



Red Hat JBoss Web Framework Kit 2.7 Hibernate Search Guide

Apache Lucene Integration for use with Red Hat JBoss Enterprise
Application Platform

Red Hat Customer Content Services

Apache Lucene Integration for use with Red Hat JBoss Enterprise Application Platform

Red Hat Customer Content Services

Legal Notice

Copyright © 2015 Red Hat, Inc.

This document is licensed by Red Hat under the [Creative Commons Attribution-ShareAlike 3.0 Unported License](#). If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, MetaMatrix, Fedora, the Infinity Logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux® is the registered trademark of Linus Torvalds in the United States and other countries.

Java® is a registered trademark of Oracle and/or its affiliates.

XFS® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack® Word Mark and OpenStack Logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This guide presents information about Hibernate Search shipped with Red Hat JBoss Web Framework Kit.

Table of Contents

Chapter 1. Hibernate Search Future Distribution	3
Chapter 2. Introduction to Red Hat JBoss Web Framework Kit	4
2.1. About Red Hat JBoss Web Framework Kit	4
2.2. About the JBoss Web Framework Kit Tiers	4
2.3. About the JBoss Web Framework Kit Distribution	5
Chapter 3. Getting Started with Hibernate Search	6
3.1. Getting Started	6
3.2. System Requirements	6
3.3. Using Maven	6
3.4. Configuration	7
3.5. Indexing	10
3.6. Searching	11
3.7. Analyzer	12
Chapter 4. Architecture	14
4.1. Overview	14
4.2. Back End Setup and Operations	15
4.3. Reader Strategies	17
Chapter 5. Configuration	19
5.1. Hibernate Search and Automatic Indexing	19
5.2. Configuring the IndexManager	19
5.3. Directory Configuration	20
5.4. Worker Configuration	25
5.5. Tuning Lucene Indexing	30
5.6. LockFactory Configuration	41
5.7. Exception Handling Configuration	43
5.8. Index Format Compatibility	43
5.9. Hibernate Search as Red Hat JBoss EAP Module	44
Chapter 6. Mapping Entities to the Index Structure	46
6.1. Mapping an Entity	46
6.2. Boosting	57
6.3. Analysis	59
6.4. Bridges	67
Chapter 7. Querying	75
7.1. Building Queries	77
7.2. Retrieving the Results	89
7.3. Filters	92
7.4. Faceting	94
7.5. Optimizing the Query Process	99
Chapter 8. Manual Index Changes	101
8.1. Adding Instances to the Index	101
8.2. Deleting Instances from the Index	101
8.3. Rebuilding the Index	102
Chapter 9. Index Optimization	106
9.1. Automatic Optimization	106
9.2. Manual Optimization	107
9.3. Adjusting Optimization	107

Chapter 10. Monitoring	109
10.1. JMX	109
Chapter 11. Advanced Features	110
11.1. Accessing the SearchFactory	110
11.2. Using an IndexReader	110
11.3. Accessing a Lucene Directory	111
11.4. Sharding Indexes	111
11.5. Sharing Indexes	113
11.6. About Using External Services	113
11.7. Customizing Lucene's Scoring Formula	116
Revision History	118

Chapter 1. Hibernate Search Future Distribution

Red Hat JBoss Web Framework Kit 2.x has a published end-of-life date of June 2015. For more information, see https://access.redhat.com/support/policy/updates/jboss_notes.

Red Hat continues to distribute and support Hibernate Search for use with Red Hat JBoss Enterprise Application Platform. Future Hibernate Search framework and documentation releases will be part of JBoss EAP releases. For more information, see <https://access.redhat.com/solutions/1314313> on the Red Hat Customer Portal.

[Report a bug](#)

Chapter 2. Introduction to Red Hat JBoss Web Framework Kit

2.1. About Red Hat JBoss Web Framework Kit

Red Hat JBoss Web Framework Kit is a set of enterprise-ready versions of popular open source frameworks. Together, these frameworks provide a solution for developing light and rich Java-based web applications.

JBoss Web Framework Kit comprises enterprise distributions of JBoss community web application frameworks and tested third party frameworks. These leading frameworks support fast and easy client-side and server-side application development and testing, with frameworks including RichFaces, jQuery, jQuery Mobile, Hibernate Search, Spring, and Arquillian.

The breadth of JBoss Web Framework Kit frameworks provides choice for Java application development. Each framework has been tested and certified for use in applications deployed to Red Hat JBoss Enterprise Application Platform, Red Hat JBoss Web Server, or Red Hat OpenShift JBoss EAP cartridge. Inclusion in JBoss Web Framework Kit ensures stable versions of frameworks are available over long-term enterprise product life cycles, with regular releases for fixes and nonintrusive feature updates. Further, Red Hat JBoss Developer Studio (an Eclipse-based development environment) provides integrated project templates, quickstarts and tooling for many of the JBoss Web Framework Kit frameworks.

For the complete list of the frameworks composing JBoss Web Framework Kit and the certified platform and framework configurations, see <https://access.redhat.com/site/articles/112543> on the Red Hat Customer Portal.

[Report a bug](#)

2.2. About the JBoss Web Framework Kit Tiers

The frameworks composing JBoss Web Framework Kit are categorized in four distinct tiers. A description of each tier and the associated Red Hat support is detailed here.

Tier 1 - Included Components

These are components that are based wholly or partly on open source technologies that support broad collaboration and where Red Hat maintains a leadership role; as such Red Hat is able to support these components and provide upgrades and fixes under our standard support terms and conditions.

Tier 2 - Tested Frameworks

These are third party frameworks where Red Hat does not have sufficient influence and does not provide upgrades and fixes under our standard support terms and conditions. Commercially reasonable support is provided by Red Hat Global Support Services for these frameworks.

Tier 3 - Frameworks in Tested Examples

These are third party frameworks where Red Hat does not have sufficient influence and does not provide upgrades and fixes under our standard support terms and conditions. Red Hat supports the examples these frameworks are used in and the generic use cases that these examples intend to demonstrate.

Tier 4 - Confirmed Frameworks

These are third party frameworks that do not receive any support from Red Hat, but Red Hat verifies that the frameworks run successfully on Red Hat JBoss Enterprise Application Platform. Frameworks and versions not listed here have not been explicitly tested and certified, and thus may be subject to support limitations.

For a list of JBoss Web Framework Kit frameworks by tier, see <https://access.redhat.com/site/articles/112543> on the Red Hat Customer Portal.

[Report a bug](#)

2.3. About the JBoss Web Framework Kit Distribution

The frameworks composing JBoss Web Framework Kit are distributed from a range of sources and in a variety of formats, as detailed here.

The component frameworks are available from the Red Hat Customer Portal. They are distributed in two alternative formats: a binary for each framework or together as one Maven repository. In addition, the source code for each framework is provided for inspection.

The third party frameworks are not distributed by Red Hat and each must be obtained from its own source.

An extensive set of examples are also provided as part of the JBoss Web Framework Kit distribution:

- TicketMonster is a moderately complex application demonstrating a number of the JBoss Web Framework Kit frameworks working together.
- Quickstarts illustrate subsets of the JBoss Web Framework Kit frameworks used to create simple applications.
- RichFaces, Snowdrop and Seam demonstrations showcase the power of each framework in web application development.

All of these examples are available from the Red Hat Customer Portal, with TicketMonster also available from <http://www.jboss.org/jdf/examples/get-started/> on the JBoss Developer Framework website.

[Report a bug](#)

Chapter 3. Getting Started with Hibernate Search

3.1. Getting Started

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](#).

[Report a bug](#)

3.2. System Requirements

Table 3.1. System requirements

Java Runtime	A JDK or JRE version 6 or greater. You can download a Java Runtime for Windows/Linux/Solaris here . If using Java version 7 make sure you avoid builds 0 and 1: those versions contained an optimisation bug which would be triggered by Lucene.
Hibernate Search	hibernate-search-4.4.4.Final-redhat-wfk-1.jar and all runtime dependencies. You can get the jar artifacts from your distribution of Red Hat JBoss Web Framework Kit.
Hibernate ORM	These instructions have been tested against Hibernate 4.2 distributed with Red Hat JBoss Enterprise Application Platform 6.
JPA 2	Even though Hibernate Search can be used without JPA annotations the following instructions will use them for basic entity configuration (<code>@Entity</code> , <code>@Id</code> , <code>@OneToMany</code> ,...). This part of the configuration could also be expressed in xml or code.

[Report a bug](#)

3.3. Using Maven

Red Hat JBoss Web Framework Kit ships with a Maven repository, which contains Hibernate Search artifacts. See the *Maven Repository User Guide* for setup information.

The following configuration can be used in your Maven project to add hibernate-search dependencies:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.jboss.bom.wfk</groupId>
      <artifactId>jboss-javaee-6.0-with-hsearch</artifactId>
      <version>2.7.0-redhat-1</version>
    </dependency>
```

```

</dependencies>
</dependencyManagement>

<dependencies>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-search</artifactId>
</dependency>
</dependencies>

```

[Report a bug](#)

3.4. 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 3.1. 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 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](#)). Once you have a working configuration you can start experimenting with other directory providers (see [Section 5.3, "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 3.2. 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.YES** will ensure that the text will be indexed, while **analyze=Analyze.YES** ensures that the text will be analyzed using the default Lucene analyzer. Usually, analyzing 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 third 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 7.1.10.5, "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 behavior is recommended since it returns managed objects whereas projections only return object arrays.

Note that **index=Index.YES**, **analyze=Analyze.YES** and **store=Store.NO** are the default values for these parameters and could be omitted.

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 6.4, "Bridges"](#).

This leaves us with **@IndexedEmbedded**. This annotation is used to index associated entities (**@ManyToMany**, **@*ToOne**, **@Embedded** and **@ElementCollection**) 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 6.1.3, "Embedded and Associated Objects"](#)

These settings should be sufficient for now. For more details on entity mapping refer to [Section 6.1, "Mapping an Entity"](#).

Example 3.3. Example entities after adding Hibernate Search annotations

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

    @Id
    @GeneratedValue
    private Integer id;

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

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

    @Field(index = Index.YES, analyze=Analyze.NO, store = Store.YES)
    @DateBridge(resolution = Resolution.DAY)
    private Date publicationDate;

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

    public Book() {
    }
}
```

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

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

    @Id
    @GeneratedValue
    private Integer id;

    @Field
    private String name;

    public Author() {
    }

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

[Report a bug](#)

3.5. 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 8.3, “Rebuilding the Index”](#)):

Example 3.4. Using the Hibernate Session to Index Data

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

Example 3.5. Using JPA to Index Data

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

After executing the above code, you should be able to see a Lucene index under `/var/lucene/indices/example.Book`. Go ahead and inspect this index with [Luke](#). It will help you to understand how Hibernate Search works.

[Report a bug](#)

3.6. Searching

To execute a search, create a Lucene query (using either the Lucene API ([Section 7.1.1, “Building a Lucene Query Using the Lucene API”](#)) or the Hibernate Search query DSL ([Section 7.1.2, “Building a Lucene Query”](#))). Wrap the query in a `org.hibernate.Query` to get the required functionality from the Hibernate API. The following code prepares a query against the indexed fields. Executing the code returns a list of **Books**.

Example 3.6. Using a Hibernate Search Session to Create and Execute a Search

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

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

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

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

Example 3.7. 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 = fullTextEntityManager.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();

```

[Report a bug](#)

3.7. Analyzer

Assuming that the title of an indexed book entity is **Refactoring: Improving the Design of Existing Code** and that hits are required for the following queries: **refactor**, **refactors**, **refactored**, and **refactoring**. Select an analyzer class in Lucene that applies word stemming when indexing and searching. Hibernate Search offers several ways to configure the analyzer (see [Section 6.3.1, “Default Analyzer and Analyzer by Class”](#) for more information):

- ✦ Set the **analyzer** property in the configuration file. The specified class becomes the default analyzer.
- ✦ Set the **@Analyzer** annotation at the entity level.
- ✦ Set the **@Analyzer** annotation at the field level.

Specify the fully qualified classname or the analyzer to use, or see an analyzer defined by the **@AnalyzerDef** annotation with the **@Analyzer** annotation. The Solr analyzer framework with its factories are utilized for the latter option. For more information about factory classes, see the Solr JavaDoc or read the corresponding section on the Solr Wiki (<http://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters>)

In the example, a **StandardTokenizerFactory** is used by two filter factories: **LowerCaseFilterFactory** and **SnowballPorterFilterFactory**. The tokenizer splits words at punctuation characters and hyphens but keeping email addresses and internet hostnames intact. The standard tokenizer is ideal for this and other general operations. The lowercase filter converts all letters in the token into lowercase and the snowball filter applies language specific stemming.

If using the Solr framework, use the tokenizer with an arbitrary number of filters.

Example 3.8. Using @AnalyzerDef and the Solr Framework to Define and Use an Analyzer

```

@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 implements Serializable {

        @Field
        @Analyzer(definition = "customanalyzer")
        private String title;

        @Field
        @Analyzer(definition = "customanalyzer")
        private String subtitle;

        @IndexedEmbedded
        private Set authors = new HashSet();

        @Field(index = Index.YES, analyze = Analyze.NO, store = Store.YES)
        @DateBridge(resolution = Resolution.DAY)
        private Date publicationDate;

        public Book() {
        }

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

```

Use **@AnalyzerDef** to define an analyzer, then apply it to entities and properties using **@Analyzer**. In the example, the **customanalyzer** is defined but not applied on the entity. The analyzer is only applied to the **title** and **subtitle** properties. An analyzer definition is global. Define the analyzer for an entity and reuse the definition for other entities as required.

[Report a bug](#)

Chapter 4. Architecture

4.1. Overview

Hibernate Search consists of an indexing component as well as 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 these updates are handled without you having to interact with the Apache Lucene APIs directly (see [Chapter 4, Architecture](#)). Instead, the interaction with the underlying Lucene indexes is handled via so called **IndexManagers**.

Each Lucene index is managed by one index manager which is uniquely identified by name. In most cases there is also a one to one relationship between an indexed entity and a single **IndexManager**. The exceptions are the use cases of index sharding and index sharing. The former can be applied when the index for a single entity becomes too big and indexing operations are slowing down the application. In this case a single entity is indexed into multiple indexes each with its own index manager (see [Section 11.4, “Sharding Indexes”](#)). The latter, index sharing, is the ability to index multiple entities into the same Lucene index (see [Section 11.5, “Sharing Indexes”](#)).

The index manager abstracts from the specific index configuration. In the case of the default index manager this includes details about the selected backend, the configured reader strategy and the chosen **DirectoryProvider**. These components will be discussed in greater detail later on. It is recommended that you start with the default index manager which uses different Lucene **Directory** types to manage the indexes (see [Section 5.3, “Directory Configuration”](#)). You can, however, also provide your own **IndexManager** implementation (see [Section 5.2, “Configuring the IndexManager”](#)).

Once the index is created, you can search for entities and return lists 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. This batching is the responsibility of the **Worker**. 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 (see [Section 5.4, “Worker Configuration”](#)).



Note

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

[Report a bug](#)

4.2. Back End Setup and Operations

4.2.1. Back End

Hibernate Search uses various back ends to process batches of work. The back end is not limited to the configuration option `default.worker.backend`. This property specifies a implementation of the **BackendQueueProcessor** interface which is a part of a back end configuration. Additional settings are required to set up a back end, for example the JMS back end.

[Report a bug](#)

4.2.2. Lucene

In the Lucene mode, all index updates for a node (JVM) are executed by the same node to the Lucene directories using the directory providers. Use this mode in a non-clustered environment or in clustered environments with a shared directory store.

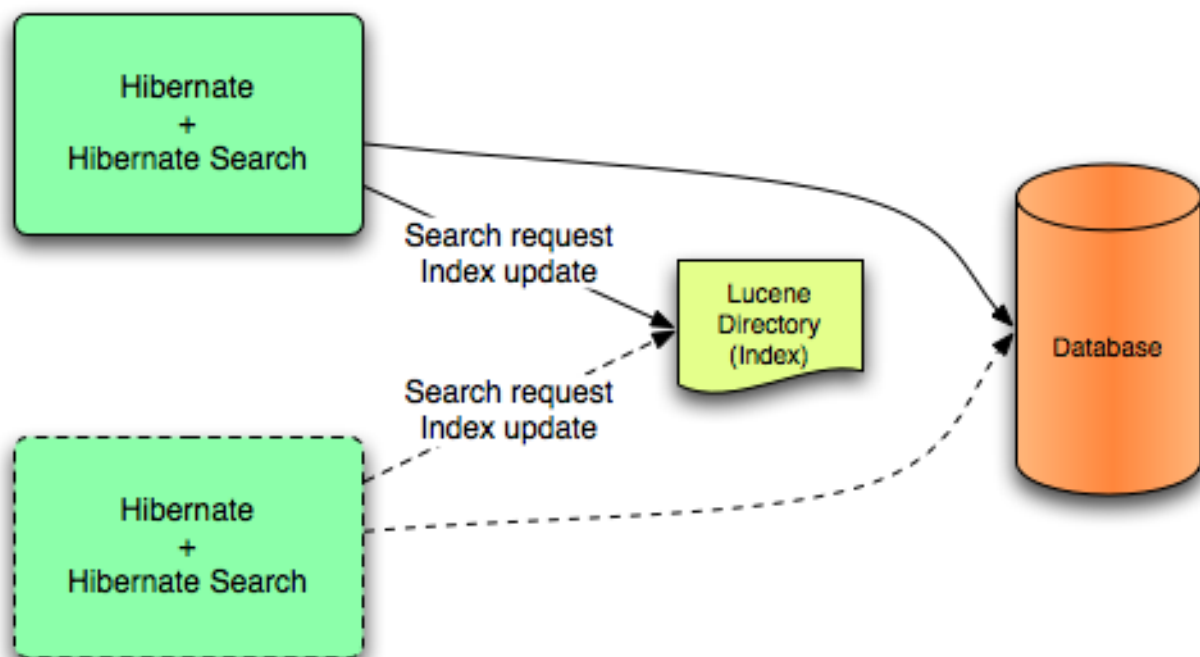


Figure 4.1. Lucene Back End Configuration

Lucene mode targets non-clustered or clustered applications where the **Directory** manages the locking strategy. The primary advantage of Lucene mode is simplicity and immediate visibility of changes in Lucene queries. The Near Real Time (NRT) back end is an alternate back end for non-clustered and non-shared index configurations.

[Report a bug](#)

4.2.3. JMS

Index updates for a node are sent to the JMS queue. A unique reader processes the queue and updates the master index. The master index is subsequently replicated regularly to slave copies to establish the master/slave pattern. The master is responsible for Lucene index updates. The slaves accept read and write operations but process read operations on local index copies. The master is the sole responsible for updating the Lucene index. Only the master applies the local changes in an update operation.

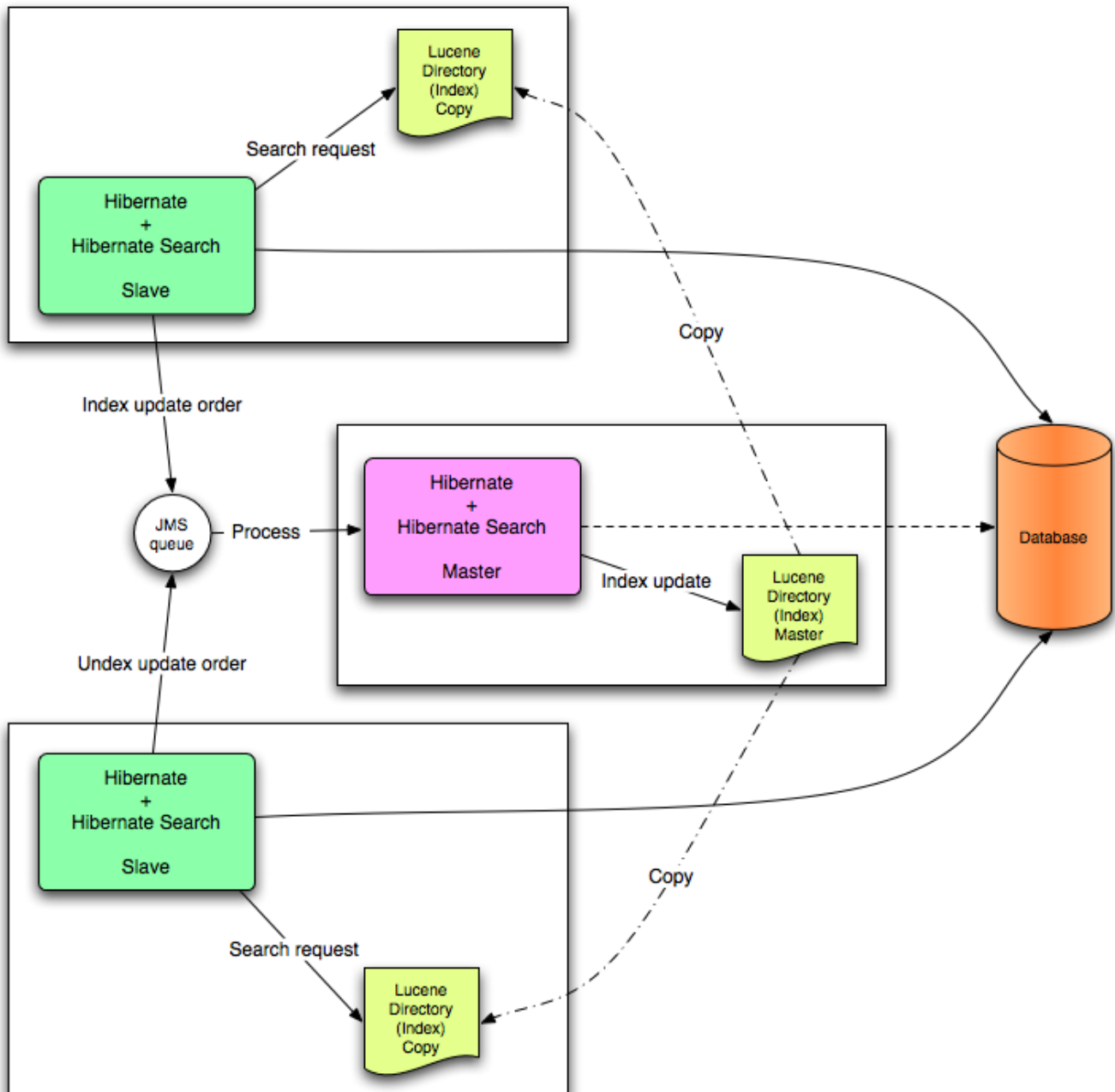


Figure 4.2. JMS Backend Configuration

This mode targets clustered environments where throughput is critical and index update delays are affordable. The JMS provider ensures reliability and uses the slaves to change the local index copies.

[Report a bug](#)

4.3. Reader Strategies

When executing a query, Hibernate Search uses a reader strategy to interact with the Apache Lucene indexes. Choose a reader strategy based on the profile of the application (frequent updates, read mostly, asynchronous index update, etc). For details, see [Section 4.3.4, “Reader Strategy Configuration”](#).

[Report a bug](#)

4.3.1. The Shared Strategy

Using the **shared** strategy, Hibernate Search shares the same **IndexReader** for a given Lucene index across multiple queries and threads provided that the **IndexReader** remains updated. If the **IndexReader** is not updated, a new one is opened and provided. Each **IndexReader** is made of several **SegmentReaders**. The shared strategy reopens segments that have been modified or created after the last opening and shares the already loaded segments from the previous instance. This is the default strategy.

[Report a bug](#)

4.3.2. The Not-shared Strategy

Using the **not-shared** strategy, a Lucene **IndexReader** opens every time a query executes. Opening and starting up a **IndexReader** is an expensive operation. As a result, opening an **IndexReader** for each query execution is not an efficient strategy.

[Report a bug](#)

4.3.3. Custom Reader Strategies

You can write a custom reader strategy using an implementation of **org.hibernate.search.reader.ReaderProvider**. The implementation must be thread safe.

[Report a bug](#)

4.3.4. Reader Strategy Configuration

Change the strategy from the default (**shared**) to **not-shared** as follows:

```
hibernate.search.[default|<indexname>].reader.strategy = not-shared
```

Alternately, customize the reader strategy by replacing **my.corp.myapp.CustomReaderProvider** with the custom strategy implementation:

```
hibernate.search.[default|<indexname>].reader.strategy =  
my.corp.myapp.CustomReaderProvider
```

[Report a bug](#)

Chapter 5. Configuration

5.1. Hibernate Search and Automatic Indexing

5.1.1. Enable and Disable Hibernate Search

Hibernate Search is enabled by default when detected on the classpath by Hibernate Core. There is no performance penalty when the listeners are enabled but no entities are annotated as indexed. Disable Hibernate Search as follows:

```
hibernate.search.autoregister_listeners = false
```

[Report a bug](#)

5.1.2. Automatic Indexing

By default, when an object is inserted, updated, or deleted, Hibernate Search updates the corresponding Lucene index. Disable automatic indexing if either the index is read-only or if index updates are done in batches.

Use the following to disable event based indexing

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

The JMS back end provides both the lightweight event-based system that tracks changes in the system and the indexing process done by a separate process or machine.

[Report a bug](#)

5.2. Configuring the IndexManager

The index manager component is discussed in detail in [Section 4.1, “Overview”](#) Hibernate Search offers two implementations for this interface:

- ✦ **directory-based**: the default implementation which uses the Lucene **Directory** abstraction to manage index files.
- ✦ **near-real-time**: avoids flushing writes to disk at each commit. This index manager is also **Directory** based, but uses Lucene's NRT functionality.

To select an alternative, specify the following property:

```
hibernate.search.[default|<indexname>].indexmanager = near-real-time
```

[Report a bug](#)

5.2.1. Directory-based

The **Directory**-based implementation is the default **IndexManager** implementation. It is highly configurable and allows separate configurations for the reader strategy, back ends, and directory providers. Refer [Section 5.3, “Directory Configuration”](#), [Section 5.4, “Worker Configuration”](#) and [Section 4.3.4, “Reader Strategy Configuration”](#) for more details.

[Report a bug](#)

5.2.2. Near Real Time

The **NRTIndexManager** is an extension of the default **IndexManager** and leverages the Lucene NRT (Near Real Time) feature for low latency index writes. However, it ignores configuration settings for alternative back ends other than **lucene** and acquires exclusive write locks on the **Directory**.

The **IndexWriter** does not flush every change to the disk to provide low latency. Queries can read the updated states from the unflushed index writer buffers. However, this means that if the **IndexWriter** is killed or the application crashes, updates can be lost so the indexes must be rebuilt.

The Near Real Time configuration is recommended for non-clustered websites with limited data due to the mentioned disadvantages and because a master node can be individually configured for improved performance as well.

[Report a bug](#)

5.2.3. Custom

Specify a fully qualified class name for the custom implementation to set up a customized **IndexManager**. Set up a no-argument constructor for the implementation as follows:

```
[default|<indexname>].indexmanager = my.corp.myapp.CustomIndexManager
```

The custom index manager implementation does not require the same components as the default implementations. For example, delegate to a remote indexing service which does not expose a **Directory** interface.

[Report a bug](#)

5.3. Directory Configuration

As we have seen in [Section 5.2, “Configuring the IndexManager”](#) the default index manager uses Lucene's 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 5.1, “List of built-in DirectoryProvider”](#) 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 11.5, “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 (recommended).

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 5.2, “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 5.1. 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 5.2. 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
```



Note

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

Table 5.1. List of built-in DirectoryProvider

Name and 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

Name and description	Properties
<p>filesystem: File system based directory. The directory used will be <indexBase>/<indexName></p>	<p>indexBase : base directory</p> <p>indexName: override @Indexed.index (useful for sharded indexes)</p> <p>locking_strategy : optional, see Section 5.6, “LockFactory Configuration”</p> <p>filesystem_access_type: allows to determine the exact type of FSDirectory implementation used by this DirectoryProvider. Allowed values are auto (the default value, selects NIOFSDirectory on non Windows systems, SimpleFSDirectory on Windows), simple (SimpleFSDirectory), nio (NIOFSDirectory), mmap (MMapDirectory). Make sure to refer to Javadocs of these Directory implementations before changing this setting. Even though NIOFSDirectory or MMapDirectory can bring substantial performance boosts they also have their issues.</p>

Name and description	Properties
<p>filesystem-master: File system based directory. Like filesystem. It also copies the index to a source directory (aka copy directory) on a regular basis.</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>DirectoryProvider typically used on the master node in a JMS back end cluster.</p> <p>The buffer_size_on_copy optimum depends on your operating system and available RAM; most people reported good results using values between 16 and 64MB.</p>	<p>indexBase: base directory</p> <p>indexName: override @Indexed.index (useful for sharded indexes)</p> <p>sourceBase: source (copy) base directory.</p> <p>source: source directory suffix (default to @Indexed.index). The actual source directory name being <sourceBase>/<source></p> <p>refresh: refresh period in seconds (the copy will take place every refresh seconds). If a copy is still in progress when the following refresh period elapses, the second copy operation will be skipped.</p> <p>buffer_size_on_copy: The amount of MegaBytes to move in a single low level copy instruction; defaults to 16MB.</p> <p>locking_strategy : optional, see Section 5.6, "LockFactory Configuration"</p> <p>filesystem_access_type: allows to determine the exact type of FSDirectory implementation used by this DirectoryProvider. Allowed values are auto (the default value, selects NIOFSDirectory on non Windows systems, SimpleFSDirectory on Windows), simple (SimpleFSDirectory), nio (NIOFSDirectory), mmap (MMapDirectory). Make sure to refer to Javadocs of these Directory implementations before changing this setting. Even though NIOFSDirectory or MMapDirectory can bring substantial performance boosts they also have their issues.</p>

Name and description	Properties
<p>filesystem-slave: File system based directory. Like filesystem, 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. If a copy is still in progress when refresh period elapses, the second copy operation will be skipped.</p> <p>DirectoryProvider typically used on slave nodes using a JMS back end.</p> <p>The buffer_size_on_copy optimum depends on your operating system and available RAM; most people reported good results using values between 16 and 64MB.</p>	<p>indexBase: Base directory</p> <p>indexName: override @Indexed.index (useful for sharded indexes)</p> <p>sourceBase: Source (copy) base directory.</p> <p>source: Source directory suffix (default to @Indexed.index). The actual source directory name being <sourceBase>/<source></p> <p>refresh: refresh period in second (the copy will take place every refresh seconds).</p> <p>buffer_size_on_copy: The amount of MegaBytes to move in a single low level copy instruction; defaults to 16MB.</p> <p>locking_strategy : optional, see Section 5.6, "LockFactory Configuration"</p> <p>retry_marker_lookup : 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>retry_initialize_period : 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 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 retry_marker_lookup instead.</p> <p>filesystem_access_type: allows to determine the exact type of FSDirectory implementation used by this DirectoryProvider. Allowed values are auto (the default value, selects NIOFSDirectory on non Windows systems, SimpleFSDirectory on Windows), simple (SimpleFSDirectory), nio (NIOFSDirectory), mmap (MMapDirectory). Make sure to refer to Javadocs of these Directory implementations before changing this setting. Even though NIOFSDirectory or MMapDirectory can bring substantial performance boosts they also have their issues.</p>



Note

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>`.

[Report a bug](#)

5.4. 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 5.2, "Scope configuration"](#).

Table 5.2. Scope configuration

Property	Description
<code>hibernate.search.worker.scope</code>	The fully qualified class name of the Worker implementation to use. If this property is not set, empty or transaction the default TransactionalWorker is used.
<code>hibernate.search.worker.*</code>	All configuration properties prefixed with hibernate.search.worker are passed to the Worker 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 Worker implementation delegates the queued work to <code>BatchedQueueingProcessor</code> (which is what the TransactionalWorker 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 5.3, "Execution configuration"](#).



Note

The following options can be different on each index; in fact they need the `indexName` prefix or use **default** to set the default value for all indexes.

Table 5.3. Execution configuration

Property	Description
hibernate.search.<indexName>.worker.execution	sync : synchronous execution (default) async : asynchronous execution
hibernate.search.<indexName>.worker.thread_pool.size	The backend can apply updates from the same transaction context (or batch) in parallel, using a threadpool. The default value is 1. You can experiment with larger values if you have many operations per transaction.
hibernate.search.<indexName>.worker.buffer_queue.max	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.default.worker.backend` - see [Table 5.4, “Backend configuration”](#). Again this option can be configured differently for each index.

Table 5.4. Backend configuration

Property	Description
hibernate.search.<indexName>.worker.backend	<p>lucene: The default backend which runs index updates in the same VM. Also used when the property is undefined or empty.</p> <p>jms: JMS backend. Index updates are send to a JMS queue to be processed by an indexing master. See Table 5.5, “JMS backend configuration” for additional configuration options and Section 5.4.1, “JMS Master/Slave Back End” for a more detailed description of this setup.</p> <p>blackhole: Mainly a test/developer setting which ignores all indexing work</p> <p>You can also specify the fully qualified name of a class implementing BackendQueueProcessor. This way you can implement your own communication layer. The implementation is responsible for returning a Runnable instance which on execution will process the index work.</p>

Table 5.5. JMS backend configuration

Property	Description
hibernate.search.<indexName>.worker.jndi.*	Defines the JNDI properties to initiate the InitialContext (if needed). JNDI is only used by the JMS back end.
hibernate.search.<indexName>.worker.jms.connection_factory	Mandatory for the JMS back end. Defines the JNDI name to lookup the JMS connection factory from (/ConnectionFactory by default in Red Hat JBoss Enterprise Application Platform)
hibernate.search.<indexName>.worker.jms.queue	Mandatory for the JMS back end. Defines the JNDI name to lookup the JMS queue from. The queue will be used to post work messages.



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 **BackendQueueProcessor** implementation.

[Report a bug](#)

5.4.1. JMS Master/Slave Back End

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

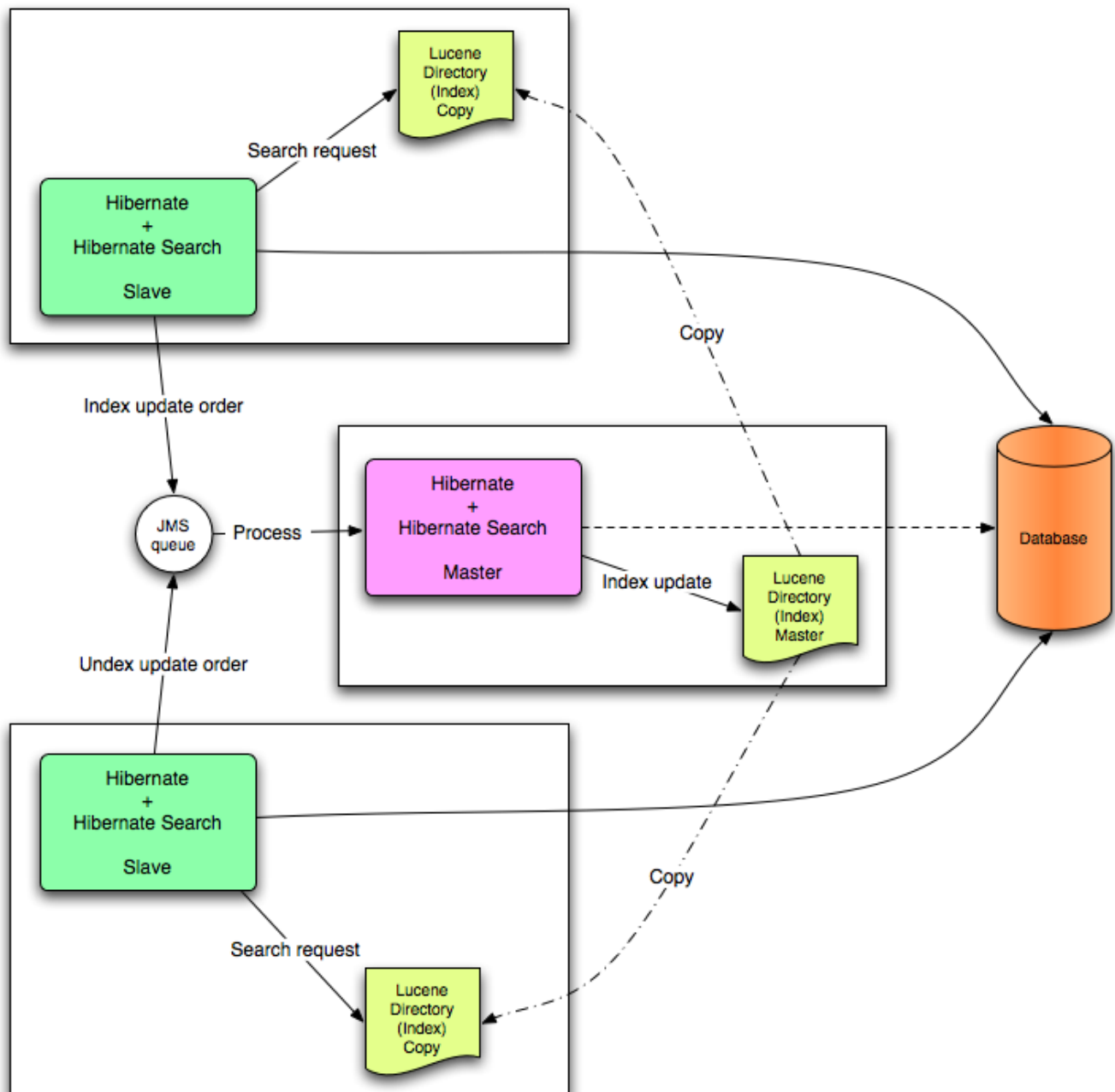


Figure 5.1. JMS Backend Configuration

[Report a bug](#)

5.4.2. Slave Nodes

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

Example 5.3. 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.default.worker.backend = jms
hibernate.search.default.worker.jms.connection_factory =
/ConnectionFactory
hibernate.search.default.worker.jms.queue = queue/hibernatesearch
#optional jndi configuration (check your JMS provider for more
information)

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



Note

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

[Report a bug](#)

5.4.3. Master Node

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

Example 5.4. 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
```

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 5.5. 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 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()`, see `AbstractJMSHibernateSearchController`'s javadoc.

[Report a bug](#)

5.5. Tuning Lucene Indexing

5.5.1. Tuning Lucene Indexing Performance

Hibernate Search is used 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`. Specify these parameters either as default values applying for all indexes, on a per index basis, or even per shard.

There are several low level **IndexWriter** settings which can be tuned for different use cases. These parameters are grouped by the **indexwriter** keyword:

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

If no value is set for an **indexwriter** value in a specific shard configuration, Hibernate Search checks the index section, then at the default section.

The configuration in the following table will result in these settings applied on the second shard of the **Animal** index:

- ✧ **max_merge_docs** = 10
- ✧ **merge_factor** = 20
- ✧ **ram_buffer_size** = 64MB
- ✧ **term_index_interval** = 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 5.6, "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, see the Lucene documentation.



Note

Previous versions of Search had the notion of **batch** and **transaction** properties. This is no longer the case as the backend will always perform work using the same settings.

Table 5.6. List of indexing performance and behavior properties

Property	Description	Default Value
hibernate.search.[default <indexname>].exclusive_index_use	Set to true when no other process will need to write to the same index. This enables Hibernate Search to work in exclusive mode on the index and improve performance when writing changes to the index.	true (improved performance, releases locks only at shutdown)
hibernate.search.[default <indexname>].max_queue_length	Each index has a separate "pipeline" which contains the updates to be applied to the index. When this queue is full adding more operations to the queue becomes a blocking operation. Configuring this setting doesn't make much sense unless the worker.execution is configured as async .	1000

Property	Description	Default Value
hibernate.search.[default <indexname>].indexwriter.max_buffered_delete_terms	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)
hibernate.search.[default <indexname>].indexwriter.max_buffered_docs	Controls the amount of documents buffered in memory during indexing. The bigger the more RAM is consumed.	Disabled (flushes by RAM usage)
hibernate.search.[default <indexname>].indexwriter.max_merge_docs	Defines the largest number of documents allowed in a segment. Smaller values perform better on frequently changing indexes, larger values provide better search performance if the index does not change often.	Unlimited (Integer.MAX_VALUE)
hibernate.search.[default <indexname>].indexwriter.merge_factor	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 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.	10

Property	Description	Default Value
hibernate.search.[default <indexname>].indexwriter.merge_min_size	<p>Controls segment merge frequency and size.</p> <p>Segments smaller than this size (in MB) are always considered for the next segment merge operation.</p> <p>Setting this too large might result in expensive merge operations, even though they are less frequent.</p> <p>See also org.apache.lucene.index.LogDocMergePolicy.minMergeSize.</p>	0 MB (actually ~1K)
hibernate.search.[default <indexname>].indexwriter.merge_max_size	<p>Controls segment merge frequency and size.</p> <p>Segments larger than this size (in MB) are never merged in bigger segments.</p> <p>This helps reduce memory requirements and avoids some merging operations at the cost of optimal search speed. When optimizing an index this value is ignored.</p> <p>See also org.apache.lucene.index.LogDocMergePolicy.maxMergeSize.</p>	Unlimited
hibernate.search.[default <indexname>].indexwriter.merge_max_optimize_size	<p>Controls segment merge frequency and size.</p> <p>Segments larger than this size (in MB) are not merged in bigger segments even when optimizing the index (see merge_max_size setting as well).</p> <p>Applied to org.apache.lucene.index.LogDocMergePolicy.maxMergeSizeForOptimize.</p>	Unlimited

Property	Description	Default Value
hibernate.search.[default <indexname>].indexwriter.merge_calibrate_by_deletes	<p>Controls segment merge frequency and size.</p> <p>Set to false to not consider deleted documents when estimating the merge policy.</p> <p>Applied to org.apache.lucene.index.LogMergePolicy.calibrateSizeByDeletes.</p>	true
hibernate.search.[default <indexname>].indexwriter.ram_buffer_size	<p>Controls the amount of RAM in MB dedicated to document buffers. When used together max_buffered_docs 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
hibernate.search.[default <indexname>].indexwriter.term_index_interval	<p>Expert: Set the interval between indexed terms.</p> <p>Large values cause less memory to be used by IndexReader, but slow random-access to terms. Small values cause more memory to be used by an IndexReader, and speed random-access to terms. See Lucene documentation for more details.</p>	128
hibernate.search.[default <indexname>].indexwriter.use_compound_file	<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 false in an attempt to improve the indexing time, but you could run out of file descriptors if mergeFactor is also large.</p> <p>Boolean parameter, use "true" or "false". The default value for this option is true.</p>	true

Property	Description	Default Value
hibernate.search.enable_dirty_check	<p>Not all entity changes require a Lucene index update. If all of the updated entity properties (dirty properties) are not indexed, Hibernate Search skips the re-indexing process.</p> <p>Disable this option if you use custom FieldBridges 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 @ClassBridge or a @DynamicBoost.</p> <p>Boolean parameter, use "true" or "false". The default value for this option is true.</p>	true



Warning

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

[Report a bug](#)

5.5.2. The Lucene IndexWriter

There are several low level **IndexWriter** settings which can be tuned for different use cases. These parameters are grouped by the **indexwriter** keyword:

```
default.<indexname>.indexwriter.<parameter_name>
```

If no value is set for **indexwriter** in a shard configuration, Hibernate Search looks at the index section and then at the default section.

[Report a bug](#)

5.5.3. Performance Option Configuration

The following configuration will result in these settings being applied on the second shard of the **Animal** index:

Example 5.6. Example performance option configuration

```
default.Animals.2.indexwriter.max_merge_docs = 10
default.Animals.2.indexwriter.merge_factor = 20
default.Animals.2.indexwriter.term_index_interval = default
default.indexwriter.max_merge_docs = 100
default.indexwriter.ram_buffer_size = 64
```

- » **max_merge_docs** = 10
- » **merge_factor** = 20
- » **ram_buffer_size** = 64MB
- » **term_index_interval** = Lucene default

All other values will use the defaults defined in Lucene.

The Lucene default values are the default setting for Hibernate Search. Therefore, the values listed in the following table depend on the version of Lucene being used. The values shown are relative to version **2.4**. For more information about Lucene indexing performance, see the Lucene documentation.



Note

The back end will always perform work using the same settings.

Table 5.7. List of indexing performance and behavior properties

Property	Description	Default Value
default. <indexname>.exclusive_index_ use	Set to true when no other process will need to write to the same index. This enables Hibernate Search to work in exclusive mode on the index and improve performance when writing changes to the index.	true (improved performance, releases locks only at shutdown)
default. <indexname>.max_queue_leng th	Each index has a separate "pipeline" which contains the updates to be applied to the index. When this queue is full adding more operations to the queue becomes a blocking operation. Configuring this setting doesn't make much sense unless the worker.execution is configured as async .	1000

Property	Description	Default Value
default. <indexname>.indexwriter.max_ buffered_delete_terms	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)
default. <indexname>.indexwriter.max_ buffered_docs	Controls the amount of documents buffered in memory during indexing. The bigger the more RAM is consumed.	Disabled (flushes by RAM usage)
default. <indexname>.indexwriter.max_ merge_docs	Defines the largest number of documents allowed in a segment. Smaller values perform better on frequently changing indexes, larger values provide better search performance if the index does not change often.	Unlimited (Integer.MAX_VALUE)
default. <indexname>.indexwriter.merg e_factor	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 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.	10

Property	Description	Default Value
default. <indexname>.indexwriter.merge_min_size	<p>Controls segment merge frequency and size.</p> <p>Segments smaller than this size (in MB) are always considered for the next segment merge operation.</p> <p>Setting this too large might result in expensive merge operations, even though they are less frequent.</p> <p>See also org.apache.lucene.index.LogDocMergePolicy.minMergeSize.</p>	0 MB (actually ~1K)
default. <indexname>.indexwriter.merge_max_size	<p>Controls segment merge frequency and size.</p> <p>Segments larger than this size (in MB) are never merged in bigger segments.</p> <p>This helps reduce memory requirements and avoids some merging operations at the cost of optimal search speed. When optimizing an index this value is ignored.</p> <p>See also org.apache.lucene.index.LogDocMergePolicy.maxMergeSize.</p>	Unlimited
default. <indexname>.indexwriter.merge_max_optimize_size	<p>Controls segment merge frequency and size.</p> <p>Segments larger than this size (in MB) are not merged in bigger segments even when optimizing the index (see merge_max_size setting as well).</p> <p>Applied to org.apache.lucene.index.LogDocMergePolicy.maxMergeSizeForOptimize.</p>	Unlimited

Property	Description	Default Value
default. <indexname>.indexwriter.merge_calibrate_by_deletes	<p>Controls segment merge frequency and size.</p> <p>Set to false to not consider deleted documents when estimating the merge policy.</p> <p>Applied to org.apache.lucene.index.LogMergePolicy.calibrateSizeByDeletes.</p>	true
default. <indexname>.indexwriter.ram_buffer_size	<p>Controls the amount of RAM in MB dedicated to document buffers. When used together max_buffered_docs 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
default. <indexname>.indexwriter.term_index_interval	<p>Expert: Set the interval between indexed terms.</p> <p>Large values cause less memory to be used by IndexReader, but slow random-access to terms. Small values cause more memory to be used by an IndexReader, and speed random-access to terms. See Lucene documentation for more details.</p>	128
default. <indexname>.indexwriter.use_compound_file	<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 false in an attempt to improve the indexing time, but you could run out of file descriptors if mergeFactor is also large.</p> <p>Boolean parameter, use "true" or "false". The default value for this option is true.</p>	true

Property	Description	Default Value
<code>default.enable_dirty_check</code>	<p>Not all entity changes require a Lucene index update. If all of the updated entity properties (dirty properties) are not indexed, Hibernate Search skips the re-indexing process.</p> <p>Disable this option if you use custom FieldBridges 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 @ClassBridge or a @DynamicBoost.</p> <p>Boolean parameter, use "true" or "false". The default value for this option is true.</p>	true

[Report a bug](#)

5.5.4. Tuning the Indexing Speed

When the architecture permits it, keep **default.exclusive_index_use=true** for improved index writing efficiency.

To tune the indexing speed, time the object loading from the database in isolation from the writes to the index. Set the **blackhole** as worker back end and start your indexing routines. This back end does not disable Hibernate Search: it generates the required change sets to the index, but discards 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.[default|<indexname>].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** back end is not meant to be used in production, only as a tool to identify indexing bottlenecks.

[Report a bug](#)

5.5.5. Control Segment Size

The following options configure the maximum size of segments created:

- ✧ ***merge_max_size***
- ✧ ***merge_max_optimize_size***
- ✧ ***merge_calibrate_by_deletes***

Example:

```
//to be fairly confident no files grow above 15MB, use:
hibernate.search.default.indexwriter.ram_buffer_size = 10
hibernate.search.default.indexwriter.merge_max_optimize_size = 7
hibernate.search.default.indexwriter.merge_max_size = 7
```

Set the ***max_size*** for merge operations to less than half of the hard limit segment size, as merging segments combines two segments into one larger segment.

A new segment may initially be a larger size than expected, however a segment is never created significantly larger than the ***ram_buffer_size***. This threshold is checked as an estimate.

[Report a bug](#)

5.6. LockFactory Configuration

The Lucene Directory can be configured with a custom locking strategy via **LockingFactory** for each index managed by Hibernate Search.

Some locking strategies require a filesystem level lock, and may be used on RAM-based indexes. When using this strategy the **IndexBase** configuration option must be specified to point to a filesystem location in which to store the lock marker files.

To select a locking factory, set the **hibernate.search.<index>.locking_strategy** option to one the following options:

- ✧ ***simple***
- ✧ ***native***
- ✧ ***single***
- ✧ ***none***

Table 5.8. List of available LockFactory implementations

name	Class	Description
simple	org.apache.lucene.store.SimpleFSLockFactory	<p>Safe 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>

name	Class	Description
native	org.apache.lucene.store. NativeFSLockFactory	<p>As does 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 the JVM is terminated the locks will be cleaned up.</p> <p>This implementation has known problems on NFS, avoid it on network shares.</p> <p>native is the default implementation for the filesystem, filesystem-master and filesystem-slave directory providers.</p>
single	org.apache.lucene.store. SingleInstanceLockFactory	<p>This LockFactory 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.</p> <p>This is the default implementation for the ram directory provider.</p>
none	org.apache.lucene.store. NoLockFactory	Changes to this index are not coordinated by a lock.

The following is an example of locking strategy configuration:

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

[Report a bug](#)

5.7. 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 follows:

```
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
```

[Report a bug](#)

5.8. Index Format Compatibility

Hibernate Search does not currently offer a backwards compatible API or tool to facilitate porting applications to newer versions. The API uses Apache Lucene for index writing and searching. Occasionally an update to the index format may be required. In this case, there is a possibility that data will need to be re-indexed if Lucene is unable to read the old format.



Warning

Back up indexes before attempting to update the index format.

Hibernate Search exposes the **hibernate.search.lucene_version** configuration property. This property instructs Analyzers and other Lucene classes to conform to their behaviour as defined in an older version of Lucene. See also **org.apache.lucene.util.Version** contained in the **lucene-core.jar**. If the option is not specified, Hibernate Search instructs Lucene to use the version default. It is recommended that the version used is explicitly defined in the configuration to prevent automatic changes when an upgrade occurs. After an upgrade, the configuration values can be updated explicitly if required.

Example 5.7. Force Analyzers to be compatible with a Lucene 3.0 created index

```
hibernate.search.lucene_version = LUCENE_30
```

The configured **SearchFactory** is global and affects all Lucene APIs that contain the relevant parameter. If Lucene is used and Hibernate Search is bypassed, apply the same value to it for consistent results.

[Report a bug](#)

5.9. Hibernate Search as Red Hat JBoss EAP Module

In JBoss EAP 6, class loading is based on modules that define explicit dependencies on other modules. Modules allow sharing of common artifacts across applications thus resulting in smaller and quicker deployments.

Hibernate Search modules are distributed as part of the JBoss Web Framework Kit Maven repository. The repository is distributed with the release in the **jboss-wfk-2.7.0-maven-repository.zip** file, and can be downloaded from [Red Hat Customer Portal](#).

Unpack the Maven repository ZIP archive. Inside the unpacked directory, there is another ZIP archive containing the actual modules. It is located at the following path:

```
org/hibernate/hibernate-search-modules/4.4.4.Final-redhat-wfk-1/hibernate-search-modules-4.4.4.Final-redhat-wfk-1-jbossas-74-dist.zip
```

Unpack the archive into the **modules/** directory in the target JBoss Enterprise Application Platform. Modules for Hibernate Search, Apache Lucene, and some useful Solr libraries will be added. The Hibernate Search modules are:

- ✱ *org.hibernate.search.orm:main* for users of Hibernate Search with Hibernate; this transitively includes Hibernate ORM.
- ✱ *org.hibernate.search.engine:main* for projects depending on the internal indexing engine that do not require other dependencies to Hibernate.

Hibernate Search offers the following two methods for including dependencies in a project:

Using MANIFEST.MF file

Add the following entry to the **MANIFEST.MF** file in the project archive:

```
Dependencies: org.hibernate.search.orm services
```

Using jboss-deployment-structure.xml file

Add **WEB-INF/jboss-deployment-structure.xml** file in the project archive with the following content:

```
<jboss-deployment-structure>
  <deployment>
    <dependencies>
      <module name="org.hibernate.search.orm"
services="export" />
    </dependencies>
  </deployment>
</jboss-deployment-structure>
```

[Report a bug](#)

Chapter 6. Mapping Entities to the Index Structure

6.1. Mapping an Entity

In [Section 3.1, “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.

[Report a bug](#)

6.1.1. Basic Mapping

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

The Lucene-based Query API uses the following common annotations to map entities:

- `@Indexed`
- `@Field`
- `@NumericField`
- `@Id`

[Report a bug](#)

6.1.1.1. `@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 6.1. 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 5.3, “Directory Configuration”](#).

[Report a bug](#)

6.1.1.2. `@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 7.1.10.5, “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 whether the property is indexed or not. The different values are **Index.NO** (no indexing, ie cannot be found by a query), **Index.YES** (the element gets indexed and is searchable). The default value is **Index.YES**. **Index.NO** can be useful for cases where a property is not required to be searchable, but should be available for projection.



Note

Index.NO in combination with **Analyze.YES** or **Norms.YES** is not useful, since **analyze** and **norms** require the property to be indexed

- ✧ **analyze**: determines whether the property is analyzed (**Analyze.YES**) or not (**Analyze.NO**). The default value is **Analyze.YES**.



Note

Whether or not you want to analyze a property depends on whether you wish to search the element as is, or by the words it contains. It make sense to analyze a text field, but probably not a date field.



Note

Fields used for sorting *must not* be analyzed.

- ✧ **norms**: describes whether index time boosting information should be stored (**Norms.YES**) or not (**Norms.NO**). Not storing it can save a considerable amount of memory, but there won't be any index time boosting information available. The default value is **Norms.YES**.
- ✧ **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
TermVector.YES	Store the term vectors of each document. This produces two synchronized arrays, one contains document terms and the other contains the term's frequency.
TermVector.NO	Do not store term vectors.

Value	Definition
TermVector.WITH_OFFSETS	Store the term vector and token offset information. This is the same as TermVector.YES plus it contains the starting and ending offset position information for the terms.
TermVector.WITH_POSITIONS	Store the term vector and token position information. This is the same as TermVector.YES plus it contains the ordinal positions of each occurrence of a term in a document.
TermVector.WITH_POSITION_OFFSETS	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 [Chapter 7, Querying](#)) to search for **null** values. It is also advisable to use this feature only with un-analyzed fields (**analyze=Analyze.NO**).



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()**).

[Report a bug](#)

6.1.1.3. @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](#). 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
forField	(Optional) Specify the name of the related @Field that will be indexed as numeric. It's only mandatory when the property contains more than a @Field declaration

Value	Definition
precisionStep	(Optional) Change the way that the Trie structure is stored in the index. Smaller precisionSteps 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 @Fields. Default value is 4.

@NumericField supports only **Double**, **Long**, **Integer** and **Float**. It is not possible to take any advantage from a similar functionality in Lucene for the other numeric types, so remaining types should use the string encoding via the default or custom **TwoWayFieldBridge**.

It is possible to use a custom **NumericFieldBridge** assuming you can deal with the approximation during type transformation:

Example 6.2. Defining a custom NumericFieldBridge

```
public class BigDecimalNumericFieldBridge extends NumericFieldBridge {
    private static final BigDecimal storeFactor =
        BigDecimal.valueOf(100);

    @Override
    public void set(String name, Object value, Document document,
        LuceneOptions luceneOptions) {
        if ( value != null ) {
            BigDecimal decimalValue = (BigDecimal) value;
            Long indexedValue = Long.valueOf( decimalValue.multiply(
                storeFactor ).longValue() );
            luceneOptions.addNumericFieldToDocument( name, indexedValue,
                document );
        }
    }

    @Override
    public Object get(String name, Document document) {
        String fromLucene = document.get( name );
        BigDecimal storedBigDecimal = new BigDecimal( fromLucene );
        return storedBigDecimal.divide( storeFactor );
    }
}
```

[Report a bug](#)

6.1.1.4. @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 6.3. Specifying indexed properties

```

@Entity
@Indexed
public class Essay {
    ...

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

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

    @Lob
    @Field
    public String getText() { return text; }

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

```

[Example 6.3, “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 precision step than the default.

[Report a bug](#)

6.1.2. 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-analyzed. If one wants to search by words in this property and still sort it, one needs to index it twice - once analyzed and once un-analyzed. `@Fields` allows to achieve this goal.

Example 6.4. Using `@Fields` to map a property multiple times

```

@Entity
@Indexed(index = "Book" )
public class Book {
    @Fields( {
        @Field,
        @Field(name = "summary_forSort", analyze = Analyze.NO,
store = Store.YES)
    } )
    public String getSummary() {
        return summary;
    }

    ...
}

```

In [Example 6.4, “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:

See below for more information about analyzers and field bridges.

[Report a bug](#)

6.1.3. 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 6.5, “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 6.5. Indexing associations

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

    @Field
    private String name;

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

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

    @Field
    private String street;

    @Field
    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 its **Address** changes, you need to mark the other side of the bidirectional relationship with **@ContainedIn**.



Note

@ContainedIn is useful on both associations pointing to entities and on embedded (collection of) objects.

Let's make [Example 6.5, "Indexing associations"](#) a bit more complex by nesting **@IndexedEmbedded** as seen in [Example 6.6, "Nested usage of @IndexedEmbedded and @ContainedIn"](#).

Example 6.6. Nested usage of @IndexedEmbedded and @ContainedIn

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

    @Field
    private String name;

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

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

    @Field
    private String street;

    @Field
    private String city;

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

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

```

}

@Embeddable
public class Owner {
    @Field
    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 6.6, “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.orderBy_name:joe
```

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

**Note**

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

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

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

Example 6.7. Using the `targetElement` property of `@IndexedEmbedded`

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

    @Field
    private String street;

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

    ...
}

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

[Report a bug](#)

6.1.4. Limiting Object Embedding to Specific Paths

The **@IndexedEmbedded** annotation provides also an attribute **includePaths** which can be used as an alternative to **depth**, or be combined with it.

When using only **depth** all indexed fields of the embedded type will be added recursively at the same depth; this makes it harder to pick only a specific path without adding all other fields as well, which might not be needed.

To avoid unnecessarily loading and indexing entities you can specify exactly which paths are needed. A typical application might need different depths for different paths, or in other words it might need to specify paths explicitly, as shown in [Example 6.8, “Using the `includePaths` property of `@IndexedEmbedded`”](#)

Example 6.8. Using the `includePaths` property of `@IndexedEmbedded`

```
@Entity
@Indexed
public class Person {

    @Id
    public int getId() {
        return id;
    }

    @Field
    public String getName() {
        return name;
    }

    @Field
    public String getSurname() {
        return surname;
    }

    @OneToMany
    @IndexedEmbedded(includePaths = { "name" })
    public Set<Person> getParents() {
        return parents;
    }

    @ContainedIn
    @ManyToOne
    public Human getChild() {
        return child;
    }

    ...//other fields omitted
}
```

Using a mapping as in [Example 6.8, “Using the `includePaths` property of `@IndexedEmbedded`”](#), you would be able to search on a **Person** by **name** and/or **surname**, and/or the **name** of the parent. It will not index the **surname** of the parent, so searching on parent's surnames will not be possible but speeds up indexing, saves space and improve overall performance.

The `@IndexedEmbeddedincludePaths` will include the specified paths *in addition to* what you would index normally specifying a limited value for **depth**. When using **includePaths**, and leaving **depth** undefined, behavior is equivalent to setting **depth=0**: only the included paths are indexed.

Example 6.9. Using the `includePaths` property of `@IndexedEmbedded`

```
@Entity
```

```

@Indexed
public class Human {

    @Id
    public int getId() {
        return id;
    }

    @Field
    public String getName() {
        return name;
    }

    @Field
    public String getSurname() {
        return surname;
    }

    @OneToMany
    @IndexedEmbedded(depth = 2, includePaths = { "parents.parents.name"
})
    public Set<Human> getParents() {
        return parents;
    }

    @ContainedIn
    @ManyToOne
    public Human getChild() {
        return child;
    }

    ...//other fields omitted

```

In [Example 6.9, “Using the `includePaths` property of `@IndexedEmbedded`”](#), every human will have its name and surname attributes indexed. The name and surname of parents will be indexed too, recursively up to second level because of the **depth** attribute. It will be possible to search by name or surname, of the person directly, his parents or of his grand parents. Beyond the second level, we will in addition index one more level but only the name, not the surname.

This results in the following fields in the index:

- ✧ **id** - as primary key
- ✧ **_hibernate_class** - stores entity type
- ✧ **name** - as direct field
- ✧ **surname** - as direct field
- ✧ **parents.name** - as embedded field at depth 1
- ✧ **parents.surname** - as embedded field at depth 1
- ✧ **parents.parents.name** - as embedded field at depth 2
- ✧ **parents.parents.surname** - as embedded field at depth 2

- » **parents.parents.parents.name** - as additional path as specified by **includePaths**. The first **parents.** is inferred from the field name, the remaining path is the attribute of **includePaths**

Having explicit control of the indexed paths might be easier if you're designing your application by defining the needed queries first, as at that point you might know exactly which fields you need, and which other fields are unnecessary to implement your use case.

[Report a bug](#)

6.2. 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.

[Report a bug](#)

6.2.1. 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 6.10. Different ways of using @Boost

```
@Entity
@Indexed
@Boost(1.7f)
public class Essay {
    ...

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

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

    @Lob
    @Field(boost=@Boost(1.2f))
    public String getText() { return text; }

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

In [Example 6.10](#), “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.

[Report a bug](#)

6.2.2. Dynamic Index Time Boosting

The **@Boost** annotation used in [Section 6.2.1, “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 depend on the actual state of the entity. In this case you can use the **@DynamicBoost** annotation together with an accompanying custom **BoostStrategy**.

Example 6.11. Dynamic boost example

```
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 6.11, “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.

[Report a bug](#)

6.3. Analysis

Analysis is the process of converting text into single terms (words) and can be considered as one of the key features of a full-text 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.

[Report a bug](#)

6.3.1. 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 6.12. Different ways of using `@Analyzer`

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

    @Field
    private String name;

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

    @Field( analyzer = @Analyzer(impl = FieldAnalyzer.class )
    private String body;

    ...
}
```

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



Warning

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.

[Report a bug](#)

6.3.2. 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 (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](#) jar. Your distribution of Hibernate Search provides these dependencies in its **lib/optional** directory.

When using Maven all required Solr dependencies are now defined as dependencies of the artifact *org.hibernate:hibernate-search-analyzers*; add the following dependency :

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-search-analyzers</artifactId>
  <version>4.4.4.Final-redhat-wfk-1</version>
</dependency>
```

Let's review a concrete example now - [Example 6.13, "@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 6.13. @AnalyzerDef and the Solr framework

```
@AnalyzerDef(name="customanalyzer",
```

```

charFilters = {
    @CharFilterDef(factory = MappingCharFilterFactory.class, params = {
        @Parameter(name = "mapping",
            value = "org/hibernate/search/test/analyser/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/analyser/solr/stoplist.properties" ),
        @Parameter(name="ignoreCase", value="true")
    })
})
public class Team {
    ...
}

```



Note

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 6.14. 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/analyser/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/analyser/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 6.15, “Referencing an analyzer by name”](#).

Example 6.15. 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 must 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.

[Report a bug](#)

6.3.3. 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 6.1. 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: "á" => "a" "ñ" => "n" "ø" => "o"	none
HTMLStripCharFilterFactory	Remove HTML standard tags, keeping the text	none	none

Table 6.2. 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 6.3. 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 ignored when comparing stop words, false otherwise	solr-core

Factory	Description	Parameters	Additional dependencies
SnowballPorterFilterFactory	Reduces a word to it's root in a given language. (example: 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	solr-core
ISOLatin1AccentFilterFactory	Remove accents for languages like French	none	solr-core
PhoneticFilterFactory	Inserts phonetically similar tokens into the token stream	encoder : One of DoubleMetaphone, Metaphone, Soundex or RefinedSoundex inject : true will add tokens to the stream, false will replace the existing token maxCodeLength : sets the maximum length of the code to be generated. Supported only for Metaphone and DoubleMetaphone encodings	solr-core and commons-codec
CollationKeyFilterFactory	Converts each token into its java.text.CollationKey , and then encodes the CollationKey with IndexableBinaryStringTools , to allow it to be stored as an index term.	custom , language , country , variant , strength , decomposition see Lucene's CollationKeyFilter javadocs for more info	solr-core and commons-io

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.

[Report a bug](#)

6.3.4. Dynamic Analyzer Selection

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 6.16, “Usage of @AnalyzerDiscriminator”](#) demonstrates the usage of this annotation.

Example 6.16. 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 dynamically 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 or **null** if the default analyzer should not be overridden. [Example 6.16, “Usage of @AnalyzerDiscriminator”](#) assumes that the language parameter is either 'de' or 'en' which matches the specified names in the **@AnalyzerDefs**.

[Report a bug](#)

6.3.5. Retrieving an Analyzer

Retrieving an analyzer can be used when multiple analyzers have been used in a domain model, in order to benefit from stemming or phonetic approximation, etc. In this case, use the same analyzers to building a query. Alternatively, use the Hibernate Search query DSL, which selects the correct analyzer automatically. See [Section 7.1.2, “Building a Lucene Query”](#)

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.



Note

When you use programmatic mapping for a child entity, you can only see the fields defined by the child entity. Fields or methods inherited from a parent entity (annotated with **@MappedSuperclass**) are not configurable. To configure properties inherited from a parent entity, either override the property in the child entity or create a programmatic mapping for the parent entity. This mimics the usage of annotations where you cannot annotate a field or method of a parent entity unless it is redefined in the child entity.

Example 6.17. 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.



Note

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

[Report a bug](#)

6.4. 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.

[Report a bug](#)

6.4.1. 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 6.1.1.2, “@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 (that is, used in ranged queries) out of the box: they have to be padded

**Note**

Using a Range query has drawbacks, an alternative approach is to use a Filter query which will filter the result query to the appropriate range.

Hibernate Search also supports the use of a custom StringBridge as described in [Section 6.4.2, “Custom Bridges”](#).

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 TermRangeQuery, 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(analyze=Analyze.NO)
    @DateBridge(resolution=Resolution.MINUTE)
    private Date date;
    ...
}
```

**Warning**

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

**Important**

The default **Date** bridge uses Lucene's **DateTools** to convert from and to **String**. This means that all dates are expressed in GMT time. If your requirements are to store dates in a fixed time zone you have to implement a custom date bridge. Make sure you understand the requirements of your applications regarding to date indexing and searching.

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

6.4.2. 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.

[Report a bug](#)

6.4.2.1. 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 6.18. 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 6.18, “Custom StringBridge implementation”](#), any property or field can use this bridge thanks to the **@FieldBridge** annotation:

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

[Report a bug](#)

6.4.2.2. Parameterized Bridge

Parameters can also be passed to the bridge implementation making it more flexible. [Example 6.19, “Passing parameters to your bridge implementation”](#) implements a **ParameterizedBridge** interface and parameters are passed through the **@FieldBridge** annotation.

Example 6.19. 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<String,String> parameters) {
        String padding = parameters.get( PADDING_PROPERTY );
        if (padding != null) this.padding = Integer.parseInt( 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.

[Report a bug](#)

6.4.2.3. 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.

[Report a bug](#)

6.4.2.4. Two-Way Bridge

If you expect to use your bridge implementation on an id property (that is, 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 6.20. 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))`).

[Report a bug](#)

6.4.2.5. 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 6.21. 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 6.21, “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.



Note

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.

[Report a bug](#)

6.4.2.6. 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 6.22, “Implementing a class bridge”](#), `@ClassBridge` supports the `termVector` attribute discussed in [section Section 6.1.1, “Basic Mapping”](#).

Example 6.22. Implementing a class bridge

```

@Entity
@Indexed
@ClassBridge(name="branchnetwork",
             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.

[Report a bug](#)

Chapter 7. Querying

Hibernate Search can execute Lucene queries and retrieve domain objects managed by an 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 following four steps:

- ✧ Creating a **FullTextSession**
- ✧ Creating a Lucene query using either Hibernate Search query DSL (recommended) or using 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, use a **FullTextSession**. This Search specific session wraps a regular **org.hibernate.Session** in order to provide query and indexing capabilities.

Example 7.1. Creating a FullTextSession

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

Use the **FullTextSession** to build a full-text query using either the Hibernate Search query DSL or the native Lucene query.

Use the following code when using the Hibernate Search query DSL:

```
final QueryBuilder b =
fullTextSession.getSearchFactory().buildQueryBuilder().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
```

As an alternative, write the Lucene query using either the Lucene query parser or the Lucene programmatic API.

Example 7.2. 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

```

A Hibernate query built on the Lucene query is a **org.hibernate.Query**. This query remains in the same paradigm as other Hibernate query facilities, such as HQL (Hibernate Query Language), Native, and Criteria. Use methods such as **list()**, **uniqueResult()**, **iterate()** and **scroll()** with the query.

The same extensions are available with the Hibernate Java Persistence APIs:

Example 7.3. 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.

[Report a bug](#)

7.1. Building Queries

Hibernate Search queries are built on Lucene queries, allowing users to use any Lucene query type. When the query is built, Hibernate Search uses `org.hibernate.Query` as the query manipulation API for further query processing.

[Report a bug](#)

7.1.1. Building a Lucene Query Using the Lucene API

With the Lucene API, use either the query parser (simple queries) or the Lucene programmatic API (complex queries). Building a Lucene query is out of scope for the Hibernate Search documentation. For details, see the online Lucene documentation or a copy of *Lucene in Action* or *Hibernate Search in Action*.

[Report a bug](#)

7.1.2. Building a Lucene Query

Using the Lucene programmatic API, it is possible to write full-text queries. However, when using Lucene programmatic API, the parameters must be converted to their string equivalent and must also apply the correct analyzer to the right field. A ngram analyzer for example uses several ngrams as the tokens for a given word and should be searched as such. It is recommended to use the **QueryBuilder** for this task.

The Hibernate Search query API is fluent. This API has a following key characteristics:

- Method names are in English. As a result, API operations can be read and understood as a series of English phrases and instructions.
- It uses IDE autocompletion which helps possible completions for the current input prefix and allows the user to choose the right option.
- It often uses the chaining method pattern.
- It is easy to use and read the API operations.

To use the API, first create a query builder that is attached to a given `indexedentitytype`. This **QueryBuilder** knows what analyzer to use and what field bridge to apply. Several **QueryBuilder**s (one for each entity type involved in the root of your query) can be created. The **QueryBuilder** is derived from the **SearchFactory**.

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

The analyzer, used for a given field or fields can also be overridden.

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


The query builder is now used to build Lucene queries. Customize queries generated using Lucene's query parser or **Query** objects assembled using the Lucene programmatic API are used with the Hibernate Search DSL.

[Report a bug](#)

7.1.3. Keyword Queries

The following example shows how to search for a specific word:

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

Table 7.1. Keyword query parameters

Parameter	Description
keyword()	Use this parameter to find a specific word
onField()	Use this parameter to specify in which lucene field to search the word
matching()	use this parameter to specify the match for search string
createQuery()	creates the Lucene query object

- ✱ The value "storm" is passed through the **history FieldBridge**. This is useful when numbers or dates are involved.
- ✱ 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, ngram, 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).

To search a property that is not of type string.

```
@Indexed
public class Myth {
    @Field(analyze = Analyze.NO)
    @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, the **Date** object had to be converted to its string representation (in this case the year)

This conversion works for any object, provided that the **FieldBridge** has an **objectToString** method (and all built-in **FieldBridge** implementations do).

The next example searches a field that uses ngram analyzers. The ngram analyzers index succession of ngrams of words, which helps to avoid 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") } )
    }
)

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 ngram will be part of the query. The user is then able to find the Sysiphus myth (with a y). All that is transparently done for the user.



Note

If the user does not want a specific field to use the field bridge or the analyzer then the **ignoreAnalyzer()** or **ignoreFieldBridge()** functions can be called.

To search for multiple possible words in the same field, 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, 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.

[Report a bug](#)

7.1.4. 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();
```

The **threshold** is the limit above which two terms are considering matching. It is a decimal between 0 and 1 and the default value is 0.5. The **prefixLength** is the length of the prefix ignored by the "fuzzyness". While the default value is 0, a non zero value is recommended for indexes containing a huge amount of distinct terms.

[Report a bug](#)

7.1.5. Wildcard Queries

Wildcard queries can also be executed (queries where some of parts of the word are unknown). The **?** 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.

[Report a bug](#)

7.1.6. Phrase Queries

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

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

Approximate sentences can be searched 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")
    .sentence("Thou kill")
    .createQuery();
```

[Report a bug](#)

7.1.7. Range Queries

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();
```

[Report a bug](#)

7.1.8. Combining Queries

Queries can be aggregated (combine) 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. Following are some 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();

```

[Report a bug](#)

7.1.9. Query Options

The following is a summary of query options for query types and fields:

- » **boostedTo** (on query type and on field) boosts the whole query or the specific field to a given factor
- » **withConstantScore** (on query) returns all results that match the query have a constant score equals to the boost
- » **filteredBy(Filter)** (on query) filters query results using the **Filter** instance
- » **ignoreAnalyzer** (on field) ignores the analyzer when processing this field
- » **ignoreFieldBridge** (on field) ignores field bridge when processing this field

The following example illustrates how to use 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!

[Report a bug](#)

7.1.10. Build a Hibernate Search Query

7.1.10.1. Generality

After building the Lucene query, wrap it within a Hibernate query. The query searches all indexed entities and returns all types of indexed classes unless explicitly configured not to do so.

Example 7.4. Wrapping a Lucene Query in a Hibernate Query

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

For improved performance, restrict the returned types as follows:

Example 7.5. Filtering the Search Result by Entity Type

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

// or

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

The first part of the second example only returns the matching **Customers**. The second part of the same example returns matching **Actors** and **Items**. The type restriction is polymorphic. As a result, if the two subclasses **Salesman** and **Customer** of the base class **Person** return, specify **Person.class** to filter based on result types.

[Report a bug](#)

7.1.10.2. Pagination

To avoid performance degradation, it is recommended to restrict the number of returned objects per query. A user navigating from one page to another page is a very common use case. The way to define pagination is similar to defining pagination in a plain HQL or Criteria query.

Example 7.6. Defining pagination for a search query

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

**Note**

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

[Report a bug](#)

7.1.10.3. Sorting

Apache Lucene contains a flexible and powerful result sorting mechanism. The default sorting is by relevance and is appropriate for a large variety of use cases. The sorting mechanism can be changed to sort by other properties using the Lucene Sort object to apply a Lucene sorting strategy.

Example 7.7. Specifying a Lucene Sort

```
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();
```



Note

Fields used for sorting must not be tokenized. For more information about tokenizing, see [Section 6.1.1.2, “@Field”](#).

[Report a bug](#)

7.1.10.4. Fetching Strategy

Hibernate Search loads objects using a single query if the return types are restricted to one class. Hibernate Search is restricted by the static fetching strategy defined in the domain model. It is useful to refine the fetching strategy for a specific use case as follows:

Example 7.8. 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.

In a criteria query definition, the type is guessed based on the provided criteria query. As a result, it is not necessary to restrict the return entity types.



Important

The fetch mode is the only adjustable property. Do not use a restriction (a where clause) on the **Criteria** query because the **getResultSize()** throws a **SearchException** if used in conjunction with a **Criteria** with restriction.

If more than one entity is expected, do not use **setCriteriaQuery**.

[Report a bug](#)

7.1.10.5. Projection

In some cases, only a small subset of the properties is required. Use Hibernate Search to return a subset of properties as follows:

Hibernate Search extracts properties from the Lucene index and converts them to their object representation and returns a list of **Object[]**. Projections prevent a time consuming database round-trip. However, they have following 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.

- ✱ Only the simple properties of the indexed entity or its embedded associations can be projected. Therefore a whole embedded entity cannot be projected.
- ✱ Projection does not work on collections or maps which are indexed via **@IndexedEmbedded**

Lucene provides metadata information about query results. Use projection constants to retrieve the metadata.

Example 7.9. Using Projection to Retrieve Metadata

```
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];
```

Fields can be mixed with the following projection 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.
- ✦ **FullTextQuery.EXPLANATION**: returns the Lucene Explanation object for the matching object/document in the given query. This is not suitable for retrieving large amounts of data. Running explanation typically is as costly as running the whole Lucene query per matching element. As a result, projection is recommended.

[Report a bug](#)

7.1.10.6. Customizing Object Initialization Strategies

By default, Hibernate Search uses the most appropriate strategy to initialize entities matching the full text query. It executes one (or several) queries to retrieve the required entities. This approach minimizes database trips where few of the retrieved entities are present in the persistence context (the session) or the second level cache.

If entities are present in the second level cache, force Hibernate Search to look into the cache before retrieving a database object.

Example 7.10. 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 to check if an object is easily accessible (without fetching it from the database). Other options are:

- ✦ **ObjectLookupMethod.PERSISTENCE_CONTEXT** is used if many matching entities are already loaded into the persistence context (loaded in the **Session** or **EntityManager**).
- ✦ **ObjectLookupMethod.SECOND_LEVEL_CACHE** checks the persistence context and then the second-level cache.

Set the following to search in the second-level cache:

- ✦ Correctly configure and activate the second-level cache.
- ✦ Enable the second-level cache for the relevant entity. This is done using annotations such as **@Cacheable**.
- ✦ Enable second-level cache read access for either **Session**, **EntityManager** or **Query**. Use **CacheMode.NORMAL** in Hibernate native APIs or **CacheRetrieveMode.USE** in Java Persistence APIs).



Warning

Unless the second-level cache implementation is EHCACHE or Infinispan, do not use **ObjectLookupMethod.SECOND_LEVEL_CACHE**. Other second-level cache providers do not implement this operation efficiently.

Customize how objects are loaded from the database using **DatabaseRetrievalMethod** as follows:

- ✦ **QUERY** (default) uses a set of queries to load several objects in each batch. This approach is recommended.
- ✦ **FIND_BY_ID** loads one object at a time using the **Session.get** or **EntityManager.find** semantic. This is recommended if the batch size is set for the entity, which allows Hibernate Core to load entities in batches.

[Report a bug](#)

7.1.10.7. Limiting the Time of a Query

Limit the time a query takes in Hibernate Guide as follows:

- ✦ Raise an exception when arriving at the limit.
- ✦ Limit to the number of results retrieved when the time limit is raised.

[Report a bug](#)

7.1.10.8. Raise an Exception on Time Limit

If a query uses more than the defined amount of time, a **QueryTimeoutException** is raised (**org.hibernate.QueryTimeoutException** or **javax.persistence.QueryTimeoutException** depending on the programmatic API).

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

Example 7.11. 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
}
```

The `getResultSize()`, `iterate()` and `scroll()` honor the timeout until the end of the method call. As a result, `Iterable` or the `ScrollableResults` ignore the timeout. Additionally, `explain()` does not honor this timeout period. This method is used for debugging and to check the reasons for slow performance of a query.

The following is the standard way to limit execution time using the Java Persistence API (JPA):

Example 7.12. 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

The example code does not guarantee that the query stops at the specified results amount.

[Report a bug](#)

7.2. Retrieving the Results

After building the Hibernate Guide, it can be executed in the same way as a HQL or Criteria query. The same paradigm and object semantic apply to Lucene Query query and all the common operations like: `list()`, `uniqueResult()`, `iterate()`, `scroll()` are available.

[Report a bug](#)

7.2.1. 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.

[Report a bug](#)

7.2.2. Result Size

It is sometimes 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)
- ✦ To provide a total search results feature, as provided by Google searches. For example, "1-10 of about 888,000,000 results".
- ✦ To implement fast pagination navigation.
- ✦ to implement a multi-step search engine that adds approximation if the restricted query returns zero 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 7.13. 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 approximation if the index is not fully up-to-date with the database (asynchronous cluster for example).

[Report a bug](#)

7.2.3. ResultTransformer

As seen in [Section 7.1.10.5, “Projection”](#) projection results are returned as **Object** arrays. This data structure is not always matching the application needs. In this case it is possible to apply a **ResultTransformer** which post query execution can build the needed data structure:

Example 7.14. Using ResultTransformer 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.

[Report a bug](#)

7.2.4. 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.

In the second approach you project the **Explanation** object using the **FullTextQuery.EXPLANATION** constant.

Example 7.15. 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. It is recommended that you do not undertake this if another object is unnecessary.

[Report a bug](#)

7.3. 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 (example, view only last month's data)
- ✧ population filter (example, search limited to a given category)
- ✧ and many more

[Report a bug](#)

7.3.1. 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 **IndexManagers** 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 IndexManagers in a array indexed by customerID
    private IndexManager[] indexManagers;

    public void initialize(Properties properties, IndexManager[]
indexManagers) {
        this.indexManagers = indexManagers;
    }

    public IndexManager[] getIndexManagersForAllShards() {
        return indexManagers;
    }
}

```

```

public IndexManager getIndexManagerForAddition(
    Class<?> entity, Serializable id, String idInString, Document
document) {
    Integer customerID =
Integer.parseInt(document.getFieldable("customerID").stringValue());
    return indexManagers[customerID];
}

public IndexManager[] getIndexManagersForDeletion(
    Class<?> entity, Serializable id, String idInString) {
    return getIndexManagersForAllShards();
}

/**
 * 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 IndexManager[] getIndexManagersForQuery(
    FullTextFilterImplementor[] filters) {
    FullTextFilter filter = getCustomerFilter(filters, "customer");
    if (filter == null) {
        return getIndexManagersForAllShards();
    }
    else {
        return new IndexManager[] { indexManagers[Integer.parseInt(
            filter.getParameter("customerID").toString())] };
    }
}

private FullTextFilter getCustomerFilter(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 7.3, “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.

```

@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.

[Report a bug](#)

7.4. Faceting

[Faceted search](#) is a technique which allows to divide the results of a query into multiple categories. This categorization includes the calculation of hit counts for each category and the ability to further restrict search results based on these facets (categories). [Example 7.16, “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 7.16. Search for Hibernate Search on Amazon

In Hibernate Search, the classes **QueryBuilder** and **FullTextQuery** are the entry point into the faceting API. The former creates faceting requests and the latter accesses the **FacetManager**. The **FacetManager** applies faceting requests on a query and selects facets that are added to an existing query to refine search results. The examples use the entity **Cd** as shown in [Example 7.17, “Entity Cd”](#):

Shop All Departments  Search Computers & Internet  Hibernate Search

Books Advanced Search Browse Subjects New Releases Bestsellers TI

Department

- < Any Department
- < Books
- Computers & Internet**
 - Programming (14)
 - Computer Science (4)
 - Databases (2)
 - Software (2)
 - Web Development (2)
 - Networking (1)
 - Home Computing (1)

Format

- ☐ Paperback (15)

Author

- Any Author
- Joe Vitale (1)

Shipping Option [\(What's this?\)](#)

- Any Shipping Option
- Free Super Saver Shipping

Avg. Customer Review

- Any Avg. Customer Review
- ★★★★★ & Up (12)
- ★★★★☆ & Up (14)
- ★★★☆☆ & Up (14)
- ★★☆☆☆ & Up (15)

Condition

- Any Condition
- Used (15)
- New (14)

Books > Computers & Internet > "Hibernate Search"

Showing 1 - 12 of 15 Results

- 

Hibernate Search in Action I

★★★★★ (3 customer reviews)

Formats

Paperback

Order in the next **2 hours** to get it by **Monday, Apr 18.** ~~\$49.95~~

Only 1 left in stock - order soon.

Eligible for **FREE** Super Saver Shipping.

Excerpt - Page 1: "... breaking the sus...
Surprise me! See a random page in thi
- 

Spring Persistence with Hib
(Nov 2, 2010)

★★★★☆ (5 customer reviews)

Formats

Paperback

Order in the next **19 hours** to get it by **Monday, Apr 18.** ~~\$44.95~~

Kindle Edition

Auto-delivered wirelessly

Other Formats: [Paperback](#)

Some formats eligible for **FREE** Super S

Excerpt - Page 11: "... In Chapter 10, y...
resolving these issues. **Hibernate-Search**
Surprise me! See a random page in thi
- 

Lucene in Action, Second Ed
Hatcher and Otis Gospodnetic (

Figure 7.1. Search for Hibernate Search on Amazon

Example 7.17. Entity Cd

```
@Indexed
public class Cd {

    private int id;

    @Fields( {
```

```

        @Field,
        @Field(name = "name_un_analyzed", analyze = Analyze.NO)
    })
    private String name;

    @Field(analyze = Analyze.NO)
    @NumericField
    private int price;

    @Field(analyze = Analyze.NO)
    @DateBridge(resolution = Resolution.YEAR)
    private Date releaseYear;

    @Field(analyze = Analyze.NO)
    private String label;

    // setter/getter
    ...

```

[Report a bug](#)

7.4.1. 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 7.18, “Creating a discrete faceting request”](#):

Example 7.18. 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.



Note

At the moment 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** and **null** values should be avoided. Furthermore the property has to be indexed with **Analyze.NO** and in case of a numeric property **@NumericField** needs to be specified.

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 7.19, “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 7.19. Creating a range faceting request

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

[Report a bug](#)

7.4.2. Applying a Faceting Request

In [Section 7.4.1, “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 7.20, “Applying a faceting request”](#)).

Example 7.20. Applying a faceting request

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

// retrieve facet manager and apply faceting request
FacetManager facetManager = fullTextQuery.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.

[Report a bug](#)

7.4.3. 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 7.21, "Restricting query results via the application of a FacetSelection"](#) shows an example.

Example 7.21. Restricting query results via the application of a FacetSelection

```

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

// retrieve facet manager and apply faceting request
FacetManager facetManager = fullTextQuery.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);

```

[Report a bug](#)

7.5. 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. For details, refer to [Section 4.3, “Reader Strategies”](#)
- caching frequently extracted values from the index: see [Section 7.5.1, “Caching Index Values: FieldCache”](#)

[Report a bug](#)

7.5.1. 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 7.1.10.5, “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 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). Performance may improve by enabling Field Caches 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).

[Report a bug](#)

Chapter 8. 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.

[Report a bug](#)

8.1. 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 8.1. 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 8.3.2, “Using a MassIndexer”](#) for more details.

[Report a bug](#)

8.2. 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 8.2. 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 8.3. 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**.

[Report a bug](#)

8.3. Rebuilding the 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**.

[Report a bug](#)

8.3.1. 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 8.4. 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.default.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 willis out of memory: with a bigger batch size objects are fetched faster from database but more memory is needed.

[Report a bug](#)

8.3.2. 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 8.5. Rebuild the Index Using a `MassIndexer`

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

This will rebuild the index, deleting it and then reloading all entities from the database. Although it is 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! If a query is performed while the `MassIndexer` is working most likely some results will be missing.

Example 8.6. Using a Tuned `MassIndexer`

```
fullTextSession
    .createIndexer( User.class )
    .batchSizeToLoadObjects( 25 )
    .cacheMode( CacheMode.NORMAL )
    .threadsToLoadObjects( 12 )
    .idFetchSize( 150 )
    .progressMonitor( monitor ) //a MassIndexerProgressMonitor
implementation
    .startAndWait();
```

This will rebuild the index of all `User` instances (and subtypes), and will create 12 parallel threads to load the `User` instances using batches of 25 objects per query. These same 12 threads will also need to process indexed embedded relations and custom **FieldBridges** or **ClassBridges** to output a Lucene document. The threads trigger lazyloading of additional attributes during the conversion process. Because of this, a high number of threads working in parallel is required. The number of threads working on actual index writing is defined by the backend configuration of each index. .

Generally we suggest 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 could increase performance if the main entity is relating to enum-like data included in the index.



Note

The ideal of number of threads to achieve best performance is highly dependent on your overall architecture, database design and data values. All internal thread groups have meaningful names so they should be easily identified with most diagnostic tools, including thread dumps.



Note

The `MassIndexer` is unaware of transactions, therefore 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.max_buffered_docs`
- » `hibernate.search.[default|<indexname>].indexwriter.max_merge_docs`
- » `hibernate.search.[default|<indexname>].indexwriter.merge_factor`
- » `hibernate.search.[default|<indexname>].indexwriter.merge_min_size`
- » `hibernate.search.[default|<indexname>].indexwriter.merge_max_size`
- » `hibernate.search.[default|<indexname>].indexwriter.merge_max_optimize_size`
- » `hibernate.search.[default|<indexname>].indexwriter.merge_calibrate_by_deletes`
- » `hibernate.search.[default|<indexname>].indexwriter.ram_buffer_size`
- » `hibernate.search.[default|<indexname>].indexwriter.term_index_interval`

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 5.5.1, "Tuning Lucene Indexing Performance"](#) for more details.

The `MassIndexer` uses a forward only scrollable result to iterate on the primary keys to be loaded, but MySQL's JDBC driver will load all values in memory; to avoid this "optimization" set `idFetchSize` to `Integer.MIN_VALUE`.

[Report a bug](#)

Chapter 9. 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 searches are least frequent.
- ✧ after a large number of index modifications are applied.

When using a **MassIndexer** (see [Section 8.3.2, “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**.

MassIndexer (see [Section 8.3.2, “Using a MassIndexer”](#)) optimizes indexes by default at the start and at the end of processing. Use **MassIndexer.optimizeAfterPurge** and **MassIndexer.optimizeOnFinish** to change this default behavior.

[Report a bug](#)

9.1. Automatic Optimization

Hibernate Search can automatically optimize an index after:

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

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

Example 9.1. 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** has priority over **hibernate.search.default.optimizer.transaction_limit.max**)

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

The default implementation of **OptimizerStrategy** can be overridden by implementing **org.hibernate.search.store.optimization.OptimizerStrategy** and setting the **optimizer.implementation** property to the fully qualified name of your implementation. This implementation must implement the interface, be a public class and have a public constructor taking

no arguments.

Example 9.2. Loading a custom OptimizerStrategy

```
hibernate.search.default.optimizer.implementation =
com.acme.worlddomination.SmartOptimizer
hibernate.search.default.optimizer.SomeOption =
CustomConfigurationValue
hibernate.search.humans.optimizer.implementation = default
```

The keyword **default** can be used to select the Hibernate Search default implementation; all properties after the **.optimizer** key separator will be passed to the implementation's **initialize** method at start.

[Report a bug](#)

9.2. Manual Optimization

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

Example 9.3. 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** and 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.

[Report a bug](#)

9.3. 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.max_buffered_docs**

- » `hibernate.search.[default]<indexname>.indexwriter.max_merge_docs`
- » `hibernate.search.[default]<indexname>.indexwriter.merge_factor`
- » `hibernate.search.[default]<indexname>.indexwriter.ram_buffer_size`
- » `hibernate.search.[default]<indexname>.indexwriter.term_index_interval`

See [Section 5.5.1, “Tuning Lucene Indexing Performance”](#) for details.

[Report a bug](#)

Chapter 10. 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.

[Report a bug](#)

10.1. JMX

10.1.1. About 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.



Note

JMX beans can be accessed remotely via JConsole by setting the system property **com.sun.management.jmxremote** to **true**.

[Report a bug](#)

10.1.2. StatisticsInfoMBean

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

[Report a bug](#)

10.1.3. IndexControlMBean

This MBean allows to build, optimize and purge the index for a given entity. Indexing occurs via the mass indexing API (see [Section 8.3.2, “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.

[Report a bug](#)

10.1.4. 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.

[Report a bug](#)

Chapter 11. 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.

[Report a bug](#)

11.1. 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 11.1. Accessing the SearchFactory

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

[Report a bug](#)

11.2. Using an IndexReader

Queries in Lucene are executed on an **IndexReader**. Hibernate Search might cache index readers to maximize performance, or provide other efficient strategies to retrieve an updated **IndexReader** minimizing IO operations. Your code can access these cached resources, but you have to follow some "good citizen" rules.

Example 11.2. Accessing an IndexReader

```
IndexReader reader =
searchFactory.getIndexReaderAccessor().open(Order.class);
try {
    //perform read-only operations on the reader
}
finally {
    searchFactory.getIndexReaderAccessor().close(reader);
}
```

In this example the **SearchFactory** figures out which indexes are needed to query this entity (considering a Sharding strategy). Using the configured **ReaderProvider** (described in [Section 4.3, "Reader Strategies"](#)) on each index, it returns a compound **IndexReader** on top of all involved indexes. Because this **IndexReader** is shared amongst several clients, you must adhere to the following rules:

- ✦ Never call `indexReader.close()`, instead use `readerProvider.closeReader(reader)` when necessary, preferably in a finally block.
- ✦ Don't use this **IndexReader** for modification operations (it's a readonly **IndexReader**, you would get an exception).

Aside from those rules, you can use the **IndexReader** freely, especially to do native Lucene queries. Using the shared **IndexReaders** will make most queries more efficient than by opening one directly from - for example - the filesystem.

As an alternative to the method **open(Class... types)** you can use **open(String... indexNames)**; in this case you pass in one or more index names; using this strategy you can also select a subset of the indexes for any indexed type if sharding is used.

Example 11.3. Accessing an IndexReader by index names

```
IndexReader reader =
searchFactory.getIndexReaderAccessor().open("Products.1",
"Products.3");
```

[Report a bug](#)

11.3. Accessing a Lucene Directory

A **Directory** is the most common abstraction used by Lucene to represent the index storage; Hibernate Search doesn't interact directly with a Lucene **Directory** but abstracts these interactions via an **IndexManager**: an index does not necessarily need to be implemented by a **Directory**.

If you know your index is represented as a **Directory** and need to access it, you can get a reference to the **Directory** via the **IndexManager**. Cast the **IndexManager** to a **DirectoryBasedIndexManager** and then use **getDirectoryProvider().getDirectory()** to get a reference to the underlying **Directory**. This is not recommended, we would encourage to use the **IndexReader** instead.

[Report a bug](#)

11.4. 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 11.4, "Enabling Index Sharding"](#). In this example 5 shards are enabled.

Example 11.4. 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 11.5. 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 7.3.1, “Using Filters in a Sharded Environment”](#)), a sharding strategy can select a subset of the shards used to answer a query (**IndexShardingStrategy.getIndexManagersForQuery**) and thus speed up the query execution.

Each shard has an independent **IndexManager** and so can be configured to use a different directory provider and back end configurations. The **IndexManager** index names for the **Animal** entity in [Example 11.6, “Sharding Configuration for Entity Animal”](#) are **Animal.0** to **Animal.4**. In other words, each shard has the name of its owning index followed by **.** (dot) and its index number (see also [Section 5.3, “Directory Configuration”](#)).

Example 11.6. 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 11.6, “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/sharded/Animal03** (overridden indexBase, overridden indexName)
- ✧ for sub-index 4: **/usr/lucene/indexes/Animal.4** (shared indexBase, default indexName)

When implementing a **IndexShardingStrategy** any field can be used to determine the sharding selection. Consider that to handle deletions, **purge** and **purgeAll** operations, the implementation might need to return one or more indexes without being able to read all the field values or the primary identifier; in case the information is not enough to pick a single index, all indexes should be returned, so that the delete operation will be propagated to all indexes potentially containing the documents to be deleted.

[Report a bug](#)

11.5. 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.

[Report a bug](#)

11.6. About Using External Services

Any of the pluggable contracts we have seen so far allows for the injection of a service. The most notable example being the **DirectoryProvider**. The full list is:

- ✧ **DirectoryProvider**
- ✧ **ReaderProvider**
- ✧ **OptimizerStrategy**
- ✧ **BackendQueueProcessor**
- ✧ **Worker**
- ✧ **ErrorHandler**

» `MassIndexerProgressMonitor`

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 contract.

[Report a bug](#)

11.6.1. 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.

[Report a bug](#)

11.6.1.1. 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 11.7. 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 11.8. Content of `META-INF/services/org.hibernate.search.spi.ServiceProvider`

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

[Report a bug](#)

11.6.1.2. Provided Services

Alternatively, the service can be provided by the environment bootstrapping 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 a service instance must be retrieved from the environment, use registry services like JNDI and look the service up in the provider.

[Report a bug](#)

11.6.2. Using a Service

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

Example 11.9. 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.

[Report a bug](#)

11.7. Customizing Lucene's Scoring Formula

Customize Lucene's scoring formula 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*:

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

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

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

Hibernate Search provides 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.

[Report a bug](#)

Revision History

Revision 2.7.0-3	Fri Feb 06 2015	Michelle Murray
WFKDOC-124: System requirements updated		
Revision 2.7.0-2	Tues Jan 06 2015	Michelle Murray
Generated for WFK 2.7 release		