to a Lucene index (except of the case where multiple entities share the same index - [Section 3.4, "Sharing indexes"](#)). The name of the index is given by the `index` property of the

`@Indexed` annotation. If the `index` property is not specified the fully qualified name of the indexed class will be used as name.

Knowing the index name, you can configure the directory provider and any additional options by using the prefix `hibernate.search.<indexname>`. The name `default` (`hibernate.search.default`) is reserved and can be used to define properties which apply to all indexes. [Example 3.2, “Configuring directory providers”](#) shows how `hibernate.search.default.directory_provider` is used to set the default directory provider to be the filesystem one. `hibernate.search.default.indexBase` sets then the default base directory for the indexes. As a result the index for the entity `Status` is created in `/usr/lucene/indexes/org.hibernate.example.Status`.

The index for the `Rule` entity, however, is using an in-memory directory, because the default directory provider for this entity is overridden by the property `hibernate.search.Rules.directory_provider`.

Finally the `Action` entity uses a custom directory provider `CustomDirectoryProvider` specified via `hibernate.search.Actions.directory_provider`.

Example 3.1. Specifying the index name

```
package org.hibernate.example;

@Indexed
public class Status { ... }

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

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

Example 3.2. Configuring directory providers

```
hibernate.search.default.directory_provider filesystem
hibernate.search.default.indexBase=/usr/lucene/indexes
hibernate.search.Rules.directory_provider ram
hibernate.search.Actions.directory_provider com.acme.hibernate.CustomDirectoryProvider
```



Tip

Using the described configuration scheme you can easily define common rules like the directory provider and base directory, and override those defaults later on on a per index basis.

Table 3.1. List of built-in `DirectoryProvider`s

Class or shortcut name	Description	Properties
ram	Memory based directory, the directory will be uniquely identified (in the same deployment unit) by the <code>@Indexed.index</code> element	none
filesystem	File system based directory. The directory used will be <code><indexBase>/<indexName></code>	<p><code>indexBase</code> : Base directory</p> <p><code>indexName</code>: override <code>@Indexed.index</code> (useful for sharded indexes)</p> <p><code>locking_strategy</code> : optional, see Section 3.11, “LockFactory configuration”</p> <p><code>filesystem_access_type</code>: allows to determine the exact type of <code>FSDirectory</code> implementation used by this <code>DirectoryProvider</code>. Allowed values are <code>auto</code> (the default value, selects <code>NIOFSDirectory</code> on non Windows systems, <code>SimpleFSDirectory</code> on Windows), <code>simple</code> (<code>SimpleFSDirectory</code>), <code>nio</code> (<code>NIOFSDirectory</code>), <code>mmap</code> (<code>MMapDirectory</code>). Make sure to refer to Javadocs of these <code>Directory</code> implementations before changing this setting. Even though <code>NIOFSDirectory</code> or <code>MMapDirectory</code> can bring substantial performance boosts they also have their issues.</p>
filesystem-master	File system based directory. Like <code>filesystem</code> . It also copies the index to a source directory (aka copy directory) on a regular basis.	<p><code>indexBase</code>: Base directory</p> <p><code>indexName</code>: override <code>@Indexed.index</code> (useful for sharded indexes)</p>

Class or shortcut name	Description	Properties
	<p>The recommended value for the refresh period is (at least) 50% higher than the time to copy the information (default 3600 seconds - 60 minutes).</p> <p>Note that the copy is based on an incremental copy mechanism reducing the average copy time.</p> <p>DirectoryProvider typically used on the master node in a JMS back end cluster.</p> <p>The <code>buffer_size_on_copy</code> optimum depends on your operating system and available RAM; most people reported good results using values between 16 and 64MB.</p>	<p><code>sourceBase</code>: Source (copy) base directory.</p> <p><code>source</code>: Source directory suffix (default to <code>@Indexed.index</code>). The actual source directory name being <code><sourceBase>/<source></code></p> <p><code>refresh</code>: refresh period in second (the copy will take place every refresh seconds).</p> <p><code>buffer_size_on_copy</code>: The amount of MegaBytes to move in a single low level copy instruction; defaults to 16MB.</p> <p><code>locking_strategy</code>: optional, see Section 3.11, "LockFactory configuration"</p> <p><code>filesystem_access_type</code>: allows to determine the exact type of <code>FSDirectory</code> implementation used by this <code>DirectoryProvider</code>. Allowed values are <code>auto</code> (the default value, selects <code>NIOFSDirectory</code> on non Windows systems, <code>SimpleFSDirectory</code> on Windows), <code>simple</code> (<code>SimpleFSDirectory</code>), <code>nio</code> (<code>NIOFSDirectory</code>), <code>mmap</code> (<code>MMapDirectory</code>). Make sure to refer to Javadocs of these <code>Directory</code> implementations before changing this setting. Even though <code>NIOFSDirectory</code> or <code>MMapDirectory</code> can bring substantial performance boosts they also have their issues.</p>

Class or shortcut name	Description	Properties
filesystem-slave	<p>File system based directory. Like <code>filesystem</code>, but retrieves a master version (source) on a regular basis. To avoid locking and inconsistent search results, 2 local copies are kept.</p> <p>The recommended value for the refresh period is (at least) 50% higher than the time to copy the information (default 3600 seconds - 60 minutes).</p> <p>Note that the copy is based on an incremental copy mechanism reducing the average copy time.</p> <p><code>DirectoryProvider</code> typically used on slave nodes using a JMS back end.</p> <p>The <code>buffer_size_on_copy</code> optimum depends on your operating system and available RAM; most people reported good results using values between 16 and 64MB.</p>	<p><code>indexBase</code>: Base directory</p> <p><code>indexName</code>: <code>override @Indexed.index</code> (useful for sharded indexes)</p> <p><code>sourceBase</code>: Source (copy) base directory.</p> <p><code>source</code>: Source directory suffix (default to <code>@Indexed.index</code>). The actual source directory name being <code><sourceBase>/<source></code></p> <p><code>refresh</code>: refresh period in second (the copy will take place every refresh seconds).</p> <p><code>buffer_size_on_copy</code>: The amount of MegaBytes to move in a single low level copy instruction; defaults to 16MB.</p> <p><code>locking_strategy</code>: optional, see Section 3.11, "LockFactory configuration"</p> <p><code>retry_marker_lookup</code>: optional, default to 0. Defines how many times we look for the marker files in the source directory before failing. Waiting 5 seconds between each try.</p> <p><code>retry_initialize_period</code>: optional, set an integer value in seconds to enable the retry initialize feature: if the slave can't find the master index it will try again until it's found in background, without preventing the application to start: fullText queries performed before the index is</p>

Class or shortcut name	Description	Properties
		<p>initialized are not blocked but will return empty results. When not enabling the option or explicitly setting it to zero it will fail with an exception instead of scheduling a retry timer. To prevent the application from starting without an invalid index but still control an initialization timeout, see <code>retry_marker_lookup</code> instead.</p> <p><code>filesystem_access_type</code>: allows to determine the exact type of <code>FSDirectory</code> implementation used by this <code>DirectoryProvider</code>. Allowed values are <code>auto</code> (the default value, selects <code>NIOFSDirectory</code> on non Windows systems, <code>SimpleFSDirectory</code> on Windows), <code>simple</code> (<code>SimpleFSDirectory</code>), <code>nio</code> (<code>NIOFSDirectory</code>), <code>mmap</code> (<code>MMapDirectory</code>). Make sure to refer to Javadocs of these <code>Directory</code> implementations before changing this setting. Even though <code>NIOFSDirectory</code> or <code>MMapDirectory</code> can bring substantial performance boosts they also have their issues.</p>
<code>infinispan</code>	Infinispan based directory. Use it to store the index in a distributed grid, making index changes visible to all elements of the cluster very quickly. Also see Section 3.8, “Infinispan Directory configuration” for additional requirements and configuration settings.	<p><code>locking_cachename</code>: name of the Infinispan cache to use to store locks.</p> <p><code>data_cachename</code> : name of the Infinispan cache to use to store the largest data chunks; this area will contain the largest objects, use replication</p>

Class or shortcut name	Description	Properties
	Infinispan needs a global configuration and additional dependencies; the settings defined here apply to each different index.	<p>if you have enough memory or switch to distribution.</p> <p><code>metadata_cachename</code>: name of the Infinispan cache to use to store the metadata relating to the index; this data is rather small and read very often, it's recommended to have this cache setup using replication.</p> <p><code>chunk_size</code>: large files of the index are split in smaller chunks, you might want to set the highest value efficiently handled by your network. Networking tuning might be useful.</p>



Tip

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. In this case, pass the fully qualified class name of your provider into the `directory_provider` property. You can pass any additional properties using the prefix `hibernate.search.<indexname>`.

3.3. Sharding indexes

In some cases it can be useful to split (shard) the indexed data of a given entity into several Lucene indexes.



Warning

This solution is not recommended unless there is a pressing need. Searches will be slower as all shards have to be opened for a single search. Don't do it until you have a real use case!

Possible use cases for sharding are:

- A single index is so huge that index update times are slowing the application down.

- A typical search will only hit a sub-set of the index, such as when data is naturally segmented by customer, region or application.

By default sharding is not enabled unless the number of shards is configured. To do this use the `hibernate.search.<indexName>.sharding_strategy.nbr_of_shards` property as seen in [Example 3.3, “Enabling index sharding”](#). In this example 5 shards are enabled.

Example 3.3. Enabling index sharding

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

Responsible for splitting the data into sub-indexes is the `IndexShardingStrategy`. The default sharding strategy splits the data according to the hash value of the id string representation (generated by the `FieldBridge`). This ensures a fairly balanced sharding. You can replace the default strategy by implementing a custom `IndexShardingStrategy`. To use your custom strategy you have to set the `hibernate.search.<indexName>.sharding_strategy` property.

Example 3.4. Specifying a custom sharding strategy

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

The `IndexShardingStrategy` also allows for optimizing searches by selecting which shard to run the query against. By activating a filter (see [Section 5.3.1, “Using filters in a sharded environment”](#)), a sharding strategy can select a subset of the shards used to answer a query (`IndexShardingStrategy.getDirectoryProvidersForQuery`) and thus speed up the query execution.

Each shard has an independent directory provider configuration. The `DirectoryProvider` index names for the `Animal` entity in [Example 3.5, “Sharding configuration for entity `Animal`”](#) are `Animal.0` to `Animal.4`. In other words, each shard has the name of it's owning index followed by `.` (dot) and its index number (see also [Section 3.2, “Directory configuration”](#)).

Example 3.5. Sharding configuration for entity `Animal`

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

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


In [Example 3.5, “Sharding configuration for entity Animal”](#), the configuration uses the default id string hashing strategy and shards the `Animal` index into 5 sub-indexes. All sub-indexes are filesystem instances and the directory where each sub-index is stored is as followed:

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

3.4. 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.



Note

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

3.5. Worker configuration

It is possible to refine how Hibernate Search interacts with Lucene through the worker configuration. There exist several architectural components and possible extension points. Let's have a closer look.

First there is a `Worker`. An implementation of the `Worker` interface is responsible for receiving all entity changes, queuing them by context and applying them once a context ends. The most intuitive context, especially in connection with ORM, is the transaction. For this reason Hibernate Search will per default use the `TransactionalWorker` to scope all changes per transaction. One can, however, imagine a scenario where the context depends for example on the number of entity changes or some other application (lifecycle) events. For this reason the `Worker` implementation is configurable as shown in [Table 3.2, “Scope configuration”](#).

Table 3.2. Scope configuration

Property	Description
<code>hibernate.search.worker.scope</code>	The fully qualified class name of the <code>Worker</code> implementation to use. If this property is not set, <code>empty</code> or <code>transaction</code> the default <code>TransactionalWorker</code> is used.
<code>hibernate.search.worker.*</code>	All configuration properties prefixed with <code>hibernate.search.worker</code> are passed to the <code>Worker</code> during initialization. This allows adding custom, worker specific parameters.
<code>hibernate.search.worker.batch_size</code>	Defines the maximum number of indexing operation batched per context. Once the limit is reached indexing will be triggered even though the context has not ended yet. This property only works if the <code>Worker</code> implementation delegates the queued work to <code>BatchedQueueingProcessor</code> (which is what the <code>TransactionalWorker</code> does)

Once a context ends it is time to prepare and apply the index changes. This can be done synchronously or asynchronously from within a new thread. Synchronous updates have the advantage that the index is at all times in sync with the databases. Asynchronous updates, on the other hand, can help to minimize the user response time. The drawback is potential discrepancies between database and index states. Lets look at the configuration options shown in [Table 3.3, “Execution configuration”](#).

Table 3.3. Execution configuration

Property	Description
<code>hibernate.search.worker.execution</code>	<code>sync</code> : synchronous execution (default) <code>async</code> : asynchronous execution
<code>hibernate.search.worker.thread_pool.size</code>	Defines the number of threads in the pool for asynchronous execution. Defaults to 1.
<code>hibernate.search.worker.buffer_queue.max</code>	Defines the maximal number of work queue if the thread pool is starved. Useful only for

	asynchronous execution. Default to infinite. If the limit is reached, the work is done by the main thread.
--	--

So far all work is done within the same Virtual Machine (VM), no matter which execution mode. The total amount of work has not changed for the single VM. Luckily there is a better approach, namely delegation. It is possible to send the indexing work to a different server by configuring `hibernate.search.worker.backend` - see [Table 3.4, “Backend configuration”](#).

Table 3.4. Backend configuration

Property	Description
<code>hibernate.search.worker.backend</code>	<p><code>lucene</code>: The default backend which runs index updates in the same VM. Also used when the property is undefined or empty.</p> <p><code>jms</code>: JMS backend. Index updates are send to a JMS queue to be processed by an indexing master. See Table 3.5, “JMS backend configuration” for additional configuration options and Section 3.6, “JMS Master/Slave configuration” for a more detailed descripton of this setup.</p> <p><code>jgroupsMaster</code> or <code>jgroupsSlave</code>: Backend using JGroups [http://www.jgroups.org/] as communication layer. See Table 3.6, “JGroups backend configuration” for additional configuration options and Section 3.7, “JGroups Master/Slave configuration” for a more detailed description of this setup.</p> <p><code>blackhole</code>: Mainly a test/developer setting which ignores all indexing work</p> <p>You can also specify the fully qualified name of a class implementing <code>BackendQueueProcessorFactory</code>. This way you can implement your own communication layer. The implementation is responsilbe for returning a <code>Runnable</code> instance which on execution will process the index work.</p>

Table 3.5. JMS backend configuration

Property	Description
----------	-------------

<code>hibernate.search.worker.jndi.*</code>	Defines the JNDI properties to initiate the InitialContext (if needed). JNDI is only used by the JMS back end.
<code>hibernate.search.worker.jms.connection_factory_name</code>	Mandatory for the JMS back end. Defines the JNDI name to lookup the JMS connection factory from (<code>/ConnectionFactory</code> by default in JBoss AS)
<code>hibernate.search.worker.jms.queue</code>	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.

Table 3.6. JGroups backend configuration

Property	Description
<code>hibernate.search.worker.jgroups.channel_name</code>	Optional for JGroups back end. Defines the name of JGroups channel.
<code>hibernate.search.worker.jgroups.configuration_name</code>	Optional JGroups network stack configuration. Defines the name of a JGroups configuration file, which must exist on classpath.
<code>hibernate.search.worker.jgroups.configuration_xml</code>	Optional JGroups network stack configuration. Defines a String representing JGroups configuration as XML.
<code>hibernate.search.worker.jgroups.configuration_text</code>	Optional JGroups network stack configuration. Provides JGroups configuration in plain text.

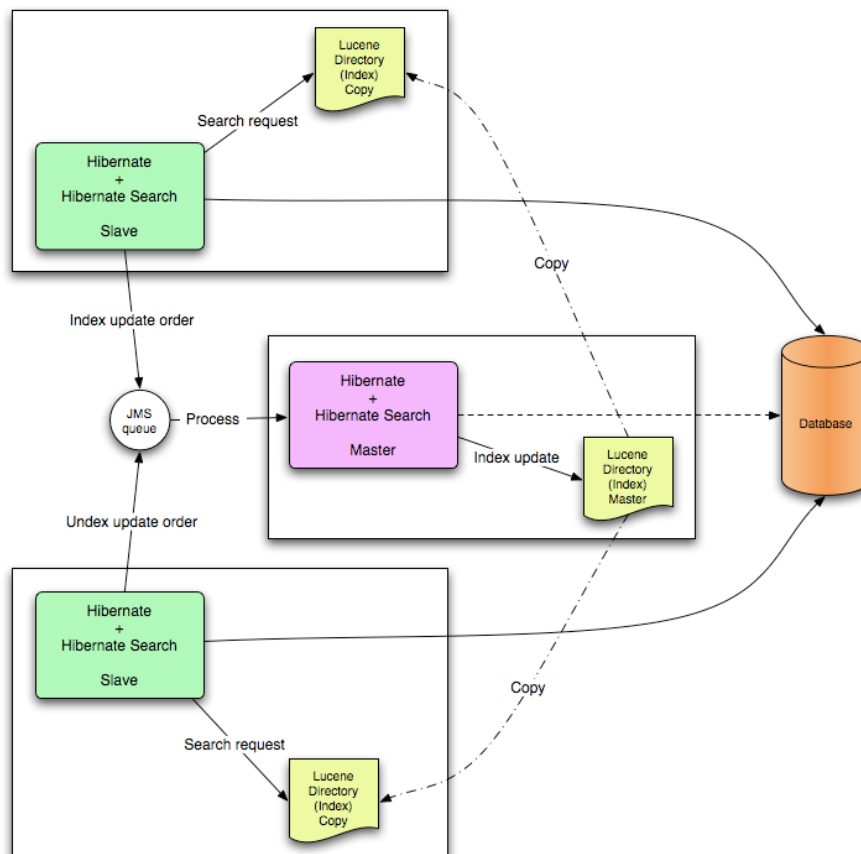


Warning

As you probably noticed, some of the shown properties are correlated which means that not all combinations of property values make sense. In fact you can end up with a non-functional configuration. This is especially true for the case that you provide your own implementations of some of the shown interfaces. Make sure to study the existing code before you write your own `Worker` or `BackendQueueProcessorFactory` implementation.

3.6. JMS Master/Slave configuration

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



JMS back end configuration.

3.6.1. 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 = filesystem-slave

## 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
```



Tip

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



Tip

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

3.6.2. 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 = filesystem-master

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



Tip

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

In addition to the Hibernate Search framework configuration, a Message Driven Bean has to 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 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.

3.7. JGroups Master/Slave configuration

This section describes how to configure the JGroups Master/Slave back end. The configuration examples illustrated in [Section 3.6, "JMS Master/Slave configuration"](#) also apply here, only a different backend (`hibernate.search.worker.backend`) needs to be set.

3.7.1. Slave nodes

Every index update operation is sent through a JGroups channel to the master node. Index querying operations are executed on a local index copy.

Example 3.9. JGroups Slave configuration

```
### slave configuration
hibernate.search.worker.backend = jgroupsSlave
```

3.7.2. Master node

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

Example 3.10. JGroups Master configuration

```
### master configuration
hibernate.search.worker.backend = jgroupsMaster
```

3.7.3. JGroups channel configuration

Optionally the configuration for the JGroups transport protocols and channel name can be defined and applied to master and slave nodes. There are several ways to configure the JGroups transport details. You can either set the `hibernate.search.worker.backend.jgroups.configurationFile` property and specify a file containing the JGroups configuration or you can use the property `hibernate.search.worker.backend.jgroups.configurationXml` or `hibernate.search.worker.backend.jgroups.configurationString` to directly embed either the xml or string JGroups configuration into your Hibernate configuration file. All three options are shown in [Example 3.11, “JGroups transport protocol configuration”](#).



Tip

If no property is explicitly specified it is assumed that the JGroups default configuration file `flush-udp.xml` is used.

Example 3.11. JGroups transport protocol configuration

```
## JGroups configuration options
# OPTION 1 - udp.xml file needs to be located in the classpath
hibernate.search.worker.backend.jgroups.configurationFile = udp.xml

# OPTION 2 - protocol stack configuration provided in XML format
hibernate.search.worker.backend.jgroups.configurationXml =

<config xmlns="urn:org:jgroups"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:org:jgroups file:schema/JGroups-2.8.xsd">
```



```

<UDP
mcast_addr="${jgroups.udp.mcast_addr:228.10.10.10}"
mcast_port="${jgroups.udp.mcast_port:45588}"
tos="8"
thread_naming_pattern="pl"
thread_pool.enabled="true"
thread_pool.min_threads="2"
thread_pool.max_threads="8"
thread_pool.keep_alive_time="5000"
thread_pool.queue_enabled="false"
thread_pool.queue_max_size="100"
thread_pool.rejection_policy="Run"/>
<PING timeout="1000" num_initial_members="3"/>
<MERGE2 max_interval="30000" min_interval="10000"/>
<FD_SOCKET/>
<FD timeout="3000" max_tries="3"/>
<VERIFY_SUSPECT timeout="1500"/>
<pbcast.STREAMING_STATE_TRANSFER/>
<pbcast.FLUSH timeout="0"/>
</config>

# OPTION 3 - protocol stack configuration provided in "old style" jgroups format
hibernate.search.worker.backend.jgroups.configurationString =

UDP(mcast_addr=228.1.2.3;mcast_port=45566;ip_ttl=32):PING(timeout=3000;
num_initial_members=6):FD(timeout=5000):VERIFY_SUSPECT(timeout=1500):
pbcast.NAKACK(gc_lag=10;retransmit_timeout=3000):UNICAST(timeout=5000):
FRAG:pbcast.GMS(join_timeout=3000;shun=false;print_local_addr=true)

```

In this JGroups master/slave configuration nodes communicate over a JGroups channel. The default channel name is `HSearchCluster` which can be configured as seen in [Example 3.12](#), “JGroups channel name configuration”.

Example 3.12. JGroups channel name configuration

```
hibernate.search.worker.backend.jgroups.clusterName = Hibernate-Search-Cluster
```

3.8. Infinispan Directory configuration

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

This section describes in greater detail how to configure Hibernate Search to use an Infinispan Lucene Directory.

When using an Infinispan Directory the index is stored in memory and shared across multiple nodes. It is considered a single directory across all participating nodes. If a node updates the index, all other nodes are updated as well. Updates on one node can be immediately searched for in the whole cluster.

The default configuration replicates all data defining the index across all nodes, thus consuming a significant amount of memory. For large indexes it's suggested to enable data distribution, so that each piece of information is replicated to a subset of all cluster members.

It is also possible to offload part or most information to a `CacheStore`, such as plain filesystem, Amazon S3, Cassandra, Berkley DB or standard relational databases. You can configure it to have a `CacheStore` on each node or have a single centralized one shared by each node.

See the [Infinispan documentation](http://www.jboss.org/infinispan/) [http://www.jboss.org/infinispan/] for all Infinispan configuration options.

3.8.1. Requirements

Infinispan requires Java 6 and an updated version of JGroups. To use the Infinispan directory via Maven, add the following dependencies:

Example 3.13. Maven dependencies for Hibernate Search

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-search</artifactId>
  <version>3.4.0.Final</version>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-search-infinispan</artifactId>
  <version>3.4.0.Final</version>
</dependency>
```

For the non-maven users, add `hibernate-search-infinispan.jar`, `infinispan-lucene-directory.jar` and `infinispan-core.jar` to your application classpath. These last two jars are distributed by [Infinispan](http://sourceforge.net/projects/infinispan/files/) [http://sourceforge.net/projects/infinispan/files/]. Also make sure to update JGroups to a version matching the Infinispan requirements. The version normally distributed with Hibernate Search is older to maintain Java 5 compatibility.

3.8.2. Architecture

Even when using an Infinispan directory it's still recommended to use the JMS Master/Slave or JGroups backend, because in Infinispan all nodes will share the same index and it is likely that `IndexWriters` being active on different nodes will try to acquire the lock on the same index. So instead of sending updates directly to the index, send it to a JMS queue or JGroups channel and have a single node apply all changes on behalf of all other nodes.

Configuring a non-default backend is not a requirement but a performance optimization as locks are enabled to have a single node writing.

To configure a JMS slave only the backend must be replaced, the directory provider must be set to `infinispan`; set the same directory provider on the master, they will connect without the need

to setup the copy job across nodes. Using the JGroups backend is very similar - just combine the backend configuration with the `infinispan` directory provider.

3.8.3. Infinispan Configuration

The most simple configuration only requires to enable the backend:

```
hibernate.search.default.directory_provider infinispan
```

That's all what is needed to get a cluster-replicated index, but the default configuration does not enable any form of permanent persistence for the index; to enable such a feature an Infinispan configuration file should be provided.

To use Infinispan, Hibernate Search requires a `CacheManager`; it can lookup and reuse an existing `CacheManager`, via JNDI, or start and manage a new one. In the latter case Hibernate Search will start and stop it (closing occurs when the Hibernate `SessionFactory` is closed).

To use an existing `CacheManager` via JNDI (optional parameter):

```
hibernate.search.infinispan.cachemanager_jndiname = [jndiname]
```

To start a new `CacheManager` from a configuration file (optional parameter):

```
hibernate.search.infinispan.configuration_resourcename = [infinispan configuration filename]
```

If both parameters are defined, JNDI will have priority. If none of these is defined, Hibernate Search will use the default Infinispan configuration included in `hibernate-search-infinispan.jar`. This configuration should work fine in most cases but does not store the index in a persistent cache store.

As mentioned in [Table 3.1, “List of built-in DirectoryProviders”](#), each index makes use of three caches, so three different caches should be configured as shown in the `default-hibernatesearch-infinispan.xml` provided in the `hibernate-search-infinispan.jar`. Several indexes can share the same caches.



Warning

Infinispan uses JGroups, which requires the JVM property `java.net.preferIPv4Stack` be set to `true`:
`Djava.net.preferIPv4Stack=true`

3.9. 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.

3.10. 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()` or via a `MassIndexer` (see [Section 6.3, “Rebuilding the whole index”](#)), the used properties are those grouped under the `batch` keyword:

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

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.

Example 3.14. Example performance option configuration

```
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
```

The configuration in [Example 3.14, “Example performance option configuration”](#) will result in these settings applied on the second shard of the `Animal` index:

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

All other values will use the defaults defined in Lucene.

The default for all values is to leave them at Lucene's own default. The values listed in [Table 3.7, “List of indexing performance and behavior properties”](#) depend for this reason on the version of Lucene you are using. The values shown are relative to version 2.4. For more information about Lucene indexing performance, please refer to the Lucene documentation.



Warning

Previous versions had the `batch` parameters inherit from `transaction` properties. This needs now to be explicitly set.

Table 3.7. List of indexing performance and behavior properties

Property	Description	Default Value
<code>hibernate.search.[default <indexname>].exclusive_index_use</code>	Set to <code>true</code> when no other process will need to write to the same index. This will enable Hibernate Search to work in exclusive mode on the index and improve performance when writing changes to the index.	<code>false</code> (releases locks as soon as possible)
<code>hibernate.search.[default <indexname>].max_queue_length</code>	Each index has a separate "pipeline" which contains the updates to be applied to the index. When this queue is full adding more	1000

Property	Description	Default Value
	operations to the queue becomes a blocking operation. Configuring this setting doesn't make much sense unless the <code>worker.execution</code> is configured as <code>async</code> .	
<code>hibernate.search. [default <indexname>].indexwriter. [transaction batch].max_buffered_delete_terms</code>	Determines the minimal number of delete terms required before the buffered in-memory delete terms are applied and flushed. If there are documents buffered in memory at the time, they are merged and a new segment is created.	Disabled (flushes by RAM usage)
<code>hibernate.search. [default <indexname>].indexwriter. [transaction batch].max_buffered_docs</code>	Controls the amount of documents buffered in memory during indexing. The bigger the more RAM is consumed.	Disabled (flushes by RAM usage)
<code>hibernate.search. [default <indexname>].indexwriter. [transaction batch].max_merge_docs</code>	Defines the largest number of documents allowed in a segment. Larger values are best for batched indexing and speedier searches. Small values are best for transaction indexing.	Unlimited (Integer.MAX_VALUE)
<code>hibernate.search. [default <indexname>].indexwriter. [transaction batch].merge_factor</code>	Controls segment merge frequency and size. Determines how often segment indexes are merged when insertion occurs. With smaller values, less RAM is used while indexing, and searches on unoptimized indexes are faster, but indexing speed is slower. With larger values, more RAM is used during indexing, and while searches on unoptimized indexes are	10

Property	Description	Default Value
	<p>slower, indexing is faster. Thus larger values (> 10) are best for batch index creation, and smaller values (< 10) for indexes that are interactively maintained. The value must not be lower than 2.</p>	
<code>hibernate.search. [default <indexname>].indexwriter. [transaction batch].ram_buffer_size</code>	<p>Controls the amount of RAM in MB dedicated to document buffers. When used together with <code>max_buffered_docs</code> a flush occurs for whichever event happens first.</p> <p>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.</p>	16 MB
<code>hibernate.search. [default <indexname>].indexwriter. [transaction batch].term_index_interval</code>	<p>Expert: Set the interval between indexed terms.</p> <p>Large values cause less memory to be used by <code>IndexReader</code>, but slow random-access to terms. Small values cause more memory to be used by an <code>IndexReader</code>, and speed random-access to terms. See Lucene documentation for more details.</p>	128
<code>hibernate.search. [default <indexname>].indexwriter. [transaction batch].use_compound_file</code>	<p>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 <code>false</code> in an attempt to improve the indexing time, but you could run out of file descriptors if <code>mergeFactor</code> is also large.</p>	true

Property	Description	Default Value
	Boolean parameter, use "true" or "false". The default value for this option is <code>true</code> .	
<code>hibernate.search.enable_dirty_check</code>	<p>Not all entity changes require an update of the Lucene index. If all of the updated entity properties (dirty properties) are not indexed Hibernate Search will skip the re-indexing work.</p> <p>Disable this option if you use custom <code>FieldBridges</code> which need to be invoked at each update event (even though the property for which the field bridge is configured has not changed).</p> <p>This optimization will not be applied on classes using a <code>@ClassBridge</code> or a <code>@DynamicBoost</code>.</p> <p>Boolean parameter, use "true" or "false". The default value for this option is <code>true</code>.</p>	<code>true</code>



Tip

When your architecture permits it, always set `hibernate.search.default.exclusive_index_use=true` as it greatly improves efficiency in index writing.



Tip

To tune the indexing speed it might be useful to time the object loading from database in isolation from the writes to the index. To achieve this set the `blackhole` as worker backend and start your indexing routines. This backend does not disable Hibernate Search: it will still generate the needed changesets to the index, but will discard them instead of flushing them to the index. In contrast to setting the `hibernate.search.indexing_strategy` to `manual`, using `blackhole` will

possibly load more data from the database, because associated entities are re-indexed as well.

```
hibernate.search.worker.backend blackhole
```

The recommended approach is to focus first on optimizing the object loading, and then use the timings you achieve as a baseline to tune the indexing process.



Warning

The `blackhole` backend is not meant to be used in production, only as a tool to identify indexing bottlenecks.

3.11. LockFactory configuration

Lucene `Directory`s have default locking strategies which work well for most cases, but it's possible to specify for each index managed by Hibernate Search which `LockingFactory` you want to use.

Some of these locking strategies require a filesystem level lock and may be used even on RAM based indexes, but this is not recommended and of no practical use.

To select a locking factory, set the `hibernate.search.<index>.locking_strategy` option to one of `simple`, `native`, `single` or `none`. Alternatively set it to the fully qualified name of an implementation of `org.hibernate.search.store.LockFactoryFactory`.

Table 3.8. List of available LockFactory implementations

name	Class	Description
simple	<code>org.apache.lucene.store.SimpleLockFactory</code>	<p>SimpleLockFactory is a LockFactory implementation based on Java's File API, it marks the usage of the index by creating a marker file.</p> <p>If for some reason you had to kill your application, you will need to remove this file before restarting it.</p> <p>This is the default implementation for the <code>filesystem</code>, <code>filesystem-master</code> and <code>filesystem-slave</code> directory providers.</p>

name	Class	Description
native	org.apache.lucene.store.NativeFSLockFactory	Simple this also marks the usage of the index by creating a marker file, but this one is using native OS file locks so that even if your application crashes the locks will be cleaned up. This implementation has known problems on NFS.
single	org.apache.lucene.store.SingleInstanceLockFactory	This lock factory doesn't use a file marker but is a Java object lock held in memory; therefore it's possible to use it only when you are sure the index is not going to be shared by any other process. This is the default implementation for the ram directory provider.
none	org.apache.lucene.store.NoLockFactory	Changes to this index are not coordinated by any lock; test your application carefully and make sure you know what it means.

Configuration example:

```
hibernate.search.default.locking_strategy simple
hibernate.search.Animals.locking_strategy native
hibernate.search.Books.locking_strategy org.custom.components.MyLockingFactory
```

3.12. Exception Handling Configuration

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

```
hibernate.search.error_handler log
```

The default exception handling occurs for both synchronous and asynchronous indexing. Hibernate Search provides an easy mechanism to override the default error handling implementation.

In order to provide your own implementation you must implement the `ErrorHandler` interface, which provides the `handle(ErrorContext context)` method. `ErrorContext` provides a reference to the primary `LuceneWork` instance, the underlying exception and any subsequent `LuceneWork` instances that could not be processed due to the primary exception.

```
public interface ErrorContext {
    List<LuceneWork> getFailingOperations();
    LuceneWork getOperationAtFault();
    Throwable getThrowable();
    boolean hasErrors();
}
```

To register this error handler with Hibernate Search you must declare the fully qualified classname of your `ErrorHandler` implementation in the configuration properties:

```
hibernate.search.error_handler CustomerErrorHandler
```


Mapping entities to the index structure

4.1. Mapping an entity

In [Chapter 1, Getting started](#) you have already learned that all the metadata information needed to index entities is described through annotations. There is no need for xml mapping files. You can still use Hibernate mapping files for the basic Hibernate configuration, but the Hibernate Search specific configuration has to be expressed via annotations.



Note

There is currently no xml configuration option available (see [HSEARCH-210](http://opensource.atlassian.com/projects/hibernate/browse/HSEARCH-210) [<http://opensource.atlassian.com/projects/hibernate/browse/HSEARCH-210>]).

4.1.1. Basic mapping

Lets start with the most commonly used annotations for mapping an entity.

4.1.1.1. @Indexed

Foremost 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 with `@Indexed`

```
@Entity
@Indexed
public class Essay {
    ...
}
```

You can optionally specify the `index` attribute of the `@Indexed` annotation to change the default name of the index. For more information see [Section 3.2, "Directory configuration"](#).

4.1.1.2. @Field

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 ignored by the indexing process. `@Field` does declare a property as indexed and allows to configure several aspects of the indexing process by setting one or more of the following attributes:

- `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 5.1.3.5, "Projection"](#)), 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`.



Tip

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 probably not a date field.



Tip

Fields used for sorting must not be tokenized.

- `termVector`: describes collections of term-frequency pairs. This attribute enables the storing of the term vectors within the documents during indexing. The default value is `TermVector.NO`.

The different values of this attribute are:

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

Value	Definition
	TermVector.YES plus it contains the ordinal positions of each occurrence of a term in a document.
TermVector.WITH_POSITION_OFFSETS	Store the term vector, token position and offset information. This is a combination of the YES, WITH_OFFSETS and WITH_POSITIONS.

- `indexNullAs` : Per default null values are ignored and not indexed. However, using `indexNullAs` you can specify a string which will be inserted as token for the null value. Per default this value is set to `Field.DO_NOT_INDEX_NULL` indicating that null values should not be indexed. You can set this value to `Field.DEFAULT_NULL_TOKEN` to indicate that a default null token should be used. This default null token can be specified in the configuration using `hibernate.search.default_null_token`. If this property is not set and you specify `Field.DEFAULT_NULL_TOKEN` the string `"_null_"` will be used as default.



Note

When the `indexNullAs` parameter is used it is important to use the same token in the search query (see [Querying](#)) to search for null values. It is also advisable to use this feature only with un-tokenized fields (`Index.UN_TOKENIZED`).



Warning

When implementing a custom `FieldBridge` or `TwoWayFieldBridge` it is up to the developer to handle the indexing of null values (see [JavaDocs of `LuceneOptions.indexNullAs\(\)`](#)).

4.1.1.3. @NumericField

There is a companion annotation to `@Field` called `@NumericField` that can be specified in the same scope as `@Field` or `@DocumentId`. It can be specified for Integer, Long, Float and Double properties. At index time the value will be indexed using a [Trie structure](http://en.wikipedia.org/wiki/Trie) [http://en.wikipedia.org/wiki/Trie]. When a property is indexed as numeric field, it enables efficient range query and sorting, orders of magnitude faster than doing the same query on standard `@Field` properties. The `@NumericField` annotation accept the following parameters:

Value	Definition
<code>forField</code>	(Optional) Specify the name of of the related <code>@Field</code> that will be indexed as numeric. It's

Value	Definition
	only mandatory when the property contains more than a <code>@Field</code> declaration
<code>precisionStep</code>	(Optional) Change the way that the Trie structure is stored in the index. Smaller <code>precisionSteps</code> lead to more disk space usage and faster range and sort queries. Larger values lead to less space used and range query performance more close to the range query in normal <code>@Fields</code> . Default value is 4.



Note

Lucene marks the numeric field API still as experimental and warns for incompatible changes in coming releases. Using Hibernate Search will hopefully shield you from any underlying API changes, but that is not guaranteed.

4.1.1.4. @Id

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 JPA and you have specified `@Id` you can omit `@DocumentId`. The chosen entity id will also be used as document id.

Example 4.2. Specifying indexed properties

```
@Entity
@Indexed
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; }

    @Field
    @NumericField( precisionStep = 6)
    public float getGrade() { return grade; }
}
```


Example 4.2, “Specifying indexed properties” defines an index with four fields: `id`, `Abstract`, `text` and `grade`. Note that by default the field name is decapitalized, following the JavaBean specification. The `grade` field is annotated as `Numeric` with a slightly larger `precisionStep` than the default.

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 needs 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;
    }

    ...
}
```

In *Example 4.3, “Using `@Fields` to map a property multiple times”* 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 *Example 4.4, “Indexing associations”* 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`). 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.

Example 4.4. Indexing 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;
    ...
}

```

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`.



Tip

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

Let's make [Example 4.4, "Indexing associations"](#) a bit more complex by nesting `@IndexedEmbedded` as seen in [Example 4.5, "Nested usage of @IndexedEmbedded and @ContainedIn"](#).

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;
    ...
}

```

As you can see, 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 [Example 4.5, “Nested usage of @IndexedEmbedded and @ContainedIn”](#) 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 (using Lucene's query syntax) 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.ownedBy_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 { ... }
```

4.2. Boosting

Lucene has the notion of *boosting* which allows you to give certain documents or fields more or less importance than others. Lucene differentiates between index and search time boosting. The following sections show you how you can achieve index time boosting using Hibernate Search.

4.2.1. Static index time boosting

To define a static boost value for an indexed class or property you can use the `@Boost` annotation. You can use this annotation within `@Field` or specify it directly on method or class level.

Example 4.7. Different ways of using `@Boost`

```
@Entity
@Indexed
@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 [Example 4.7, “Different ways of using @Boost”](#), `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, because `@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 is wrong in strictest terms, 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.

4.2.2. Dynamic index time boosting

The `@Boost` annotation used in [Section 4.2.1, “Static index time boosting”](#) defines a static boost factor which is independent of the state of the indexed entity at runtime. However, there are usecases in which the boost factor may depends on the actual state of the entity. In this case you can use the `@DynamicBoost` annotation together with an accompanying custom `BoostStrategy`.

Example 4.8. Dynamic boost examle

```
public enum PersonType {
    NORMAL,
    VIP
}

@Entity
@Indexed
@DynamicBoost(impl = VIPBoostStrategy.class)
public class Person {
    private PersonType type;

    // ....
}

public class VIPBoostStrategy implements BoostStrategy {
    public float defineBoost(Object value) {
        Person person = ( Person ) value;
        if ( person.getType().equals( PersonType.VIP ) ) {
            return 2.0f;
        }
    }
}
```

```

    }
    else {
        return 1.0f;
    }
}
}

```

In [Example 4.8, “Dynamic boost example”](#) a dynamic boost is defined on class level specifying `VIPBoostStrategy` as implementation of the `BoostStrategy` interface to be used at indexing time. You can place the `@DynamicBoost` either at class or field level. Depending on the placement of the annotation either the whole entity is passed to the `defineBoost` method or just the annotated field/property value. It's up to you to cast the passed object to the correct type. In the example all indexed values of a VIP person would be double as important as the values of a normal person.



Note

The specified `BoostStrategy` implementation must define a public no-arg constructor.

Of course you can mix and match `@Boost` and `@DynamicBoost` annotations in your entity. All defined boost factors are cumulative.

4.3. Analysis

Analysis is the process of converting text into single terms (words) and can be considered as one of the key features of a fulltext search engine. Lucene uses the concept of `Analyzers` to control this process. In the following section we cover the multiple ways Hibernate Search offers to configure the analyzers.

4.3.1. Default analyzer and analyzer by class

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 4.9. Different ways of using `@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.

4.3.2. Named analyzers

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

- a name: the unique string used to refer to the definition
- a list of char filters: each char filter is responsible to pre-process input characters before the tokenization. Char filters can add, change or remove characters; one common usage is for characters normalization
- 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 list of char filters, and 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 char filters do some pre-processing in the character input, then the `Tokenizer` starts the tokenizing process by turning the character input into tokens which are then further processed by the `TokenFilters`. Hibernate Search supports this infrastructure by utilizing the Solr analyzer framework.



Note

Some of the analyzers and filters will require additional dependencies. For example to use the snowball stemmer you have to also include the `lucene-snowball` jar and for the `PhoneticFilterFactory` you need the [commons-codec](http://commons.apache.org/codec) [http://commons.apache.org/codec] jar. Your distribution of Hibernate Search provides these dependencies in its `lib/optional` directory. Have a look at [Table 4.2, "Example of available tokenizers"](#) and [Table 4.3, "Examples of available filters"](#) to see which analyzers and filters have additional dependencies

Prior to Search version 3.3.0.Beta2 it was required to add the Solr dependency `org.apache.solr:solr-core` when you wanted to use the analyzer definition framework. In case you are using Maven this is no longer needed: all required Solr dependencies are now defined as dependencies of the artifact `org.hibernate:hibernate-search-analyzers`; just add the following dependency :

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-search-analyzers</artifactId>
  <version>3.4.0.Final</version>
</dependency>
```

Let's have a look at a concrete example now - [Example 4.10, "@AnalyzerDef and the Solr framework"](#). First a char filter is defined by its factory. In our example, a mapping char filter is used, and will replace characters in the input based on the rules specified in the mapping file. Next a tokenizer is defined. This example uses the standard tokenizer. Last but not least, a list of filters is defined by their factories. In our example, the `StopFilter` filter is built reading the dedicated words property file. The filter is also expected to ignore case.

Example 4.10. @AnalyzerDef and the Solr framework

```
@AnalyzerDef(name="customanalyzer",
  charFilters = {
    @CharFilterDef(factory = MappingCharFilterFactory.class, params = {
      @Parameter(name = "mapping",
        value = "org/hibernate/search/test/analyzer/solr/mapping-chars.properties")
    })
  },
  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 {  
    ...  
}
```



Tip

Filters and char filters are applied in the order they are defined in the `@AnalyzerDef` annotation. Order matters!

Some tokenizers, token filters or char filters load resources like a configuration or metadata file. This is the case for the stop filter and the synonym filter. If the resource charset is not using the VM default, you can explicitly specify it by adding a `resource_charset` parameter.

Example 4.11. Use a specific charset to load the property file

```
@AnalyzerDef(name="customanalyzer",  
    charFilters = {  
        @CharFilterDef(factory = MappingCharFilterFactory.class, params = {  
            @Parameter(name = "mapping",  
                value = "org/hibernate/search/test/analyzer/solr/mapping-chars.properties")  
        })  
    },  
    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="resource_charset", value = "UTF-16BE"),  
            @Parameter(name="ignoreCase", value="true")  
        })  
    })  
})  
public class Team {  
    ...  
}
```

Once defined, an analyzer definition can be reused by an `@Analyzer` declaration as seen in [Example 4.12, “Referencing an analyzer by name”](#).

Example 4.12. 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 also available by their name in the `SearchFactory` which is quite useful when building queries.

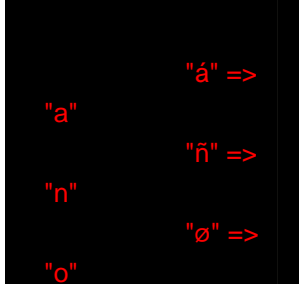
```
Analyzer analyzer = fullTextSession.getSearchFactory().getAnalyzer("customanalyzer");
```

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.

4.3.2.1. Available analyzers

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

Table 4.1. Example of available char filters

Factory	Description	Parameters	Additional dependencies
MappingCharFilterFactory	Replaces one or more characters with one or more characters, based on mappings specified in the resource file	mapping: points to a resource file containing the mappings using the format:	none
		 <pre> "á" => "a" "ñ" => "n" "ø" => "o" "ö" => "o" </pre>	

Factory	Description	Parameters	Additional dependencies
HTMLStripCharFilterFactory	Remove HTML tags, keeping the text	none	none

Table 4.2. Example of available tokenizers

Factory	Description	Parameters	Additional dependencies
StandardTokenizerFactory	Use the Lucene StandardTokenizer	none	none
HTMLStripCharFilterFactory	Remove HTML tags, keep the text and pass it to a StandardTokenizer.	none	solr-core
PatternTokenizerFactory	Breaks text at the specified regular expression pattern.	pattern: the regular expression to use for tokenizing group: says which pattern group to extract into tokens	solr-core

Table 4.3. Examples of available filters

Factory	Description	Parameters	Additional dependencies
StandardFilterFactory	Remove dots from acronyms and 's from words	none	solr-core
LowerCaseFilterFactory	Lowercases all words	none	solr-core
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	solr-core

Factory	Description	Parameters	Additional dependencies
<code>SnowballPorterFilter</code>	Reduces a word to its 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	<code>solr-core</code>
<code>ISOLatinAccentFilter</code>	Remove accents for languages like French	none	<code>solr-core</code>
<code>PhoneticFilterFactory</code>	Inserts phonetically similar tokens into the token stream	encoder: One of <code>DoubleMetaphone</code> , <code>Metaphone</code> , <code>Soundex</code> or <code>RefinedSoundex</code> inject: <code>true</code> will add tokens to the stream, <code>false</code> will replace the existing token maxCodeLength: sets the maximum length of the code to be generated. Supported only for <code>Metaphone</code> and <code>DoubleMetaphone</code> encodings	<code>solr-core</code> and <code>commons-codec</code>
<code>CollationKeyFilterFactory</code>	Converts each token into its <code>java.text.CollationKey</code> , and then encodes the <code>CollationKey</code> with <code>IndexableBinaryString</code> to allow it to be stored as an index term.	custom, language, country, variant, <code>java.text.CollationKey</code> , length, decomposition see Lucene's <code>CollationKeyFilter</code> javadocs for more info	<code>solr-core</code> and <code>commons-io</code>

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.

4.3.3. Dynamic analyzer selection (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 a multilingual applications. 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. [Example 4.13, “Usage of @AnalyzerDiscriminator”](#) demonstrates the usage of this annotation.

Example 4.13. Usage of @AnalyzerDiscriminator

```
@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 getAnalyzerDefinitionName(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 `getAnalyzerDefinitionName()` 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 implementation 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 `@AnalyzerDefs`.



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.

4.3.4. Retrieving an analyzer

In some situations retrieving analyzers can be handy. For example, 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. If you use the Hibernate Search query DSL (see [Section 5.1.2, "Building a Lucene query with the Hibernate Search query DSL"](#)), you don't have to think about it. The query DSL does use the right analyzer transparently for you.

Whether you are using the Lucene programmatic API or the Lucene query parser, you can retrieve the scoped analyzer for a given entity. A scoped analyzer is an analyzer which applies the right analyzers depending on the field indexed. Remember, multiple analyzers can be defined on a given entity each one working on an individual field. A scoped analyzer unifies 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 4.14. 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.



Tip

You can also retrieve analyzers defined via `@AnalyzerDef` by their definition name using `searchFactory.getAnalyzer(String)`.

4.4. Bridges

When discussing the basic mapping for an entity one important fact was so far disregarded. In Lucene all index fields have to be represented as strings. All entity properties annotated with `@Field` have to be converted to strings to be indexed. The reason we have not mentioned it so far is, that for most of your properties Hibernate Search does the translation job for you thanks to set of built-in bridges. However, in some cases you need a more fine grained control over the translation process.

4.4.1. 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

Per default `null` elements are not indexed. Lucene does not support null elements. However, in some situation it can be useful to insert a custom token representing the `null` value. See [Section 4.1.1.2, “@Field”](#) for more information.

java.lang.String

Strings are indexed as are

short, Short, integer, Integer, long, Long, float, Float, double, Double, BigInteger, BigDecimal

Numbers are converted into 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

4.4.2. Custom bridges

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.

4.4.2.1. 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 implement the `org.hibernate.search.bridge.StringBridge` interface. All implementations have to be thread-safe as they are used concurrently.

Example 4.15. Custom `StringBridge` implementation

```
/**
 * 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();
    }
}
```

Given the string bridge defined in [Example 4.15, “Custom StringBridge implementation”](#), any property or field can use this bridge thanks to the `@FieldBridge` annotation:

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

4.4.2.1.1. Parameterized bridge

Parameters can also be passed to the bridge implementation making it more flexible. [Example 4.16](#), “*Passing parameters to your bridge implementation*” implements a `ParameterizedBridge` interface and parameters are passed through the `@FieldBridge` annotation.

Example 4.16. 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.

4.4.2.1.2. Type aware bridge

It is sometimes useful to get the type the bridge is applied on:

- the return type of the property for field/getter-level bridges
- the class type for class-level bridges

An example is a bridge that deals with enums in a custom fashion but needs to access the actual enum type. Any bridge implementing `AppliedOnTypeAwareBridge` will get the type the bridge is applied on injected. Like parameters, the type injected needs no particular care with regard to thread-safety.

4.4.2.1.3. Two-way bridge

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 no difference in the way the `@FieldBridge` annotation is used.

Example 4.17. Implementing a `TwoWayStringBridge` usable 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;
```



Important

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

4.4.2.2. 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`. You can for example store a property in two different document fields. The interface is very similar in its concept to the Hibernate `UserTypes`.

Example 4.18. Implementing the FieldBridge interface

```
/**
 * 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
        luceneOptions.addFieldToDocument(
            name + ".year",
            String.valueOf( year ),
            document );

        // set month and pad it if needed
        luceneOptions.addFieldToDocument(
            name + ".month",
            month < 10 ? "0" : "" + String.valueOf( month ),
            document );

        // set day and pad it if needed
        luceneOptions.addFieldToDocument(
            name + ".day",
            day < 10 ? "0" : "" + String.valueOf( day ),
            document );
    }
}
```

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

In [Example 4.18, “Implementing the FieldBridge interface”](#) the fields are not added directly to Document. Instead the addition is delegated to the `LuceneOptions` helper; this helper will apply the options you have selected on `@Field`, like `Store` or `TermVector`, or apply the chosen `@Boost` value. It is especially useful to encapsulate the complexity of `COMPRESS` implementations. Even though it is recommended to delegate to `LuceneOptions` to add fields to the Document, nothing stops you from editing the Document directly and ignore the `LuceneOptions` in case you need to.



Tip

Classes like `LuceneOptions` are created to shield your application from changes in Lucene API and simplify your code. Use them if you can, but if you need more flexibility you're not required to.

4.4.2.3. 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` respectively `@ClassBridges` annotations can be defined at 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 [Example 4.19, “Implementing a class bridge”](#), `@ClassBridge` supports the `termVector` attribute discussed in section [Section 4.1.1, “Basic mapping”](#).

Example 4.19. 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.

4.5. 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.

4.5.1. The `ProvidedId` annotation

Unlike `@DocumentId` which is applied on field level, `@ProvidedId` is used on the class level. Optionally you can specify your own bridge implementation using the `bridge` property. 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 4.20. Providing your own id

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

4.6. Programmatic API



Warning

This feature is considered experimental. While stable code-wise, the API is subject to change in the future.

Although the recommended approach for mapping indexed entities is to use annotations, it is sometimes more convenient to use a different approach:

- the same entity is mapped differently depending on deployment needs (customization for clients)
- some automatization process requires the dynamic mapping of many entities sharing common traits

While it has been a popular demand in the past, the Hibernate team never found the idea of an XML alternative to annotations appealing due to its heavy duplication, lack of code refactoring safety, because it did not cover all the use case spectrum and because we are in the 21st century :)

The idea of a programmatic API was much more appealing and has now become a reality. You can programmatically define your mapping using a programmatic API: you define entities and fields as indexable by using mapping classes which effectively mirror the annotation concepts in Hibernate Search. Note that fan(s) of XML approach can design their own schema and use the programmatic API to create the mapping while parsing the XML stream.

In order to use the programmatic model you must first construct a `SearchMapping` object. This object is passed to Hibernate Search via a property set to the `Configuration` object. The property key is `hibernate.search.model_mapping` or its type-safe representation `Environment.MODEL_MAPPING`.

```
SearchMapping mapping = new SearchMapping();
[...]
configuration.setProperty( Environment.MODEL_MAPPING, mapping );

//or in JPA
```



```
SearchMapping mapping = new SearchMapping();
[...]
Map<String,String> properties = new HashMap<String,String>(1);
properties.put( Environment.MODEL_MAPPING, mapping );
EntityManagerFactory emf = Persistence.createEntityManagerFactory( "userPU", properties );
```

The `SearchMapping` is the root object which contains all the necessary indexable entities and fields. From there, the `SearchMapping` object exposes a fluent (and thus intuitive) API to express your mappings: it contextually exposes the relevant mapping options in a type-safe way. Just let your IDE autocompletion feature guide you through.

Today, the programmatic API cannot be used on a class annotated with Hibernate Search annotations, chose one approach or the other. Also note that the same default values apply in annotations and the programmatic API. For example, the `@Field.name` is defaulted to the property name and does not have to be set.

Each core concept of the programmatic API has a corresponding example to depict how the same definition would look using annotation. Therefore seeing an annotation example of the programmatic approach should give you a clear picture of what Hibernate Search will build with the marked entities and associated properties.

4.6.1. Mapping an entity as indexable

The first concept of the programmatic API is to define an entity as indexable. Using the annotation approach a user would mark the entity as `@Indexed`, the following example demonstrates how to programmatically achieve this.

Example 4.21. Marking an entity indexable

```
SearchMapping mapping = new SearchMapping();

mapping.entity(Address.class)
    .indexed()
    .indexName( "Address_Index" ); //optional

cfg.getProperties().put( "hibernate.search.model_mapping", mapping );
```

As you can see you must first create a `SearchMapping` object which is the root object that is then passed to the `Configuration` object as property. You must declare an entity and if you wish to make that entity as indexable then you must call the `indexed()` method. The `indexed()` method has an optional `indexName(String indexName)` which can be used to change the default index name that is created by Hibernate Search. Using the annotation model the above can be achieved as:

Example 4.22. Annotation example of indexing entity

```
@Entity
```

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

4.6.2. Adding DocumentId to indexed entity

To set a property as a document id:

Example 4.23. Enabling document id with programmatic model

```
SearchMapping mapping = new SearchMapping();

mapping.entity(Address.class).indexed()
    .property("addressId", ElementType.FIELD) //field access
    .documentId()
    .name("id");

cfg.getProperties().put("hibernate.search.model_mapping", mapping);
```

The above is equivalent to annotating a property in the entity as `@DocumentId` as seen in the following example:

Example 4.24. DocumentId annotation definition

```
@Entity
@Indexed
public class Address {
    @Id
    @GeneratedValue
    @DocumentId(name="id")
    private Long addressId;

    ....
}
```

The next section demonstrates how to programmatically define analyzers.

4.6.3. Defining analyzers

Analyzers can be programmatically defined using the `analyzerDef(String analyzerDef, Class<? extends TokenizerFactory> tokenizerFactory)` method. This method also enables you to define filters for the analyzer definition. Each filter that you define can optionally take in parameters as seen in the following example :

Example 4.25. Defining analyzers using programmatic model

```
SearchMapping mapping = new SearchMapping();

mapping
    .analyzerDef( "ngram", StandardTokenizerFactory.class )
        .filter( LowerCaseFilterFactory.class )
        .filter( NGramFilterFactory.class )
            .param( "minGramSize", "3" )
            .param( "maxGramSize", "3" )
    .analyzerDef( "en", StandardTokenizerFactory.class )
        .filter( LowerCaseFilterFactory.class )
        .filter( EnglishPorterFilterFactory.class )
    .analyzerDef( "de", StandardTokenizerFactory.class )
        .filter( LowerCaseFilterFactory.class )
        .filter( GermanStemFilterFactory.class )
    .entity(Address.class).indexed()
        .property("addressId", ElementType.METHOD) //getter access
        .documentId()
            .name("id");

cfg.getProperties().put( "hibernate.search.model_mapping", mapping );
```

The analyzer mapping defined above is equivalent to the annotation model using `@AnalyzerDef` in conjunction with `@AnalyzerDefs`:

Example 4.26. Analyzer definition using annotation

```
@Indexed
@Entity
@AnalyzerDefs({
    @AnalyzerDef(name = "ngram",
        tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
        filters = {
            @TokenFilterDef(factory = LowerCaseFilterFactory.class),
            @TokenFilterDef(factory = NGramFilterFactory.class,
                params = {
                    @Parameter(name = "minGramSize", value = "3"),
                    @Parameter(name = "maxGramSize", value = "3")
                }
            )
        }
    ),
    @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 Address {  
    ...  
}
```

4.6.4. Defining full text filter definitions

The programmatic API provides easy mechanism for defining full text filter definitions which is available via `@FullTextFilterDef` and `@FullTextFilterDefs` (see [Section 5.3, “Filters”](#)). The next example depicts the creation of full text filter definition using the `fullTextFilterDef` method.

Example 4.27. Defining full text definition programmatically

```
SearchMapping mapping = new SearchMapping();  
  
mapping  
    .analyzerDef( "en", StandardTokenizerFactory.class )  
        .filter( LowerCaseFilterFactory.class )  
        .filter( EnglishPorterFilterFactory.class )  
    .fullTextFilterDef( "security", SecurityFilterFactory.class )  
        .cache( FilterCacheModeType.INSTANCE_ONLY )  
    .entity( Address.class )  
        .indexed()  
        .property( "addressId", ElementType.METHOD )  
            .documentId()  
            .name( "id" )  
        .property( "street1", ElementType.METHOD )  
            .field()  
                .analyzer( "en" )  
                .store( Store.YES )  
            .field()  
                .name( "address_data" )  
                .analyzer( "en" )  
                .store( Store.NO );  
  
cfg.getProperties().put( "hibernate.search.model_mapping", mapping );
```

The previous example can effectively be seen as annotating your entity with `@FullTextFilterDef` like below:

Example 4.28. Using annotation to define full text filter definition

```
@Entity  
@Indexed  
@AnalyzerDefs({  
    @AnalyzerDef(name = "en",  
        tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),  
        filters = {  
            @TokenFilterDef(factory = LowerCaseFilterFactory.class),
```

```

        @TokenFilterDef(factory = EnglishPorterFilterFactory.class)
    })
})
@FullTextFilterDefs({
    @FullTextFilterDef(name = "security", impl = SecurityFilterFactory.class, cache = FilterCacheModeType.INSTANCE_ON)
})
public class Address {

    @Id
    @GeneratedValue
    @DocumentId(name="id")
    public Long getAddressId() {...};

    @Fields({
        @Field(index=Index.TOKENIZED, store=Store.YES,
            analyzer=@Analyzer(definition="en")),
        @Field(name="address_data", analyzer=@Analyzer(definition="en"))
    })
    public String getAddress1() {...};

    .....
}

```

4.6.5. Defining fields for indexing

When defining fields for indexing using the programmatic API, call `field()` on the `property(String propertyName, ElementType elementType)` method. From `field()` you can specify the name, index, store, bridge and analyzer definitions.

Example 4.29. Indexing fields using programmatic API

```

SearchMapping mapping = new SearchMapping();

mapping
    .analyzerDef( "en", StandardTokenizerFactory.class )
    .filter( LowerCaseFilterFactory.class )
    .filter( EnglishPorterFilterFactory.class )
    .entity(Address.class).indexed()
    .property("addressId", ElementType.METHOD)
        .documentId()
        .name("id")
    .property("street1", ElementType.METHOD)
        .field()
            .analyzer("en")
            .store(Store.YES)
            .index(Index.TOKENIZED) //no useful here as it's the default
        .field()
            .name("address_data")
            .analyzer("en");

cfg.getProperties().put( "hibernate.search.model_mapping", mapping );

```

The above example of marking fields as indexable is equivalent to defining fields using `@Field` as seen below:

Example 4.30. Indexing fields using annotation

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

    @Id
    @GeneratedValue
    @DocumentId(name="id")
    private Long getAddressId() {...};

    @Fields({
        @Field(index=Index.TOKENIZED, store=Store.YES,
            analyzer=@Analyzer(definition="en")),
        @Field(name="address_data", analyzer=@Analyzer(definition="en"))
    })
    public String getAddress1() {...}

    .....
}
```

4.6.6. Programmatically defining embedded entities

In this section you will see how to programmatically define entities to be embedded into the indexed entity similar to using the `@IndexEmbedded` model. In order to define this you must mark the property as `indexEmbedded`. There is the option to add a prefix to the embedded entity definition which can be done by calling `prefix` as seen in the example below:

Example 4.31. Programmatically defining embedded entites

```
SearchMapping mapping = new SearchMapping();

mapping
    .entity(ProductCatalog.class)
        .indexed()
        .property("catalogId", ElementType.METHOD)
            .documentId()
            .name("id")
        .property("title", ElementType.METHOD)
            .field()
            .index(Index.TOKENIZED)
```

```

        .store(Store.NO)
        .property("description", ElementType.METHOD)
        .field()
        .index(Index.TOKENIZED)
        .store(Store.NO)
        .property("items", ElementType.METHOD)
        .indexEmbedded()
        .prefix("catalog.items"); //optional

cfg.getProperties().put( "hibernate.search.model_mapping", mapping );

```

The next example shows the same definition using annotation (`@IndexEmbedded`):

Example 4.32. Using `@IndexEmbedded`

```

@Entity
@Indexed
public class ProductCatalog {
    @Id
    @GeneratedValue
    @DocumentId(name="id")
    public Long getCatalogId() {...}

    @Field(store=Store.NO, index=Index.TOKENIZED)
    public String getTitle() {...}

    @Field(store=Store.NO, index=Index.TOKENIZED)
    public String getDescription();

    @OneToMany(fetch = FetchType.LAZY)
    @IndexColumn(name = "list_position")
    @Cascade(org.hibernate.annotations.CascadeType.ALL)
    @IndexEmbedded(prefix="catalog.items")
    public List<Item> getItems() {...}

    ...
}

```

4.6.7. Contained In definition

`@ContainedIn` can be define as seen in the example below:

Example 4.33. Programmatically defining `ContainedIn`

```

SearchMapping mapping = new SearchMapping();

mapping
    .entity(ProductCatalog.class)
    .indexed()
    .property("catalogId", ElementType.METHOD)
    .documentId()

```

```
.property("title", ElementType.METHOD)
    .field()
.property("description", ElementType.METHOD)
    .field()
.property("items", ElementType.METHOD)
    .indexEmbedded()

.entity(Item.class)
    .property("description", ElementType.METHOD)
        .field()
    .property("productCatalog", ElementType.METHOD)
        .containedIn();

cfg.getProperties().put( "hibernate.search.model_mapping", mapping );
```

This is equivalent to defining `@ContainedIn` in your entity:

Example 4.34. Annotation approach for ContainedIn

```
@Entity
@Indexed
public class ProductCatalog {

    @Id
    @GeneratedValue
    @DocumentId
    public Long getCatalogId() {...}

    @Field
    public String getTitle() {...}

    @Field
    public String getDescription() {...}

    @OneToMany(fetch = FetchType.LAZY)
    @IndexColumn(name = "list_position")
    @Cascade(org.hibernate.annotations.CascadeType.ALL)
    @IndexEmbedded
    private List<Item> getItems() {...}

    ...
}

@Entity
public class Item {

    @Id
    @GeneratedValue
    private Long itemId;

    @Field
    public String getDescription() {...}

    @ManyToOne( cascade = { CascadeType.PERSIST, CascadeType.REMOVE } )
    @ContainedIn
    public ProductCatalog getProductCatalog() {...}
```



```
...
}
```

4.6.8. Date/Calendar Bridge

In order to define a calendar or date bridge mapping, call the `dateBridge(Resolution resolution)` OR `calendarBridge(Resolution resolution)` methods after you have defined a `field()` in the `SearchMapping` hierarchy.

Example 4.35. Programmatic model for defining calendar/date bridge

```
SearchMapping mapping = new SearchMapping();

mapping
    .entity(Address.class)
    .indexed()
    .property("addressId", ElementType.FIELD)
    .documentId()
    .property("street1", ElementType.FIELD())
    .field()
    .property("createdOn", ElementType.FIELD)
    .field()
    .dateBridge(Resolution.DAY)
    .property("lastUpdated", ElementType.FIELD)
    .calendarBridge(Resolution.DAY);

cfg.getProperties().put( "hibernate.search.model_mapping", mapping );
```

See below for defining the above using `@CalendarBridge` and `@DateBridge`:

Example 4.36. @CalendarBridge and @DateBridge definition

```
@Entity
@Indexed
public class Address {

    @Id
    @GeneratedValue
    @DocumentId
    private Long addressId;

    @Field
    private String address1;

    @Field
    @DateBridge(resolution=Resolution.DAY)
    private Date createdOn;

    @CalendarBridge(resolution=Resolution.DAY)
    private Calendar lastUpdated;
```

```
...  
}
```

4.6.9. Defining bridges

It is possible to associate bridges to programmatically defined fields. When you define a `field()` programmatically you can use the `bridge(Class<?> impl)` to associate a `FieldBridge` implementation class. The bridge method also provides optional methods to include any parameters required for the bridge class. The below shows an example of programmatically defining a bridge:

Example 4.37. Defining field bridges programmatically

```
SearchMapping mapping = new SearchMapping();  
  
mapping  
    .entity(Address.class)  
    .indexed()  
    .property("addressId", ElementType.FIELD)  
    .documentId()  
    .property("street1", ElementType.FIELD)  
    .field()  
    .field()  
        .name("street1_abridged")  
        .bridge(ConcatStringBridge.class)  
        .param("size", "4");  
  
cfg.getProperties().put("hibernate.search.model_mapping", mapping);
```

The above can equally be defined using annotations, as seen in the next example.

Example 4.38. Defining field bridges using annotation

```
@Entity  
@Indexed  
public class Address {  
  
    @Id  
    @GeneratedValue  
    @DocumentId(name="id")  
    private Long addressId;  
  
    @Fields({  
        @Field,  
        @Field(name="street1_abridged",  
            bridge = @FieldBridge(impl = ConcatStringBridge.class,  
                params = @Parameter(name="size", value="4"))  
    })  
    private String address1;  
  
    ...  
}
```

```
}
```

4.6.10. Mapping class bridge

You can define class bridges on entities programmatically. This is shown in the next example:

Example 4.39. Defining class bridges using API

```
SearchMapping mapping = new SearchMapping();

mapping
    .entity(Departments.class)
        .classBridge(CatDeptsFieldsClassBridge.class)
            .name("branchnetwork")
            .index(Index.TOKENIZED)
            .store(Store.YES)
            .param("sepChar", " ")
        .classBridge(EquipmentType.class)
            .name("equiptype")
            .index(Index.TOKENIZED)
            .store(Store.YES)
            .param("C", "Cisco")
            .param("D", "D-Link")
            .param("K", "Kingston")
            .param("3", "3Com")
        .indexed();

cfg.getProperties().put( "hibernate.search.model_mapping", mapping );
```

The above is similar to using `@ClassBridge` as seen in the next example:

Example 4.40. Using `@ClassBridge`

```
@Entity
@Indexed
@ClassBridges ( {
    @ClassBridge(name="branchnetwork",
        index= Index.TOKENIZED,
        store= Store.YES,
        impl = CatDeptsFieldsClassBridge.class,
        params = @Parameter( name="sepChar", value=" " ) ),
    @ClassBridge(name="equiptype",
        index= Index.TOKENIZED,
        store= Store.YES,
        impl = EquipmentType.class,
        params = {@Parameter( name="C", value="Cisco" ),
            @Parameter( name="D", value="D-Link" ),
            @Parameter( name="K", value="Kingston" ),
            @Parameter( name="3", value="3Com" )
        }
    }
})

public class Departments {
```

```
....  
}
```

4.6.11. Mapping dynamic boost

You can apply a dynamic boost factor on either a field or a whole entity:

Example 4.41. DynamicBoost mapping using programmatic model

```
SearchMapping mapping = new SearchMapping();  
  
mapping  
    .entity(DynamicBoostedDescLibrary.class)  
    .indexed()  
    .dynamicBoost(CustomBoostStrategy.class)  
    .property("libraryId", ElementType.FIELD)  
    .documentId().name("id")  
    .property("name", ElementType.FIELD)  
    .dynamicBoost(CustomFieldBoostStrategy.class);  
    .field()  
    .store(Store.YES)  
  
cfg.getProperties().put( "hibernate.search.model_mapping", mapping );
```

The next example shows the equivalent mapping using the `@DynamicBoost` annotation:

Example 4.42. Using the `@DynamicBoost`

```
@Entity  
@Indexed  
@DynamicBoost(impl = CustomBoostStrategy.class)  
public class DynamicBoostedDescriptionLibrary {  
  
    @Id  
    @GeneratedValue  
    @DocumentId  
    private int id;  
  
    private float dynScore;  
  
    @Field(store = Store.YES)  
    @DynamicBoost(impl = CustomFieldBoostStrategy.class)  
    private String name;  
  
    public DynamicBoostedDescriptionLibrary() {  
        dynScore = 1.0f;  
    }  
  
    .....  
}
```

Querying

The second most important capability of Hibernate Search is the ability to execute Lucene queries and retrieve entities managed by a Hibernate session. The search provides the power of Lucene without leaving the Hibernate paradigm, 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 either via the Hibernate Search query DSL (recommended) or by utilizing the Lucene query API
- Wrapping the Lucene query using an `org.hibernate.Query`
- Executing the search by calling for example `list()` or `scroll()`

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

Example 5.1. Creating a `FullTextSession`

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

Once you have a `FullTextSession` you have two options to build the full-text query: the Hibernate Search query DSL or the native Lucene query.

If you use the Hibernate Search query DSL, it will look like this:

```
final QueryBuilder b = fullTextSession.getSearchFactory()
    .createQueryBuilder().forEntity( Myth.class ).get();

org.apache.lucene.search.Query luceneQuery =
    b.keyword()
        .onField( "history" ).boostedTo(3)
        .matching( "storm" )
        .createQuery();

org.hibernate.Query fullTextQuery = fullTextSession.createFullTextQuery( luceneQuery );
List result = fullTextQuery.list(); //return a list of managed objects
```

You can alternatively write your Lucene query either using the Lucene query parser or Lucene programmatic API.

Example 5.2. Creating a Lucene query via the `QueryParser`

```
SearchFactory searchFactory = fullTextSession.getSearchFactory();
org.apache.lucene.queryParser.QueryParser parser =
    new QueryParser("title", searchFactory.getAnalyzer(Myth.class));
try {
    org.apache.lucene.search.Query luceneQuery = parser.parse( "history:storm^3" );
}
catch (ParseException e) {
    //handle parsing failure
}

org.hibernate.Query fullTextQuery = fullTextSession.createFullTextQuery(luceneQuery);
List result = fullTextQuery.list(); //return a list of managed objects
```



Note

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, the same extensions exist:

Example 5.3. Creating a Search query using the JPA API

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

FullTextEntityManager fullTextEntityManager =
    org.hibernate.search.jpa.Search.getFullTextEntityManager(em);

...
final QueryBuilder b = fullTextEntityManager.getSearchFactory()
    .buildQueryBuilder().forEntity( Myth.class ).get();

org.apache.lucene.search.Query luceneQuery =
    b.keyword()
        .onField( "history" ).boostedTo(3)
        .matching( "storm" )
        .createQuery();

javax.persistence.Query fullTextQuery =
    fullTextEntityManager.createFullTextQuery( luceneQuery );

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



Note

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 using the Lucene API

Using the Lucene API, you have several options. You can use the query parser (fine for simple queries) or the Lucene programmatic API (for more complex use cases). 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 *Lucene In Action* or *Hibernate Search in Action*.

5.1.2. Building a Lucene query with the Hibernate Search query DSL

Writing full-text queries with the Lucene programmatic API is quite complex. It's even more complex to understand the code once written. Besides the inherent API complexity, you have to remember to convert your parameters to their string equivalent as well as make sure to apply the correct analyzer to the right field (a ngram analyzer will for example use several ngrams as the tokens for a given word and should be searched as such).

The Hibernate Search query DSL makes use of a style of API called a fluent API. This API has a few key characteristics:

- it has meaningful method names making a succession of operations reads almost like English
- it limits the options offered to what makes sense in a given context (thanks to strong typing and IDE autocompletion).
- It often uses the chaining method pattern
- it's easy to use and even easier to read

Let's see how to use the API. You first need to create a query builder that is attached to a given indexed entity type. This `QueryBuilder` will know what analyzer to use and what field bridge to apply. You can create several `QueryBuilder`s (one for each entity type involved in the root of your query). You get the `QueryBuilder` from the `SearchFactory`.

```
QueryBuilder mythQB = searchFactory.buildQueryBuilder().forEntity( Myth.class ).get();
```

You can also override the analyzer used for a given field or fields. This is rarely needed and should be avoided unless you know what you are doing.

```
QueryBuilder mythQB = searchFactory.buildQueryBuilder()  
    .forEntity( Myth.class )  
    .overridesForField( "history", "stem_analyzer_definition" )  
    .get();
```

Using the query builder, you can then build queries. It is important to realize that the end result of a `QueryBuilder` is a Lucene query. For this reason you can easily mix and match queries generated via Lucene's query parser or `Query` objects you have assembled with the Lucene programmatic API and use them with the Hibernate Search DSL. Just in case the DSL is missing some features.

5.1.2.1. Keyword queries

Let's start with the most basic use case - searching for a specific word:

```
Query luceneQuery = mythQB.keyword().onField( "history" ).matching( "storm" ).createQuery();
```

`keyword()` means that you are trying to find a specific word. `onField()` specifies in which Lucene field to look. `matching()` tells what to look for. And finally `createQuery()` creates the Lucene query object. A lot is going on with this line of code.

- The value `storm` is passed through the `history` `FieldBridge`: it does not matter here but you will see that it's quite handy when dealing with numbers or dates.
- The field bridge value is then passed to the analyzer used to index the field `history`. This ensures that the query uses the same term transformation than the indexing (lower case, n-gram, stemming and so on). If the analyzing process generates several terms for a given word, a boolean query is used with the `SHOULD` logic (roughly an `OR` logic).

Let's see how you can search a property that is not of type string.

```
@Entity  
@Indexed  
public class Myth {  
    @Field(index = Index.UN_TOKENIZED)  
    @DateBridge(resolution = Resolution.YEAR)  
    public Date getCreationDate() { return creationDate; }  
    public Date setCreationDate(Date creationDate) { this.creationDate = creationDate; }  
    private Date creationDate;
```



```

    ...
}

Date birthdate = ...;
Query luceneQuery = mythQb.keyword().onField("creationDate").matching(birthdate).createQuery();

```



Note

In plain Lucene, you would have had to convert the `Date` object to its string representation (in this case the year).

This conversion works for any object, not just `Date`, provided that the `FieldBridge` has an `objectToString` method (and all built-in `FieldBridge` implementations do).

We make the example a little more advanced now and have a look at how to search a field that uses ngram analyzers. ngram analyzers index succession of ngrams of your words which helps to recover from user typos. For example the 3-grams of the word hibernate are hib, ibe, ber, rna, nat, ate.

```

@AnalyzerDef(name = "ngram",
    tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
    filters = {
        @TokenFilterDef(factory = StandardFilterFactory.class),
        @TokenFilterDef(factory = LowerCaseFilterFactory.class),
        @TokenFilterDef(factory = StopFilterFactory.class),
        @TokenFilterDef(factory = NGramFilterFactory.class,
            params = {
                @Parameter(name = "minGramSize", value = "3"),
                @Parameter(name = "maxGramSize", value = "3") } )
    }
)
@Entity
@Indexed
public class Myth {
    @Field(analyzer=@Analyzer(definition="ngram"))
    @DateBridge(resolution = Resolution.YEAR)
    public String getName() { return name; }
    public String setName(Date name) { this.name = name; }
    private String name;

    ...
}

Date birthdate = ...;
Query luceneQuery = mythQb.keyword().onField("name").matching("Sisiphus")
    .createQuery();

```

The matching word "Sisiphus" will be lower-cased and then split into 3-grams: sis, isi, sip, phu, hus. Each of these n-gram will be part of the query. We will then be able to find the Sysiphus myth (with a y). All that is transparently done for you.



Note

If for some reason you do not want a specific field to use the field bridge or the analyzer you can call the `ignoreAnalyzer()` or `ignoreFieldBridge()` functions

To search for multiple possible words in the same field, simply add them all in the matching clause.

```
//search document with storm or lightning in their history
Query luceneQuery =
    mythQB.keyword().onField("history").matching("storm lightning").createQuery();
```

To search the same word on multiple fields, use the `onFields` method.

```
Query luceneQuery = mythQB
    .keyword()
    .onFields("history", "description", "name")
    .matching("storm")
    .createQuery();
```

Sometimes, one field should be treated differently from another field even if searching the same term, you can use the `andField()` method for that.

```
Query luceneQuery = mythQB.keyword()
    .onField("history")
    .andField("name")
    .boostedTo(5)
    .andField("description")
    .matching("storm")
    .createQuery();
```

In the previous example, only field name is boosted to 5.

5.1.2.2. Fuzzy queries

To execute a fuzzy query (based on the Levenshtein distance algorithm), start like a `keyword` query and add the fuzzy flag.

```
Query luceneQuery = mythQB
    .keyword()
    .fuzzy()
    .withThreshold( .8f )
    .withPrefixLength( 1 )
    .onField("history")
    .matching("starm")
```

```
.createQuery();
```

`threshold` is the limit above which two terms are considering matching. It's a decimal between 0 and 1 and defaults to 0.5. `prefixLength` is the length of the prefix ignored by the "fuzzyness": while it defaults to 0, a non zero value is recommended for indexes containing a huge amount of distinct terms.

5.1.2.3. Wildcard queries

You can also execute wildcard queries (queries where some of parts of the word are unknown). `?` represents a single character and `*` represents any character sequence. Note that for performance purposes, it is recommended that the query does not start with either `?` or `*`.

```
Query luceneQuery = mythQB
    .keyword()
    .wildcard()
    .onField("history")
    .matching("sto*")
    .createQuery();
```



Note

Wildcard queries do not apply the analyzer on the matching terms. Otherwise the risk of `*` or `?` being mangled is too high.

5.1.2.4. Phrase queries

So far we have been looking for words or sets of words, you can also search exact or approximate sentences. Use `phrase()` to do so.

```
Query luceneQuery = mythQB
    .phrase()
    .onField("history")
    .matching("Thou shalt not kill")
    .createQuery();
```

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

```
Query luceneQuery = mythQB
    .phrase()
    .withSlop(3)
    .onField("history")
    .matching("Thou kill")
```

```
.createQuery();
```

5.1.2.5. Range queries

After looking at all these query examples for searching for a given word, it is time to introduce range queries (on numbers, dates, strings etc). A range query searches for a value in between given boundaries (included or not) or for a value below or above a given boundary (included or not).

```
//look for 0 <= starred < 3
Query luceneQuery = mythQB
    .range()
    .onField("starred")
    .from(0).to(3).excludeLimit()
    .createQuery();

//look for myths strictly BC
Date beforeChrist = ...;
Query luceneQuery = mythQB
    .range()
    .onField("creationDate")
    .below(beforeChrist).excludeLimit()
    .createQuery();
```

5.1.2.6. Combining queries

Finally, you can aggregate (combine) queries to create more complex queries. The following aggregation operators are available:

- **SHOULD**: the query should contain the matching elements of the subquery
- **MUST**: the query must contain the matching elements of the subquery
- **MUST NOT**: the query must not contain the matching elements of the subquery

The subqueries can be any Lucene query including a boolean query itself. Let's look at a few examples:

```
//look for popular modern myths that are not urban
Date twentiethCentury = ...;
Query luceneQuery = mythQB
    .bool()
    .must( mythQB.keyword().onField("description").matching("urban").createQuery() )
    .not()
    .must( mythQB.range().onField("starred").above(4).createQuery() )
    .must( mythQB
        .range()
        .onField("creationDate")
        .above(twentiethCentury)
```

```

        .createQuery() )
        .createQuery();

//look for popular myths that are preferably urban
Query luceneQuery = mythQB
    .bool()
        .should( mythQB.keyword().onField("description").matching("urban").createQuery() )
        .must( mythQB.range().onField("starred").above(4).createQuery() )
    .createQuery();

//look for all myths except religious ones
Query luceneQuery = mythQB
    .all()
        .except( mythQB
            .keyword()
            .onField( "description_stem"
                .matching( "religion" )
                .createQuery()
            )
        )
    .createQuery();

```

5.1.2.7. Query options

We already have seen several query options in the previous example, but let's summarize again the options for query types and fields:

- `boostedTo` (on query type and on field): boost the whole query or the specific field to a given factor
- `withConstantScore` (on query): all results matching the query have a constant score equals to the boost
- `filteredBy(Filter)` (on query): filter query results using the `Filter` instance
- `ignoreAnalyzer` (on field): ignore the analyzer when processing this field
- `ignoreFieldBridge` (on field): ignore field bridge when processing this field

Let's check out an example using some of these options

```

Query luceneQuery = mythQB
    .bool()
        .should( mythQB.keyword().onField("description").matching("urban").createQuery() )
        .should( mythQB
            .keyword()
            .onField("name")
            .boostedTo(3)
            .ignoreAnalyzer()
            .matching("urban").createQuery() )
        .must( mythQB
            .range()
            .boostedTo(5).withConstantScore()
            .onField("starred").above(4).createQuery() )

```

```
.createQuery();
```

As you can see, the Hibernate Search query DSL is an easy to use and easy to read query API and by accepting and producing Lucene queries, you can easily incorporate query types not (yet) supported by the DSL. Please give us feedback!

5.1.3. Building a Hibernate Search query

So far we only covered the process of how to create your Lucene query (see [Section 5.1, “Building queries”](#)). However, this is only the first step in the chain of actions. Let's now see how to build the Hibernate Search query from the Lucene query.

5.1.3.1. Generality

Once the Lucene query is built, it needs to be wrapped into an Hibernate Query. If not specified otherwise, the query will be executed against all indexed entities, potentially returning all types of indexed classes.

Example 5.4. Wrapping a Lucene query into a Hibernate Query

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

It is advised, from a performance point of view, to restrict the returned types:

Example 5.5. Filtering the search result by entity type

```
fullTextQuery = fullTextSession
    .createFullTextQuery( luceneQuery, Customer.class );

// or

fullTextQuery = fullTextSession
    .createFullTextQuery( luceneQuery, Item.class, Actor.class );
```

In [Example 5.5, “Filtering the search result by entity type”](#) the first example returns only matching `Customers`, the second returns matching `Actors` and `Items`. 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.

5.1.3.2. 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 another. 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
```



Tip

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

5.1.3.3. 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", SortField.STRING));
query.setSort(sort);
List results = query.list();
```



Tip

Be aware that fields used for sorting must not be tokenized (see [Section 4.1.1.2](#), “`@Field`”).

5.1.3.4. 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 necessary to restrict the returned entity types when creating the Hibernate Search query from the full text session: the type is guessed from the criteria query itself.



Important

Only fetch mode can be adjusted, refrain from applying any other restriction.



Important

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

5.1.3.5. Projection

For some use cases, returning the domain object (including its associations) 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). However, it also has several constraints:

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



Note

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 also useful for another kind of use case. Lucene can provide metadata information about the results. By using some special projection constants, the projection mechanism can retrieve this metadata:

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 projection constants. Here is the list of the available constants:

- `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!

5.1.3.6. Customizing object initialization strategies

By default, Hibernate Search uses the most appropriate strategy to initialize entities matching your full text query. It executes one (or several) queries to retrieve the required entities. This is the best approach to minimize database round trips in a scenario where none / few of the retrieved entities are present in the persistence context (ie the session) or the second level cache.

If most of your entities are present in the second level cache, you can force Hibernate Search to look into the cache before retrieving an object from the database.

Example 5.11. Check the second-level cache before using a query

```
FullTextQuery query = session.createFullTextQuery(luceneQuery, User.class);
query.initializeObjectWith(
    ObjectLookupMethod.SECOND_LEVEL_CACHE,
    DatabaseRetrievalMethod.QUERY
);
```

`ObjectLookupMethod` defines the strategy used to check if an object is easily accessible (without database round trip). Other options are:

- `ObjectLookupMethod.PERSISTENCE_CONTEXT`: useful if most of the matching entities are already in the persistence context (ie loaded in the `Session` or `EntityManager`)
- `ObjectLookupMethod.SECOND_LEVEL_CACHE`: check first the persistence context and then the second-level cache.



Note

Note that to search in the second-level cache, several settings must be in place:

- the second level cache must be properly configured and active
- the entity must have enabled second-level cache (eg via `@Cacheable`)
- the `Session`, `EntityManager` or `Query` must allow access to the second-level cache for read access (ie `CacheMode.NORMAL` in Hibernate native APIs or `CacheRetrieveMode.USE` in JPA 2 APIs).



Warning

Avoid using `ObjectLookupMethod.SECOND_LEVEL_CACHE` unless your second level cache implementation is either `EHCache` or `Infinispan`; other second level cache providers don't currently implement this operation efficiently.

You can also customize how objects are loaded from the database (if not found before). Use `DatabaseRetrievalMethod` for that:

- `QUERY` (default): use a (set of) queries to load several objects in batch. This is usually the best approach.
- `FIND_BY_ID`: load objects one by one using the `Session.get` or `EntityManager.find` semantic. This might be useful if batch-size is set on the entity (in which case, entities will be loaded in batch by Hibernate Core). `QUERY` should be preferred almost all the time.

5.1.3.7. Limiting the time of a query

You can limit the time a query takes in Hibernate Search in two ways:

- raise an exception when the limit is reached
- limit to the number of results retrieved when the time limit is raised

5.1.3.7.1. Raise an exception on time limit

You can decide to stop a query if when it takes more than a predefined amount of time. Note that this is a best effort basis but if Hibernate Search still has significant work to do and if we are beyond the time limit, a `QueryTimeoutException` will be raised (`org.hibernate.QueryTimeoutException` or `javax.persistence.QueryTimeoutException` depending on your programmatic API).

To define the limit when using the native Hibernate APIs, use one of the following approaches

Example 5.12. Defining a timeout in query execution

```
Query luceneQuery = ...;
FullTextQuery query = fullTextSession.createFullTextQuery(luceneQuery, User.class);

//define the timeout in seconds
query.setTimeout(5);

//alternatively, define the timeout in any given time unit
query.setTimeout(450, TimeUnit.MILLISECONDS);

try {
    query.list();
}
catch (org.hibernate.QueryTimeoutException e) {
    //do something, too slow
}
```

```
}
```

Likewise `getResultSize()`, `iterate()` and `scroll()` honor the timeout but only until the end of the method call. That simply means that the methods of `Iterable` or the `ScrollableResults` ignore the timeout.



Note

`explain()` does not honor the timeout: this method is used for debug purposes and in particular to find out why a query is slow

When using JPA, simply use the standard way of limiting query execution time.

Example 5.13. Defining a timeout in query execution

```
Query luceneQuery = ...;
FullTextQuery query = fullTextEM.createFullTextQuery(luceneQuery, User.class);

//define the timeout in milliseconds
query.setHint( "javax.persistence.query.timeout", 450 );

try {
    query.getResultList();
}
catch (javax.persistence.QueryTimeoutException e) {
    //do something, too slow
}
```



Important

Remember, this is a best effort approach and does not guarantee to stop exactly on the specified timeout.

5.1.3.7.2. Limit the number of results when the time limit is reached (EXPERIMENTAL)

Alternatively, you can return the number of results which have already been fetched by the time the limit is reached. Note that only the Lucene part of the query is influenced by this limit. It is possible that, if you retrieve managed object, it takes longer to fetch these objects.



Warning

This approach is not compatible with the `setTimeout` approach.

To define this soft limit, use the following approach

Example 5.14. Defining a time limit in query execution

```
Query luceneQuery = ...;
FullTextQuery query = fullTextSession.createFullTextQuery(luceneQuery, User.class);

//define the timeout in seconds
query.limitExecutionTimeTo(500, TimeUnit.MILLISECONDS);
List results = query.list();
```

Likewise `getResultSize()`, `iterate()` and `scroll()` honor the time limit but only until the end of the method call. That simply means that the methods of `Iterable` or the `ScrollableResults` ignore the timeout.

You can determine if the results have been partially loaded by invoking the `hasPartialResults` method.

Example 5.15. Determines when a query returns partial results

```
Query luceneQuery = ...;
FullTextQuery query = fullTextSession.createFullTextQuery(luceneQuery, User.class);

//define the timeout in seconds
query.limitExecutionTimeTo(500, TimeUnit.MILLISECONDS);
List results = query.list();

if ( query.hasPartialResults() ) {
    displayWarningToUser();
}
```

If you use the JPA API, `limitExecutionTimeTo` and `hasPartialResults` are also available to you.



Warning

This approach is considered experimental

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.



Important

Pagination is preferred over scrolling.

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 5.16. Determining the result size of a query

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

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

```
assert 3245 == query.getResultSize();
```



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

As seen in [Section 5.1.3.5, “Projection”](#) projection results are returns as `Object` arrays. This data structure is not always matching the application needs. In this cases It is possible to apply a `ResultTransformer` which post query execution can build the needed data structure:

Example 5.17. 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 lets you project the `Explanation` object using the `FullTextQuery.EXPLANATION` constant.

Example 5.18. 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 5.19. 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 5.20. 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 5.21. 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.

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 5.22. 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 5.23. 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 `Documents` (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 `SoftReferences` 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
<code>FilterCacheModeType.NONE</code>	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.
<code>FilterCacheModeType.INSTANCE_ONLY</code>	The filter instance is cached and reused across concurrent <code>Filter.getDocIdSet()</code> calls. <code>DocIdSet</code> results are not cached. This setting is useful when a filter uses its own specific caching mechanism or the filter results change dynamically due to application specific events making <code>DocIdSet</code> caching in both cases unnecessary.
<code>FilterCacheModeType.INSTANCE_AND_DOCIDSET_RESULTS</code>	Both the filter instance and the <code>DocIdSet</code> 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)

5.3.1. Using filters in a sharded environment

It is possible, in a sharded environment to execute queries on a subset of the available shards. This can be done in two steps:

- create a sharding strategy that does select a subset of `DirectoryProviders` depending on some filter configuration
- activate the proper filter at query time

Let's first look at an example of sharding strategy that query on a specific customer shard if the customer filter is activated.

```
public class CustomerShardingStrategy implements IndexShardingStrategy {

    // stored DirectoryProviders in a array indexed by customerID
    private DirectoryProvider<?>[] providers;

    public void initialize(Properties properties, DirectoryProvider<?>[] providers) {
        this.providers = providers;
    }

    public DirectoryProvider<?>[] getDirectoryProvidersForAllShards() {
        return providers;
    }
}
```

```

public DirectoryProvider<?> getDirectoryProviderForAddition(
    Class<?> entity, Serializable id, String idInString, Document document) {
    Integer customerID = Integer.parseInt(document.getField("customerID").stringValue());
    return providers[customerID];
}

public DirectoryProvider<?>[] getDirectoryProvidersForDeletion(
    Class<?> entity, Serializable id, String idInString) {
    return getDirectoryProvidersForAllShards();
}

/**
 * Optimization; don't search ALL shards and union the results; in this case, we
 * can be certain that all the data for a particular customer Filter is in a single
 * shard; simply return that shard by customerID.
 */
public DirectoryProvider<?>[] getDirectoryProvidersForQuery(
    FullTextFilterImplementor[] filters) {
    FFullTextFilter filter = getCustomerFilter(filters, "customer");
    if (filter == null) {
        return getDirectoryProvidersForAllShards();
    }
    else {
        return new DirectoryProvider[] { providers[Integer.parseInt(
            filter.getParameter("customerID").toString())] };
    }
}

private FullTextFilter getFilter(FullTextFilterImplementor[] filters, String name) {
    for (FullTextFilterImplementor filter: filters) {
        if (filter.getName().equals(name)) return filter;
    }
    return null;
}
}

```

In this example, if the filter named `customer` is present, we make sure to only use the shard dedicated to this customer. Otherwise, we return all shards. A given Sharding strategy can react to one or more filters and depends on their parameters.

The second step is simply to activate the filter at query time. While the filter can be a regular filter (as defined in [Section 5.3, "Filters"](#)) which also filters Lucene results after the query, you can make use of a special filter that will only be passed to the sharding strategy and otherwise ignored for the rest of the query. Simply use the `ShardSensitiveOnlyFilter` class when declaring your filter.

```

@Entity @Indexed
@FullTextFilterDef(name="customer", impl=ShardSensitiveOnlyFilter.class)
public class Customer {
    ...
}

FullTextQuery query = ftEm.createFullTextQuery(luceneQuery, Customer.class);
query.enableFullTextFilter("customer").setParameter("CustomerID", 5);
@SuppressWarnings("unchecked")

```

```
List<Customer> results = query.getResultList();
```

Note that by using the `ShardSensitiveOnlyFilter`, you do not have to implement any Lucene filter. Using filters and sharding strategy reacting to these filters is recommended to speed up queries in a sharded environment.

5.4. Faceting

Faceted search [http://en.wikipedia.org/wiki/Faceted_search] is a technique which allows to divide the results of a query into multiple categories. This categorisation includes the calculation of hit counts for each category and the ability to further restrict search results based on these facets (categories). *Example 5.24, "Search for 'Hibernate Search' on Amazon"* shows a faceting example. The search results in fifteen hits which are displayed on the main part of the page. The navigation bar on the left, however, shows the category *Computers & Internet* with its subcategories *Programming*, *Computer Science*, *Databases*, *Software*, *Web Development*, *Networking* and *Home Computing*. For each of these subcategories the number of books is shown matching the main search criteria and belonging to the respective subcategory. This division of the category *Computers & Internet* is one concrete search facet. Another one is for example the average customer review.

Example 5.24. Search for 'Hibernate Search' on Amazon

The screenshot shows an Amazon search results page for the query "Hibernate Search". The page is faceted on the left with categories like Department, Format, Author, Shipping Option, Avg. Customer Review, and Condition. The main results area shows three books:

- Hibernate Search in Action I** by Doug Spring. It has 3 customer reviews, is available in Paperback format, and is eligible for FREE Super Saver Shipping. The price is \$49.99.
- Spring Persistence with Hib** (Nov 2, 2010) by Phil Stropker and Brian C. Maples. It has 5 customer reviews, is available in Paperback and Kindle Edition formats, and is eligible for FREE Super Saver Shipping. The price is \$44.99.
- Lucene in Action, Second Ed** by Hatcher and Otis Gospodnetic.

In Hibernate Search the classes `QueryBuilder` and `FullTextQuery` are the entry point into the faceting API. The former allows to create faceting requests whereas the latter gives access to the so called `FacetManager`. With the help of the `FacetManager` faceting requests can be applied on a query and selected facets can be added to an existing query in order to refine search results. The following sections will describe the faceting process in more detail. The examples will use the entity `cd` as shown in [Example 5.25, "Entity Cd"](#):

Example 5.25. Entity Cd

```
@Entity
@Indexed
public class Cd {
```

```
@Id
@GeneratedValue
private int id;

@Fields( {
    @Field,
    @Field(name = "name_un_analyzed", index = Index.UN_TOKENIZED)
})
private String name;

@Field(index = Index.UN_TOKENIZED)
@NumericField
private int price;

Field(index = Index.UN_TOKENIZED)
@DateBridge(resolution = Resolution.YEAR)
private Date releaseYear;

@Field(index = Index.UN_TOKENIZED)
private String label;

// setter/getter
...
```

5.4.1. Creating a faceting request

The first step towards a faceted search is to create the `FacetingRequest`. Currently two types of faceting requests are supported. The first type is called *discrete faceting* and the second type *range faceting* request. In the case of a discrete faceting request you specify on which index field you want to facet (categorize) and which faceting options to apply. An example for a discrete faceting request can be seen in [Example 5.26, “Creating a discrete faceting request”](#):

Example 5.26. Creating a discrete faceting request

```
QueryBuilder builder = fullTextSession.getSearchFactory()
    .buildQueryBuilder()
    .forEntity( Cd.class )
    .get();
FacetingRequest labelFacetingRequest = builder.facet()
    .name( "labelFaceting" )
    .onField( "label" )
    .discrete()
    .orderBy( FacetSortOrder.COUNT_DESC )
    .includeZeroCounts( false )
    .maxFacetCount( 1 )
    .createFacetingRequest();
```

When executing this faceting request a `Facet` instance will be created for each discrete value for the indexed field `label`. The `Facet` instance will record the actual field value including how often this particular field value occurs within the original query results. `orderBy`, `includeZeroCounts` and `maxFacetCount` are optional parameters which can be applied on any faceting request.

`orderBy` allows to specify in which order the created facets will be returned. The default is `FacetSortOrder.COUNT_DESC`, but you can also sort on the field value or the order in which ranges were specified. `includeZeroCount` determines whether facets with a count of 0 will be included in the result (per default they are) and `maxFacetCount` allows to limit the maximum amount of facets returned.



Tip

There are several preconditions an indexed field has to meet in order to apply faceting on it. The indexed property must be of type `String`, `Date` or a subtype of `Number`. Furthermore the property has to be indexed with `Index.UN_TOKENIZED` and in case of a numeric property you need to specify `@NumericField`.

The creation of a range faceting request is quite similar except that we have to specify ranges for the field values we are faceting on. A range faceting request can be seen in [Example 5.27, “Creating a range faceting request”](#) where three different price ranges are specified. `below` and `above` can only be specified once, but you can specify as many `from` - `to` ranges as you want. For each range boundary you can also specify via `excludeLimit` whether it is included into the range or not.

Example 5.27. Creating a range faceting request

```
QueryBuilder builder = fullTextSession.getSearchFactory()
    .buildQueryBuilder()
    .forEntity( Cd.class )
    .get();
FacetingRequest priceacetingRequest = queryBuilder( Cd.class ).facet()
    .name( "priceFaceting" )
    .onField( "price" )
    .range()
    .below( 1000 )
    .from( 1001 ).to( 1500 )
    .above( 1500 ).excludeLimit()
    .createFacetingRequest();
```

5.4.2. Applying a faceting request

In [Section 5.4.1, “Creating a faceting request”](#) we have seen how to create a faceting request. Now it is time to apply it on a query. The key is the `FacetManager` which can be retrieved via the `FullTextQuery` (see [Example 5.28, “Applying a faceting request”](#)).

Example 5.28. Applying a faceting request

```
// create a fulltext query
QueryBuilder builder = queryBuilder( Cd.class );
```

```
Query luceneQuery = builder.all().createQuery(); // match all query
FullTextQuery fullTextQuery = fullTextSession.createFullTextQuery( luceneQuery, Cd.class );

// retrieve facet manager and apply faceting request
FacetManager facetManager = query.getFacetManager();
facetManager.enableFaceting( priceFacetingRequest );

// get the list of Cds
List<Cd> cds = fullTextQuery.list();
...

// retrieve the faceting results
List<Facet> facets = facetManager.getFacets( "priceFaceting" );
...
```

You can enable as many faceting requests as you like and retrieve them afterwards via `getFacets()` specifying the faceting request name. There is also a `disableFaceting()` method which allows you to disable a faceting request by specifying its name.

5.4.3. Restricting query results

Last but not least, you can apply any of the returned `Facets` as additional criteria on your original query in order to implement a "drill-down" functionality. For this purpose `FacetSelection` can be utilized. `FacetSelections` are available via the `FacetManager` and allow you to select a facet as query criteria (`selectFacets`), remove a facet restriction (`deselectFacets`), remove all facet restrictions (`clearSelectedFacets`) and retrieve all currently selected facets (`getSelectedFacets`). [Example 5.29, "Restricting query results via the application of a `FacetSelection`"](#) shows an example.

Example 5.29. Restricting query results via the application of a `FacetSelection`

```
// create a fulltext query
QueryBuilder builder = queryBuilder( Cd.class );
Query luceneQuery = builder.all().createQuery(); // match all query
FullTextQuery fullTextQuery = fullTextSession.createFullTextQuery( luceneQuery, clazz );

// retrieve facet manager and apply faceting request
FacetManager facetManager = query.getFacetManager();
facetManager.enableFaceting( priceFacetingRequest );

// get the list of Cd
List<Cd> cds = fullTextQuery.list();
assertTrue(cds.size() == 10);

// retrieve the faceting results
List<Facet> facets = facetManager.getFacets( "priceFaceting" );
assertTrue(facets.get(0).getCount() == 2)

// apply first facet as additional search criteria
facetManager.getFacetGroup( "priceFaceting" ).selectFacets( facets.get( 0 ) );

// re-execute the query
cds = fullTextQuery.list();
```

```
assertTrue(cds.size() == 2);
```

5.5. 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](#).
- caching frequently extracted values from the index: see [Section 5.5.1, "Caching index values: FieldCache"](#).

5.5.1. Caching index values: FieldCache

The primary function of a Lucene index is to identify matches to your queries, still after the query is performed the results must be analyzed to extract useful information: typically Hibernate Search might need to extract the Class type and the primary key.

Extracting the needed values from the index has a performance cost, which in some cases might be very low and not noticeable, but in some other cases might be a good candidate for caching.

What is exactly needed depends on the kind of Projections being used (see [Section 5.1.3.5, "Projection"](#)), and in some cases the Class type is not needed as it can be inferred from the query context or other means.

Using the `@CacheFromIndex` annotation you can experiment different kinds of caching of the main metadata fields required by Hibernate Search:

```
import static org.hibernate.search.annotations.FieldCacheType.CLASS;
import static org.hibernate.search.annotations.FieldCacheType.ID;

@Indexed
@CacheFromIndex( { CLASS, ID } )
public class Essay {
    ...
}
```

It is currently possible to cache Class types and IDs using this annotation:

- `CLASS`: Hibernate Search will use a Lucene FieldCache to improve performance of the Class type extraction from the index.

This value is enabled by default, and is what Hibernate Search will apply if you don't specify the `@CacheFromIndex` annotation.

- `ID`: Extracting the primary identifier will use a cache. This is likely providing the best performing queries, but will consume much more memory which in turn might reduce performance.



Note

Measure the performance and memory consumption impact after warmup (executing some queries): enabling Field Caches is likely to improve performance but this is not always the case.

Using a `FieldCache` has two downsides to consider:

- **Memory usage**: these caches can be quite memory hungry. Typically the `CLASS` cache has lower requirements than the `ID` cache.
- **Index warmup**: when using field caches, the first query on a new index or segment will be slower than when you don't have caching enabled.

With some queries the classtype won't be needed at all, in that case even if you enabled the `CLASS` field cache, this might not be used; for example if you are targeting a single class, obviously all returned values will be of that type (this is evaluated at each Query execution).

For the `ID` `FieldCache` to be used, the ids of targeted entities must be using a `TwoWayFieldBridge` (as all building bridges), and all types being loaded in a specific query must use the fieldname for the id, and have ids of the same type (this is evaluated at each Query execution).

Manual index changes

As Hibernate core applies changes to the Database, Hibernate Search detects these changes and will update the index automatically (unless the EventListeners are disabled). Sometimes changes are made to the database without using Hibernate, as when backup is restored or your data is otherwise affected; for these cases Hibernate Search exposes the Manual Index APIs to explicitly update or remove a single entity from the index, or rebuild the index for the whole database, or remove all references to a specific type.

All these methods affect the Lucene Index only, no changes are applied to the Database.

6.1. Adding instances to the index

Using `FullTextSession.index(T entity)` you can directly add or update a specific object instance to the index. If this entity was already indexed, then the index will be updated. Changes to the index are only applied at transaction commit.

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

```
FullTextSession fullTextSession = Search.getFullTextSession(session);
Transaction tx = fullTextSession.beginTransaction();
Object customer = fullTextSession.load( Customer.class, 8 );
fullTextSession.index(customer);
tx.commit(); //index only updated at commit time
```

In case you want to add all instances for a type, or for all indexed types, the recommended approach is to use a `MassIndexer`: see [Section 6.3.2, “Using a MassIndexer”](#) for more details.

6.2. Deleting instances from the index

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 6.2. 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 is updated 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 removes all entities of the type passed as a parameter as well as all its subtypes.

Example 6.3. 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 changes are applied 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.



Note

All manual indexing methods (`index`, `purge` and `purgeAll`) only affect the index, not the database, nevertheless they are transactional and as such they won't be applied until the transaction is successfully committed, or you make use of `flushToIndexes`.

6.3. Rebuilding the whole index

If you change the entity mapping to the index, chances are that the whole Index needs to be updated; For example if you decide to index a an existing field using a different analyzer you'll need to rebuild the index for affected types. Also if the Database is replaced (like restored from a backup, imported from a legacy system) you'll want to be able to rebuild the index from existing data. Hibernate Search provides two main strategies to choose from:

- Using `FullTextSession.flushToIndexes()` periodically, while using `FullTextSession.index()` on all entities.
- Use a `MassIndexer`.

6.3.1. Using `flushToIndexes()`

This strategy consists in removing the existing index and then adding all entities back to the index using `FullTextSession.purgeAll()` and `FullTextSession.index()`, however there are some memory and efficiency constraints. For maximum efficiency Hibernate Search batches index

operations and executes them at commit time. If you expect to index a lot of data 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` if you don't empty the queue periodically: to do this you can use `fullTextSession.flushToIndexes()`. Every time `fullTextSession.flushToIndexes()` is called (or if the transaction is committed), the batch queue is processed applying all index changes. Be aware that, once flushed, the changes cannot be rolled back.

Example 6.4. Index rebuilding using `index()` and `flushToIndexes()`

```
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(); //free memory since the queue is processed
    }
}
transaction.commit();
```



Note

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

Try to use a batch size that guarantees that your application will not run out of memory: with a bigger batch size objects are fetched faster from database but more memory is needed.

6.3.2. Using a MassIndexer

Hibernate Search's `MassIndexer` uses several parallel threads to rebuild the index; you can optionally select which entities need to be reloaded or have it reindex all entities. This approach is optimized for best performance but requires to set the application in maintenance mode: making queries to the index is not recommended when a `MassIndexer` is busy.

Example 6.5. Index rebuilding using a `MassIndexer`

```
fullTextSession.createIndexer().startAndWait();
```

This will rebuild the index, deleting it and then reloading all entities from the database. Although it's simple to use, some tweaking is recommended to speed up the process: there are several parameters configurable.



Warning

During the progress of a `MassIndexer` the content of the index is undefined, make sure that nobody will try to make some query during index rebuilding! If somebody should query the index it will not corrupt but most results will likely be missing.

Example 6.6. Using a tuned `MassIndexer`

```
fullTextSession
    .createIndexer( User.class )
    .batchSizeToLoadObjects( 25 )
    .cacheMode( CacheMode.NORMAL )
    .threadsToLoadObjects( 5 )
    .threadsForIndexWriter( 3 )
    .threadsForSubsequentFetching( 20 )
    .progressMonitor( monitor ) //a MassIndexerProgressMonitor implementation
    .startAndWait();
```

This will rebuild the index of all `User` instances (and subtypes), and will create 5 parallel threads to load the `User` instances using batches of 25 objects per query; these loaded `User` instances are then pipelined to 20 parallel threads to load the attached lazy collections of `User` containing some information needed for the index. Finally, 3 parallel threads are being used to Analyze the text and write to the index.

It is recommended to leave `cacheMode` to `CacheMode.IGNORE` (the default), as in most reindexing situations the cache will be a useless additional overhead; it might be useful to enable some other `CacheMode` depending on your data: it might increase performance if the main entity is relating to enum-like data included in the index.



Tip

The "sweet spot" of number of threads to achieve best performance is highly dependent on your overall architecture, database design and even data values. To find out the best number of threads for your application it is recommended to use a profiler: all internal thread groups have meaningful names to be easily identified with most tools.



Note

The MassIndexer was designed for speed and is unaware of transactions, so there is no need to begin one or committing. Also because it is not transactional it is not recommended to let users use the system during its processing, as it is unlikely people will be able to find results and the system load might be too high anyway.

Other parameters which affect indexing time and memory consumption are:

- `hibernate.search.[default|<indexname>].exclusive_index_use`
- `hibernate.search.[default|<indexname>].indexwriter.batch.max_buffered_docs`
- `hibernate.search.[default|<indexname>].indexwriter.batch.max_merge_docs`
- `hibernate.search.[default|<indexname>].indexwriter.batch.merge_factor`
- `hibernate.search.[default|<indexname>].indexwriter.batch.ram_buffer_size`
- `hibernate.search.[default|<indexname>].indexwriter.batch.term_index_interval`
- `hibernate.search.batchbackend.concurrent_writers`

Previous versions also had a `max_field_length` but this was removed from Lucene, it's possible to obtain a similar effect by using a `LimitTokenCountAnalyzer`.

All `.indexwriter` parameters are Lucene specific and Hibernate Search is just passing these parameters through - see [Section 3.10, "Tuning Lucene indexing performance"](#) for more details.

`hibernate.search.batchbackend.concurrent_writers` defaults to 2 and represent the number of threads being used at the Analysis and indexwriter stage of the MassIndexing pipeline. The `MassIndexer.threadsForIndexWriter(int)` method overrides this value.

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

When using a `MassIndexer` (see [Section 6.3.2, “Using a MassIndexer”](#)) it will optimize involved indexes by default at the start and at the end of processing; you can change this behavior by using respectively `MassIndexer.optimizeAfterPurge` and `MassIndexer.optimizeOnFinish`.

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 `Orders`; 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_docs`
- `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_interval`

See [Section 3.10, “Tuning Lucene indexing performance”](#) for more details.

Monitoring

Hibernate Search offers access to a `Statistics` object via `SearchFactory.getStatistics()`. It allows you for example to determine which classes are indexed and how many entities are in the index. This information is always available. However, by specifying the `hibernate.search.generate_statistics` property in your configuration you can also collect total and average Lucene query and object loading timings.

8.1. JMX

You can also enable access to the statistics via JMX. Setting the property `hibernate.search.jmx_enabled` will automatically register the `StatisticsInfoMBean`. Depending on your the configuration the `IndexControlMBean` and `IndexingProgressMonitorMBean` will also be registered. Lets have a closer look at the different MBeans.



Tip

If you want to access your JMX beans remotely via JConsole make sure to set the system property `com.sun.management.jmxremote` to `true`.

8.1.1. StatisticsInfoMBean

This MBean gives you access to `Statistics` object as described in the previous section.

8.1.2. IndexControlMBean

This MBean allows to build, optimize and purge the index for a given entity. Indexing occurs via the mass indexing API (see [Section 6.3.2, “Using a MassIndexer”](#)). A requirement for this bean to be registered in JMX is, that the Hibernate `SessionFactory` is bound to JNDI via the `hibernate.session_factory_name` property. Refer to the Hibernate Core manual for more information on how to configure JNDI. The `IndexControlMBean` and its API are for now experimental.

8.1.3. IndexingProgressMonitorMBean

This MBean is an implementation `MassIndexerProgressMonitor` interface. If `hibernate.search.jmx_enabled` is enabled and the mass indexer API is used the indexing progress can be followed via this bean. The bean will only be bound to JMX while indexing is in progress. Once indexing is completed the MBean is not longer available.

Advanced features

In this final chapter we are offering a smorgasbord of tips and tricks which might become useful as you dive deeper and deeper into Hibernate Search.

9.1. Accessing the SearchFactory

The `SearchFactory` object keeps track of the underlying Lucene resources for Hibernate Search. It is a convenient way to access Lucene natively. The `SearchFactory` can be accessed from a `FullTextSession`:

Example 9.1. Accessing the `SearchFactory`

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

9.2. 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 `DirectoryProviders` 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 `DirectoryProviders` if the index is sharded (see [Section 3.3, “Sharding indexes”](#)).

Example 9.2. 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 `Orders` information. Note that the obtained Lucene directory must not be closed (this is Hibernate Search's responsibility).

9.3. 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 9.3. 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 `IndexReaders` will make most queries more efficient.

9.4. Use external services in Hibernate Search components (experimental)

By components, this section means any of the pluggable contracts - `DirectoryProvider` being the most useful use case:

- `DirectoryProvider`
- `ReaderProvider`
- `OptimizerStrategy`
- `BackendQueueProcessorFactory`
- `Worker`

Some of these components need to access a service which is either available in the environment or whose lifecycle is bound to the `SearchFactory`. Sometimes, you even want the same service to be shared amongst several instances of these contracts. One example is the ability to share an Infinispan cache instance between several directory providers to store the various indexes using the same underlying infrastructure.

9.4.1. Exposing a service

To expose a service, you need to implement `org.hibernate.search.spi.ServiceProvider<T>`. `T` is the type of the service you want to use. Services are retrieved by components via their `ServiceProvider` class implementation.

9.4.1.1. Managed services

If your service ought to be started when Hibernate Search starts and stopped when Hibernate Search stops, you can use a managed service. Make sure to properly implement the `start` and `stop` methods of `ServiceProvider`. When the service is requested, the `getService` method is called.

Example 9.4. Example of `ServiceProvider` implementation

```
public class CacheServiceProvider implements ServiceProvider<Cache> {
    private CacheManager manager;

    public void start(Properties properties) {
        //read configuration
        manager = new CacheManager(properties);
    }

    public Cache getService() {
        return manager.getCache(DEFAULT);
    }

    void stop() {
        manager.close();
    }
}
```



Note

The `ServiceProvider` implementation must have a no-arg constructor.

To be transparently discoverable, such service should have an accompanying `META-INF/services/org.hibernate.search.spi.ServiceProvider` whose content list the (various) service provider implementation(s).

Example 9.5. Content of `META-INF/services/org.hibernate.search.spi.ServiceProvider`

```
com.acme.infra.hibernate.CacheServiceProvider
```

9.4.1.2. Provided services

Alternatively, the service can be provided by the environment bootstrapping Hibernate Search. For example, Infinispan which uses Hibernate Search as its internal search engine can pass the `CacheContainer` to Hibernate Search. In this case, the `CacheContainer` instance is not managed by Hibernate Search and the `start/stop` methods of its corresponding service provider will not be used.



Note

Provided services have priority over managed services. If a provider service is registered with the same `ServiceProvider` class as a managed service, the provided service will be used.

The provided services are passed to Hibernate Search via the `SearchConfiguration` interface (`getProvidedServices`).



Important

Provided services are used by frameworks controlling the lifecycle of Hibernate Search and not by traditional users.

If, as a user, you want to retrieve a service instance from the environment, use registry services like JNDI and look the service up in the provider.

9.4.2. Using a service

Many of the pluggable contracts of Hibernate Search can use services. Services are accessible via the `BuildContext` interface.

Example 9.6. Example of a directory provider using a cache service

```
public CustomDirectoryProvider implements DirectoryProvider<RAMDirectory> {
    private BuildContext context;

    public void initialize(
        String directoryProviderName,
        Properties properties,
        BuildContext context) {
        //initialize
        this.context = context;
    }

    public void start() {
        Cache cache = context.requestService( CacheServiceProvider.class );
        //use cache
    }
}
```

```

    }

    public RAMDirectory getDirectory() {
        // use cache
    }

    public stop() {
        //stop services
        context.releaseService( CacheServiceProvider.class );
    }
}

```

When you request a service, an instance of the service is served to you. Make sure to then release the service. This is fundamental. Note that the service can be released in the `DirectoryProvider.stop` method if the `DirectoryProvider` uses the service during its lifetime or could be released right away if the service is simply used at initialization time.

9.5. 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*:

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

Factor	Description
<code>tf(t ind)</code>	Term frequency factor for the term (t) in the document (d).
<code>idf(t)</code>	Inverse document frequency of the term.
<code>coord(q,d)</code>	Score factor based on how many of the query terms are found in the specified document.
<code>queryNorm(q)</code>	Normalizing factor used to make scores between queries comparable.
<code>t.getBoost()</code>	Field boost.
<code>norm(t,d)</code>	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 three 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`.

You can also override the similarity used for a specific index by setting the `similarity` property

```
hibernate.search.default.similarity my.custom.Similarity
```

Finally 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.



Warning

When two entities share the same index they must declare the same `Similarity` implementation. Classes in the same class hierarchy always share the index, so it's not allowed to override the `Similarity` implementation in a subtype.

Likewise, it does not make sense to define the similarity via the index setting and the class-level setting as they would conflict. Such a configuration will be rejected.

Further reading

Last but not least, a few pointers to further information. We highly recommend you to get a copy *Hibernate Search in Action* [<http://www.manning.com/bernard/>]. This excellent book covers Hibernate Search in much more depth than this online documentation can and has a great range of additional examples. If you want to increase your knowledge in Lucene we recommend *Lucene in Action (Second Edition)* [<http://www.manning.com/hatcher3/>]. Because Hibernate Search's functionality is tightly coupled to Hibernate Core it is a good idea to understand Hibernate in more detail. Start with the [online documentation](http://www.hibernate.org/docs) [<http://www.hibernate.org/docs>] or get hold of a copy of *Java Persistence with Hibernate* [<http://www.manning.com/bauer2/>].

If you have any further questions regarding Hibernate Search or want to share some of your use cases have a look at the [Hibernate Search Wiki](http://community.jboss.org/en/hibernate/search) [<http://community.jboss.org/en/hibernate/search>] and the [Hibernate Search Forum](https://forum.hibernate.org/viewforum.php?f=9) [<https://forum.hibernate.org/viewforum.php?f=9>]. We are looking forward hearing from you.

In case you would like to report a bug use the [Hibernate Search Jira](http://opensource.atlassian.com/projects/hibernate/browse/HSEARCH) [<http://opensource.atlassian.com/projects/hibernate/browse/HSEARCH>] instance. Feedback is always welcome!

