Hibernate Search

Apache Lucene™ Integration

Reference Guide

Emmanuel Bernard
Hardy Ferentschik
Gustavo Fernandes
Sanne Grinovero
Nabeel Ali Memon
Gunnar Morling

Hibernate Search: Apache Lucene™ Integration: Reference Guide

by Emmanuel Bernard, Hardy Ferentschik, Gustavo Fernandes, Sanne Grinovero, Nabeel Ali Memon, and Gunnar Morling

5.3.0.Final

Preface

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

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

Chapter 1. Getting started

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

1.1. System Requirements

Table 1.1. System requirements

Java Runtime	Requires Java version *7* or greater. You can download a Java Runtime for Windows/Linux/Solaris here [http://www.oracle.com/technetwork/java/javase/downloads/index.html].
Hibernate Search	`hibernate-search-5.3.0.Final.jar` and all runtime dependencies. You can get the jar artifacts either from the `dist/lib` directory of the Hibernate Search distribution [http://sourceforge.net/projects/hibernate/files/hibernate-search/] or you can download them from the JBoss maven repository [http://repository.jboss.org/nexus/content/groups/public-jboss/org/hibernate/].
Hibernate Core	You will need `hibernate-core-4.3.10.Final.jar` and its dependencies (either from the distribution bundle [http://sourceforge.net/projects/hibernate/files/hibernate4/] or the maven repository).
JPA 2.1	Hibernate Search can be used without JPA but the following instructions will use JPA annotations for basic entity configuration (`@Entity`, `@Id`, `@OneToMany`,…).

1.2. Migration notes

If you are upgrading an existing application from an earlier version of Hibernate Search to the latest release, make sure to check the out the migration guide [http://hibernate.org/search/documentation/migrate/5.0/].

1.3. Required libraries

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

as the Java Persistence API and the Java Transactions API. Other dependencies are optional, providing additional integration points. To get the correct jar files on your classpath we highly recommend to use a dependency manager such as Maven [http://maven.apache.org/], or similar tools such as Gradle [http://www.gradle.org/] or Ivy [http://ant.apache.org/ivy/]. These alternatives are also able to consume the artifacts from the Section 1.3.1, "Using Maven" section.

1.3.1. Using Maven

The Hibernate Search artifacts can be found in Maven's Central Repository [http://central.sonatype.org/] but are released first in the JBoss Maven Repository [http://repository.jboss.org/nexus/content/groups/public-jboss/]. See also the Maven Getting Started wiki page [https://community.jboss.org/wiki/MavenGettingStarted-Users] to use the JBoss repository.

All you have to add to your pom.xml is:

Example 1.1. Maven artifact identifier for Hibernate Search

```xml
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-search-orm</artifactId>
   <version>5.3.0.Final</version>
</dependency>
```

Example 1.2. Optional Maven dependencies for Hibernate Search

```xml
<!-- If using JPA, add: -->
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-entitymanager</artifactId>
   <version>4.3.10.Final</version>
</dependency>
<!-- Infinispan integration: -->
<dependency>
   <groupId>org.infinispan</groupId>
   <artifactId>infinispan-directory-provider</artifactId>
   <version>7.2.0.Final</version>
</dependency>
```

Only the *hibernate-search-orm* dependency is mandatory. *hibernate-entitymanager* is only required if you want to use Hibernate Search in conjunction with JPA.

1.3.2. Manual library management

You can download zip bundles from Sourceforge containing all needed Hibernate Search [http://sourceforge.net/projects/hibernate/files/hibernate-search/5.3.0.Final/] dependencies. This includes - among others - the latest compatible version of Hibernate ORM. However, only the essential parts you need to start experimenting with are included. You will probably need to com-

bine this with downloads from the other projects, for example the Hibernate ORM distribution on Sourceforge [http://sourceforge.net/projects/hibernate/files/hibernate4/4.3.10.Final/] also provides the modules to enable caching or use a connection pool.

1.4. Deploying on WildFly

If you are creating an application to be deployed on WildFly you're lucky: Hibernate Search is included in the application server. This means that you don't need to package it along with your application but remember that you need to activate the module, see Section 3.9, "Hibernate Search as a WildFly module" for details.

Due to he modular design of WildFly, you can also bundle a more recent version of Hibernate Search than the one included in the popular application server. This is also explained further in Section 3.9, "Hibernate Search as a WildFly module".

1.5. Configuration

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

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

```
...
<property name="hibernate.search.default.directory_provider"
          value="filesystem"/>

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

First you have to tell Hibernate Search which `DirectoryProvider` to use. This can be achieved by setting the `hibernate.search.default.directory_provider` property. Apache Lucene has the notion of a `Directory` to store the index files. Hibernate Search handles the initialization and configuration of a Lucene `Directory` instance via a `DirectoryProvider`. In this tutorial we will use a a directory provider which stores the index on the file system. This will give us the ability to inspect the Lucene indexes created by Hibernate Search (eg via Luke [https://github.com/DmitryKey/luke/]). Once you have a working configuration you can start experimenting with other directory providers (see Section 3.3, "Directory configuration"). You also have to specify the default base directory for all indexes via `hibernate.search.default.indexBase`. This defines the path where indexes are stored.

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

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

```java
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
  ...
}
```

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

  @Id
  @GeneratedValue
  private Integer id;

  private String name;

  public Author() {}

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

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 *untok-*

enized id in the index to ensure index uniqueness for a given entity (for now don't worry if you don't know what *untokenized* means, it will soon be clear).

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 or tokenizing means chunking a sentence into individual words and potentially excluding common words like "a" or "the". We will talk more about analyzers a little later on. The third parameter we specify is `store=Store.NO`, which ensures that the actual data will not be stored in the index. Whether data is stored in the index or not has nothing to do with the ability to search for it. It is not necessary to store fields in the index to allow Lucene to search for them: the benefit of storing them is the ability to retrieve them via projections (see Section 5.1.3.5, "Projection").

Without projections, Hibernate Search will per default execute a Lucene query in order to find the database identifiers of the entities matching the query criteria and use these identifiers to retrieve managed objects from the database. The decision for or against projection has to be made on a case by case basis.

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 mostly string based, with special support for encoding numbers. Hibernate Search must convert the data types of the indexed fields to their respective Lucene encoding and vice versa. A range of predefined bridges is provided for this purpose, including the `DateBridge` which will convert a `java.util.Date` into a numeric value (a `long`) with the specified resolution. For more details see Section 4.4.1, "Built-in 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 author names will be searchable you have to make sure that the names are indexed as part of the book itself. On top of `@IndexedEmbedded` you will also have to mark all fields of the associated entity you want to have included in the index with `@Indexed`. For more details see Section 4.1.3, "Embedded and associated objects".

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

Example 1.5. Example entities after adding Hibernate Search annotations

```java
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
  ...
}
```

```
@Entity
public class Author {

  @Id
  @GeneratedValue
  private Integer id;

  @Field
  private String name;

  public Author() {
  }

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

1.6. Indexing

Hibernate Search will transparently index every entity persisted, updated or removed through Hibernate ORM. However, you have to create an initial Lucene index for the data already present in your database. Once you have added the above properties and annotations it is time to trigger an initial batch index of your books. You can achieve this by using one of the following code snippets (see also Section 6.3, "Rebuilding the whole index"):

Example 1.6. Using Hibernate Session to index data

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

Example 1.7. Using JPA to index data

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

After executing the above code, you should be able to see a Lucene index under `/var/lucene/indexes/example.Book` (or based on a different path depending how you configured the property `hibernate.search.default.directory_provider`).

Go ahead an inspect this index with Luke [https://github.com/DmitryKey/luke/]: it will help you to understand how Hibernate Search works.

1.7. Searching

Now it is time to execute a first search. The general approach is to create a Lucene query, either via the Lucene API (Section 5.1.1, "Building a Lucene query using the Lucene API") or via the Hibernate Search query DSL (Section 5.1.2, "Building a Lucene query with the Hibernate Search query DSL"), and then wrap this query into a `org.hibernate.Query` in order to get all the functionality one is used to from the Hibernate API. The following code will prepare a query against the indexed fields, execute it and return a list of `Book` instances.

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

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

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

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

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

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

Example 1.9. Using JPA to create and execute a search

```java
EntityManager em = entityManagerFactory.createEntityManager();
FullTextEntityManager fullTextEntityManager =
    org.hibernate.search.jpa.Search.getFullTextEntityManager(em);
em.getTransaction().begin();

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

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

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

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

1.8. Analyzer

Let's make things a little more interesting now. Assume that one of your indexed book entities has the title "Refactoring: Improving the Design of Existing Code" and you want to get hits for all of the following queries: "refactor", "refactors", "refactored" and "refactoring". In Lucene this can be achieved by choosing an analyzer class which applies word stemming during the indexing **as well as** the search process. Hibernate Search offers several ways to configure the analyzer to be used (see Section 4.3.1, "Default analyzer and analyzer by class"):

- Setting the `hibernate.search.analyzer` property in the configuration file. The specified class will then be the default analyzer.

- Setting the `@Analyzer` annotation at the entity level.

- Setting the `@Analyzer` annotation at the field level.

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

To find out more about the factory classes available you can either browse the Lucene JavaDoc or read the corresponding section on the Solr Wiki [http://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters].

You can use `@AnalyzerDef` or `@AnalyzerDefs` on any: *`@Indexed` entity regardless of where the analyzer is applied to; * parent class of an `@Indexed` entity; * package-info.java of a package containing an `@Indexed` entity.

This implies that analyzer definitions are global and their names must be unique.

> **Note**
>
> Why the reference to the Apache Solr wiki?
>
> Apache Solr was historically an indepedent sister project of Apache Lucene and the analyzer factory framework was originally created in Solr. Since then the Apache Lucene and Solr projects have merged, but the documentation for these additional analyzers can still be found in the Solr Wiki. You might find other documentation referring to the "Solr Analyzer Framework" - just remember you don't need to depend on Apache Solr anymore to use it. The required classes are part of the core Lucene distribution.

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

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

Example 1.10. Using `@AnalyzerDef` and the Analyzer Framework to define and use an analyzer

```
@Entity
@Indexed
@AnalyzerDef(name = "customanalyzer",
  tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
  filters = {
    @TokenFilterDef(factory = LowerCaseFilterFactory.class),
    @TokenFilterDef(factory = SnowballPorterFilterFactory.class, params = {
      @Parameter(name = "language", value = "English")
    })
```

```
  })
public class Book {

  @Id
  @GeneratedValue
  @DocumentId
  private Integer id;

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

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

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

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

  public Book() {
  }

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

Using @AnalyzerDef only defines an Analyzer, you still have to apply it to entities and or properties using @Analyzer. Like in the above example the customanalyzer is defined but not applied on the entity: it's applied on the title and subtitle properties only. An analyzer definition is global, so you can define it on any entity and reuse the definition on other entities.

1.9. What's next

The above paragraphs helped you getting an overview of Hibernate Search. The next step after this tutorial is to get more familiar with the overall architecture of Hibernate Search (Chapter 2, *Architecture*) and explore the basic features in more detail. Two topics which were only briefly touched in this tutorial were analyzer configuration (Section 4.3.1, "Default analyzer and analyzer by class") and field bridges (Section 4.4, "Bridges"). Both are important features required for more fine-grained indexing. More advanced topics cover clustering (Section 3.4.1, "JMS Master/Slave back end", Section 3.3.1, "Infinispan Directory configuration") and large index handling (Section 10.5, "Sharding indexes").

Chapter 2. Architecture

2.1. 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 Section 3.1, "Enabling Hibernate Search and automatic indexing"). 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 10.5, "Sharding indexes"). The latter, index sharing, is the ability to index multiple entities into the same Lucene index (see Section 10.6, "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 3.3, "Directory configuration"). You can, however, also provide your own IndexManager implementation (see Section 3.2, "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 3.4, "Worker configuration").

Tip

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

Note

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

2.2. Back end

Hibernate Search offers the ability to let the batched work being processed by different back ends. Several back ends are provided out of the box and you have the option to plugin your own. It is important to understand that in this context back end encompasses more than just the configuration option `hibernate.search.default.worker.backend`. This property just specifies a implementation of the BackendQueueProcessor interface which is a part of a back end configuration. In most cases, however, additional configuration settings are needed to successfully configure a specific backend setup, like for example the JMS back end.

2.2.1. Lucene

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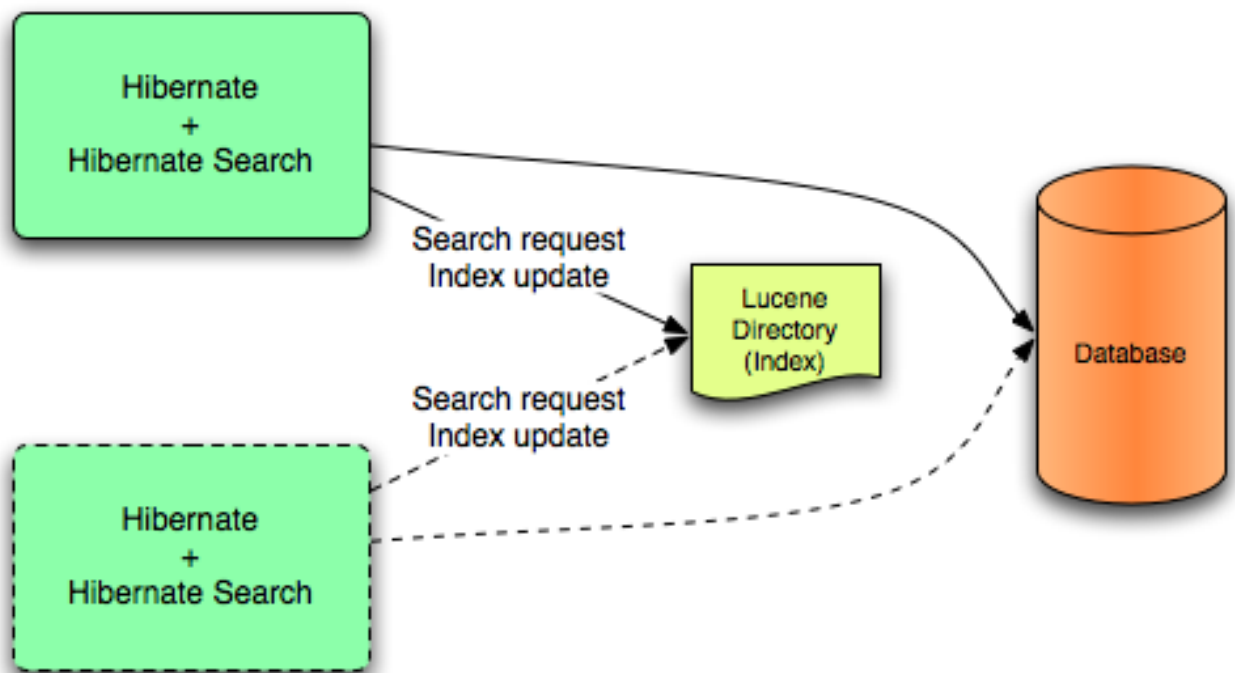
This mode targets non clustered applications, or clustered applications where the Directory is taking care of the locking strategy.

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

An alternative back end viable for non-clustered and non-shared index configurations is the near-real-time backend.

2.2.2. JMS

All index update operations applied on a given node are sent to a JMS queue. A unique reader will then process the queue and update the master index. The master index is then replicated on a regular basis to the slave copies. This is known as the master/slaves pattern. The master is the sole responsible for updating the Lucene index. The slaves can accept read as well as write operations. However, they only process the read operation on their local index copy and delegate the update operations to the master.
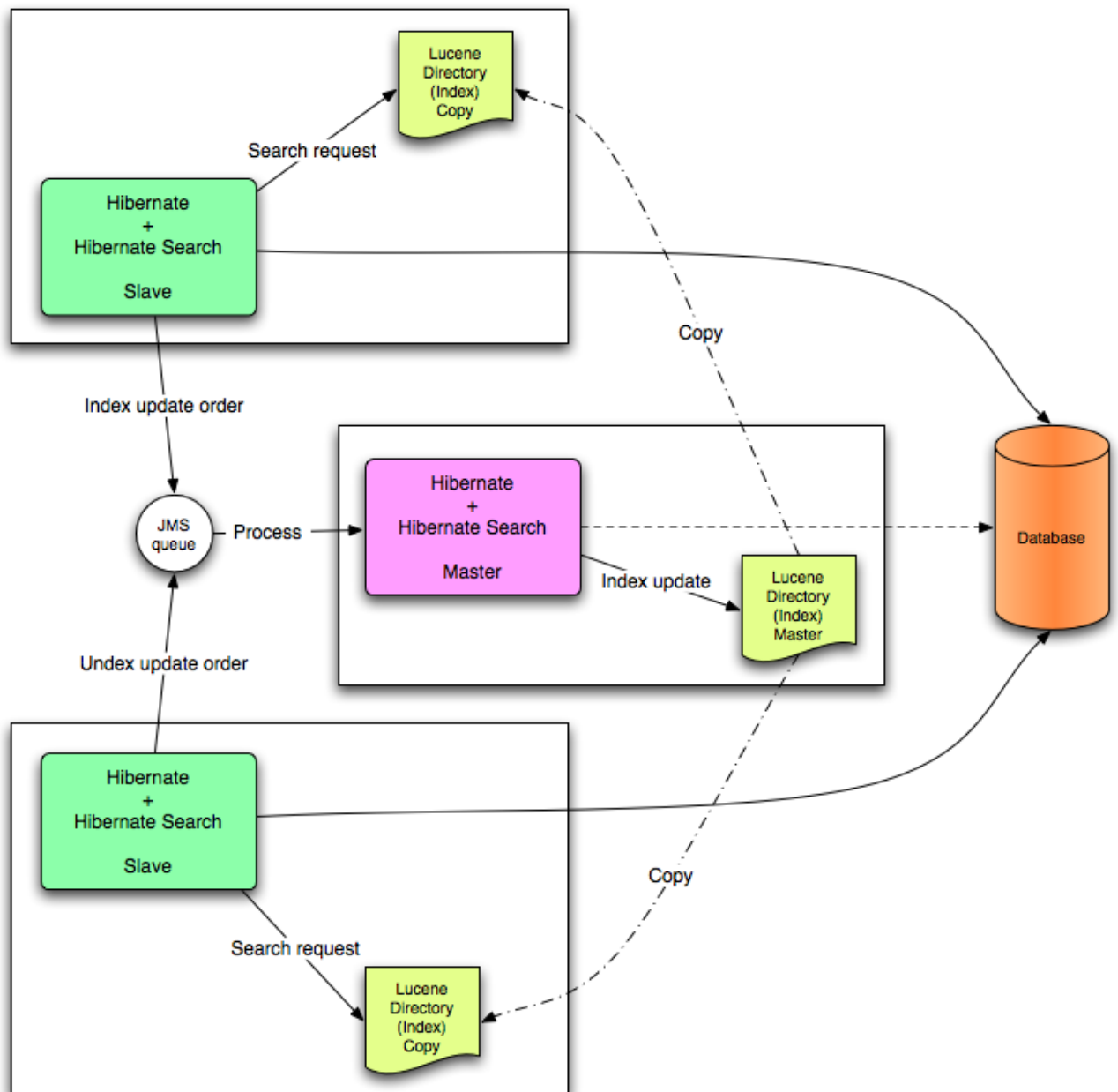
This mode targets clustered environments where throughput is critical, and index update delays are affordable. Reliability is ensured by the JMS provider and by having the slaves working on a local copy of the index.

2.2.3. JGroups

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

Note that while JMS can usually be configured to use persistent queues, JGroups talks directly to other nodes over network. Message delivery to other reachable nodes is guaranteed, but if no master node is available, index operations are silently discarded. This backend can be configured to use asynchronous messages, or to wait for each indexing operation to be completed on the remote node before returning.

The JGroups backend can be configured with static master or slave roles, or can be setup to perform an auto-election of the master. This mode is particularly useful to have the system automatically pick a new master in case of failure, but during a reelection process some indexing operations might be lost. For this reason this mode is not suited for use cases requiring strong consistency guarantees. When configured to perform an automatic election, the master node is defined as an hash on the index name: the role is therefore possibly different for each index or shard.

2.3. Reader strategy

When executing a query, Hibernate Search interacts with the Apache Lucene indexes through a reader strategy. Choosing a reader strategy will depend on the profile of the application (frequent updates, read mostly, asynchronous index update etc). See also Section 3.5, "Reader strategy configuration"

2.3.1. shared

With this strategy, Hibernate Search will share the same IndexReader, for a given Lucene index, across multiple queries and threads provided that the IndexReader is still up-to-date. If the IndexReader is not up-to-date, a new one is opened and provided. Each IndexReader is made of several SegmentReaders. This strategy only reopens segments that have been modified or created after last opening and shares the already loaded segments from the previous instance. This strategy is the default.

The name of this strategy is `shared`.

2.3.2. not-shared

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

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

2.3.3. Custom

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

Chapter 3. Configuration

3.1. Enabling Hibernate Search and automatic indexing

Let's start with the most basic configuration question - how do I enable Hibernate Search?

3.1.1. Enabling Hibernate Search

The good news is that Hibernate Search is enabled out of the box when detected on the classpath by Hibernate Core. If, for some reason you need to disable it, set `hibernate.search.autoregister_listeners` to false. Note that there is no performance penalty when the listeners are enabled but no entities are annotated as indexed.

3.1.2. Automatic indexing

By default, every time an object is inserted, updated or deleted through Hibernate, Hibernate Search updates the according Lucene index. It is sometimes desirable to disable that features if either your index is read-only or if index updates are done in a batch way (see Section 6.3, "Rebuilding the whole index").

To disable event based indexing, set

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

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

3.2. Configuring the IndexManager

The role of the index manager component is described in Chapter 2, *Architecture*. Hibernate Search provides two possible implementations for this interface to choose from.

- `directory-based`: the default implementation which uses the Lucene Directory abstraction to manage index files.

- `near-real-time`: avoid flushing writes to disk at each commit. This index manager is also Directory based, but also makes uses of Lucene's NRT functionality.

To select an alternative you specify the property:

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

3.2.1. directory-based

The default IndexManager implementation. This is the one mostly referred to in this documentation. It is highly configurable and allows you to select different settings for the reader strategy, back ends and directory providers. Refer to Section 3.3, "Directory configuration", Section 3.4, "Worker configuration" and Section 3.5, "Reader strategy configuration" for more details.

3.2.2. near-real-time

The NRTIndexManager is an extension of the default IndexManager, leveraging the Lucene NRT (Near Real Time) features for extreme low latency index writes. As a trade-off it requires a non-clustered and non-shared index. In other words, it will ignore configuration settings for alternative back ends other than `lucene` and will acquire exclusive write locks on the Directory.

To achieve this low latency writes, the IndexWriter will not flush every change to disk. Queries will be allowed to read updated state from the unflushed index writer buffers; the downside of this strategy is that if the application crashes or the IndexWriter is otherwise killed you'll have to rebuild the indexes as some updates might be lost.

Because of these downsides, and because a master node in cluster can be configured for good performance as well, the NRT configuration is only recommended for non clustered websites with a limited amount of data.

3.2.3. Custom

It is also possible to configure a custom IndexManager implementation by specifying the fully qualified class name of your custom implementation. This implementation must have a no-argument constructor:

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

> **Tip**
>
> Your custom index manager implementation doesn't need to use the same components as the default implementations. For example, you can delegate to a remote indexing service which doesn't expose a Directory interface.

3.3. Directory configuration

As we have seen in Section 3.2, "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. Di-

rectoryProvider is the Hibernate Search abstraction around a Lucene Directory and handles the configuration and the initialization of the underlying Lucene resources. Table 3.1, "List of built-in 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 10.6, "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 3.2, "Configuring directory providers" shows how `hibernate.search.default.directory_provider` is used to set the default directory provider to be the filesystem one. `hibernate.search.default.indexBase` sets then the default base directory for the indexes. As a result the index for the entity Status is created in `/usr/lucene/indexes/org.hibernate.example.Status`.

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

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

Example 3.1. Specifying the index name

```java
package org.hibernate.example;

@Indexed
public class Status { ... }

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

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

Example 3.2. Configuring directory providers

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

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

Table 3.1. List of built-in DirectoryProvider

Name and description	Properties
ram: Memory based directory. The directory will be uniquely identified (in the same deployment unit) by the `@Indexed.index` element	none
filesystem: File system based directory. The directory used will be <indexBase>/<indexName>	`indexBase` : base directory `indexName`: override @Indexed.index (useful for sharded indexes) `locking_strategy` : optional, see Section 3.7.2, "LockFactory configuration" `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.
filesystem-master: File system based directory. Like `filesystem`. It also copies the index to a source directory (aka copy directory) on a regular basis. The recommended value for the refresh period is (at least) 50% higher that the time to copy the information (default 3600 seconds - 60 minutes).	`indexBase`: base directory `indexName`: override @Indexed.index (useful for sharded indexes) `sourceBase`: source (copy) base directory. `source`: source directory suffix (default to `@Indexed.index`). The actual source directory name being *<sourceBase>/<source>* `refresh`: refresh period in 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. `buffer_size_on_copy`: The

Name and description	Properties
Note that the copy is based on an incremental copy mechanism reducing the average copy time. DirectoryProvider typically used on the master node in a JMS back end cluster. 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.	amount of MegaBytes to move in a single low level copy instruction; defaults to 16MB. `locking_strategy` : optional, see Section 3.7.2, "LockFactory configuration" `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.
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. The recommended value for the refresh period is (at least) 50% higher that the time to copy the information (default 3600 seconds - 60 minutes). Note that the copy is based on an incremental copy mechanism reducing the average copy time. If a copy is still in progress when refresh period elapses, the second copy operation will be skipped. DirectoryProvider typically used on slave nodes using a JMS back end. The `buffer_size_on_copy` optimum depends on your operating system and available RAM; most people reported good results using values between 16 and 64MB.	`indexBase`: Base directory `indexName`: override @Indexed.index (useful for sharded indexes) `sourceBase`: Source (copy) base directory. `source`: Source directory suffix (default to `@Indexed.index`). The actual source directory name being *<sourceBase>/<source>* `refresh`: refresh period in second (the copy will take place every refresh seconds). `buffer_size_on_copy`: The amount of MegaBytes to move in a single low level copy instruction; defaults to 16MB. `locking_strategy` : optional, see Section 3.7.2, "LockFactory configuration" `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. `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: full-text queries performed before the index is initialized are not blocked but will return empty results. When not enabling the option or explicitly setting

Name and description	Properties
	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 time-out, see `retry_marker_lookup` instead. `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.
infinispan: Infinispan based directory. Use it to store the index in a distributed grid, making index changes visible to all elements of the cluster very quickly. Also see Section 3.3.1, "Infinispan Directory configuration" for additional requirements and configuration settings. Infinispan needs a global configuration and additional dependencies; the settings defined here apply to each different index.	`locking_cachename`: name of the Infinispan cache to use to store locks. ` `data_cachename` : name of the Infinispan cache to use to store the largest data chunks; this area will contain the largest objects, use replication if you have enough memory or switch to distribution. `metadata_cachename`: name of the Infinispan cache to use to store the metadata relating to the index; this data is rather small and read very often, it's recommended to have this cache setup using replication. `chunk_size`: large files of the index are split in smaller chunks, you might want to set the highest value efficiently handled by your network. Networking tuning might be useful.

> **Tip**
>
> If the built-in directory providers do not fit your needs, you can write your own directory provider by implementing the org.hibernate.store.DirectoryProvider interface. In this case, pass the fully qualified class name of your provider into the

`directory_provider` property. You can pass any additional properties using the prefix `hibernate.search.<indexname>`.

3.3.1. Infinispan Directory configuration

Infinispan is a distributed, scalable, cloud friendly data grid platform, which Hibernate Search can use to store the Lucene index. Your application can benefits in this case from Infinispan's distribution capabilities making index updates available on all nodes with short latency.

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

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

The default configuration replicates all data which defines the index across all nodes, thus consuming a significant amount of memory but providing the best query performance. For large indexes it's suggested to enable data distribution, so that each piece of information is replicated to a subset of all cluster members. The distribution option will reduce the amount of memory required for each node but is less efficient as it will cause high network usage among the nodes.

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

A popular choice is to use a replicated index aiming to keep the whole index in memory, combined with a `CacheStore` as safety valve in case the index gets larger than expected.

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

3.3.1.1. Requirements

To use the Infinispan directory via Maven, add the following dependencies:

Example 3.3. Maven dependencies for Hibernate Search using Infinispan

```xml
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-search-orm</artifactId>
    <version>5.3.0.Final</version>
</dependency>
<dependency>
    <groupId>org.infinispan</groupId>
    <artifactId>infinispan-directory-provider</artifactId>
    <version>7.2.0.Final</version>
</dependency>
```

> **Important**
>
> This dependency changed in Hibernate Search version 5.2.
>
> Previously the DirectoryProvider was provided by the Hibernate Search project and had Maven coordinates 'org.hibernate:hibernate-search-infinispan', but the Infinispan team is now maintaining this extension point so since this version please use the Maven definition as in the previous example.
>
> The version printed above was the latest compatible at the time of publishing this Hibernate Search version: it's possible that more recently improved versions of Infinispan have been published which are compatible with this same Hibernate Search version.

3.3.1.2. Architecture

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

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

To configure a JMS slave only the backend must be replaced, the directory provider must be set to `infinispan`; set the same directory provider on the master, they will connect without the need to setup the copy job across nodes. Using the JGroups backend is very similar - just combine the backend configuration with the `infinispan` directory provider.

3.3.1.3. Infinispan Configuration

The most simple configuration only requires to enable the backend:

```
hibernate.search.[default|<indexname>].directory_provider = infinispan
```

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

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

To use and existing CacheManager via JNDI (optional parameter):

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

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

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

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

As mentioned in Table 3.1, "List of built-in DirectoryProvider", each index makes use of three caches, so three different caches should be configured as shown in the `default-hibernate-search-infinispan.xml` provided in the `infinispan-directory-provider.jar`. Several indexes can share the same caches.

Infinispan relies on JGroups for its networking functionality, so unless you are using Infinispan on a single node, an Infinispan configuration file will refer to a JGroups configuration file. This coupling is not always practical and we provide a property to override the used JGroups configuration file:

```
hibernate.search.infinispan.configuration.transport_override_resourcename =
  jgroups-ec2.xml
```

This allows to just switch the JGroups configuration while keeping the rest of the Infinispan configuration.

The file `jgroups-ec2.xml` used in the example above is one of the several JGroups configurations included in Infinispan. It is a good starting point to run on Amazon EC2 networks. For more details and examples see usage of pre-configured JGroups stacks [http://infinispan.org/docs/7.2.x/user_guide/user_guide.html#_use_one_of_the_pre_configured_jgroups_files] in the Infinispan configuration guide.

3.4. 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 3.2, "Scope configuration".

Table 3.2. Scope configuration

Property	Description

hibernate.search.default.worker.scope	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.
hibernate.search.default.worker.*	All configuration properties prefixed with `hibernate.search.default.worker` are passed to the Worker during initialization. This allows adding custom, worker specific parameters.

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

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

Table 3.3. Execution configuration

Property	Description
hibernate.search.<indexName>. worker.execution	`sync`: synchronous execution (default) `async`: asynchronous execution

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 3.4, "Backend configuration". Again this option can be configured differently for each index.

Table 3.4. Backend configuration

Property	Description
hibernate.search.<indexName>. worker.backend	`lucene`: The default backend which runs index updates in the same VM. Also used when the property is undefined or empty. `jms`: JMS backend. Index updates are send to a JMS queue to be processed by an indexing

master. See Table 3.5, "JMS backend con-figuration" for additional configuration options and Section 3.4.1, "JMS Master/Slave back end" for a more detailed description of this setup.

`jgroupsMaster`, `jgroupsSlave` or `jgroups`: Backend using JGroups [http://www.jgroups.org/] as communication layer. See Section 3.4.2, "JGroups Master/Slave back end" for a more detailed description of this setup.

`blackhole`: Mainly a test/developer setting which ignores all indexing work

You can also specify the fully qualified name of a class implementing BackendQueue-Processor. 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.

Table 3.5. JMS backend configuration

Property	Description
hibernate.search.<indexName>.worker.jndi.*	Defines the JNDI properties to initiate the Ini-tialContext (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 fac-tory from (`/ConnectionFactory` by default in JBoss AS)
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 mes-sages.
hibernate.search.<indexName>.worker.jms.login	Optional for the JMS slaves. Use it when your queue requires login credentials to define your login.
hibernate.search.<indexName>.worker.jms.login	Optional for the JMS slaves. Use it when your queue requires login credentials to define your password.

> **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.

3.4.1. JMS Master/Slave back end

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

JMS back end configuration.

3.4.1.1. Slave nodes

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

Example 3.4. 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
#optionally authentication credentials:
hibernate.search.default.worker.jms.login = myname
hibernate.search.default.worker.jms.password = wonttellyou
```

```
#optional jndi configuration (check your JMS provider for more information)

## Optional asynchronous execution strategy
# hibernate.search.default.worker.execution = async
```

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

3.4.1.2. Master node

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

Example 3.5. JMS Master configuration

```
### master configuration

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

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

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

# appropriate directory provider
hibernate.search.default.directory_provider = filesystem-master

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

> **Tip**
>
> It is recommended that the refresh period be higher than the expected copy time; if a copy operation is still being performed when the next refresh triggers, the second refresh is skipped: it's safe to set this value low even when the copy time is not known.

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

Example 3.6. Message Driven Bean processing the indexing queue

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

    @PersistenceContext EntityManager em;

    @Override
    protected SearchIntegrator getSearchIntegrator() {
        FullTextEntityManager fullTextEntityManager = Search.getFullTextEntityManager(em);
        return fullTextEntityManager.getSearchFactory().unwrap(SearchIntegrator.class);
    }
}
```

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. Essentially what you need to do is to connect the specific JMS Queue with the `SearchFactory` instance of the EntityManager. As an advanced alternative, you can implement your own logic by not extending `AbstractJMSHibernateSearchController` but rather to use it as an implementation example.

3.4.2. JGroups Master/Slave back end

This section describes how to configure the JGroups Master/Slave back end. The master and slave roles are similar to what is illustrated in Section 3.4.1, "JMS Master/Slave back end", only a different backend (hibernate.search.default.worker.backend) needs to be set.

A specific backend can be configured to act either as a slave using `jgroupsSlave`, as a master using `jgroupsMaster`, or can automatically switch between the roles as needed by using `jgroups`.

> **Note**
>
> Either you specify a single `jgroupsMaster` and a set of `jgroupsSlave` instances, or you specify all instances as `jgroups`. Never mix the two approaches!

All backends configured to use JGroups share the same channel. The JGroups JChannel is the main communication link across all nodes participating in the same cluster group; since it is convenient to have just one channel shared across all backends, the Channel configuration properties are not defined on a per-worker section but are defined globally. See Section 3.4.2.4, "JGroups channel configuration".

Table Table 3.6, "JGroups backend configuration properties" contains all configuration options which can be set independently on each index backend. These apply to all three variants of the backend: `jgroupsSlave, jgroupsMaster, jgroups`. It is very unlikely that you need to change any of these from their defaults.

Table 3.6. JGroups backend configuration properties

Property	Description
hibernate.search.<indexName>.jgroups.block_waiting_ack	Set to either `true` or `false`. False is more efficient but will not wait for the operation to be delivered to the peers. Defaults to `true` when the backend is synchronous, to `false` when the backend is `async`.
hibernate.search.<indexName>.jgroups. messages_timeout	The timeout of waiting for a single command to be acknowledged and executed when `block_waiting_ack` is `true`, or just acknowledged otherwise. Value in milliseconds, defaults to `20000`.
hibernate.search.<indexName>.jgroups.delegate_backend	The master node receiving indexing operations forwards them to a standard backend to be performed. Defaults to `lucene`. See also Table 3.4, "Backend configuration" for other options, but probably the only useful option is `blackhole`, or a custom implementation, to help isolating network latency problems.

3.4.2.1. Slave nodes

Every index update operation is sent through a JGroups channel to the master node. Index querying operations are executed on a local index copy. Enabling the JGroups worker only makes sure the index operations are sent to the master, you still have to synchronize configuring an appropriate directory (See `filesystem-master, filesystem-slave` or `infinispan` options in Section 3.3, "Directory configuration").

Example 3.7. JGroups Slave configuration

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

3.4.2.2. Master node

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

Example 3.8. JGroups Master configuration

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

3.4.2.3. Automatic master election

> **Important**
>
> This feature is considered experimental. In particular during a re-election process there is a small window of time in which indexing requests could be lost.

In this mode the different nodes will autonomously elect a master node. When a master fails, a new node is elected automatically.

When setting this backend it is expected that all Hibernate Search instances in the same cluster use the same backend for each specific index: this configuration is an alternative to the static `jgroupsMaster` and `jgroupsSlave` approach so make sure to not mix them.

To synchronize the indexes in this configuration avoid `filesystem-master` and `filesystem-slave` directory providers as their behaviour can not be switched dynamically; use the Infinispan `Directory` instead, which has no need for different configurations on each instance and allows dynamic switching of writers; see also Section 3.3.1, "Infinispan Directory configuration".

Example 3.9. JGroups configuration for automatic master configuration

```
### automatic configuration
hibernate.search.default.worker.backend = jgroups
```

> **Tip**
>
> Should you use `jgroups` or the couple `jgroupsMaster, jgroupsSlave`?
>
> The dynamic `jgroups` backend is better suited for environments in which your master is more likely to need to failover to a different machine, as in clouds. The static configuration has the benefit of keeping the master at a well known location: your architecture might take advantage of it by sending most write requests to the known master. Also optimisation and MassIndexer operations need to be triggered on the master node.

3.4.2.4. JGroups channel configuration

Configuring the JGroups channel essentially entails specifying the transport in terms of a network protocol stack. To configure the JGroups transport, point the configuration property

hibernate.search.services.jgroups.configurationFile to a JGroups configuration file; this can be either a file path or a Java resource name.

> **Tip**
>
> If no property is explicitly specified it is assumed that the JGroups default configuration file `flush-udp.xml` is used. This example configuration is known to work in most scenarios, with the notable exception of Amazon AWS; refer to the JGroups manual [http://www.jgroups.org/manual-3.x/html/] for more examples and protocol configuration details.

The default cluster name is `Hibernate Search Cluster` which can be configured as seen in Example 3.10, "JGroups cluster name configuration".

Example 3.10. JGroups cluster name configuration

```
hibernate.search.services.jgroups.clusterName = My-Custom-Cluster-Id
```

The cluster name is what identifies a group: by changing the name you can run different clusters in the same network in isolation.

3.4.2.4.1. JGroups channel instance injection

For programmatic configurations, one additional option is available to configure the JGroups channel: to pass an existing channel instance to Hibernate Search directly using the property `hibernate.search.services.jgroups.providedChannel`, as shown in the following example.

```
import org.hibernate.search.backend.impl.jgroups.JGroupsChannelProvider;

org.jgroups.JChannel channel = ...
Map<String,String> properties = new HashMap<String,String>(1);
properties.put( JGroupsChannelProvider.CHANNEL_INJECT, channel );
EntityManagerFactory emf = Persistence.createEntityManagerFactory( "userPU", properties );
```

3.5. Reader strategy configuration

The different reader strategies are described in Section 2.3, "Reader strategy". Out of the box strategies are:

- `shared`: share index readers across several queries. This strategy is the most efficient.

- `not-shared`: create an index reader for each individual query

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

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

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

Or if you have a custom reader strategy:

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

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

3.6. Exception handling

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

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

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

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

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

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

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

Alternatively, an ErrorHandler instance may be passed via the configuration value map used when bootstrapping Hibernate Search programmatically.

3.7. Lucene configuration

Even though Hibernate Search will try to shield you as much as possible from Lucene specifics, there are several Lucene specifics which can be directly configured, either for performance reasons or for satisfying a specific use case. The following sections discuss these configuration options.

3.7.1. Tuning indexing performance

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

There are 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 will look at the index section, then at the default section.

Example 3.11. Example performance option configuration

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

The configuration in Example 3.11, "Example performance option configuration" 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 3.7, "List of indexing performance and behavior properties" depend for this reason on the version of Lucene you are using. The values shown are relative to version `2.4`. For more information about Lucene indexing performance, please refer to the Lucene documentation.

Table 3.7. List of indexing performance and behavior properties

Property	Description	Default Value
hibernate.search.[default|<index-name>].exclusive_index_use	Set to `true` when no other process will need to write to the same index. This will enable Hibernate Search to work in exclusive mode on the in-	`true` (improved performance, re-leases locks only at shutdown)

Property	Description	Default Value
	dex and improve performance when writing changes to the index.	
hibernate.search.[default|<index-name>].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`
hibernate.search.[default|<indexname>].index_flush_interval	The interval in milliseconds between flushes of write operations to the index storage. Ignored unless `worker.execution` is configured as `async`.	`1000`
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|<index-name>].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 index-	10

Property	Description	Default Value
	ing 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.	
hibernate.search.[default|<index-name>].indexwriter.merge_min_size	Controls segment merge frequency and size. Segments smaller than this size (in MB) are always considered for the next segment merge operation. ` Setting this too large might result in expensive merge operations, even tough they are less frequent. See also `org.apache.lucene.index.LogDocMergePolicy.minMergeSize`.	0 MB (actually ~1K)
hibernate.search.[default|<indexname>]. indexwriter.merge_max_size	Controls segment merge frequency and size. Segments larger than this size (in MB) are never merged in bigger segments. 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. ` See also `org.apache.lucene.index.LogDocMergePolicy.maxMergeSize`.	Unlimited
hibernate.search.[default|<indexname>]. indexwriter.merge_max_optimize_size	Controls segment merge frequency and size. 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). Applied to `org.apache.lucene.index.LogDocMergePolicy.maxMergeSizeForOptim`	Unlimited
hibernate.search.[default|<indexname>]. indexwriter.merge_calibrate_by_deletes	Controls segment merge frequency and size. Set to `false` to not consider deleted documents when estimating the merge policy. Applied to `org.apache.lucene.index.LogMergePolicy.calibrateSizeByDeletes`.	`true`

Property	Description	Default Value
hibernate.search.[default|<index-name>].indexwriter.ram_buffer_size	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. 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.	16 MB
hibernate.search.[default|<indexname>]. indexwriter.term_index_interval	Expert: Set the interval between indexed terms. Large values cause less memory to be used by IndexReader, but slow random-access to terms. Small values cause more memory to be used by an IndexReader, and speed random-access to terms. See Lucene documentation for more details.	128
hibernate.search. enable_dirty_check	Not all entity changes require an update of the Lucene index. If all of the updated entity properties (dirty properties) are not indexed Hibernate Search will skip the re-indexing work. Disable this option if you use a custom `FieldBridge` which need to be invoked at each update event (even though the property for which the field bridge is configured has not changed). This optimization will not be applied on classes using a `@ClassBridge` or a `@DynamicBoost`. Boolean parameter, use "true" or "false".	true
hibernate.search.[default|<indexname>]. indexwriter.max_thread_states	Lucene's `IndexWriter` can apply writes in parallel, but this property controls the limit of parallelism. If you have many cores and contention on the internal structures of the `IndexWriter` becomes a bottleneck you should configure an high-	8

Property	Description	Default Value
	er value, at the cost of slightly higher memory consumption.	
hibernate.search.[default|<index-name>].indexwriter.infostream	Enable low level trace information about Lucene's internal components. Will cause significant performance degradation: should only be used for troubleshooting purposes.	false

> **Tip**
>
> When your architecture permits it, always keep `hibernate.search.default.exclusive_index_use=true` as it greatly improves efficiency in index writing. This is the default since Hibernate Search version 4.

> **Tip**
>
> To tune the indexing speed it might be useful to time the object loading from database in isolation from the writes to the index. To achieve this set the `blackhole` as worker backend and start your indexing routines. This backend does not disable Hibernate Search: it will still generate the needed changesets to the index, but will discard them instead of flushing them to the index. In contrast to setting the `hibernate.search.indexing_strategy` to `manual`, using `blackhole` will possibly load more data from the database. because associated entities are re-indexed as well.
>
> ```
> hibernate.search.[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` backend is not meant to be used in production, only as a tool to identify indexing bottlenecks.

3.7.1.1. Control segment size

The options `merge_max_size`, `merge_max_optimize_size`, `merge_calibrate_by_deletes` give you control on the maximum size of the segments being created, but you need to understand how they affect file sizes. If you need to hard limit the size, consider that merging a segment is

about adding it together with another existing segment to form a larger one, so you might want to set the `max_size` for merge operations to less than half of your hard limit. Also segments might initially be generated larger than your expected size at first creation time: before they are ever merged. A segment is never created much larger than `ram_buffer_size`, but the threshold is checked as an estimate.

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
```

> **Tip**
>
> When using the Infinispan Directory to cluster indexes make sure that your segments are smaller than the `chunk_size` so that you avoid fragmenting segments in the grid. Note that the `chunk_size` of the Infinispan Directory is expressed in bytes, while the index tuning options are in MB.

3.7.1.2. Troubleshooting: enable Lucene's Infostream

Apache Lucene allows to log a very detailed trace log from its internals using a feature called "infostream". To access these details, Hibernate Search can be configured to capture this internal trace from Apache Lucene and redirect it to your logger.

- Enable `TRACE` level logging for the category `org.hibernate.search.backend.lucene.infostream`

- Activate the feature on the index you want to inspect: `hibernate.search.[default|<index-name>].indexwriter.infostream=true`

Keep in mind that this feature has a performance cost, and although most logger frameworks allow the `TRACE` level to be reconfigured at runtime, enabling the `infostream` property will slow you down even if the logger is disabled.

3.7.2. LockFactory configuration

Lucene Directorys have default locking strategies which work generally good enough for most cases, but it's possible to specify for each index managed by Hibernate Search a specific LockingFactory you want to use. This is generally not needed but could be useful.

Some of these locking strategies require a filesystem level lock and may be used even on RAM based indexes, this combination is valid but in this case the `indexBase` configuration option usually needed only for filesystem based Directory instances must be specified to point to a filesystem location where to store the lock marker files.

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

Table 3.8. List of available LockFactory implementations

name	Class	Description
simple	org.apache.lucene.store.SimpleFSLockFactory	Safe implementation based on Java's File API, it marks the usage of the index by creating a marker file. If for some reason you had to kill your application, you will need to remove this file before restarting it.
native	org.apache.lucene.store.NativeFSLockFactory	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. This implementation has known problems on NFS, avoid it on network shares. `native` is the default implementation for the `filesystem`, `filesystem-master` and `filesystem-slave` directory providers.
single	org.apache.lucene.store.SingleInstanceLockFactory	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. This is the default implementation for the `ram` directory provider.

name	Class	Description
none	org.apache.lucene.store. NoLockFactory	All changes to this index are not coordinated by any lock; test your application carefully and make sure you know what it means.

Configuration example:

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

The Infinispan Directory uses a custom implementation; it's still possible to override it but make sure you understand how that will work, especially with clustered indexes.

3.7.3. Index format compatibility

While Hibernate Search strives to offer a backwards compatible API making it easy to port your application to newer versions, it still delegates to Apache Lucene to handle the index writing and searching. This creates a dependency to the Lucene index format. The Lucene developers of course attempt to keep a stable index format, but sometimes a change in the format can not be avoided. In those cases you either have to re-index all your data or use an index upgrade tool. Sometimes Lucene is also able to read the old format so you don't need to take specific actions (besides making backup of your index).

While an index format incompatibility is a rare event, it can happen more often that Lucene's Analyzer implementations might slightly change its behavior. This can lead to a poor recall score, possibly missing many hits from the results.

Hibernate Search exposes a configuration property `hibernate.search.lucene_version` which instructs the analyzers and other Lucene classes to conform to their behavior as defined in an (older) specific version of Lucene. See also `org.apache.lucene.util.Version` contained in the *lucene-core.jar*. Depending on the specific version of Lucene you're using you might have different options available. When this option is not specified, Hibernate Search will instruct Lucene to use the default version, which is usually the best option for new projects. Still it's recommended to define the version you're using explicitly in the configuration so that when you happen to upgrade Lucene the analyzers will not change behavior. You can then choose to update this value at a later time, when you for example have the chance to rebuild the index from scratch.

Example 3.12. Force Analyzers to be compatible with a Lucene 4.7 created index

```
hibernate.search.lucene_version = LUCENE_47
```

This option is global for the configured SearchFactory and affects all Lucene APIs having such a parameter, as this should be applied consistently. So if you are also making use of Lucene bypassing Hibernate Search, make sure to apply the same value too.

3.8. Metadata API

After looking at all these different configuration options, it is time to have a look at an API which allows you to programmatically access parts of the configuration. Via the metadata API you can determine the indexed types and also how they are mapped (see Chapter 4, *Mapping entities to the index structure*) to the index structure. The entry point into this API is the SearchFactory. It offers two methods, namely `getIndexedTypes()` and `getIndexedTypeDescriptor(Class<?>)`. The former returns a set of all indexed type, where as the latter allows to retrieve a so called IndexedTypeDescriptorfor a given type. This descriptor allows you determine whether the type is indexed at all and, if so, whether the index is for example sharded or not (see Section 10.5, "Sharding indexes"). It also allows you to determine the static boost of the type (see Section 4.2.1, "Static index time boosting") as well as its dynamic boost strategy (see Section 4.2.2, "Dynamic index time boosting"). Most importantly, however, you get information about the indexed properties and generated Lucene Document fields. This is exposed via PropertyDescriptors respectively FieldDescriptors. The easiest way to get to know the API is to explore it via the IDE or its javadocs.

> **Note**
>
> All descriptor instances of the metadata API are read only. They do not allow to change any runtime configuration.

3.9. Hibernate Search as a WildFly module

Hibernate Search is included in the WildFly 8 application server, but you need to activate the module for your deployment to be able to take advantage of it.

You can either activate the modules already included in your WildFly distribution, or download and unzip a different version provided by Hibernate Search. The modules system in WildFly allows to safely run multiple versions of Hibernate ORM and Hibernate Search in parallel. You can pick the one you need for each deployment.

3.9.1. Use the Hibernate Search version included in WildFly

To activate the Hibernate Search modules in wildfly, you have to declare the dependency by either adding a line in the `MANIFEST.MF` of your deployment archive, or by including a resource named `jboss-deployment-structure.xml`. This resource should be placed in the top level deployment, in META-INF (or WEB-INF for web deployments).

Using the manifest

Add the following entry to the `MANIFEST.MF` in your archive:

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

Using jboss-deployment-structure.xml

add a resource named `jboss-deployment-structure.xml` in your top level deployment, in META-INF (or WEB-INF for web deployments).

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

3.9.2. Update and activate latest Hibernate Search version in WildFly

You can also download the latest Hibernate Search provided module and install it. This is often the best approach as you will benefit from all the latest improvements of Hibernate Search. Because of the modular design in WildFly, these additional modules can coexist with the embedded modules and won't affect any other application, unless you explicitly reconfigure it to use the newer module.

You can download the latest pre-packaged Hibernate Search modules from Sourceforge [http://sourceforge.net/projects/hibernate/files/hibernate-search/5.3.0.Final/hiber-nate-search-modules-5.3.0.Final-wildfly-8-dist.zip/download]. As a convenience these zip files are also distributed as Maven artifacts: org.hibernate:hibernate-search-modules-5.3.0.Final-wildfly-8-dist:zip [https://repository.jboss.org/nexus/index.html#nexus-search;gav~org.hibernate~hibernate-search-modules~{hibernateSearchVersion}~~].

Unpack the modules in your WildFly `modules` directory: this will create modules for Hibernate Search and Apache Lucene. The Hibernate Search modules are:

- *org.hibernate.search.orm*, for users of Hibernate Search with Hibernate; this will transitively include Hibernate ORM.

- *org.hibernate.search.engine*, for projects depending on the internal indexing engine that don't require other dependencies to Hibernate.

- *org.hibernate.search.backend-jms*, in case you want to use the JMS backend described in JMS Architecture.

Using the manifest

Add the following entry to the `MANIFEST.MF` in your archive:

```
Dependencies: org.hibernate.search.orm:5.3.0.Final services
```

Using jboss-deployment-structure.xml

> add a resource named `jboss-deployment-structure.xml` in your top level deployment, in META-INF (or WEB-INF for web deployments).

```xml
<jboss-deployment-structure>
    <deployment>
        <dependencies>
            <module
                name="org.hibernate.search.orm"
                slot="5.3.0.Final"
                services="export" />
        </dependencies>
    </deployment>
</jboss-deployment-structure>
```

Tip

Modular classloading is a feature of JBoss EAP 6 as well, but if you are using JBoss EAP, you're reading the wrong version of the user guide! JBoss EAP subscriptions include official support for Hibernate Search (as part of the WFK) and come with a different edition of this guide specifically tailored for EAP users.

More information about the modules configuration in WildFly can be found in the Class Loading in WildFly 8 [https://docs.jboss.org/author/display/WFLY8/Class+Loading+in+WildFly] wiki.

3.9.3. Using Infinispan with Hibernate Search on WildFly

The Infinispan project is also included in WildFly so you can use the feature without additional downloads.

If you are updating the version of Hibernate Search in WildFly as described in the previous paragraph, you might need to update Infinispan as well. The process is very similar: download the modules from Infinispan project downloads [http://infinispan.org/download/], picking a compatible version, and decompress the modules into the `modules` directory of your WildFly installation.

Hibernate Search version `5.3.0.Final` was compiled and tested with Infinispan version `7.2.0.Final`; generally a more recent version of either project is expected to be backwards compatible for cross-project integration purposes as long as they have the same "major.minor" family version.

For example for a version of Hibernate Search depending on Infinispan `7.0.3.Final` it should be safe to upgrade Infinispan to `7.0.6.Final`, but an upgrade to `7.1.0.Final` might not work.

Chapter 4. Mapping entities to the index structure

4.1. Mapping an entity

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

> **Note**
>
> There is no XML configuration available for Hibernate Search but we provide a programmatic mapping API that elegantly replaces this kind of deployment form (see Section 4.7, "Programmatic API" for more information).
>
> If you want to contribute the XML mapping implementation, see HSEARCH-210 [https://hibernate.onjira.com/browse/HSEARCH-210].

4.1.1. Basic mapping

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

4.1.1.1. @Indexed

Foremost you must declare a persistent class as indexable by annotating the class with `@Indexed`. All entities not annotated with `@Indexed` will be ignored by the indexing process.

Example 4.1. Making a class indexable with `@Indexed`

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

You can optionally specify the `Indexed.index` attribute to change the default name of the index. For more information regarding index naming see Section 3.3, "Directory configuration".

You can also specify an optional indexing interceptor. For more information see conditional indexing.

4.1.1.2. @Field

For each property of your entity, you have the ability to describe whether and how it will be indexed. Adding the `@Field` annotation declares a property as indexed and allows you to configure various aspects of the indexing process. Without `@Field` the property is ignored by the indexing process.

Hibernate Search tries to determine the best way to index your property. In most cases this will be as string, but for the types int, long, double and float (and their respective Java wrapper types) Lucene's numeric field encoding (see Section 4.1.1.3, "@NumericField") is used. This numeric encoding uses a so called Trie structure [http://en.wikipedia.org/wiki/Trie] which allows for efficient range queries and sorting, resulting in query response times being orders of magnitude faster than with the plain string encoding. Byte and short properties will only be encoded in numeric fields if explicitly marked with the `@NumericField` annotation.

> **Caution**
>
> Prior to Search 5, numeric field encoding was only chosen if explicitly requested via `@NumericField`. As of Search 5 this encoding is automatically chosen for numeric types. To avoid numeric encoding you can explicitly specify a non numeric field bridge via `@Field.bridge` or `@FieldBridge`. The package `org.hibernate.search.bridge.builtin` contains a set of bridges which encode numbers as strings, for example `org.hibernate.search.bridge.builtin.IntegerBridge`.

The following attributes of the `@Field` annotation help you control the indexing outcome:

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

- `store`: describes 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), 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. Storing the property has no impact on whether the value is searchable or not.

- `index`: describes whether the property is indexed or not. The different values are `Index.NO` (no indexing, meaning the value 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 needed for projection.

> **Tip**
>
> `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`.

> **Tip**
>
> 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.

> **Tip**
>
> Fields used for sorting or faceting *must not* be analyzed.

- `norms`: describes whether index time boosting information should be stored (`Norms.YES`) or not (`Norms.NO`). Not storing the norms can save a considerable amount of memory, but index time boosting will not be available in this case. 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.
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

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

- `indexNullAs`: Per default null values are ignored and not indexed. However, using `indexNullAs` you can specify a string which will be inserted as token for the null value. Per default this value is set to `org.hibernate.search.annotations.Field.DO_NOT_INDEX_NULL` indicating that null values should not be indexed. You can set this value to `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 the string "*null*" will be used as default.

> **Note**
>
> When indexNullAs is used, it is important to use the chosen null token in search queries (see Chapter 5, *Querying*) in order to find null values. It is also advisable to use this feature only with un-analyzed fields (`analyze=Analyze.NO`).

> **Note**
>
> 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()`).

- `boost`: Refer to section about boosting

- `bridge`: Refer to section about field bridges

4.1.1.3. @NumericField

`@NumericField` is a companion annotation to `@Field`. It can be specified in the same scope as `@Field`, but only on properties of numeric type like byte, short, int, long, double and float (and their respective Java wrapper types). It allows to define a custom `precisionStep` for the numeric encoding of the property value.

`@NumericField` accepts the following parameters:

Value	Definition
forField	(Optional) Specify the name of of the related `@Field` that will be indexed numerically. It's only mandatory when the property contains more than a `@Field` declaration
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 using string encoding. Default value is 4.

Lucene supports the numeric types: `Double`, `Long`, `Integer` and `Float`. For properties of types `Byte` and `Short`, an `Integer` field will be used in the index. Other numeric types should use the default string encoding (via `@Field`), unless the application can deal with a potential loss in precision, in which case a custom `NumericFieldBridge` can be used. See Example 4.2, "Defining a custom NumericFieldBridge for BigDecimal".

Example 4.2. Defining a custom NumericFieldBridge for BigDecimal

```java
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 tmpLong = decimalValue.multiply( storeFactor ).longValue();
            Long indexedValue = Long.valueOf( tmpLong );
            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 );
    }
}
```

You would use this custom bridge like seen in Example 4.3, "Use of `BigDecimalNumericFieldBridge`". In this case three annotations are used - `@Field`, `@NumericField` and `@FieldBridge`. `@Field` is required to mark the property for being indexed (a standalone `@NumericField` is never allowed). `@NumericField` might be omitted in this specific case, because the used `@FieldBridge` annotation refers already to a `NumericFieldBridge` instance. However, the use of `@NumericField` makes the use of the property as numeric value explicit.

Example 4.3. Use of `BigDecimalNumericFieldBridge`

```
@Entity
@Indexed
public static class Item {
    @Id
    @GeneratedValue
    private int id;

    @Field
    @NumericField
    @FieldBridge(impl = BigDecimalNumericFieldBridge.class)
    private BigDecimal price;

    public int getId() {
        return id;
    }

    public BigDecimal getPrice() {
        return price;
    }

    public void setPrice(BigDecimal price) {
        this.price = price;
    }
}
```

4.1.1.4. @Id

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

Example 4.4. 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 4.4, "Specifying indexed properties" defines an index with four fields: `id`, `Abstract`, `text` and `grade`. Note that by default the field name is de-capitalized, following the JavaBean specification. The `grade` field is annotated as Numeric with a slightly larger precisionStep than the default.

4.1.2. Mapping properties multiple times

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

Example 4.5. 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 4.5, "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 un-tokenized way. @Field supports 2 attributes useful when @Fields is used:

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

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

See below for more information about analyzers and field bridges.

4.1.3. Embedded and associated objects

Associated objects as well as embedded objects can be indexed as part of the root entity index. This is useful if you expect to search a given entity based on properties of the associated objects.

In the example Example 4.6, "Indexing associations" the aim is to return places where the associated city is Atlanta (in Lucene query parser language, it would translate into `address.city:Atlanta`). All place fields are added to the `Place` index, but also the address related fields `address.street`, and `address.city` will be added and made queryable. The embedded object id, `address.id`, is not added per default. To include it you need to also set `@IndexedEmbedded(includeEmbeddedObjectId=true, …)`.

> **Tip**
>
> Only actual indexed fields (properties annotated with `@Field`) are added to the root entity index when embedded objects are indexed. The embedded object identifiers are treated differently and need to be included explicitly.

Example 4.6. Indexing associations

```
@Entity
@Indexed
public class Place {
    @Id
    @GeneratedValue
    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 de-normalized in the Lucene index when using the `@IndexedEm-bedded` technique, Hibernate Search needs to be aware of any change in the `Place` object and any change in the `Address` object to keep the index up to date. To make sure the `Place` Lucene document is updated when it's `Address` changes, you need to mark the other side of the bidirectional relationship with `@ContainedIn`.

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

Let's make Example 4.6, "Indexing associations" a bit more complex by nesting `@IndexedEmbedded` as seen in Example 4.7, "Nested usage of `@IndexedEmbedded` and `@ContainedIn`".

Example 4.7. Nested usage of `@IndexedEmbedded` and `@ContainedIn`

```
@Entity
@Indexed
public class Place {
    @Id
    @GeneratedValue
    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` or `@Embedded` attribute can be annotated with `@In-dexedEmbedded`. The attributes of the associated class will then be added to the main entity index. In Example 4.7, "Nested usage of `@IndexedEmbedded` and `@ContainedIn`" the index will contain the following fields

- id

- name

- address.street

- address.city

- address.ownedBy_name

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

> **Note**
>
> The prefix cannot be set to the empty string.

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

Using `@IndexedEmbedded` for object associations allows you to express queries (using Lucene's query syntax) such as:

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

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

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

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

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

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

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

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

Example 4.8. Using the targetElement property of `@IndexedEmbedded`

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

    @Field
    private String street;

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

    // ...
}
```

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

4.1.3.1. Limiting object embedding to specific paths

The `@IndexedEmbedded` annotation provides also an attribute `includePaths` which can be used as an alternative to `depth`, or in combination 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 4.9, "Using the includePaths property of @IndexedEmbedded"

Example 4.9. 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 4.9, "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 `@IndexedEmbedded.includePaths` will include the specified paths *in addition to* what you would index normally specifying a limited value for depth. Using `includePaths` with a undefined (default) value for `depth` is equivalent to setting `depth=0`: only the included paths are indexed.

Example 4.10. Using the includePaths property of @IndexedEmbedded

```java
@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 4.10, "Using the includePaths property of @IndexedEmbedded", every human will have it's name and surname attributes indexed. The name and surname of parents will be indexed too, recursively up to second line 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

> **Tip**
>
> You can explicitly include the id of the embedded object using `includePath`, for example `@IndexedEmbedded(includePaths = { "parents.id" })`. This will work regardless of the `includeEmbeddedObjectId` attribute. However, it is recommended to just set `includeEmbeddedObjectId=true`.

> **Tip**
>
> 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.

4.1.4. Associated objects: building a dependency graph with @ContainedIn

While `@ContainedIn` is often seen as the counterpart of `@IndexedEmbedded`, it can also be used on its own to build an indexing dependency graph.

When an entity is reindexed, all the entities pointed by `@ContainedIn` are also going to be reindexed.

4.2. Boosting

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

4.2.1. Static index time boosting

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

Example 4.11. Different ways of using @Boost

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

4.2.2. Dynamic index time boosting

The `@Boost` annotation used in Section 4.2.1, "Static index time boosting" defines a static boost factor which is independent of the state of of the indexed entity at runtime. However, there are use cases 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 4.12. Dynamic boost example

```java
public enum PersonType {
    NORMAL,
    VIP
```

```
}
```

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

    // ...
}
```

```
public class VIPBoostStrategy implements BoostStrategy {
    public float defineBoost(Object value) {
        Person person = ( Person ) value;
        if ( person.getType().equals( PersonType.VIP ) ) {
            return 2.0f;
        }
        else {
            return 1.0f;
        }
    }
}
```

In Example 4.12, "Dynamic boost example" a dynamic boost is defined on class level specifying VIPBoostStrategy as implementation of the BoostStrategy interface to be used at indexing time. You can place the `@DynamicBoost` either at class or field level. Depending on the placement of the annotation either the whole entity is passed to the defineBoost method or just the annotated field/property value. It's up to you to cast the passed object to the correct type. In the example all indexed values of a VIP person would be double as important as the values of a normal person.

> **Note**
>
> The specified BoostStrategy implementation must define a public no-arg constructor.

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

4.3. Analysis

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

4.3.1. Default analyzer and analyzer by class

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

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

Example 4.13. Different ways of using @Analyzer

```java
@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 (eg. `name`), except `summary` and `body` which are indexed with PropertyAnalyzer and FieldAnalyzer respectively.

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

4.3.2. Named analyzers

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

- a name: the unique string used to refer to the definition

- a list of char filters: each char filter is responsible to pre-process input characters before the tokenization. Char filters can add, change or remove characters; one common usage is for characters normalization

- a tokenizer: responsible for tokenizing the input stream into individual words

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

This separation of tasks - a list of char filters, and a tokenizer followed by a list of filters - allows for easy reuse of each individual component and let you build your customized analyzer in a very flexible way (just like Lego). Generally speaking the char filters do some pre-processing in the character input, then the Tokenizer starts the tokenizing process by turning the character input into tokens which are then further processed by the TokenFilters. Hibernate Search supports this infrastructure by utilizing the advanced analyzers provided by Lucene; this is often referred to as the Analyzer Framework.

> **Note**
>
> Some of the analyzers and filters will require additional dependencies. For example to use the snowball stemmer you have to also include the `lucene-snowball` jar and for the PhoneticFilterFactory you need the commons-codec [http://commons.apache.org/codec] jar. Your distribution of Hibernate Search provides these dependencies in its *lib/optional* directory. Have a look at Table 4.2, "Example of available tokenizers" and Table 4.3, "Examples of available filters" to see which analyzers and filters have additional dependencies
>
> Prior to Hibernate Search 5 it was required to add the Apache Solr dependency to your project as well; this is no longer required.

Let's have a look at a concrete example now - Example 4.14, "@AnalyzerDef and the Analyzer Framework". First a char filter is defined by its factory. In our example, a mapping char filter is used, and will replace characters in the input based on the rules specified in the mapping file. Next a tokenizer is defined. This example uses the standard tokenizer. Last but not least, a list of filters is defined by their factories. In our example, the StopFilter filter is built reading the dedicated words property file. The filter is also expected to ignore case.

Example 4.14. @AnalyzerDef and the Analyzer Framework

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

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

Some tokenizers, token filters or char filters load resources like a configuration or metadata file.
This is the case for the stop filter and the synonym filter.

Example 4.15. Use a specific charset to load the property file

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

```
    // ...
}
```

Once defined, an analyzer definition can be reused by an @Analyzer declaration as seen in Example 4.16, "Referencing an analyzer by name".

Example 4.16. 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 wen building queries.

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

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

4.3.2.1. Available analyzers

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

Table 4.1. Example of available char filters

Factory	Description	Parameters	Additional dependencies
MappingCharFilter-Factory	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" $\Rightarrow$ "a" "ñ" $\Rightarrow$ "n" "ø" $\Rightarrow$ "o"	lucene-analyzers-common
HTMLStripCharFilter-Factory	Remove HTML standard tags, keeping the text	none	lucene-analyzers-common

Table 4.2. Example of available tokenizers

Factory	Description	Parameters	Additional dependencies
StandardTokenizer-Factory	Use the Lucene StandardTokenizer	none	lucene-analyzers-common
HTMLStripCharFilter-Factory	Remove HTML tags, keep the text and pass it to a StandardTokenizer.	none	`lucene-analyzers-common`
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	`lucene-analyzers-common`

Table 4.3. Examples of available filters

Factory	Description	Parameters	Additional dependencies
StandardFilterFactory	Remove dots from acronyms and 's from words	none	`lucene-analyzers-common`
LowerCaseFilterFactory	Lowercases all words	none	`lucene-analyzers-common`

Factory	Description	Parameters	Additional dependencies
StopFilterFactory	Remove words (tokens) matching a list of stop words	`words`: points to a resource file containing the stop words ignoreCase: true if `case` should be ignore when comparing stop words, `false` otherwise	`lucene-analyzers-common`
SnowballPorterFilterFactory	Reduces a word to it's root in a given language. (eg. protect, protects, protection share the same root). Using such a filter allows searches matching related words.	`language`: Danish, Dutch, English, Finnish, French, German, Italian, Norwegian, Portuguese, Russian, Spanish, Swedish and a few more	`lucene-analyzers-common`
ASCIIFoldingFilterFactory	Remove accents for languages like French	none	`lucene-analyzers-common`
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	`lucene-analyzers-phonetic` and `commons-codec`
CollationKeyFilterFactory	Converts each token into its java.text.CollationKey, and then encodes the CollationKey with IndexableBinaryString-	`custom`, `language`, `country`, `variant`, `strength`, \`decomposition \`see Lucene's CollationKeyFilter	`lucene-analyzers-common` and `commons-io`

Factory	Description	Parameters	Additional dependencies
	Tools, to allow it to be stored as an index term.	javadocs for more info	

We recommend to check out the implementations of org.apache.lucene.analysis.util.TokenizerFactory and org.apache.lucene.analysis.util.TokenFilterFactory in your IDE to see the implementations available.

4.3.3. 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 4.17, "Usage of @AnalyzerDiscriminator" demonstrates the usage of this annotation.

Example 4.17. 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 4.17, "Usage of @AnalyzerDiscriminator" assumes that the language parameter is either 'de' or 'en' which matches the specified names in the @AnalyzerDefs.

4.3.4. Retrieving an analyzer

In some situations retrieving analyzers can be handy. For example, if your domain model makes use of multiple analyzers (maybe to benefit from stemming, use phonetic approximation and so on), you need to make sure to use the same analyzers when you build your query.

> **Note**
>
> This rule can be broken but you need a good reason for it. If you are unsure, use the same analyzers. If you use the Hibernate Search query DSL (see Section 5.1.2,

> "Building a Lucene query with the Hibernate Search query DSL"), you don't have
> to think about it. The query DSL does use the right analyzer transparently for you.

Whether you are using the Lucene programmatic API or the Lucene query parser, you can retrieve the scoped analyzer for a given entity. A scoped analyzer is an analyzer which applies the right analyzers depending on the field indexed. Remember, multiple analyzers can be defined on a given entity each one working on an individual field. A scoped analyzer unifies all these analyzers into a context-aware analyzer. While the theory seems a bit complex, using the right analyzer in a query is very easy.

Example 4.18. Using the scoped analyzer when building a full-text query

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

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

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

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

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

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

4.4. Bridges

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

4.4.1. Built-in bridges

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

null

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

`java.lang.String`

Strings are indexed as are

`short`, `Short`, `integer`, `Integer`, `long`, `Long`, `float`, `Float`, `double`, `Double`

Are per default indexed numerically using a Trie structure [http://en.wikipedia.org/wiki/Trie]. You need to use a `NumericRangeQuery` to search for values. See also Section 4.1.1.2, "@Field" and Section 4.1.1.3, "@NumericField"

`BigInteger`, `BigDecimal`

`BigInteger` and `BigDecimal` are converted into their string representation and indexed. Note that in this form the values cannot be compared by Lucene using for example a `TermRange-Query`. For that the string representation would need to be padded. An alternative using numeric encoding with a potential loss in precision can be seen in Example 4.2, "Defining a custom NumericFieldBridge for BigDecimal".

`java.util.Date`, `java.util.Calendar`

Dates are indexed as `long` value representing the number of milliseconds since *January 1, 1970, 00:00:00 GMT*. You shouldn't really bother with the internal format. It is important, however, to query a numerically indexed date via a `NumericRangeQuery`.

Usually, storing the date up to the millisecond is not necessary. `@DateBridge` defines the appropriate resolution you are willing to store in the index.

```
@Entity
@Indexed
public class Meeting {
    @Field(analyze=Analyze.NO)
    @DateBridge(resolution=Resolution.MINUTE)
    private Date date;
    // ...
```

You can also choose to encode the date as string using the `encoding=EncodingType.STRING` of `DateBridge`. In this case the dates are stored in the format *yyyyMMddHHmmssSSS* (using GMT time).

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

> **Important**
>
> The default date bridge uses Lucene's `DateTools` to convert from `Date` or `Calendar` to its indexed value. 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`

Classes are converted to their fully qualified class name. The thread context classloader is used when the class is rehydrated

4.4.2. Tika bridge

Hibernate Search allows you to extract text from various document types using the built-in TikaBridge which utilizes Apache Tika [http://tika.apache.org] to extract text and metadata from the provided documents. The TikaBridge annotation can be used with String, URI, byte[] or java.sql.Blob properties. In the case of String and URI the bridge interprets the values are file paths and tries to open a file to parse the document. In the case of byte[] and Blob the values are directly passed to Tika for parsing.

Tika uses metadata as in- and output of the parsing process and it also allows to provide additional context information. This process is described in Parser interface [http://tika.apache.org/1.1/parser.html#apiorgapachetikametadataMetadata.html]. The Hibernate Search Tika bridge allows you to make use of these additional configuration options by providing two interfaces in conjunction with TikaBridge. The first interface is the TikaParseContextProvider. It allows you to create a custom ParseContext for the document parsing. The second interface is TikaMetadataProcessor which has two methods - prepareMetadata() and set(String, Object, Document, LuceneOptions, Metadata metadata). The former allows to add additional metadata to the parsing process (for example the file name) and the latter allows you to index metadata discovered during the parsing process.

TikaParseContextProvider as well as TikaMetadataProcessor implementation classes can both be specified as parameters on the TikaBridge annotation.

Example 4.19. Example mapping with Apache Tika

```
@Entity
@Indexed
public class Song {
    @Id
    @GeneratedValue
    long id;

    @Field
    @TikaBridge(metadataProcessor = Mp3TikaMetadataProcessor.class)
    String mp3FileName;

    // ...
}
```

```
QueryBuilder queryBuilder = fullTextSession.getSearchFactory()
    .buildQueryBuilder()
    .forEntity( Song.class )
    .get();
Query query = queryBuilder.keyword()
    .onField( "mp3FileName" )
    .ignoreFieldBridge() //mandatory
    .matching( "Apes" )
    .createQuery();
List result = fullTextSession.createFullTextQuery( query ).list();
```

In the Example 4.19, "Example mapping with Apache Tika" the property mp3FileName represents a path to an MP3 file; the headers of this file will be indexed and so the performed query will be able to match the MP3 metadata.

> **Warning**
>
> TikaBridge does not implement TwoWayFieldBridge: queries built using the DSL (as in the Example 4.19, "Example mapping with Apache Tika") need to explicitly enable the option ignoreFieldBridge().

4.4.3. Custom bridges

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

4.4.3.1. StringBridge

The simplest custom solution is to give Hibernate Search an implementation of your expected Object to String bridge. To do so you need to implement the

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

Example 4.20. Custom `StringBridge` implementation

```java
/**
 * 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("Number too big to be padded");
        StringBuilder paddedInteger = new StringBuilder();
        for (int padIndex = rawInteger.length(); padIndex < padding; padIndex++) {
            paddedInteger.append('0');
        }
        return paddedInteger.append( rawInteger ).toString();
    }
}
```

Given the string bridge defined in Example 4.20, "Custom `StringBridge` implementation", any property or field can use this bridge thanks to the `@FieldBridge` annotation:

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

4.4.3.1.1. Parameterized bridge

Parameters can also be passed to the bridge implementation making it more flexible. Example 4.21, "Passing parameters to your bridge implementation" implements a ParameterizedBridge interface and parameters are passed through the `@FieldBridge` annotation.

Example 4.21. Passing parameters to your bridge implementation

```java
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 );
    }
```

```java
    public String objectToString(Object object) {
        String rawInteger = ((Integer) object).toString();
        if (rawInteger.length() > padding)
            throw new IllegalArgumentException("Number too big to be padded");
        StringBuilder paddedInteger = new StringBuilder( );
        for (int padIndex = rawInteger.length(); padIndex < padding; padIndex++) {
            paddedInteger.append('0');
        }
        return paddedInteger.append(rawInteger).toString();
    }
}
```

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

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

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

4.4.3.1.2. Type aware bridge

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

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

- the class type for class-level bridges

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

4.4.3.1.3. Two-way bridge

If you expect to use your bridge implementation on an id property (ie annotated with `@DocumentId`), you need to use a slightly extended version of `StringBridge` named TwoWayStringBridge. Hibernate Search needs to read the string representation of the identifier and generate the object out of it. There is no difference in the way the `@FieldBridge` annotation is used.

Example 4.22. Implementing a TwoWayStringBridge usable for id properties

```java
public class PaddedIntegerBridge implements TwoWayStringBridge, ParameterizedBridge {
```

```java
    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("Number too big to be padded");
        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);
    }
}
```

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

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

4.4.3.2. FieldBridge

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

Example 4.23. Implementing the FieldBridge interface

```java
/**
 * Store the date in 3 different fields - year, month, day - to ease the creation of RangeQuery per
 * year, month or day (eg get all the elements of December for the last 5 years).
```

```java
 * @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 );
    }
}
```

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

In Example 4.23, "Implementing the FieldBridge interface" the fields are not added directly to `Document`. Instead the addition is delegated to the `LuceneOptions` helper; this helper will apply the options you have selected on `@Field`, like `Store` or `TermVector`, or apply the chosen `@Boost` value. It is especially useful to encapsulate the complexity of `COMPRESS` implementations. Even though it is recommended to delegate to `LuceneOptions` to add fields to the `Document`, nothing stops you from editing the `Document` directly and ignore the LuceneOptions in case you need to.

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

4.4.3.3. ClassBridge

It is sometimes useful to combine more than one property of a given entity and index this combination in a specific way into the Lucene index. The @ClassBridge respectively @ClassBridges annotations can be defined at class level (as opposed to the property level). In this case the custom field bridge implementation receives the entity instance as the value parameter instead of a particular property. Though not shown in Example 4.24, "Implementing a class bridge", @ClassBridge supports the termVector attribute discussed in section Section 4.1.1, "Basic mapping".

Example 4.24. 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.

4.4.4. BridgeProvider: associate a bridge to a given return type

Custom field bridges are very flexible, but it can be tedious and error prone to apply the same custom @FieldBridge annotation every time a property of a given type is present in your domain model. That is what BridgeProviders are for.

Let's imagine that you have a type Currency in your application and that you want to apply your very own CurrencyFieldBridge every time an indexed property returns Currency. You can do it the hard way:

Example 4.25. Applying the same @FieldBridge for a type the hard way

```
@Entity @Indexed
public class User {
    @FieldBridge(impl=CurrencyFieldBridge.class)
    public Currency getDefaultCurrency();

    // ...
}

@Entity @Indexed
public class Account {
    @FieldBridge(impl=CurrencyFieldBridge.class)
    public Currency getCurrency();

    // ...
}

// continue to add @FieldBridge(impl=CurrencyFieldBridge.class) everywhere Currency is
```

Or you can write your own BridgeProvider implementation for Currency.

Example 4.26. Writing a BridgeProvider

```
public class CurrencyBridgeProvider implements BridgeProvider {

    //needs a default no-arg constructor

    @Override
    public FieldBridge provideFieldBridge(BridgeContext bridgeProviderContext) {
        if ( bridgeProviderContext.getReturnType().equals( Currency.class ) ) {
            return CurrencyFieldBridge.INSTANCE;
        }
        return null;
```

```
    }
}
```

```
# service file named META-INF/services/
org.hibernate.search.bridge.spi.BridgeProvider
com.acme.myapps.hibernatesearch.CurrencyBridgeProvider
```

You need to implement BridgeProvider and create a service file named *META-INF/ser-vices/org.hibernate.search.bridge.spi.BridgeProvider*. This file must contain the fully qualified class name(s) of the BridgeProvider implementations. This is the classic Service Loader discovery mechanism.

Now, any indexed property of type Currency will use CurrencyFieldBridge automatically.

Example 4.27. An explicit @FieldBrige is no longer needed

```java
@Entity @Indexed
public class User {

    @Field
    public Currency getDefaultCurrency();

    // ...
}

@Entity @Indexed
public class Account {

    @Field
    public Currency getCurrency();

    // ...
}

//CurrencyFieldBridge is applied automatically everywhere Currency is found on an indexed property
```

A few more things you need to know:

• a BridgeProvider must have a no-arg constructor

• if a BridgeProvider only returns FieldBridge instances if it is meaningful for the calling context. Null otherwise. In our example, the return type must be Currency to be meaningful to our provider.

• if two or more bridge providers return a FieldBridge instance for a given return type, an exception will be raised.

> ### ℹ What is a calling context
>
> A calling context is represented by the BridgeContext object and represents the environment for which we are looking for a bridge. BridgeContext gives access to the return type of the indexed property as well as the ServiceManager which gives access to the ClassLoaderService for everything class loader related.
>
> ```
> ClassLoaderService                                          classLoaderSer
> vice = serviceManager.requestService( ClassLoaderService.class );
> //use the classLoaderService
> serviceManager.releaseService( ClassLoaderService.class );
> ```

4.5. Conditional indexing

> ### ★ Important
>
> This feature is considered experimental. More operation types might be added in the future depending on user feedback.

In some situations, you want to index an entity only when it is in a given state, for example:

- only index blog entries marked as published

- no longer index invoices when they are marked archived

This serves both functional and technical needs. You don't want your blog readers to find your draft entries and filtering them off the query is a bit annoying. Very few of your entities are actually required to be indexed and you want to limit indexing overhead and keep indexes small and fast.

Hibernate Search lets you intercept entity indexing operations and override them. It is quite simple:

- Write an EntityIndexingInterceptor class with your entity state based logic

- Mark the entity as intercepted by this implementation

Let's look at the blog example at Example 4.28, "Index blog entries only when they are published and remove them when they are in a different state"

Example 4.28. Index blog entries only when they are published and remove them when they are in a different state

```
/**
 * Only index blog when it is in published state
 *
 * @author Emmanuel Bernard <emmanuel@hibernate.org>
```

```
 */
public class IndexWhenPublishedInterceptor implements EntityIndexingInterceptor<Blog> {
    @Override
    public IndexingOverride onAdd(Blog entity) {
        if (entity.getStatus() == BlogStatus.PUBLISHED) {
            return IndexingOverride.APPLY_DEFAULT;
        }
        return IndexingOverride.SKIP;
    }

    @Override
    public IndexingOverride onUpdate(Blog entity) {
        if (entity.getStatus() == BlogStatus.PUBLISHED) {
            return IndexingOverride.UPDATE;
        }
        return IndexingOverride.REMOVE;
    }

    @Override
    public IndexingOverride onDelete(Blog entity) {
        return IndexingOverride.APPLY_DEFAULT;
    }

    @Override
    public IndexingOverride onCollectionUpdate(Blog entity) {
        return onUpdate(entity);
    }
}
```

```
@Entity
@Indexed(interceptor=IndexWhenPublishedInterceptor.class)
public class Blog {
    @Id
    @GeneratedValue
    public Integer getId() { return id; }
    public void setId(Integer id) {  this.id = id; }
    private Integer id;

    @Field
    public String getTitle() { return title; }
    public void setTitle(String title) {  this.title = title; }
    private String title;

    public BlogStatus getStatus() { return status; }
    public void setStatus(BlogStatus status) {  this.status = status; }
    private BlogStatus status;

    // ...
}
```

We mark the Blog entity with @Indexed.interceptor. As you can see, IndexWhenPublishedInterceptor implements EntityIndexingInterceptor and accepts Blog entities (it could have accepted super classes as well - for example Object if you create a generic interceptor.

You can react to several planned indexing events:

- when an entity is added to your datastore

- when an entity is updated in your datastore

- when an entity is deleted from your datastore

- when a collection own by this entity is updated in your datastore

For each occurring event you can respond with one of the following actions:

- `APPLY_DEFAULT`: that's the basic operation that lets Hibernate Search update the index as expected - creating, updating or removing the document

- `SKIP`: ask Hibernate Search to not do anything to the index for this event - data will not be created, updated or removed from the index in any way

- `REMOVE`: ask Hibernate Search to remove indexing data about this entity - you can safely ask for `REMOVE` even if the entity has not yet been indexed

- `UPDATE`: ask Hibernate Search to either index or update the index for this entity - it is safe to ask for `UPDATE` even if the entity has never been indexed

> ### Note
>
> Be careful, not every combination makes sense: for example, asking to `UPDATE` the index upon onDelete. Note that you could ask for `SKIP` in this situation if saving indexing time is critical for you. That's rarely the case though.

By default, no interceptor is applied on an entity. You have to explicitly define an interceptor via the @Indexed annotation (see Section 4.1.1.1, "@Indexed") or programmatically (see Section 4.7, "Programmatic API"). This class and all its subclasses will then be intercepted. You can stop or change the interceptor used in a subclass by overriding @Indexed.interceptor. Hibernate Search provides DontInterceptEntityInterceptor which will explicitly not intercept any call. This is useful to reset interception within a class hierarchy.

> ### Note
>
> Dirty checking optimization is disabled when interceptors are used. Dirty checking optimization does check what has changed in an entity and only triggers an index update if indexed properties are changed. The reason is simple, your interceptor might depend on a non indexed property which would be ignored by this optimization.

> **Warning**
>
> An `EntityIndexingInterceptor` can never override an explicit indexing operation such as `index(T)`, `purge(T, id)` or `purgeAll(class)`.

4.6. Providing your own id

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

4.6.1. The ProvidedId annotation

Unlike @DocumentIdwhich is applied on field level, @ProvidedId is used on the class level. Optionally you can specify your own bridge implementation using the bridge property. Also, if you annotate a class with @ProvidedId, your subclasses will also get the annotation - but it is not done by using the java.lang.annotations.@Inherited. Be sure however, to *not* use this annotation with @DocumentId as your system will break.

Example 4.29. Providing your own id

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

4.7. Programmatic API

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

- the same entity is mapped differently depending on deployment needs (customization for clients)

- some automation process requires the dynamic mapping of many entities sharing common traits

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

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

In order to use the programmatic model you must first construct a SearchMapping object which you can do in two ways:

- directly

- via a factory

You can pass the SearchMapping object directly via the property key `hibernate.search.model_mapping` or the constant Environment.MODEL_MAPPING. Use the Configuration API or the Map passed to the JPA Persistence bootstrap methods.

Example 4.30. Programmatic mapping

```
SearchMapping mapping = new SearchMapping();
// ... configure mapping
Configuration config = new Configuration();
config.getProperties().put( Environment.MODEL_MAPPING, mapping );
SessionFactory sf = config.buildSessionFactory();
```

Example 4.31. Programmatic mapping with JPA

```
SearchMapping mapping = new SearchMapping();
// ... configure mapping
Map props = new HashMap();
props.put( Environment.MODEL_MAPPING, mapping );
EntityManagerFactory emf = Persistence.createEntityManagerFactory( "userPU", props );
```

Alternatively, you can create a factory class (ie hosting a method annotated with @Factory) whose factory method returns the SearchMapping object. The factory class must have a no-arg constructor and its fully qualified class name is passed to the property key `hibernate.search.model_mapping` or its type-safe representation Environment.MODEL_MAPPING. This approach is useful when you do not necessarily control the bootstrap process like in a Java EE, CDI or Spring Framework container.

Example 4.32. Use a mapping factory

```
public class MyAppSearchMappingFactory {
    @Factory
    public SearchMapping getSearchMapping() {
```

```
        SearchMapping mapping = new SearchMapping();
        mapping
                .analyzerDef( "ngram", StandardTokenizerFactory.class )
                    .filter( LowerCaseFilterFactory.class )
                    .filter( NGramFilterFactory.class )
                        .param( "minGramSize", "3" )
                        .param( "maxGramSize", "3" );
        return mapping;
    }
}
```

```xml
<persistence ...>
    <persistence-unit name="users">
        ...
        <properties>
            <property name="hibernate.search.model_mapping"
                    value="com.acme.MyAppSearchMappingFactory"/>
        </properties>
    </persistence-unit>
</persistence>
```

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

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

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

4.7.1. Mapping an entity as indexable

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

Example 4.33. Marking an entity indexable

```
SearchMapping mapping = new SearchMapping();

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

```
                .interceptor(IndexWhenPublishedInterceptor.class); //optional

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

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

Example 4.34. Annotation example of indexing entity

```
@Entity
@Indexed(index="Address_Index", interceptor=IndexWhenPublishedInterceptor.class)
public class Address {
   // ...
}
```

4.7.2. Adding DocumentId to indexed entity

To set a property as a document id:

Example 4.35. Enabling document id with programmatic model

```
SearchMapping mapping = new SearchMapping();

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

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

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

Example 4.36. DocumentId annotation definition

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

```
 // ...
}
```

4.7.3. Defining analyzers

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

Example 4.37. Defining analyzers using programmatic model

```
SearchMapping mapping = new SearchMapping();

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

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

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

Example 4.38. Analyzer definition using annotation

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

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

  @AnalyzerDef(name = "de",
    tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
    filters = {
      @TokenFilterDef(factory = LowerCaseFilterFactory.class),
      @TokenFilterDef(factory = GermanStemFilterFactory.class)
  })

})
public class Address {
   // ...
}
```

4.7.4. Defining full text filter definitions

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

Example 4.39. Defining full text definition programmatically

```
SearchMapping mapping = new SearchMapping();

mapping
    .analyzerDef( "en", StandardTokenizerFactory.class )
        .filter( LowerCaseFilterFactory.class )
        .filter( EnglishPorterFilterFactory.class )
    .fullTextFilterDef("security", SecurityFilterFactory.class)
            .cache(FilterCacheModeType.INSTANCE_ONLY)
    .entity(Address.class)
        .indexed()
        .property("addressId", ElementType.METHOD)
            .documentId()
                .name("id")
        .property("street1", ElementType.METHOD)
            .field()
                .analyzer("en")
                .store(Store.YES)
            .field()
                .name("address_data")
                .analyzer("en")
                .store(Store.NO);

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

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

Example 4.40. Using annotation to define full text filter definition

```
@Entity
@Indexed
@AnalyzerDefs({
  @AnalyzerDef(name = "en",
    tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
    filters = {
      @TokenFilterDef(factory = LowerCaseFilterFactory.class),
      @TokenFilterDef(factory = EnglishPorterFilterFactory.class)
   })
})
@FullTextFilterDefs({
 @FullTextFilterDef(name = "security", impl = SecurityFilterFactory.class, cache = FilterCacheModeType.INSTANCE_ON
})
public class Address {

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

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

 // ...

}
```

4.7.5. Defining fields for indexing

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

Example 4.41. Indexing fields using programmatic API

```
SearchMapping mapping = new SearchMapping();

mapping
    .analyzerDef( "en", StandardTokenizerFactory.class )
        .filter( LowerCaseFilterFactory.class )
        .filter( EnglishPorterFilterFactory.class )
    .entity(Address.class).indexed()
        .property("addressId", ElementType.METHOD)
            .documentId()
                .name("id")
        .property("street1", ElementType.METHOD)
            .field()
                .analyzer("en")
```

```
                .store(Store.YES)
            .field()
                .name("address_data")
                .analyzer("en");

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

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

Example 4.42. Indexing fields using annotation

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

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

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

 // ...
}
```

Note

When using a programmatic mapping for a given type X, you can only refer to fields defined on X. Fields or methods inherited from a super type are not configurable. In case you need to configure a super class property, you need to either override the property in X or create a programmatic mapping for the super class. This mimics the usage of annotations where you cannot annotate a field or method of a super class either, unless it is redefined in the given type.

4.7.6. Programmatically defining embedded entities

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

Example 4.43. Programmatically defining embedded entities

```
SearchMapping mapping = new SearchMapping();

mapping
    .entity(ProductCatalog.class)
        .indexed()
        .property("catalogId", ElementType.METHOD)
            .documentId()
                .name("id")
        .property("title", ElementType.METHOD)
            .field()
                .index(Index.YES)
                .store(Store.NO)
        .property("description", ElementType.METHOD)
            .field()
                .index(Index.YES)
                .store(Store.NO)
        .property("items", ElementType.METHOD)
            .indexEmbedded()
                .prefix("catalog.items"); //optional

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

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

Example 4.44. Using @IndexedEmbedded

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

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

 @Field
 public String getDescription();

 @OneToMany(fetch = FetchType.LAZY)
 @IndexColumn(name = "list_position")
 @Cascade(org.hibernate.annotations.CascadeType.ALL)
```

```
@IndexedEmbedded(prefix="catalog.items")
public List<Item> getItems() {...}

// ...
}
```

4.7.7. Contained In definition

@ContainedIn can be defined as seen in the example below:

Example 4.45. Programmatically defining ContainedIn

```
SearchMapping mapping = new SearchMapping();

mapping
    .entity(ProductCatalog.class)
        .indexed()
        .property("catalogId", ElementType.METHOD)
            .documentId()
        .property("title", ElementType.METHOD)
            .field()
        .property("description", ElementType.METHOD)
            .field()
        .property("items", ElementType.METHOD)
            .indexEmbedded()

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

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

This is equivalent to defining @ContainedIn in your entity:

Example 4.46. Annotation approach for ContainedIn

```
@Entity
@Indexed
public class ProductCatalog {

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

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

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

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

 // ...
}
```

```
@Entity
public class Item {

 @Id
 @GeneratedValue
 private Long itemId;

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

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

 // ...
}
```

4.7.8. Date/Calendar Bridge

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

Example 4.47. Programmatic model for defining calendar/date bridge

```
SearchMapping mapping = new SearchMapping();

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

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

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

Example 4.48. @CalendarBridge and @DateBridge definition

```
@Entity
@Indexed
public class Address {

 @Id
 @GeneratedValue
 @DocumentId
 private Long addressId;

 @Field
 private String address1;

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

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

 // ...
}
```

4.7.9. Declaring bridges

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

Example 4.49. Declaring field bridges programmatically

```
SearchMapping mapping = new SearchMapping();

mapping
    .entity(Address.class)
        .indexed()
        .property("addressId", ElementType.FIELD)
            .documentId()
        .property("street1", ElementType.FIELD)
            .field()
            .field()
                .name("street1_abridged")
                .bridge( ConcatStringBridge.class )
                    .param( "size", "4" );

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

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

Example 4.50. Declaring field bridges using annotation

```
@Entity
@Indexed
public class Address {

 @Id
 @GeneratedValue
 @DocumentId(name="id")
 private Long addressId;

 @Fields({
      @Field,
      @Field(name="street1_abridged",
             bridge = @FieldBridge( impl = ConcatStringBridge.class,
             params = @Parameter( name="size", value="4" ))
 })
 private String address1;


 // ...
}
```

4.7.10. Mapping class bridge

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

Example 4.51. Defining class bridges using API

```
SearchMapping mapping = new SearchMapping();

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

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

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

Example 4.52. Using @ClassBridge

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

4.7.11. Mapping dynamic boost

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

Example 4.53. DynamicBoost mapping using programmatic model

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

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

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

Example 4.54. Using the @DynamicBoost

```
@Entity
@Indexed
@DynamicBoost(impl = CustomBoostStrategy.class)
public class DynamicBoostedDescriptionLibrary {
```

```java
@Id
@GeneratedValue
@DocumentId
private int id;

private float dynScore;

@Field(store = Store.YES)
@DynamicBoost(impl = CustomFieldBoostStrategy.class)
private String name;

public DynamicBoostedDescriptionLibrary() {
 dynScore = 1.0f;
}

// ...
}
```

Chapter 5. Querying

The second most important capability of Hibernate Search is the ability to execute Lucene queries and retrieve entities managed by a Hibernate session. The search provides the power of Lucene without leaving the Hibernate paradigm, giving another dimension to the Hibernate classic search mechanisms (HQL, Criteria query, native SQL query).

Preparing and executing a query consists of four simple steps:

- Creating a `FullTextSession`

- Creating a Lucene query either via the Hibernate Search query DSL (recommended) or by utilizing the Lucene query API

- Wrapping the Lucene query using an `org.hibernate.Query`

- Executing the search by calling for example `list()` or `scroll()`

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

Example 5.1. Creating a FullTextSession

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

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

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

```
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
```

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

Example 5.2. Creating a Lucene query via the QueryParser

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

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

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

In case you are using the Java Persistence APIs of Hibernate, the same extensions exist:

Example 5.3. Creating a Search query using the JPA API

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

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

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

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

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

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

5.1. Building queries

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

5.1.1. Building a Lucene query using the Lucene API

Using the Lucene API, you have several options. You can use the query parser (fine for simple queries) or the Lucene programmatic API (for more complex use cases). It is out of the scope of this documentation on how to exactly build a Lucene query. Please refer to the online Lucene documentation or get hold of a copy of Lucene In Action or Hibernate Search in Action.

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

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

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

- it has meaningful method names making a succession of operations reads almost like English

- it limits the options offered to what makes sense in a given context (thanks to strong typing and IDE auto-completion).

- it often uses the chaining method pattern

- it's easy to use and even easier to read

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

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

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

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

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

5.1.2.1. Keyword queries

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

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

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

- The value storm is passed through the `history` FieldBridge: it does not matter here but you will see that it's quite handy when dealing with numbers or dates.

- The field bridge value is then passed to the analyzer used to index the field `history`. This ensures that the query uses the same term transformation than the indexing (lower case, n-gram, stemming and so on). If the analyzing process generates several terms for a given word, a boolean query is used with the `SHOULD` logic (roughly an `OR` logic).

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

```
@AnalyzerDef(name = "ngram",
  tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class ),
  filters = {
    @TokenFilterDef(factory = StandardFilterFactory.class),
    @TokenFilterDef(factory = LowerCaseFilterFactory.class),
    @TokenFilterDef(factory = StopFilterFactory.class),
```

```
    @TokenFilterDef(factory = NGramFilterFactory.class,
      params = {
        @Parameter(name = "minGramSize", value = "3"),
        @Parameter(name = "maxGramSize", value = "3") } )
  }
)
@Entity
@Indexed
public class Myth {
  @Field(analyzer=@Analyzer(definition="ngram")
  public String getName() { return name; }
  public String setName(String name) { this.name = name; }
  private String name;

  ...
}

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

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

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

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

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

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

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

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

```
Query luceneQuery = mythQB.keyword()
```

```
    .onField("history")
    .andField("name")
      .boostedTo(5)
    .andField("description")
    .matching("storm")
    .createQuery();
```

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

5.1.2.2. Fuzzy queries

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

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

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

5.1.2.3. Wildcard queries

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

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

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

5.1.2.4. Phrase queries

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

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

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

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

5.1.2.5. Range queries

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

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

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

5.1.2.6. Spatial (or geolocation) queries

This set of queries has its own chapter, check out Chapter 9, *Spatial*.

5.1.2.7. More Like This queries

> **Important**
>
> This feature is considered experimental.

Have you ever looked at an article or document and thought: "I want to find more like this"? Have you ever appreciated an e-commerce website that gives you similar articles to the one you are exploring?

More Like This queries are achieving just that. You feed it an entity (or its identifier) and Hibernate Search returns the list of entities that are similar.

> **How does it work?**
>
> For each (selected) field of the targeted entity, we look at the most meaningful terms. Then we create a query matching the most meaningful terms per field. This is a slight variation compared to the original Lucene `MoreLikeThisQuery` implementation.

The query DSL API should be self explaining. Let's look at some usage examples.

```
QueryBuilder qb = fullTextSession.getSearchFactory()
        .buildQueryBuilder()
        .forEntity( Coffee.class )
        .get();

Query mltQuery = qb
        .moreLikeThis()
            .comparingAllFields()
            .toEntityWithId( coffeeId )
            .createQuery();
List<Object[]> results = (List<Object[]>) fullTextSession
        .createFullTextQuery( mltQuery, Coffee.class )
        .setProjection( ProjectionConstants.THIS, ProjectionConstants.SCORE )
        .list();
```

This first example takes the id of an Coffee entity and finds the matching coffees across all fields. To be fair, this is not across *all* fields. To be included in the More Like This query, fields need to store term vectors or the actual field value. Id fields (of the root entity as well as embedded entities) and numeric fields are excluded. The latter exclusion might change in future versions.

Looking at the `Coffee` class, the following fields are considered: `name` as it is stored, `description` as it stores the term vector. `id` and `internalDescription` are excluded.

```
@Entity @Indexed
public class Coffee {

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

    @Field(termVector = TermVector.NO, store = Store.YES)
    public String getName() { return name; }

    @Field(termVector = TermVector.YES)
    public String getSummary() { return summary; }

    @Column(length = 2000)
    @Field(termVector = TermVector.YES)
    public String getDescription() { return description; }

    public int getIntensity() { return intensity; }

    // Not stored nor term vector, i.e. cannot be used for More Like This
    @Field
    public String getInternalDescription() { return internalDescription; }

    // ...
}
```

In the example above we used projection to retrieve the relative score of each element. We might use the score to only display the results for which the score is high enough.

> **Tip**
>
> For best performance and best results, store the term vectors for the fields you want to include in a More Like This query.

Often, you are only interested in a few key fields to find similar entities. Plus some fields are more important than others and should be boosted.

```
Query mltQuery = qb
        .moreLikeThis()
            .comparingField("summary").boostedTo(10f)
            .andField("description")
            .toEntityWithId( coffeeId )
            .createQuery();
```

In this example, we look for similar entities by summary and description. But similar summaries are more important than similar descriptions. This is a critical tool to make More Like This meaningful for your data set.

Instead of providing the entity id, you can pass the full entity object. If the entity contains the identifier, we will use it to find the term vectors or field values. This means that we will compare

the entity state as stored in the Lucene index. If the identifier cannot be retrieved (for example if the entity has not been persisted yet), we will look at each of the entity properties to find the most meaningful terms. The latter is slower and won't give the best results - avoid it if possible.

Here is how you pass the entity instance you want to compare with:

```
Coffee coffee = ...; //managed entity from somewhere

Query mltQuery = qb
        .moreLikeThis()
            .comparingField("summary").boostedTo(10f)
            .andField("description")
            .toEntity( coffee )
            .createQuery();
```

> ### Note
>
> By default, the results contain at the top the entity you are comparing with. This is particularly useful to compare relative scores. If you don't need it, you can exclude it.

```
Query mltQuery = qb
        .moreLikeThis()
            .excludeEntityUsedForComparison()
            .comparingField("summary").boostedTo(10f)
            .andField("description")
            .toEntity( coffee )
            .createQuery();
```

You can ask Hibernate Search to give a higher score to the very similar entities and downgrade the score of mildly similar entities. We do that by boosting each meaningful terms by their individual overall score. Start with a boost factor of 1 and adjust from there.

```
Query mltQuery = qb
        .moreLikeThis()
            .favorSignificantTermsWithFactor(1f)
            .comparingField("summary").boostedTo(10f)
            .andField("description")
            .toEntity( coffee )
            .createQuery();
```

Remember, more like this is a very subjective meaning and will vary depending on your data and the rules of your domain. With the various options offered, Hibernate Search arms you with the tools to adjust this weapon. Make sure to continuously test the results against your data set.

5.1.2.8. Combining queries

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

- `SHOULD`: the query 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 sub-queries can be any Lucene query including a boolean query itself. Let's look at a few examples:

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

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

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

5.1.2.9. Query options

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

- `boostedTo` (on query type and on field): boost the whole query or the specific field to a given factor

- `withConstantScore` (on query): all results matching the query have a constant score equals to the boost

- `filteredBy(Filter)` (on query): filter query results using the Filter instance

- `ignoreAnalyzer` (on field): ignore the analyzer when processing this field

- `ignoreFieldBridge` (on field): ignore field bridge when processing this field

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

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

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

5.1.3. Building a Hibernate Search query

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

5.1.3.1. Generality

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

Example 5.4. Wrapping a Lucene query into a Hibernate Query

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

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

Example 5.5. Filtering the search result by entity type

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

// or

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

In Example 5.5, "Filtering the search result by entity type" the first example returns only matching `Customer` instances, the second returns matching `Actor` and `Item` instances. The type restriction is fully polymorphic which means that if there are two indexed subclasses `Salesman` and `Customer` of the baseclass `Person`, it is possible to just specify `Person.class` in order to filter on result types.

5.1.3.2. Pagination

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

Example 5.6. Defining pagination for a search query

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

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

5.1.3.3. Sorting

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

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

```
org.hibernate.search.FullTextQuery query = s.createFullTextQuery( query, Book.class );
```

```
org.apache.lucene.search.Sort sort = new Sort(
    new SortField("title", SortField.STRING));
query.setSort(sort);
List results = query.list();
```

> **Tip**
>
> Be aware that fields used for sorting must not be tokenized (see Section 4.1.1.2, "@Field").

5.1.3.4. Fetching strategy

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

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

Example 5.8. Specifying FetchMode on a query

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

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

When defining a criteria query, it is not necessary to restrict the returned entity types when creating the Hibernate Search query from the full text session: the type is guessed from the criteria query itself.

> **Important**
>
> Only fetch mode can be adjusted, refrain from applying any other restriction. While it is known to work as of Hibernate Search 4, using restriction (ie a where clause) on your Criteria query should be avoided when possible. getResultSize() will throw a SearchException if used in conjunction with a Criteria with restriction.

> **Important**
>
> You cannot use setCriteriaQuery if more than one entity type is expected to be returned.

5.1.3.5. Projection

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

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

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

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

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

- the properties projected must use a `FieldBridge` implementing org.hibernate.search.bridge.TwoWayFieldBridge or `org.hibernate.search.bridge.TwoWayStringBridge`, the latter being the simpler version.

> **Note**
>
> All Hibernate Search built-in types are two-way.

- you can only project simple properties of the indexed entity or its embedded associations. This means you cannot project a whole embedded entity.

- projection does not work on collections or maps which are indexed via `@IndexedEmbedded`

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

Example 5.10. Using projection in order to retrieve meta data

```
org.hibernate.search.FullTextQuery query =
```

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

You can mix and match regular fields and projection constants. Here is the list of the available constants:

- `FullTextQuery.THIS`: returns the initialized and managed entity (as a non projected query would have done).

- `FullTextQuery.DOCUMENT`: returns the Lucene Document related to the object projected.

- `FullTextQuery.OBJECT_CLASS`: returns the class of the indexed entity.

- `FullTextQuery.SCORE`: returns the document score in the query. Scores are handy to compare one result against an other for a given query but are useless when comparing the result of different queries.

- `FullTextQuery.ID`: the id property value of the projected object.

- `FullTextQuery.DOCUMENT_ID`: the Lucene document id. Careful, Lucene document id can change overtime between two different IndexReader opening.

- `FullTextQuery.EXPLANATION`: returns the Lucene Explanation object for the matching object/document in the given query. Do not use if you retrieve a lot of data. Running explanation typically is as costly as running the whole Lucene query per matching element. Make sure you use projection!

5.1.3.6. Customizing object initialization strategies

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

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

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

```
FullTextQuery query = session.createFullTextQuery(luceneQuery, User.class);
query.initializeObjectWith(
```

```
    ObjectLookupMethod.SECOND_LEVEL_CACHE,
    DatabaseRetrievalMethod.QUERY
);
```

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

- `ObjectLookupMethod.PERSISTENCE_CONTEXT`: useful if most of the matching entities are already in the persistence context (ie loaded in the Session or EntityManager)

- `ObjectLookupMethod.SECOND_LEVEL_CACHE`: check first the persistence context and then the second-level cache.

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

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

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

- `QUERY` (default): use a (set of) queries to load several objects in batch. This is usually the best approach.

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

The defaults for both methods, the object lookup as well as the database retrieval can also be configured via configuration properties. This way you don't have to specify your preferred methods on each query cre-

ation. The property names are `hibernate.search.query.object_lookup_method` and `hibernate.search.query.database_retrieval_method` respectively. As value use the name of the method (upper- or lowercase). For example:

Example 5.12. Setting object lookup and database retrieval methods via configuration properties

```
hibernate.search.query.object_lookup_method = second_level_cache
hibernate.search.query.database_retrieval_method = query
```

5.1.3.7. Limiting the time of a query

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

- raise an exception when the limit is reached

- limit to the number of results retrieved when the time limit is raised

5.1.3.7.1. Raise an exception on time limit

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

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

Example 5.13. Defining a timeout in query execution

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

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

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

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

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

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

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

Example 5.14. Defining a timeout in query execution

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

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

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

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

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

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

> **Warning**
>
> This approach is not compatible with the setTimeout approach.

To define this soft limit, use the following approach

Example 5.15. Defining a time limit in query execution

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

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

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

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

Example 5.16. Determines when a query returns partial results

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

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

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

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

5.2. Retrieving the results

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

5.2.1. Performance considerations

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

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

> **Important**
>
> Pagination is preferred over scrolling.

5.2.2. Result size

It is 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)

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

Example 5.17. 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 an approximation if the index is not fully up-to-date with the database (asynchronous cluster for example).

5.2.3. ResultTransformer

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

Example 5.18. Using ResultTransformer in conjunction with projections

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

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

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

5.2.4. Understanding results

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

- Use the fullTextQuery.explain(int) method

- Use projection

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

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

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

Example 5.19. Retrieving the Lucene Explanation object using projection

```
FullTextQuery ftQuery = s.createFullTextQuery( luceneQuery, Dvd.class )
        .setProjection(
```

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

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

5.3. Filters

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

- security

- temporal data (eg. view only last month's data)

- population filter (eg. search limited to a given category)

- and many more

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

Example 5.20. Enabling fulltext filters for a given query

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

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

Declaring filters is done through the `@FullTextFilterDef` annotation. You can use `@Full-TextFilterDef` or `@FullTextFilterDefs` on any: *`@Indexed` entity regardless of the query the filter is later applied to * Parent class of an `@Indexed` entity * package-info.java of a package containing an `@Indexed` entity

This implies that filter definitions are global and their names must be unique. A `SearchException` is thrown in case two different `@FullTextFilterDef` annotations with the same name are defined. Each named filter has to specify its actual filter implementation.

Example 5.21. Defining and implementing a Filter

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

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

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

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

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

Example 5.22. Creating a filter using the factory pattern

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

public class BestDriversFilterFactory {

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

Hibernate Search will look for a `@Factory` annotated method and use it to build the filter instance. The factory must have a no-arg constructor.

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

Example 5.23. Passing parameters to a defined filter

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

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

Example 5.24. Using parameters in the actual filter implementation

```
public class SecurityFilterFactory {
    private Integer level;

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

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

Filters will be cached once created, based on all their parameter names and values. Caching happens using a combination of hard and soft references to allow disposal of memory when needed. The hard reference cache keeps track of the most recently used filters and transforms the ones least used to SoftReferences when needed. Once the limit of the hard reference cache is reached additional filters are cached as SoftReferences. To adjust the size of the hard reference cache, use `hibernate.search.filter.cache_strategy.size` (defaults to 128). For advanced use of filter caching, you can implement your own FilterCachingStrategy. The classname is defined by `hibernate.search.filter.cache_strategy`.

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

tially on a different set of Documents (either from a different index or simply because the index has changed), hence the cached DocIdSet has to be recomputed.

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

Value	Definition
FilterCacheModeType.NONE	No filter instance and no result is cached by Hibernate Search. For every filter call, a new filter instance is created. This setting might be useful for rapidly changing data sets or heavily memory constrained environments.
FilterCacheModeType.INSTANCE_ONLY	The filter instance is cached and reused across concurrent Filter.getDocIdSet() calls. DocIdSet results are not cached. This setting is useful when a filter uses its own specific caching mechanism or the filter results change dynamically due to application specific events making DocIdSet caching in both cases unnecessary.
FilterCacheModeType.INSTANCE_AND_DOCIDSETRESULTS	Both the filter instance and the DocIdSet results are cached. This is the default value.

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

- the system does not update the targeted entity index often (in other words, the IndexReader is reused a lot)

- the Filter's DocIdSet is expensive to compute (compared to the time spent to execute the query)

5.3.1. 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.

```java
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 5.3, "Filters") which also filters Lucene results after the query, you can make use of a special filter that will only be passed to the sharding strategy and otherwise ignored for the rest of the query. Simply use the `ShardSensitiveOnlyFilter` class when declaring your filter.

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

```
FullTextQuery query = ftEm.createFullTextQuery(luceneQuery, Customer.class);
query.enableFulltextFilter("customer").setParameter("CustomerID", 5);
@SuppressWarnings("unchecked")
List<Customer> results = query.getResultList();
```

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

5.4. Faceting

Faceted search [http://en.wikipedia.org/wiki/Faceted_search] is a technique which allows to divide the results of a query into multiple categories. This categorization includes the calculation of hit counts for each category and the ability to further restrict search results based on these facets (categories). Figure 5.1, "Facets Example on Amazon" shows a faceting example. The search for 'Hibernate 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 facet of this search. Another one is for example the average customer review rating.

Figure 5.1. Facets Example on Amazon

In Hibernate Search the classes `QueryBuilder` and `FullTextQuery` are the entry point to the faceting API. The former allows to create faceting requests whereas the latter gives access to the so called `FacetManager`. With the help of the `FacetManager` faceting requests can be applied on a query and selected facets can be added to an existing query in order to refine search results.

The following sections will describe the faceting process in more detail. The examples will use the entity `Cd` as shown in Example 5.25, "Example entity for faceting":

Example 5.25. Example entity for faceting

```
@Entity
@Indexed
public class Cd {

    @Id
    @GeneratedValue
    private int id;

    @Field,
    private String name;

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

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

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


// setter/getter
// ...
```

In order to facet on a given indexed field, the field needs to be configured with the `@Facet` annotation. Also, the field itself cannot be analyzed.

`@Facet` contains a `name` and `forField` parameter. The `name` is arbitrary and used to identify the facet. Per default it matches the field name it belongs to. `forField` is relevant in case the property is mapped to multiple fields using `@Fields` (see also Section 4.1.2, "Mapping properties multiple times"). In this case `forField` can be used to identify the index field to which it applies. Mirroring `@Fields` there also exists a `@Facets` annotation in case multiple fields need to be targeted by faceting.

Last but not least, `@Facet` contains a `encoding` parameter. Usually, Hibernate Search automatically selects the encoding:

- String fields are encoded as `FacetEncodingType.STRING`

- `byte, short, int, long` (including corresponding wrapper types) and `Date` as `FacetEncodingType.LONG`

- and `float` and `double` (including corresponding wrapper types) as FacetEncodingType.DOUBLE`

In some cases it can make sense, however, to explicitly set the encoding. Discrete faceting requests for example only work for string encoded facets. In order to use a discrete facet for numbers the encoding must be explicitly set to `FacetEncodingType.STRING`.

> **Note**
>
> Pre Hibernate Search 5.2 there was no need to explicitly use a `@Facet` annotation. In 5.2 it became necessary in order to use Lucene's native faceting API.

5.4.1. Creating a faceting request

The first step towards a faceted search is to create the `FacetingRequest`. Currently two types of faceting requests are supported. The first type is called *discrete faceting* and the second type *range faceting* request.

5.4.1.1. Discrete faceting request

In the case of a discrete faceting request, you start with giving the request a unique name. This name will later be used to retrieve the facet values (see Section 5.4.4, "Interpreting a Facet result"). Then you need to specify on which index field you want to categorize on and which faceting options to apply. An example for a discrete faceting request can be seen in Example 5.26, "Creating a discrete faceting request":

Example 5.26. Creating a discrete faceting request

```
QueryBuilder builder = fullTextSession.getSearchFactory()
    .buildQueryBuilder().forEntity(Cd.class).get();

FacetingRequest labelFacetingRequest = builder.facet()
    .name("labelFacetRequest")
    .onField("label")
    .discrete()
    .orderedBy(FacetSortOrder.COUNT_DESC)
    .includeZeroCounts(false)
    .maxFacetCount(3)
    .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. Parameters `orderedBy`, `includeZeroCounts` and `maxFacetCount` are optional and can be applied on any faceting request. Parameter `orderedBy` 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. Parameter `in-`

cludeZeroCount determines whether facets with a count of 0 will be included in the result (by default they are not) and maxFacetCount allows to limit the maximum amount of facets returned.

> **Note**
>
> There are several preconditions an indexed field has to meet in order to categorize (facet) on it:
>
> - The indexed property must be of type String, Date or of the numeric type byte, shirt, int, long, double or float (or their respective Java wrapper types).
>
> - The property has to be indexed with Analyze.NO.
>
> - *null* values should be avoided.
>
> When you need conflicting options, we suggest to index the property twice and use the appropriate field depending on the use case:
>
> ```
> @Fields({
>   @Field(name="price"),
>   @Field(name="price_facet",
>          analyze=Analyze.NO,
>          bridge=@FieldBridge(impl = IntegerBridge.class))
> })
> private int price;
> ```

5.4.1.2. Creating a range faceting request

The creation of a range faceting request is similar. We also start with a name for the request and the field to facet on. Then we have to specify ranges for the field values. A range faceting request can be seen in Example 5.27, "Creating a range faceting request". There, three different price ranges are specified. below and above can only be specified once, but you can specify as many from - to ranges as you want. For each range boundary you can also specify via excludeLimit whether it is included into the range or not.

Example 5.27. Creating a range faceting request

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

```
    .above(1500).excludeLimit()
    .createFacetingRequest();
```

5.4.2. Setting the facet sort order

The result of applying a faceting request is a list of `Facet` instances as seen in Example 5.28, "Applying a faceting request". The order within the list is given by the `FacetSortOrder` parameter specified via `orderedBy` when creating the faceting request. The default value is `FacetSortOrder.COUNT_DESC`, meaning facets are ordered by their count in descending order (highest count first). Other values are `COUNT_ASC`, `FIELD_VALUE` and `RANGE_DEFINITION_ORDER`. `COUNT_ASC` returns the facets in ascending count order whereas `FIELD_VALUE` will return them in alphabetical order of the facet/category value (see Section 5.4.4, "Interpreting a Facet result"). `RANGE_DEFINITION_ORDER` only applies for range faceting request and returns the facets in the same order in which the ranges are defined. For Example 5.27, "Creating a range faceting request" this would mean the facet for the range of below 1000 would be returned first, followed by the facet for the range 1001 to 1500 and finally the facet for above 1500.

5.4.3. Applying a faceting request

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

Example 5.28. Applying a faceting request

```
// create a fulltext query
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 need to enable the faceting request before you execute the query. You do that via `facetManager.enableFaceting(<facetName>)`. You can enable as many faceting requests as you like. Then you execute the query and retrieve the facet results for a given request via `facetManager.getFacets(<facetname>)`. For each request you will get a list of `Facet` instances. Facet requests stay active and get applied to the fulltext query until they are either explicitly disabled via `disableFaceting(<facetName>)` or the query is `discarded`.

5.4.4. Interpreting a Facet result

Each facet request results in a list of `Facet` instances. Each instance represents one facet/category value. In the CD example (Example 5.26, "Creating a discrete faceting request") where we want to categorize on the CD labels, there would for example be a `Facet` for each of the record labels Universal, Sony and Warner. Example 5.29, "Facet API" shows the API of Facet.

Example 5.29. Facet API

```java
public interface Facet {
    /**
     * @return the faceting name this {@code Facet} belongs to.
     *
     * @see org.hibernate.search.query.facet.FacetingRequest#getFacetingName()
     */
    String getFacetingName();

    /**
     * Return the {@code Document} field name this facet is targeting.
     * The field needs to be indexed with {@code Analyze.NO}.
     *
     * @return the {@code Document} field name this facet is targeting.
     */
    String getFieldName();

    /**
     * @return the value of this facet. In case of a discrete facet it is the actual
     *         {@code Document} field value. In case of a range query the value is a
     *         string representation of the range.
     */
    String getValue();

    /**
     * @return the facet count.
     */
    int getCount();

    /**
     * @return a Lucene {@link Query} which can be executed to retrieve all
     *         documents matching the value of this facet.
     */
    Query getFacetQuery();
}
```

`getFacetingName()` and `getFieldName()` are returning the facet request name and the targeted document field name as specified by the underlying `FacetRequest`. For example "Example 5.26, "Creating a discrete faceting request"" that would be `labelFacetRequest` and `label` respectively. The interesting information is provided by `getValue()` and `getCount()`. The former is the actual facet/category value, for example a concrete record label like Universal. The latter returns the count for this value. To stick with the example again, the count value tells you how many Cds are released under the Universal label. Last but not least, `getFacetQuery()` returns a Lucene query which can be used to retrieve the entities counted in this facet.

5.4.5. Restricting query results

A common use case for faceting is a "drill-down" functionality which allows you to narrow your original search by applying a given facet on it. To do this, you can apply any of the returned `Facet` instances as additional criteria on your original query via `FacetSelection`. `FacetSelection` is available via the `FacetManager` and allow you to select a facet as query criteria (`selectFacets`), remove a facet restriction (`deselectFacets`), remove all facet restrictions (`clearSelectedFacets`) and retrieve all currently selected facets (`getSelectedFacets`). Example 5.30, "Restricting query results via the application of a `FacetSelection`" shows an example.

Example 5.30. 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
FacetSelection facetSelection = facetManager.getFacetGroup( "priceFaceting" );
facetSelection.selectFacets( facets.get( 0 ) );

// re-execute the query
cds = fullTextQuery.list();
assertTrue(cds.size() == 2);
```

Per default selected facets are combined via disjunction (OR). In case a field has multiple values, like a potential `Cd.artists` association, you can also use conjunction (AND) for the facet selection.

Example 5.31. Using conjunction in `FacetSelection`

```
FacetSelection facetSelection = facetManager.getFacetGroup( "artistsFaceting" );
facetSelection.selectFacets( FacetCombine.AND, facets.get( 0 ), facets.get( 1 ) );
```

5.5. Optimizing the query process

Query performance depends on several criteria:

- the Lucene query itself: read the literature on this subject.

- the number of loaded objects: use pagination and / or index projection (if needed).

- the way Hibernate Search interacts with the Lucene readers: defines the appropriate Section 2.3, "Reader strategy".

- caching frequently extracted values from the index: see Section 5.5.1, "Caching index values: FieldCache".

5.5.1. Caching index values: FieldCache

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

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

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

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

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

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

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

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

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

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

> **Note**
>
> Measure the performance and memory consumption impact after warm-up (executing some queries): enabling Field Caches is likely to improve performance but this is not always the case.

Using a FieldCache has two downsides to consider:

- Memory usage: these caches can be quite memory hungry. Typically the CLASS cache has lower requirements than the ID cache.

- Index warm-up: 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 class type 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 built-in bridges), and all types being loaded in a specific query must use the field name for the id, and have ids of the same type (this is evaluated at each Query execution).

Chapter 6. Manual index changes

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

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

6.1. Adding instances to the index

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

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

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

In case you want to add all instances for a type, or for all indexed types, the recommended approach is to use a MassIndexer: see Section 6.3.2, "Using a MassIndexer" for more details.

The method `FullTextSession.index(T entity)` is considered an explicit indexing operation, so any registered EntityIndexingInterceptor won't be applied in this case. For more information on EntityIndexingInterceptor see Section 4.5, "Conditional indexing".

6.2. Deleting instances from the index

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

Example 6.2. Purging a specific instance of an entity from the index

```
FullTextSession fullTextSession = Search.getFullTextSession(session);
Transaction tx = fullTextSession.beginTransaction();
for (Customer customer : customers) {
    fullTextSession.purge( Customer.class, customer.getId() );
}
```

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

Purging will remove the entity with the given id from the Lucene index but will not touch the data-base.

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

Example 6.3. Purging all instances of an entity from the index

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

As in the previous example, it is suggested to optimize the index after many purge operation to actually free the used space.

As is the case with method `FullTextSession.index(T entity)`, also `purge` and `purgeAll` are considered explicit indexing operations: any registered `EntityIndexingInterceptor` won't be applied. For more information on `EntityIndexingInterceptor` see Section 4.5, "Conditional indexing".

> **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 flush-ToIndexes.

6.3. Rebuilding the whole index

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

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

- Use a MassIndexer.

6.3.1. Using flushToIndexes()

This strategy consists in removing the existing index and then adding all entities back to the index using `FullTextSession.purgeAll()` and `FullTextSession.index()`, however there are some memory and efficiency constraints. For maximum efficiency Hibernate Search batches index operations and executes them at commit time. If you expect to index a lot of data you need to be careful about memory consumption since all documents are kept in a queue until the transaction commit. You can potentially face an `OutOfMemoryException` if you don't empty the queue periodically: to do this you can use `fullTextSession.flushToIndexes()`. Every time `fullTextSession.flushToIndexes()` is called (or if the transaction is committed), the batch queue is processed applying all index changes. Be aware that, once flushed, the changes cannot be rolled back.

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

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

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

6.3.2. Using a MassIndexer

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

Example 6.5. Index rebuilding using a MassIndexer

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

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

> **Warning**
>
> During the progress of a MassIndexer the content of the index is undefined! If a query is performed while the MassIndexer is working most likely some results will be missing.

Example 6.6. 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 finally output a Lucene document. In this conversion process these threads are likely going to need to trigger lazy loading of additional attributes, so you will probably need a high number of threads working in parallel.

As of Hibernate Search 4.4.0, instead of indexing all the types in parallel, the MassIndexer is configured by default to index only one type in parallel. It prevents resource exhaustion especially database connections and usually does not slow down the indexing. You can however configure this behavior using `MassIndexer.typesToIndexInParallel(int threadsToIndexObjects)`:

Example 6.7. Configuring the MassIndexer to index several types in parallel

```
fullTextSession
 .createIndexer( User.class, Customer.class )
 .typesToIndexInParallel( 2 )
 .batchSizeToLoadObjects( 25 )
 .cacheMode( CacheMode.NORMAL )
```

```
.threadsToLoadObjects( 5 )
.idFetchSize( 150 )
.progressMonitor( monitor ) //a MassIndexerProgressMonitor implementation
.startAndWait();
```

Generally we suggest to leave cacheMode to `CacheMode.IGNORE` (the default), as in most rein-dexing 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.

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

6.3.2.1. MassIndexer using threads and JDBC connections

The MassIndexer was designed to finish the re-indexing task as quickly as possible, but this requires a bit of care in its configuration to behave fairly with your server resources.

There is a simple formula to understand how the different options applied to the MassIndexer affect the number of used worker threads and connections: each thread will require a JDBC connection.

```
threads = typesToIndexInParallel * (threadsToLoadObjects + 1);
required JDBC connections = threads;
```

Let's see some suggestions for a roughly sane tuning starting point:

1. Option `typesToIndexInParallel` should probably be a low value, like 1 or 2, depending on how much of your CPUs have spare cycles and how slow a database round trip will be.

2. Before tuning a parallel run, experiment with options to tune your primary indexed entities in isolation.

3. Making `threadsToLoadObjects` higher increases the pre-loading rate for the picked entities from the database, but also increases memory usage and the pressure on the threads working on subsequent indexing.

4. Increasing parallelism usually helps as the bottleneck usually is the latency to the database connection: it's probably worth it to experiment with values significantly higher than the number of actual cores available, but make sure your database can handle all the multiple requests.

5. This advice might not apply to you: always measure the effects! We're providing this as a means to help you understand how these options are related.

Warning

Running the MassIndexer with many threads will require many connections to the database. If you don't have a sufficiently large connection pool, the MassIndexer itself and/or your other applications could starve being unable to serve other requests: make sure you size your connection pool accordingly to the options as explained in the above paragraph.

Tip

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

6.3.2.2. Using a custom MassIndexer implementation

The provided MassIndexer is quite general purpose, and while we believe it's a robust approach, you might be able to squeeze some better performance by writing a custom implementation. To run your own MassIndexer instead of using the one shipped with Hibernate Search you have to:

1. create an implementation of the `org.hibernate.search.spi.MassIndexerFactory` interface;

2. set the property `hibernate.search.massindexer.factoryclass` with the qualified class name of the factory implementation.

Example 6.8. Custom MassIndexerFactory example

```java
package org.myproject
import org.hibernate.search.spi.MassIndexerFactory

// ...

public class CustomIndexerFactory implements MassIndexerFactory {

  public void initialize(Properties properties) {
  }

  public MassIndexer createMassIndexer(...) {
    return new CustomIndexer();
  }

}
```

```
hibernate.search.massindexer.factoryclass =
  org.myproject.CustomIndexerFactory
```

6.3.3. Useful parameters for batch indexing

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 3.7.1, "Tuning 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 `id-FetchSize` to `Integer.MIN_VALUE`.

Chapter 7. Index Optimization

This section explains some low level tricks to keep your indexes at peak performance. We cover some Lucene details which in most cases you don't have to know about: Hibernate Search will handle these operations optimally and transparently in most cases without the need for further configuration. Still, it is good to know that there are ways to configure the behavior, if the need arises.

The index is physically stored in several smaller segments. Each segment is immutable and represents a generation of index writes. Index segments are periodically compacted, both to merge smaller segments and to remove stale entries; this merging process happens constantly in the background and can be tuned with the options specified in Section 3.7.1, "Tuning indexing performance", but you can also define policies to fully run index optimizations when it is most suited for your specific workload.

With older versions of Lucene it was important to frequently optimize the index to maintain good performance, but with current Lucene versions this doesn't apply anymore. The benefit of explicit optimization is very low, and in certain cases even counter-productive. During an explicit optimization the whole index is processed and rewritten inflicting a significant performance cost. Optimization is for this reason a double-edged sword.

Another reason to avoid optimizing the index too often is that an optimization will, as a side effect, invalidate cached filters and field caches and internal buffers need to be refreshed.

> **Tip**
>
> Optimizing the index is often not needed, does not benefit write (update) performance at all, and is a slow operation: make sure you need it before activating it.

Of course optimizing the index does not only present drawbacks: after the optimization process is completed and new IndexReader instances have loaded their buffers, queries will perform at peak performance and you will have reclaimed all disk space potentially used by stale entries.

It is recommended to not schedule any optimization, but if you wish to perform it periodically you should run it:

- on an idle system or when the searches are less frequent

- after a lot of index modifications

When using a MassIndexer (see Section 6.3.2, "Using a MassIndexer") it will optimize involved indexes by default at the start and at the end of processing; you can change this behavior by using MassIndexer.optimizeAfterPurge and MassIndexer.optimizeOnFinish respectively. The initial optimization is actually very cheap as it is performed on an empty index: its purpose is to release the storage space occupied by the old index.

7.1. Automatic optimization

While in most cases this is not needed, Hibernate Search can automatically optimize an index after:

- a certain amount of write operations

- or after a certain amount of transactions

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

Example 7.1. Defining automatic optimization parameters

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

With the above example an optimization will be triggered to the `Animal` index as soon as either:

- the number of additions and deletions reaches 1000

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

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

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 7.2. 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.

7.2. Manual optimization

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

Example 7.3. 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; the second, optimizes all indexes.

> **Note**
>
> `searchFactory.optimize()` has no effect on a JMS or JGroups backend: you must apply the optimize operation on the Master node.

7.3. Adjusting optimization

The Lucene index is constantly being merged in the background to keep a good balance between write and read performance; in a sense this is a form of background optimization which is always applied.

The following match attributes of Lucene's IndexWriter and are commonly used to tune how often merging occurs and how aggressive it is applied. They are exposed by Hibernate Search via:

- `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 3.7.1, "Tuning indexing performance" for a description of these properties.

Chapter 8. Monitoring

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

8.1. JMX

You can also enable access to the statistics via JMX. Setting the property `hibernate.search.jmx_enabled` will automatically register the `StatisticsInfoMBean`. Depending on your the configuration the `IndexControlMBean` and `IndexingProgressMonitorM-Bean` will also be registered. In case you are having more than one JMX enabled Hibernate Search instance running within a single JVM, you should also set `hibernate.search.jmx_bean_suffix` to a different value for each of the instances. The specified suffix will be used to distinguish between the different MBean instances. Let's have a closer look at the mentioned MBeans.

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

8.1.1. StatisticsInfoMBean

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

8.1.2. IndexControlMBean

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

8.1.3. IndexingProgressMonitorMBean

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

Chapter 9. Spatial

With the spatial extensions you can combine full-text queries with distance restrictions, filter results based on distances or sort results on such a distance criteria.

The spatial support of Hibernate Search has the following goals:

- Enable spatial search on entities: find entities within x km from a given location (latitude, longitude) on Earth

- Provide an easy way to enable spatial indexing via expressive annotations

- Provide a simple way for querying

- Hide geographical complexity

For example, you might search for restaurants somewhere in a 2 km radius around your office.

In order to use the spatial extensions for an indexed entity, you need to add the `@Spatial` annotation (`org.hibernate.search.annotations.Spatial`) and specify one or more sets of coordinates.

9.1. Enable indexing of Spatial Coordinates

There are different techniques to index point coordinates. Hibernate Search Spatial offers a choice between two strategies:

- index as numbers

- index as labeled spatial hashes

We will now describe both methods, so you can make a suitable choice. You can pick a different strategy for each set of coordinates. The strategy is selected by specifying the `spatialMode` attribute of the `@Spatial` annotation.

9.1.1. Indexing coordinates for range queries

When setting the `@Spatial.spatialMode` attribute to `SpatialMode.RANGE` (which is the default) coordinates are indexed as numeric fields, so that range queries can be performed to narrow down the initial area of interest.

Pros:

- Is quick on small data sets (< 100k entities)

- Is very simple: straightforward to debug/analyze

- Impact on index size is moderate

Cons:

- Poor performance on large data sets

- Poor performance if your data set is distributed across the whole world (for example when indexing points of interest in the United States, in Europe and in Asia, large areas collide because they share the same latitude. The latitude range query returns large amounts of data that need to be cross checked with those returned by the longitude range).

To index your entities for range querying you have to:

- add the `@Spatial` annotation on your entity

- add the `@Latitude` and `@Longitude` annotations on your properties representing the coordinates; these must be of type `Double`

Example 9.1. Sample Spatial indexing: Hotel class

```
import org.hibernate.search.annotations.*;

@Entity
@Indexed
@Spatial
public class Hotel {

  @Latitude
  Double latitude

  @Longitude
  Double longitude

  // ...
```

9.1.2. Indexing coordinates in a grid with spatial hashes

When setting `@Spatial.spatialMode` to `SpatialMode.HASH` the coordinates are encoded in several fields representing different zoom levels. Each box for each level is labeled so coordinates are assigned matching labels for each zoom level. This results in a grid encoding of labels called `spatial hashes`.

Pros :

- Good performance even with large data sets

- World wide data distribution independent

Cons :

- Index size is larger: need to encode multiple labels per pair of coordinates

To index your entities you have to:

- add the `@Spatial` annotation on the entity with the `SpatialMode` set to GRID : @Spatial(spatialMode = SpatialMode.HASH)

- add the `@Latitude` and `@Longitude` annotations on the properties representing your coordinates; these must be of type `Double`

Example 9.2. Indexing coordinates in a grid using spatial hashes

```
@Spatial(spatialMode = SpatialMode.HASH)
@Indexed
@Entity
public class Hotel {

  @Latitude
  Double latitude;

  @Longitude
  Double longitude;

  // ...
```

9.1.3. Implementing the Coordinates interface

Instead of using the `@Latitude` and `@Longitude` annotations you can choose to implement the `org.hibernate.search.spatial.Coordinates` interface.

Example 9.3. Implementing the Coordinates interface

```
import org.hibernate.search.annotations.*;
import org.hibernate.search.spatial.Coordinates;

@Entity
@Indexed
@Spatial
public class Song implements Coordinates {

  @Id long id;
  double latitude;
  double longitude;
  // ...

  @Override
  Double getLatitude() {
    return latitude;
```

```
  }

  @Override
  Double getLongitude() {
    return longitude;
  }

  // ...
```

As we will see in the section Section 9.3, "Multiple Coordinate pairs", an entity can have multiple `@Spatial` annotations; when having the entity implement `Coordinates`, the implemented methods refer to the default `@Spatial` annotation with the default pair of coordinates.

> **Tip**
>
> The default (field) name in case `@Spatial` is placed on the entity level is `org.hibernate.search.annotations.Spatial.COORDINATES_DEFAULT_FIELD`.

An alternative is to use properties implementing the `Coordinates` interface; this way you can have multiple `Spatial` instances:

Example 9.4. Using attributes of type Coordinates

```
@Entity
@Indexed
public class Event {
  @Id
  Integer id;

  @Field(store = Store.YES)
  String name;

  double latitude;
  double longitude;

  @Spatial(spatialMode = SpatialMode.HASH)
  public Coordinates getLocation() {
    return new Coordinates() {
      @Override
      public Double getLatitude() {
        return latitude;
      }

      @Override
      public Double getLongitude() {
        return longitude;
      }
    };
  }

// ...
```

When using this form the `@Spatial.name` automatically defaults to the property name. In the above case to `location`.

9.2. Performing Spatial Queries

You can use the Hibernate Search query DSL to build a query to search around a pair of coordinates (latitude, longitude) or around a bean implementing the `Coordinates` interface.

As with any full-text query, the spatial query creation flow looks like:

1. retrieve a `QueryBuilder` from the `SearchFactory`

2. use the DSL to build a spatial query, defining search center and radius

3. optionally combine the resulting `Query` with other filters

4. call the `createFullTextQuery()` and use the resulting query like any standard Hibernate or JPA query

Example 9.5. Search for an Hotel by distance

```
QueryBuilder builder = fullTextSession.getSearchFactory()
  .buildQueryBuilder().forEntity( Hotel.class ).get();

org.apache.lucene.search.Query luceneQuery = builder
  .spatial()
  .within( radius, Unit.KM )
    .ofLatitude( centerLatitude )
    .andLongitude( centerLongitude )
  .createQuery();

org.hibernate.Query hibQuery = fullTextSession
  .createFullTextQuery( luceneQuery, Hotel.class );
List results = hibQuery.list();
```

> **Note**
>
> In the above example we did not explicitly specify the field name to use. The default coordinates field name was used implicitly. To target an alternative pair of coordinates at query time, we need to specify the field name as well. See Section 9.3, "Multiple Coordinate pairs".

A fully working example can be found in the test-suite of the source code [https://github.com/hibernate/hibernate-search]. Refer to `SpatialIndexingTest.testSpatialAnnotationOnClassLevel()` and its corresponding `Hotel` test class.

Alternatively to passing separate latitude and longitude values, you can also pass an instance implementing the `Coordinates` interface:

Example 9.6. DSL example with Coordinates

```
Coordinates coordinates = Point.fromDegrees(24d, 31.5d);
Query query = builder
  .spatial()
    .within( 51, Unit.KM )
      .ofCoordinates( coordinates )
  .createQuery();

List results = fullTextSession.createFullTextQuery( query, POI.class ).list();
```

9.2.1. Returning distance to query point in the search results

9.2.1.1. Returning distance to the center in the results

To retrieve the actual distance values you need to use projection (see Section 5.1.3.5, "Projection"):

Example 9.7. Distance projection example

```
double centerLatitude = 24.0d;
double centerLongitude= 32.0d;

QueryBuilder builder = fullTextSession.getSearchFactory()
  .buildQueryBuilder().forEntity(POI.class).get();
org.apache.lucene.search.Query luceneQuery = builder
  .spatial()
    .onField("location")
    .within(100, Unit.KM)
      .ofLatitude(centerLatitude)
      .andLongitude(centerLongitude)
  .createQuery();

FullTextQuery hibQuery = fullTextSession.createFullTextQuery(luceneQuery, POI.class);
hibQuery.setProjection(FullTextQuery.SPATIAL_DISTANCE, FullTextQuery.THIS);
hibQuery.setSpatialParameters(centerLatitude, centerLongitude, "location");
List results = hibQuery.list();
```

- Use `FullTextQuery.setProjection` with `FullTextQuery.SPATIAL_DISTANCE` as one of the projected fields.

- Call `FullTextQuery.setSpatialParameters` with the latitude, longitude and the name of the spatial field used to build the spatial query. Note that using coordinates different than the center used for the query will have unexpected results.

> **Tip**
>
> The default (field) name in case `@Spatial` is placed on the entity level is `org.hibernate.search.annotations.Spatial.COORDINATES_DEFAULT_FIELD`.

> **Distance projection and null values**
>
> Using distance projection on non `@Spatial` enabled entities and/or with a non spatial Query will have unexpected results as entities not spatially indexed and/or having `null` values for latitude or longitude will be considered to be at (0,0)/(lat,0)/ (0,long).
>
> Using distance projection with a spatial query on spatially indexed entities having, eventually, `null` values for latitude and/or longitude is safe as they will not be found by the spatial query and won't have distance calculated.

9.2.1.2. Sorting by distance

To sort the results by distance to the center of the search you will have to build a `Sort` instance using a `DistanceSortField`:

Example 9.8. Distance sort example

```
double centerLatitude = 24.0d;
double centerLongitude = 32.0d;

QueryBuilder builder = fullTextSession.getSearchFactory()
    .buildQueryBuilder().forEntity( POI.class ).get();
org.apache.lucene.search.Query luceneQuery = builder
  .spatial()
    .onField("location")
      .within(100, Unit.KM)
      .ofLatitude(centerLatitude)
      .andLongitude(centerLongitude)
  .createQuery();

FullTextQuery hibQuery = fullTextSession.createFullTextQuery(luceneQuery, POI.class);
Sort distanceSort = new Sort(
    new DistanceSortField(centerLatitude, centerLongitude, "location"));
hibQuery.setSort(distanceSort);
```

The `DistanceSortField` must be constructed using the same coordinates on the same spatial field used to build the spatial query otherwise the sorting will occur with another center than the query. This repetition is needed to allow you to define Queries with any tool.

> **Sorting and null values**
>
> Using distance sort on non `@Spatial` enabled entities and/or with a non spatial
> Query will have also unexpected results as entities non spatially indexed and/or
> with null values for latitude or longitude will be considered to be at (0,0)/(lat,0)/
> (0,long)
>
> Using distance sort with a spatial query on spatially indexed entities having, poten-
> tially, `null` values for latitude and/or longitude is safe as they will not be found by
> the spatial query and so won't be sorted

9.3. Multiple Coordinate pairs

You can associate multiple pairs of coordinates to the same entity, as long as each pair is uniquely
identified by using a different name. This is achieved by stacking multiple `@Spatial` annotations
within a single `@Spatials` annotation and specifying the `name` attribute on the individual `@Spatial`
annotations.

Example 9.9. Multiple sets of coordinates

```java
import org.hibernate.search.annotations.*;

@Entity
@Indexed
@Spatials({
  @Spatial,
  @Spatial(name="work",  spatialMode = SpatialMode.HASH)
})
public class UserEx {

  @Id
  Integer id;

  @Latitude
  Double homeLatitude;

  @Longitude
  Double homeLongitude;

  @Latitude(of="work")
  Double workLatitude;

  @Longitude(of="work")
  Double workLongitude;
```

To target an alternative pair of coordinates at query time, we need to specify the pair by name
using `onField(String)`:

Example 9.10. Querying on non-default coordinate set

```
QueryBuilder builder = fullTextSession.getSearchFactory()
  .buildQueryBuilder().forEntity( UserEx.class ).get();

org.apache.lucene.search.Query luceneQuery = builder
  .spatial()
  .onField( "work" )
  .within( radius, Unit.KM )
    .ofLatitude( centerLatitude )
    .andLongitude( centerLongitude )
  .createQuery();

org.hibernate.Query hibQuery = fullTextSession.createFullTextQuery( luceneQuery,
    Hotel.class );
List results = hibQuery.list();
```

9.4. Insight: implementation details of spatial hashes indexing

The following chapter is meant to provide a technical insight in spatial hash (grid) indexing. It discusses how coordinates are mapped to the index and how queries are implemented.

9.4.1. At indexing level

When Hibernate Search indexes an entity annotated with `@Spatial`, it instantiates a `Spatial-FieldBridge` to transform the latitude and longitude fields accessed via the `Coordinates` interface to the multiple index fields stored in the Lucene index.

Principle of the spatial index: the spatial index used in Hibernate Search is a grid based spatial index [http://en.wikipedia.org/wiki/Grid_(spatial_index)#Grid-based_spatial_indexing] where grid ids are hashes derived from latitude and longitude.

To make computations easier the latitude and longitude field values will be projected into a flat coordinate system with the help of a sinusoidal projection [http://en.wikipedia.org/wiki/Sinusoidal_projection]. Origin value space is :

$[-90 \rightarrow +90]$,$]-180 \rightarrow 180]$

for latitude,longitude coordinates and projected space is:

$]-pi \rightarrow +pi]$,$[-pi/2 \rightarrow +pi/2]$

for Cartesian x,y coordinates (beware of fields order inversion: x is longitude and y is latitude).

The index is divided into n levels labeled from 0 to n-1.

At the level 0 the projected space is the whole Earth. At the level 1 the projected space is divided into 4 rectangles (called boxes as in bounding box):

[-pi,-pi/2]#[0,0], [-pi,0]#[0,+pi/2], [0,-pi/2]#[+pi,0] and [0,0]#[+pi,+pi/2]

At level n+1 each box of level n is divided into 4 new boxes and so on. The numbers of boxes at a given level is 4^n.

Each box is given an id, in this format: [Box index on the X axis]|[Box index on the Y axis]. To calculate the index of a box on an axis we divide the axis range in 2^n slots and find the slot the box belongs to. At the n level the indexes on an axis are from -(2^n)/2 to (2^n)/2. For instance, the 5th level has 4^5 = 1024 boxes with 32 indexes on each axis (32x32 is 1024) and the box of Id "0|8" is covering the [0,8/32*pi/2]#[1/32*pi,9/32*pi/2] rectangle is projected space.

Beware! The boxes are rectangles in projected space but the related area on Earth is not rectangular!

Now that we have all these boxes at all these levels, we index points "into" them.

For a point (lat,long) we calculate its projection (x,y) and then we calculate for each level of the spatial index, the ids of the boxes it belongs to.

At each level the point is in one and only one box. For points on the edges the box are considered exclusive n the left side and inclusive on the right i-e]start,end] (the points are normalized before projection to [-90,+90],]-180,+180]).

We store in the Lucene document corresponding to the entity to index one field for each level of the spatial hash grid. The field is named: *HSSI*[n]. [spatial index fields name] is given either by the parameter at class level annotation or derived from the name of the spatial annotated method of the entity, HSSI stands for Hibernate Search Spatial Index and n is the level of the spatial hashes grid.

We also store the latitude and longitude as a numeric field under [spatial index fields name]_HSSI_Latitude and [spatial index fields name]_HSSI_Longitude fields. They will be used to filter precisely results by distance in the second stage of the search.

9.4.2. At search level

Now that we have all these fields, what are they used for?

When you ask for a spatial search by providing a search discus (center+radius) we will calculate the box ids that do cover the search discus in the projected space, fetch all the documents that belong to these boxes (thus narrowing the number of documents for which we will have to calculate distance to the center) and then filter this subset with a real distance calculation. This is called two level spatial filtering.

9.4.2.1. Step 1: Compute the best spatial hashes grid level for the search discus

For a given search radius there is an optimal hash grid level where the number of boxes to retrieve shall be minimal without bringing back to many documents (level 0 has only 1 box but retrieve all documents). The optimal hash grid level is the maximum level where the width of each box is

larger than the search area. Near the equator line where projection deformation is minimal, this will lead to the retrieval of at most 4 boxes. Towards the poles where the deformation is more significant, it might need to examine more boxes but as the sinusoidal projection has a simple Tissot's indicatrix (see Sinusoidal projection [http://en.wikipedia.org/wiki/Sinusoidal_projection]) in populated areas, the overhead is minimal.

9.4.2.2. Step 2: Compute ids of the corresponding covering boxes at that level

Now that we have chosen the optimal level, we can compute the ids of the boxes covering the search discus (which is not a discus in projected space anymore).

This is done by `org.hibernate.search.spatial.impl.SpatialHelper.getSpatialHashCellsIds(Point center, double radius, int spatialHashLevel)`

It will calculate the bounding box of the search discus and then call `org.hibernate.search.spatial.impl.SpatialHelper.getSpatialHashCellsIds(Point lowerLeft, Point upperRight, int spatialHashLevel)` that will do the actual computation. If the bounding box crosses the meridian line it will cut the search in two and make two calls to `getSpatialHashCellsIds(Point lowerLeft, Point upperRight, int spatialHashLevel)` with left and right parts of the box.

There are some geo related hacks (search radius too large, search radius crossing the poles) that are handled in bounding box computations done by `Rectangle.fromBoundingCircle(Coordinates center, double radius)` (see http://janmatuschek.de/LatitudeLongitudeBoundingCoordinates for reference on those subjects).

The `SpatialHelper.getSpatialHashCellsIds(Point lowerLeft, Point upperRight, int spatialHashLevel)` project the defining points of the bounding box and compute the boxes they belong to. It returns all the box Ids between the lower left to the upper right corners, thus covering the area.

9.4.2.3. Step 3: Lucene index lookup

The query is built with theses Ids searching for documents having a *HSSI*[n] (n the level found at Step 1) field valued with one of the ids of Step 2.

See also the implementation of `org.hibernate.search.spatial.impl.SpatialHashFilter`.

This query will return all documents in the boxes covering the projected bounding box of the search discus. So it is too large and needs refining. But we have narrowed the distance calculation problems to a subset of our data.

9.4.2.4. Step 4: Refine

A distance calculation filter is set after the Lucene index lookup query of Step 3 to exclude false candidates from the result list.

See `SpatialQueryBuilderFromCoordinates.buildSpatialQuery(Coordinates center, double radius, String fieldName)`

Chapter 10. Advanced features

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

10.1. 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 10.1. Accessing the `SearchFactory`

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

10.2. Accessing the `SearchIntegrator`

The interface `SearchIntegrator` gives access to lower level APIs of Hibernate Search. You can access the `SearchIntegrator` SPI using the SearchFactory (Section 10.1, "Accessing the `SearchFactory`"):

Example 10.2. Accessing the `SearchIntegrator`

```
SearchIntegrator searchIntegrator = searchFactory.unwrap(SearchIntegrator.class);
```

10.3. Using an `IndexReader`

Queries in Lucene are executed on an `IndexReader`. Hibernate Search caches index readers to maximize performance and implements other strategies to retrieve updated IndexReaders in order to minimize IO operations. Your code can access these cached resources, but you have to follow some "good citizen" rules.

Example 10.3. 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. Using the configured ReaderProvider (described in Section 2.3, "Reader strategy") 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()`, but always call `readerProvider.closeReader(reader)`, using a finally block.

- Don't use this `IndexReader` for modification operations: it's a read-only instace, you would get an exception.

Aside from those rules, you can use the `IndexReader` freely, especially to do native Lucene queries. Using this shared IndexReaders will be 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 10.4. Accessing an `IndexReader` by index names

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

10.4. 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 are certain that your index is represented as a Directory and need to access it, you can get a reference to the Directory via the IndexManager. You will have to cast the IndexManager instance to a DirectoryBasedIndexManager and then use `getDirectoryProvider().getDirectory()` to get a reference to the underlying Directory. This is not recommended, if you need low level access to the index using Lucene APIs we suggest to see Section 10.3, "Using an `IndexReader`" instead.

10.5. Sharding indexes

In some cases it can be useful to split (shard) the data into several Lucene indexes. There are two main use use cases:

- A single index is so big that index update times are slowing the application down. In this case static sharding can be used to split the data into a pre-defined number of shards.

- Data is naturally segmented by customer, region, language or other application parameter and the index should be split according to these segments. This is a use case for dynamic sharding.

> **Tip**
>
> By default sharding is not enabled.

10.5.1. Static sharding

To enable static sharding set the hibernate.search.<indexName>.sharding_strategy.nbr_of_shards property as seen in Example 10.5, "Enabling index sharding".

Example 10.5. Enabling index sharding

```
hibernate.search.[default|<indexName>].sharding_strategy.nbr_of_shards = 5
```

The default sharding strategy which gets enabled by setting this property, splits the data according to the hash value of the document id (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.[default|<indexName>].sharding_strategy property to the fully qualified class name of your custom IndexShardingStrategy.

Example 10.6. Registering a custom IndexShardingStrategy

```
hibernate.search.[default|<indexName>].sharding_strategy =
  my.custom.RandomShardingStrategy
```

10.5.2. Dynamic sharding

Dynamic sharding allows you to manage the shards yourself and even create new shards on the fly. To do so you need to implement the interface ShardIdentifierProvider and set the hibernate.search.[default|<indexName>].sharding_strategy property to the fully qualified name of this class. Note that instead of implementing the interface directly, you should rather derive your implementation from org.hibernate.search.store.ShardIdentifierProviderTemplate which provides a basic implementation. Let's look at Example 10.7, "Custom ShardIdentifierProvider" for an example.

Example 10.7. Custom ShardIdentifierProvider

```java
public static class AnimalShardIdentifierProvider extends ShardIdentifierProviderTemplate {

 @Override
 public String getShardIdentifier(Class<?> entityType, Serializable id,
         String idAsString, Document document) {
```

```
    if (entityType.equals(Animal.class)) {
        String typeValue = document.getField("type").stringValue();
        addShard(typeValue);
        return typeValue;
    }
    throw new RuntimeException("Animal expected but found " + entityType);
}


@Override
protected Set<String> loadInitialShardNames(Properties properties, BuildContext buildContext) {
    ServiceManager serviceManager = buildContext.getServiceManager();
    SessionFactory sessionFactory = serviceManager.requestService(
        HibernateSessionFactoryService.class).getSessionFactory();
    Session session = sessionFactory.openSession();
    try {
        Criteria initialShardsCriteria = session.createCriteria(Animal.class);
        initialShardsCriteria.setProjection(Projections.distinct(Property.forName("type")));
        List<String> initialTypes = initialShardsCriteria.list();
        return new HashSet<String>(initialTypes);
    }
    finally {
        session.close();
    }
  }
}
```

The are several things happening in `AnimalShardIdentifierProvider`. First off its purpose is to create one shard per animal type (e.g. mammal, insect, etc.). It does so by inspecting the class type and the Lucene document passed to the `getShardIdentifier()` method. It extracts the type field from the document and uses it as shard name. `getShardIdentifier()` is called for every addition to the index and a new shard will be created with every new animal type encountered. The base class `ShardIdentifierProviderTemplate` maintains a set with all known shards to which any identifier must be added by calling `addShard()`.

It is important to understand that Hibernate Search cannot know which shards already exist when the application starts. When using `ShardIdentifierProviderTemplate` as base class of a `ShardIdentifierProvider` implementation, the initial set of shard identifiers must be returned by the `loadInitialShardNames()` method. How this is done will depend on the use case. However, a common case in combination with Hibernate ORM is that the initial shard set is defined by the the distinct values of a given database column. Example 10.7, "Custom ShardIdentifierProvider" shows how to handle such a case. `AnimalShardIdentifierProvider` makes in its `loadInitialShardNames()` implementation use of a service called `HibernateSessionFactoryService` (see also Section 10.7, "Using external services") which is available within an ORM environment. It allows to request a Hibernate `SessionFactory` instance which can be used to run a Criteria query in order to determine the initial set of shard identifiers.

Last but not least, the `ShardIdentifierProvider` also allows for optimizing searches by selecting which shard to run a query against. By activating a filter (see Section 5.3.1, "Using filters in a sharded environment"), a sharding strategy can select a subset of the shards used to answer a query (`getShardIdentifiersForQuery()`, not shown in the example) and thus speed up the query execution.

> **Important**
>
> This ShardIdentifierProvider is considered experimental. We might need to apply some changes to the defined method signatures to accommodate for unforeseen use cases. Please provide feedback if you have ideas, or just to let us know how you're using this API.

10.6. 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` entities. We just set `indexName` for both entities to "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.

10.7. Using external services

A `Service` in Hibernate Search is a class implementing the interface `org.hibernate.search.engine.service.spi.Service` and providing a default no-arg constructor. Theoretically that's all that is needed to request a given service type from the Hibernate Search `ServiceManager`. In practice you want probably want to add some service life cycle methods (implement `Startable` and `Stoppable`) as well as actual methods providing some functionality.

Hibernate Search uses the service approach to decouple different components of the system. Let's have a closer look at services and how they are used.

10.7.1. Using a Service

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

Example 10.8. Example of a custom DirectoryProvider using a ClassLoaderService

```java
public CustomDirectoryProvider implements DirectoryProvider<RAMDirectory> {
    private ServiceManager serviceManager;
    private ClassLoaderService classLoaderService;

    public void initialize(
        String directoryProviderName,
        Properties properties,
        BuildContext context) {
        //get a reference to the ServiceManager
        this.serviceManager = context.getServiceManager();
    }

    public void start() {
        //get the current ClassLoaderService
        classLoaderService = serviceManager.requestService(ClassLoaderService.class);
    }

    public RAMDirectory getDirectory() {
        //use the ClassLoaderService
    }

    public stop() {
        //make sure to release all services
        serviceManager.releaseService(ClassLoaderService.class);
    }
}
```

When you request a service, an instance of the requested service type is returned to you. Make sure release the service via `ServiceManager.releaseService` once you don't need it anymore. 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 only needed during initialization time.

10.7.2. Implementing a Service

To implement a service, you need to create an interface which identifies it and extends `org.hibernate.search.engine.service.spi.Service`. You can then add additional methods to your service interface as needed.

Naturally you will also need to provide an implementation of your service interface. This implementation must have a public no-arg constructor. Optionally your service can also implement the life cycle methods `org.hibernate.search.engine.service.spi.Startable` and/or

org.hibernate.search.engine.service.spi.Stoppable. These methods will be called by the ServiceManager when the service is created respectively the last reference to a requested service is released.

Services are retrieved from the ServiceManager.requestService using the Class object of the interface you define as a key.

10.7.2.1. Managed services

To transparently discover services Hibernate Search uses the Java ServiceLoader mechanism. This means you need to add a service file to your jar under /META-INF/services/ named after the fully qualified classname of your service interface. The content of the file contains the fully qualified classname of your service implementation.

Example 10.9. Service file for the Infinispan CacheManagerService service

```
/META-INF/services/org.infinispan.hibernate.search.spi.CacheManagerService
```

Example 10.10. Content of META-INF/services/ org.infinispan.hibernate.search.spi.CacheManagerService

```
org.infinispan.hibernate.search.impl.DefaultCacheManagerService
```

> **Note**
>
> Hibernate Search only supports a single service implementation of a given service. There is no mechanism to select between multiple versions of a service. It is an error to have multiple jars defining each a different implementation for the same service. If you want to override the implementation of a already existing service at runtime you will need to look at Section 10.7.2.2, "Provided services".

10.7.2.2. Provided services

> **Important**
>
> Provided services are usually used by frameworks integrating with Hibernate Search and not by library users themselves.

As an alternative to manages services, a service can be provided by the environment bootstrapping Hibernate Search. For example, Infinispan which uses Hibernate Search as its internal search engine, passes the CacheContainer to Hibernate Search. In this case, the CacheContainer instance is not managed by Hibernate Search and the start/stop methods defined by optional Stoppable and Startable interfaces will be ignored.

A Service implementation which is only used as a Provided Service doesn't need to have a public constructor taking no arguments.

> **Note**
>
> Provided services have priority over managed services. If a provided service is registered with the same `ServiceManager` instance as a managed service, the provided service will be used.

The provided services are passed to Hibernate Search via the `SearchConfiguration` interface: as implementor of method `getProvidedServices` you can return a `Map` of all services you need to provide.

> **Note**
>
> When implementing a custom `org.hibernate.search.cfg.spi.SearchConfiguration` we recommend you extend the base class `org.hibernate.search.cfg.spi.SearchConfigurationBase`: that will improve compatibility by not breaking your code when we need to add new methods to this interface.

10.8. Customizing Lucene's scoring formula

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

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

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

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

Hibernate Search provides two ways to modify Lucene's similarity calculation.

First you can set the default similarity by specifying the fully specified classname of your Similarity implementation using the property hibernate.search.similarity. The default value is org.apache.lucene.search.similarities.DefaultSimilarity.

Secondly, you can override the similarity used for a specific index by setting the `similarity` property for this index (see Section 3.3, "Directory configuration" for more information about index configuration):

```
hibernate.search.[default|<indexname>].similarity = my.custom.Similarity
```

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.

> **Note**
>
> When two entities share the same index they must declare the same Similarity implementation.

10.9. Multi-tenancy

10.9.1. What is multi-tenancy?

The term multi-tenancy in general is applied to software development to indicate an architecture in which a single running instance of an application simultaneously serves multiple clients (tenants). Isolating information (data, customizations, etc) pertaining to the various tenants is a particular challenge in these systems. This includes the data owned by each tenant stored in the database. You will find more details on how to enable multi-tenancy with Hibernate in the Hibernate ORM developer's guide [http://docs.jboss.org/hibernate/orm/4.3/devguide/en-US/html/ch16.html].

10.9.2. Using a tenant-aware `FullTextSession`

Hibernate Search supports multi-tenancy on top of Hibernate ORM, it stores the tenant identifier in the document and automatically filters the query results.

The `FullTextSession` will be bound to the specific tenant ("client-A" in the example) and the mass indexer will only index the entities associated to that tenant identifier.

Example 10.11. Bind the session to a tenant

```
Session session = getSessionFactory()
                    .withOptions()
                        .tenantIdentifier( "client-A" )
                .openSession();

FullTextSession session = Search.getFullTextSession( session );
```

The use of a tenant identifier will have the following effects:

1. Every document saved or updated in the index will have an additional field `__HSearch_TenantId` containing the tenant identifier.

2. Every search will be filtered using the tenant identifier.

3. The MassIndexer (see Section 6.3.2, "Using a MassIndexer") will only affect the currently selected tenant.

Note that not using a tenant will return all the matching results for all the tenants in the index.

Chapter 11. Further reading

Last but not least, a few pointers to further information. We highly recommend you to get a copy of Hibernate Search in Action [http://www.manning.com/bernard/]. This excellent book covers Hibernate Search in much more depth than this online documentation can and has a great range of additional examples. If you want to increase your knowledge of Lucene we recommend Lucene in Action (Second Edition) [http://www.manning.com/hatcher3/].

Because Hibernate Search's functionality is tightly coupled to Hibernate ORM it is a good idea to understand Hibernate. Start with the online documentation [http://hibernate.org/orm/documentation/] or get hold of a copy of Java Persistence with Hibernate, Second Edition [http://www.manning.com/bauer3/].

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

In case you would like to report a bug use the Hibernate Search JIRA [https://hibernate.atlassian.net/browse/HSEARCH] instance. Feedback is always welcome!