Red Hat JBoss Enterprise Application Platform 7.1

Developing Hibernate Applications

For Use with Red Hat JBoss Enterprise Application Platform 7.1

Last Updated: 2018-10-11

Red Hat JBoss Enterprise Application Platform 7.1 Developing Hibernate Applications

For Use with Red Hat JBoss Enterprise Application Platform 7.1

Abstract

This document provides information for developers and administrators who want to develop and deploy JPA/Hibernate applications with JBoss EAP 7.1.

Table of Contents

CHAPTER 1. INTRODUCTION

1.1. ABOUT HIBERNATE CORE

Hibernate Core is an object-relational mapping framework for the Java language. It provides a framework for mapping an object-oriented domain model to a relational database, allowing applications to avoid direct interaction with the database. Hibernate solves object-relational impedance mismatch problems by replacing direct, persistent database accesses with high-level object handling functions.

1.2. HIBERNATE ENTITYMANAGER

Hibernate EntityManager implements the programming interfaces and lifecycle rules as defined by the Java Persistence 2.1 specification. Together with Hibernate Annotations, this wrapper implements a complete (and standalone) JPA persistence solution on top of the mature Hibernate Core. You may use a combination of all three together, annotations without JPA programming interfaces and lifecycle, or even pure native Hibernate Core, depending on the business and technical needs of your project. You can at all times fall back to Hibernate native APIs, or if required, even to native JDBC and SQL. It provides JBoss EAP with a complete Java Persistence solution.

JBoss EAP is 100% compliant with the Java Persistence 2.1 specification. Hibernate also provides additional features to the specification. To get started with JPA and JBoss EAP, see the **`bean-validation`**, **`greeter`**, and **`kitchensink`** quickstarts that ship with JBoss EAP. For information about how to download and run the quickstarts, see Using the Quickstart Examples in the JBoss EAP *Getting Started Guide*.

Persistence in JPA is available in containers like EJB 3 or the more modern CDI, Java Context and Dependency Injection, as well as in standalone Java SE applications that execute outside of a particular container. The following programming interfaces and artifacts are available in both environments.

EntityManagerFactory

An entity manager factory provides entity manager instances, all instances are configured to connect to the same database, to use the same default settings as defined by the particular implementation, etc. You can prepare several entity manager factories to access several data stores. This interface is similar to the SessionFactory in native Hibernate.

EntityManager

The EntityManager API is used to access a database in a particular unit of work. It is used to create and remove persistent entity instances, to find entities by their primary key identity, and to query over all entities. This interface is similar to the Session in Hibernate.

Persistence context

A persistence context is a set of entity instances in which for any persistent entity identity there is a unique entity instance. Within the persistence context, the entity instances and their lifecycle is managed by a particular entity manager. The scope of this context can either be the transaction, or an extended unit of work.

Persistence unit

The set of entity types that can be managed by a given entity manager is defined by a persistence unit. A persistence unit defines the set of all classes that are related or grouped by the application, and which must be collocated in their mapping to a single data store.

Container-managed entity manager

An entity manager whose lifecycle is managed by the container.

Application-managed entity manager

An entity manager whose lifecycle is managed by the application.

JTA entity manager

Entity manager involved in a JTA transaction.

Resource-local entity manager

Entity manager using a resource transaction (not a JTA transaction).

CHAPTER 2. HIBERNATE CONFIGURATION

2.1. HIBERNATE CONFIGURATION

The configuration for entity managers both inside an application server and in a standalone application reside in a persistence archive. A persistence archive is a JAR file which must define a **persistence.xml** file that resides in the **META-INF/** folder.

You can connect to the database using the **persistence.xml** file. There are two ways of doing this:

- Specifying a data source which is configured in the **datasources** subsystem in JBoss EAP.
  The **jta-data-source** points to the JNDI name of the data source this persistence unit maps to. The **java:jboss/datasources/ExampleDS** here points to the **H2 DB** embedded in the JBoss EAP.

  Example of object-relational-mapping in the persistence.xml File

```
<persistence>
   <persistence-unit name="myapp">
      <provider>org.hibernate.ejb.HibernatePersistence</provider>
      <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-
source>
      <properties>
         ... ...
      </properties>
   </persistence-unit>
</persistence>
```

- Explicitly configuring the **persistence.xml** file by specifying the connection properties.

  Example of Specifying Connection Properties in the persistence.xml file

```
<property name="javax.persistence.jdbc.driver"
value="org.hsqldb.jdbcDriver"/>
<property name="javax.persistence.jdbc.user" value="sa"/>
<property name="javax.persistence.jdbc.password" value=""/>
<property name="javax.persistence.jdbc.url" value="jdbc:hsqldb:."/>
```

  For the complete list of connection properties, see Connection Properties Configurable in the **persistence.xml** File.

There are a number of properties that control the behavior of Hibernate at runtime. All are optional and have reasonable default values. These Hibernate properties are all used in the **persistence.xml** file. For the complete list of all configurable Hibernate properties, see Hibernate Properties.

2.2. SECOND-LEVEL CACHES

2.2.1. About Second-level Caches

A second-level cache is a local data store that holds information persisted outside the application session. The cache is managed by the persistence provider, improving runtime by keeping the data separate from the application.

JBoss EAP supports caching for the following purposes:

- Web Session Clustering
- Stateful Session Bean Clustering
- SSO Clustering
- Hibernate/JPA Second-level Cache

WARNING

Each cache container defines a **`repl`** and a **`dist`** cache. These caches should not be used directly by user applications.

2.2.2. Configure a Second-level Cache for Hibernate

The configuration of Infinispan to act as the second-level cache for Hibernate can be done in two ways:

- It is recommended to configure the second-level cache through JPA applications, using the **`persistence.xml`** file, as explained in the JBoss EAP*Development Guide*.
- Alternatively, you can configure the second-level cache through Hibernate native applications, using the **`hibernate.cfg.xml`** file, as explained below.

Configuring a Second-level Cache for Hibernate Using Hibernate Native Applications

1. Create the **`hibernate.cfg.xml`** file in the deployment’s class path.
2. Add the following XML to the **`hibernate.cfg.xml`** file. The XML needs to be within the **`<session-factory>`** tag:

```
<property
name="hibernate.cache.use_second_level_cache">true</property>
<property name="hibernate.cache.use_query_cache">true</property>
<property
name="hibernate.cache.region.factory_class">org.jboss.as.jpa.hiberna
te5.infinispan.InfinispanRegionFactory</property>
```

3. In order to use the Hibernate native APIs within your application, you must add the following dependencies to the **`MANIFEST.MF`** file:

```
Dependencies: org.infinispan,org.hibernate
```

CHAPTER 3. HIBERNATE ANNOTATIONS

3.1. HIBERNATE ANNOTATIONS

The **`org.hibernate.annotations`** package contains some annotations which are offered by Hibernate, on top of the standard JPA annotations.

Table 3.1. General Annotations

Annotation	Description
`Check`	Arbitrary SQL check constraints which can be defined at the class, property or collection level.
`Immutable`	Mark an Entity or a Collection as immutable. No annotation means the element is mutable. An immutable entity may not be updated by the application. Updates to an immutable entity will be ignored, but no exception is thrown. **`@Immutable`** placed on a collection makes the collection immutable, meaning additions and deletions to and from the collection are not allowed. A **`HibernateException`** is thrown in this case.

Table 3.2. Caching Entities

Annotation	Description
`Cache`	Add caching strategy to a root entity or a collection.

Table 3.3. Collection Related Annotations

Annotation	Description
`MapKeyType`	Defines the type of key of a persistent map.
`ManyToAny`	Defines a **`ToMany`** association pointing to different entity types. Matching the entity type is done through a metadata discriminator column. This kind of mapping should be only marginal.
`OrderBy`	Order a collection using SQL ordering (not HQL ordering).

Annotation	Description
`OnDelete`	Strategy to use on collections, arrays and on joined subclasses delete. `OnDelete` of secondary tables is currently not supported.
`Persister`	Specify a custom persister.
`Sort`	Collection sort (Java level sorting).
`Where`	Where clause to add to the element Entity or target entity of a collection. The clause is written in SQL.
`WhereJoinTable`	Where clause to add to the collection join table. The clause is written in SQL.

Table 3.4. Custom SQL for CRUD Operations

Annotation	Description
`Loader`	Overwrites Hibernate default `FIND` method.
`SQLDelete`	Overwrites the Hibernate default `DELETE` method.
`SQLDeleteAll`	Overwrites the Hibernate default `DELETE ALL` method.
`SQLInsert`	Overwrites the Hibernate default `INSERT INTO` method.
`SQLUpdate`	Overwrites the Hibernate default `UPDATE` method.
`Subselect`	Maps an immutable and read-only entity to a given SQL subselect expression.
`Synchronize`	Ensures that auto-flush happens correctly and that queries against the derived entity do not return stale data. Mostly used with `Subselect`.

Table 3.5. Entity

Annotation	Description
`Cascade`	Apply a cascade strategy on an association.
`Entity`	Adds additional metadata that may be needed beyond what is defined in the standard **`@Entity`**. • **`mutable`**: whether this entity is mutable or not • **`dynamicInsert`**: allow dynamic SQL for inserts • **`dynamicUpdate`**: allow dynamic SQL for updates • **`selectBeforeUpdate`**: Specifies that Hibernate should never perform an SQL UPDATE unless it is certain that an object is actually modified. • **`polymorphism`**: whether the entity polymorphism is of PolymorphismType.IMPLICIT (default) or PolymorphismType.EXPLICIT • **`optimisticLock`**: optimistic locking strategy (OptimisticLockType.VERSION, OptimisticLockType.NONE, OptimisticLockType.DIRTY or OptimisticLockType.ALL) **NOTE** The annotation "Entity" is deprecated and scheduled for removal in future releases. Its individual attributes or values should become annotations.
`Polymorphism`	Used to define the type of polymorphism Hibernate will apply to entity hierarchies.
`Proxy`	Lazy and proxy configuration of a particular class.
`Table`	Complementary information to a table either primary or secondary.
`Tables`	Plural annotation of Table.

Annotation	Description
`Target`	Defines an explicit target, avoiding reflection and generics resolving.
`Tuplizer`	Defines a tuplizer for an entity or a component.
`Tuplizers`	Defines a set of tuplizers for an entity or a component.

Table 3.6. Fetching

Annotation	Description
`BatchSize`	Batch size for SQL loading.
`FetchProfile`	Defines the fetching strategy profile.
`FetchProfiles`	Plural annotation for **`@FetchProfile`**.
`LazyGroup`	Specifies that an entity attribute should be fetched along with all the other attributes belonging to the same group. In order to load entity attributes lazily, bytecode enhancement is needed. By default, all non-collection attributes are loaded in one group named **`DEFAULT`**. This annotation allows defining different groups of attributes to be initialized together when accessing one attribute in the group.

Table 3.7. Filters

Annotation	Description
`Filter`	Adds filters to an entity or a target entity of a collection.
`FilterDef`	Filter definition.
`FilterDefs`	Array of filter definitions.
`FilterJoinTable`	Adds filters to a join table collection.
`FilterJoinTables`	Adds multiple **`@FilterJoinTable`** to a collection.

Annotation	Description
`Filters`	Adds multiple **`@Filter`**.
`ParamDef`	A parameter definition.

Table 3.8. Primary Keys

Annotation	Description
`Generated`	This annotated property is generated by the database.
`GenericGenerator`	Generator annotation describing any kind of Hibernate generator in a detyped manner.
`GenericGenerators`	Array of generic generator definitions.
`NaturalId`	Specifies that a property is part of the natural id of the entity.
`Parameter`	Key/value pattern.
`RowId`	Support for **`ROWID`** mapping feature of Hibernate.

Table 3.9. Inheritance

Annotation	Description
`DiscriminatorFormula`	Discriminator formula to be placed at the root entity.
`DiscriminatorOptions`	Optional annotation to express Hibernate specific discriminator properties.
`MetaValue`	Maps a given discriminator value to the corresponding entity type.

Table 3.10. Mapping JP-QL/HQL Queries

Annotation	Description
`NamedNativeQueries`	Extends **`NamedNativeQueries`** to hold Hibernate NamedNativeQuery objects.

Annotation	Description
`NamedNativeQuery`	Extends **`NamedNativeQuery`** with Hibernate features.
`NamedQueries`	Extends **`NamedQueries`** to hold Hibernate **`NamedQuery`** objects.
`NamedQuery`	Extends **`NamedQuery`** with Hibernate features.

Table 3.11. Mapping Simple Properties

Annotation	Description
`AccessType`	Property access type.
`Columns`	Support an array of columns. Useful for component user type mappings.
`ColumnTransformer`	Custom SQL expression used to read the value from and write a value to a column. Use for direct object loading/saving as well as queries. The write expression must contain exactly one '?' placeholder for the value.
`ColumnTransformers`	Plural annotation for **`@ColumnTransformer`**. Useful when more than one column is using this behavior.

Table 3.12. Property

Annotation	Description
`Formula`	To be used as a replacement for **`@Column`** in most places. The formula has to be a valid SQL fragment.
`Index`	Defines a database index.
`JoinFormula`	To be used as a replacement for **`@JoinColumn`** in most places. The formula has to be a valid SQL fragment.
`Parent`	Reference the property as a pointer back to the owner (generally the owning entity).
`Type`	Hibernate type.

Annotation	Description
`TypeDef`	Hibernate type definition.
`TypeDefs`	Hibernate type definition array.

Table 3.13. Single Association Related Annotations

Annotation	Description
`Any`	Defines a **ToOne** association pointing to several entity types. Matching the according entity type is done through a metadata discriminator column. This kind of mapping should be only marginal.
`AnyMetaDef`	Defines `@Any` and `@ManyToAny` metadata.
`AnyMetaDefs`	Defines `@Any` and `@ManyToAny` set of metadata. Can be defined at the entity level or the package level.
`Fetch`	Defines the fetching strategy used for the given association.
`LazyCollection`	Defines the lazy status of a collection.
`LazyToOne`	Defines the lazy status of a ToOne association (i.e. `OneToOne` or `ManyToOne`).
`NotFound`	Action to do when an element is not found on an association.

Table 3.14. Optimistic Locking

Annotation	Description
`OptimisticLock`	Whether or not a change of the annotated property will trigger an entity version increment. If the annotation is not present, the property is involved in the optimistic lock strategy (default).
`OptimisticLocking`	Used to define the style of optimistic locking to be applied to an entity. In a hierarchy, only valid on the root entity.

Annotation	Description
`Source`	Optional annotation in conjunction with Version and timestamp version properties. The annotation value decides where the timestamp is generated.

CHAPTER 4. HIBERNATE QUERY LANGUAGE

4.1. ABOUT HIBERNATE QUERY LANGUAGE

Introduction to JPQL

The Java Persistence Query Language (JPQL) is a platform-independent object-oriented query language defined as part of the Java Persistence API (JPA) specification. JPQL is used to make queries against entities stored in a relational database. It is heavily inspired by SQL, and its queries resemble SQL queries in syntax, but operate against JPA entity objects rather than directly with database tables.

Introduction to HQL

The Hibernate Query Language (HQL) is a powerful query language, similar in appearance to SQL. Compared with SQL, however, HQL is fully object-oriented and understands notions like inheritance, polymorphism and association.

HQL is a superset of JPQL. An HQL query is not always a valid JPQL query, but a JPQL query is always a valid HQL query.

Both HQL and JPQL are non-type-safe ways to perform query operations. Criteria queries offer a type-safe approach to querying.

4.2. ABOUT HQL STATEMENTS

Both HQL and JPQL allow **`SELECT`**, **`UPDATE`**, and **`DELETE`** statements. HQL additionally allows **`INSERT`** statements, in a form similar to a**`SQL INSERT-SELECT`**.

The following table shows the syntax in Backus-Naur Form (BNF) notation for the various HQL statements.

Table 4.1. HQL Statements

Statement	Description

`SELECT`	The BNF for **`SELECT`** statements in HQL is:
	```
	select_statement :: =
	        [select_clause]
	        from_clause
	        [where_clause]
	        [groupby_clause]
	        [having_clause]
	        [orderby_clause]
	```

Statement	Description

`UPDATE`	The BNF for **`UPDATE`** statement in HQL is the same as it is in JPQL.
	
	`update_statement ::=`
	`update_clause [where_clause]`
	
	`update_clause ::= UPDATE`
	`entity_name [[AS]`
	`identification_variable]`
	`        SET update_item {,`
	`update_item}*`
	
	`update_item ::=`
	`[identification_variable.]`
	`{state_field |`
	`single_valued_object_field}`
	`        = new_value`
	
	`new_value ::= scalar_expression |`
	
	`simple_entity_expression |`
	`                NULL`

`DELETE`	The BNF for **`DELETE`** statements in HQL is the same as it is in JPQL.
	
	`delete_statement ::=`
	`delete_clause [where_clause]`
	
	`delete_clause ::= DELETE FROM`
	`entity_name [[AS]`
	`identification_variable]`

`INSERT`	The BNF for **`INSERT`** statement in HQL is:
	
	`insert_statement ::=`
	`insert_clause select_statement`
	
	`insert_clause ::= INSERT INTO`
	`entity_name (attribute_list)`
	
	`attribute_list ::= state_field[,`
	`state_field ]*`
	
	There is no JPQL equivalent to this.

WARNING

Hibernate allows the use of Data Manipulation Language (DML) to bulk insert, update and delete data directly in the mapped database through the Hibernate Query Language (HQL).

Using DML may violate the object/relational mapping and may affect object state. Object state stays in memory and by using DML, the state of an in-memory object is not affected, depending on the operation that is performed on the underlying database. In-memory data must be used with care if DML is used.

About the UPDATE and DELETE Statements

The pseudo-syntax for **`UPDATE`** and **`DELETE`** statements is:

`( UPDATE | DELETE ) FROM? EntityName (WHERE where_conditions)?`.

NOTE

The **`FROM`** keyword and the **`WHERE`** Clause are optional. The **`FROM`** clause is responsible for defining the scope of object model types available to the rest of the query. It also is responsible for defining all the identification variables available to the rest of the query. The **`WHERE`** clause allows you to refine the list of instances returned.

The result of execution of a **`UPDATE`** or **`DELETE`** statement is the number of rows that are actually affected (updated or deleted).

Example: Bulk Update Statement

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlUpdate = "update Company set name = :newName where name =
:oldName";
int updatedEntities = s.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

Example: Bulk Delete Statement

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlDelete = "delete Company where name = :oldName";
int deletedEntities = s.createQuery( hqlDelete )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

The **int** value returned by the**Query.executeUpdate()** method indicates the number of entities within the database that were affected by the operation.

Internally, the database might use multiple SQL statements to execute the operation in response to a DML **Update** or **Delete** request. This might be because of relationships that exist between tables and the join tables that need to be updated or deleted.

For example, issuing a delete statement, as in the example above, may actually result in deletes being executed against not just the **Company** table for companies that are named with **oldName**, but also against joined tables. Therefore a**Company** table in a bidirectional, many-to-many relationship with an **Employee** table would also lose rows from the corresponding join table, **Company_Employee**, as a result of the successful execution of the previous example.

The **deletedEntries** value above will contain a count of all the rows affected due to this operation, including the rows in the join tables.

IMPORTANT

Care should be taken when executing bulk update or delete operations because they may result in inconsistencies between the database and the entities in the active persistence context. In general, bulk update and delete operations should only be performed within a transaction in a new persistence context or before fetching or accessing entities whose state might be affected by such operations.

About the INSERT Statement

HQL adds the ability to define **INSERT** statements. There is no JPQL equivalent to this. The Backus-Naur Form (BNF) for an HQL **INSERT** statement is:

```
insert_statement ::= insert_clause select_statement

insert_clause ::= INSERT INTO entity_name (attribute_list)

attribute_list ::= state_field[, state_field ]*
```

The **attribute_list** is analogous to the column specification in the SQL**INSERT** statement. For entities involved in mapped inheritance, only attributes directly defined on the named entity can be used in the **attribute_list**. Superclass properties are not allowed and subclass properties do not make sense. In other words, **INSERT** statements are inherently non-polymorphic.

WARNING

The **select_statement** can be any valid HQL select query, with the caveat that the return types must match the types expected by the insert. Currently, this is checked during query compilation rather than allowing the check to relegate to the database. This can cause problems with Hibernate Types that are *equivalent* as opposed to *equal*. For example, this might cause mismatch issues between an attribute mapped as an **org.hibernate.type.DateType** and an attribute defined as a **org.hibernate.type.TimestampType**, even though the database might not make a distinction or might be able to handle the conversion.

For the **id** attribute, the insert statement gives you two options. You can either explicitly specify the **id** property in the **attribute_list**, in which case its value is taken from the corresponding select expression, or omit it from the **attribute_list** in which case a generated value is used. This latter option is only available when using **id** generators that operate "in the database"; attempting to use this option with any "in memory" type generators will cause an exception during parsing.

For optimistic locking attributes, the insert statement again gives you two options. You can either specify the attribute in the **attribute_list** in which case its value is taken from the corresponding select expressions, or omit it from the **attribute_list** in which case the **seed value** defined by the corresponding **org.hibernate.type.VersionType** is used.

Example: INSERT Query Statements

```
String hqlInsert = "insert into DelinquentAccount (id, name) select c.id,
c.name from Customer c where ...";
int createdEntities = s.createQuery(hqlInsert).executeUpdate();
```

Example: Bulk Insert Statement

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlInsert = "insert into Account (id, name) select c.id, c.name
from Customer c where ...";
int createdEntities = s.createQuery( hqlInsert )
        .executeUpdate();
tx.commit();
session.close();
```

If you do not supply the value for the **id** attribute using the **SELECT** statement, an identifier is generated for you, as long as the underlying database supports auto-generated keys. The return value of this bulk insert operation is the number of entries actually created in the database.

4.3. ABOUT HQL ORDERING

The results of the query can also be ordered. The **ORDER BY** clause is used to specify the selected values to be used to order the result. The types of expressions considered valid as part of the order-by clause include:

- state fields
- component/embeddable attributes
- scalar expressions such as arithmetic operations, functions, etc.
- identification variable declared in the select clause for any of the previous expression types

HQL does not mandate that all values referenced in the order-by clause must be named in the select clause, but it is required by JPQL. Applications desiring database portability should be aware that not all databases support referencing values in the order-by clause that are not referenced in the select clause.

Individual expressions in the order-by can be qualified with either **ASC** (ascending) or **DESC** (descending) to indicate the desired ordering direction.

Example: Order By

```
// legal because p.name is implicitly part of p
select p
from Person p
order by p.name

select c.id, sum( o.total ) as t
from Order o
    inner join o.customer c
group by c.id
order by t
```

4.4. ABOUT COLLECTION MEMBER REFERENCES

References to collection-valued associations actually refer to the *values* of that collection.

Example: Collection References

```
select c
from Customer c
    join c.orders o
    join o.lineItems l
    join l.product p
where o.status = 'pending'
  and p.status = 'backorder'

// alternate syntax
select c
from Customer c,
    in(c.orders) o,
    in(o.lineItems) l
    join l.product p
where o.status = 'pending'
  and p.status = 'backorder'
```

In the example, the identification variable **o** actually refers to the object model type Order which is the type of the elements of the Customer#orders association.

The example also shows the alternate syntax for specifying collection association joins using the **IN** syntax. Both forms are equivalent. Which form an application chooses to use is simply a matter of taste.

4.5. ABOUT QUALIFIED PATH EXPRESSIONS

It was previously stated that collection-valued associations actually refer to the *values* of that collection. Based on the type of collection, there are also available a set of explicit qualification expressions.

Table 4.2. Qualified Path Expressions

Expression	Description
VALUE	Refers to the collection value. Same as not specifying a qualifier. Useful to explicitly show intent. Valid for any type of collection-valued reference.
INDEX	According to HQL rules, this is valid for both Maps and Lists which specify a javax.persistence.OrderColumn annotation to refer to the Map key or the List position (aka the OrderColumn value). JPQL however, reserves this for use in the List case and adds **KEY** for the MAP case. Applications interested in JPA provider portability should be aware of this distinction.
KEY	Valid only for Maps. Refers to the map’s key. If the key is itself an entity, can be further navigated.
ENTRY	Only valid only for Maps. Refers to the Map’s logical java.util.Map.Entry tuple (the combination of its key and value). **ENTRY** is only valid as a terminal path and only valid in the select clause.

Example: Qualified Collection References

```
// Product.images is a Map<String,String> : key = a name, value = file
path

// select all the image file paths (the map value) for Product#123
select i
```

```
from Product p
    join p.images i
where p.id = 123

// same as above
select value(i)
from Product p
    join p.images i
where p.id = 123

// select all the image names (the map key) for Product#123
select key(i)
from Product p
    join p.images i
where p.id = 123

// select all the image names and file paths (the 'Map.Entry') for
Product#123
select entry(i)
from Product p
    join p.images i
where p.id = 123

// total the value of the initial line items for all orders for a customer
select sum( li.amount )
from Customer c
        join c.orders o
        join o.lineItems li
where c.id = 123
  and index(li) = 1
```

4.6. ABOUT HQL FUNCTIONS

HQL defines some standard functions that are available regardless of the underlying database in use. HQL can also understand additional functions defined by the dialect and the application.

4.6.1. About HQL Standardized Functions

The following functions are available in HQL regardless of the underlying database in use.

Table 4.3. HQL Standardized Functions

Function	Description
`BIT_LENGTH`	Returns the length of binary data.
`CAST`	Performs an SQL cast. The cast target should name the Hibernate mapping type to use.

Function	Description
`EXTRACT`	Performs an SQL extraction on datetime values. An extraction returns a part of the date/time value, for example, the year. See the abbreviated forms below.
`SECOND`	Abbreviated extract form for extracting the second.
`MINUTE`	Abbreviated extract form for extracting the minute.
`HOUR`	Abbreviated extract form for extracting the hour.
`DAY`	Abbreviated extract form for extracting the day.
`MONTH`	Abbreviated extract form for extracting the month.
`YEAR`	Abbreviated extract form for extracting the year.
`STR`	Abbreviated form for casting a value as character data.

4.6.2. About HQL Non-Standardized Functions

Hibernate dialects can register additional functions known to be available for that particular database product. They would only be available when using that database or dialect. Applications that aim for database portability should avoid using functions in this category.

Application developers can also supply their own set of functions. This would usually represent either custom SQL functions or aliases for snippets of SQL. Such function declarations are made by using the **`addSqlFunction`** method of **`org.hibernate.cfg.Configuration`**.

4.6.3. About the Concatenation Operation

HQL defines a concatenation operator in addition to supporting the concatenation (**`CONCAT`**) function. This is not defined by JPQL, so portable applications should avoid using it. The concatenation operator is taken from the SQL concatenation operator (**`||`**).

Example: Concatenation Operation Example

```
select 'Mr. ' || c.name.first || ' ' || c.name.last
from Customer c
where c.gender = Gender.MALE
```

4.7. ABOUT DYNAMIC INSTANTIATION

There is a particular expression type that is only valid in the select clause. Hibernate calls this "dynamic instantiation". JPQL supports some of this feature and calls it a "constructor expression".

Example: Dynamic Instantiation Example - Constructor

```
select new Family( mother, mate, offspr )
from DomesticCat as mother
    join mother.mate as mate
    left join mother.kittens as offspr
```

So rather than dealing with the Object[] here we are wrapping the values in a type-safe java object that will be returned as the results of the query. The class reference must be fully qualified and it must have a matching constructor.

The class here does not need to be mapped. If it does represent an entity, the resulting instances are returned in the NEW state (not managed!).

This is the part JPQL supports as well. HQL supports additional "dynamic instantiation" features. First, the query can specify to return a List rather than an Object[] for scalar results:

Example: Dynamic Instantiation Example - List

```
select new list(mother, offspr, mate.name)
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

The results from this query will be a List<List> as opposed to a List<Object[]>.

HQL also supports wrapping the scalar results in a Map.

Example: Dynamic Instantiation Example - Map

```
select new map( mother as mother, offspr as offspr, mate as mate )
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr

select new map( max(c.bodyWeight) as max, min(c.bodyWeight) as min,
count(*) as n )
from Cat cxt
```

The results from this query will be a List<Map<String,Object>> as opposed to a List<Object[]>. The keys of the map are defined by the aliases given to the select expressions.

4.8. ABOUT HQL PREDICATES

Predicates form the basis of the **where** clause, the **having** clause and searched case expressions. They are expressions which resolve to a truth value, generally **TRUE** or **FALSE**, although boolean comparisons involving NULL values generally resolve to **UNKNOWN**.

HQL Predicates

- Null Predicate
  Check a value for null. Can be applied to basic attribute references, entity references and parameters. HQL additionally allows it to be applied to component/embeddable types.

  Example: NULL Check

  ```
  // select everyone with an associated address
  select p
  from Person p
  where p.address is not null

  // select everyone without an associated address
  select p
  from Person p
    where p.address is null
  ```

- Like Predicate
  Performs a like comparison on string values. The syntax is:

  ```
  like_expression ::=
         string_expression
         [NOT] LIKE pattern_value
         [ESCAPE escape_character]
  ```

  The semantics follow that of the SQL like expression. The **pattern_value** is the pattern to attempt to match in the **string_expression**. Just like SQL, **pattern_value** can use **_** (underscore) and **%** (percent) as wildcards. The meanings are the same. The **_** matches any single character. The**%** matches any number of characters.

  The optional **escape_character** is used to specify an escape character used to escape the special meaning of **_** and **%** in the **pattern_value**. This is useful when needing to search on patterns including either **_** or **%**.

  Example: LIKE Predicate

  ```
  select p
  from Person p
  where p.name like '%Schmidt'

  select p
  from Person p
  where p.name not like 'Jingleheimmer%'

  // find any with name starting with "sp_"
  select sp
  from StoredProcedureMetadata sp
  where sp.name like 'sp|_%' escape '|'
  ```

- Between Predicate
  Analogous to the SQL **BETWEEN** expression. Perform an evaluation that a value is within the range of 2 other values. All the operands should have comparable types.

  Example: BETWEEN Predicate

  ```
  select p
  from Customer c
      join c.paymentHistory p
  where c.id = 123
    and index(p) between 0 and 9

  select c
  from Customer c
  where c.president.dateOfBirth
          between {d '1945-01-01'}
              and {d '1965-01-01'}

  select o
  from Order o
  where o.total between 500 and 5000

  select p
  from Person p
  where p.name between 'A' and 'E'
  ```

- IN Predicate
  The **IN** predicate performs a check that a particular value is in a list of values. Its syntax is:

  ```
  in_expression ::= single_valued_expression
              [NOT] IN single_valued_list

  single_valued_list ::= constructor_expression |
              (subquery) |
              collection_valued_input_parameter

  constructor_expression ::= (expression[, expression]*)
  ```

  The types of the **single_valued_expression** and the individual values in the **single_valued_list** must be consistent. JPQL limits the valid types here to string, numeric, date, time, timestamp, and enum types. In JPQL, **single_valued_expression** can only refer to:

  - "state fields", which is its term for simple attributes. Specifically this excludes association and component/embedded attributes.

  - entity type expressions.
    In HQL, **single_valued_expression** can refer to a far more broad set of expression types. Single-valued association are allowed. So are component/embedded attributes, although that feature depends on the level of support for tuple or "row value constructor syntax" in the underlying database. Additionally, HQL does not limit the value type in any way, though application

developers should be aware that different types may incur limited support based on the underlying database vendor. This is largely the reason for the JPQL limitations.

The list of values can come from a number of different sources. In the **`constructor_expression`** and **`collection_valued_input_parameter`**, the list of values must not be empty; it must contain at least one value.

Example: IN Predicate

```
select p
from Payment p
where type(p) in (CreditCardPayment, WireTransferPayment)

select c
from Customer c
where c.hqAddress.state in ('TX', 'OK', 'LA', 'NM')

select c
from Customer c
where c.hqAddress.state in ?

select c
from Customer c
where c.hqAddress.state in (
    select dm.state
    from DeliveryMetadata dm
    where dm.salesTax is not null
)

// Not JPQL compliant!
select c
from Customer c
where c.name in (
    ('John','Doe'),
    ('Jane','Doe')
)

// Not JPQL compliant!
select c
from Customer c
where c.chiefExecutive in (
    select p
    from Person p
    where ...
)
```

4.9. ABOUT RELATIONAL COMPARISONS

Comparisons involve one of the comparison operators - =, >, >=, <, ⇐, <>. HQL also defines != as a comparison operator synonymous with <>. The operands should be of the same type.

Example: Relational Comparison Examples

```
// numeric comparison
select c
from Customer c
where c.chiefExecutive.age < 30

// string comparison
select c
from Customer c
where c.name = 'Acme'

// datetime comparison
select c
from Customer c
where c.inceptionDate < {d '2000-01-01'}

// enum comparison
select c
from Customer c
where c.chiefExecutive.gender = com.acme.Gender.MALE

// boolean comparison
select c
from Customer c
where c.sendEmail = true

// entity type comparison
select p
from Payment p
where type(p) = WireTransferPayment

// entity value comparison
select c
from Customer c
where c.chiefExecutive = c.chiefTechnologist
```

Comparisons can also involve subquery qualifiers - **`ALL`**, **`ANY`**, **`SOME`**. **`SOME`** and **`ANY`** are synonymous.

The **`ALL`** qualifier resolves to true if the comparison is true for all of the values in the result of the subquery. It resolves to false if the subquery result is empty.

Example: ALL Subquery Comparison Qualifier Example

```
// select all players that scored at least 3 points
// in every game.
select p
from Player p
where 3 > all (
   select spg.points
   from StatsPerGame spg
   where spg.player = p
)
```

The **`ANY`**/**`SOME`** qualifier resolves to true if the comparison is true for at least one of the values in the result of the subquery. It resolves to false if the subquery result is empty.

4.10. BYTECODE ENHANCEMENT

4.10.1. Lazy Attribute Loading

Lazy attribute loading is a bytecode enhancement which allows you to tell Hibernate that only certain parts of an entity should be loaded upon fetching from the database, and when the other remaining parts should be loaded as well. This is different from proxy-based idea of lazy loading which is entity-centric where the entity’s state is loaded at once as needed. With bytecode enhancement, individual attributes or groups of attributes are loaded as needed.

Lazy attributes can be designated to be loaded together and this is called a *lazy group*. By default, all singular attributes are part of a single group. When one lazy singular attribute is accessed, all lazy singular attributes are loaded. Contrary to lazy singular group, lazy plural attributes are each a discrete lazy group. This behavior is explicitly controllable through the **`@org.hibernate.annotations.LazyGroup`** annotation.

```
@Entity
public class Customer {

    @Id
    private Integer id;

    private String name;

    @Basic( fetch = FetchType.LAZY )
    private UUID accountsPayableXrefId;

    @Lob
    @Basic( fetch = FetchType.LAZY )
    @LazyGroup( "lobs" )
    private Blob image;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public UUID getAccountsPayableXrefId() {
        return accountsPayableXrefId;
    }

    public void setAccountsPayableXrefId(UUID accountsPayableXrefId) {
        this.accountsPayableXrefId = accountsPayableXrefId;
    }

    public Blob getImage() {
        return image;
    }

    public void setImage(Blob image) {
        this.image = image;
    }
}
```

In the example above, there are two lazy attributes: **`accountsPayableXrefId`** and **`image`**. Each of these attributes is part of a different fetch group. The **`accountsPayableXrefId`** attribute is a part of the default fetch group, which means that accessing **`accountsPayableXrefId`** will not force the loading of the**`image`** attribute, and vice versa.

CHAPTER 5. HIBERNATE SERVICES

5.1. ABOUT HIBERNATE SERVICES

Services are classes that provide Hibernate with pluggable implementations of various types of functionality. Specifically they are implementations of certain service contract interfaces. The interface is known as the service role; the implementation class is known as the service implementation. Generally speaking, users can plug in alternate implementations of all standard service roles (overriding); they can also define additional services beyond the base set of service roles (extending).

5.2. ABOUT SERVICE CONTRACTS

The basic requirement for a service is to implement the marker interface org.hibernate.service.Service. Hibernate uses this internally for some basic type safety.

Optionally, the service can also implement the org.hibernate.service.spi.Startable and org.hibernate.service.spi.Stoppable interfaces to receive notifications of being started and stopped. Another optional service contract is org.hibernate.service.spi.Manageable which marks the service as manageable in JMX provided the JMX integration is enabled.

5.3. TYPES OF SERVICE DEPENDENCIES

Services are allowed to declare dependencies on other services using either of the following approaches:

@org.hibernate.service.spi.InjectService

Any method on the service implementation class accepting a single parameter and annotated with `@InjectService` is considered requesting injection of another service. By default the type of the method parameter is expected to be the service role to be injected. If the parameter type is different than the service role, the `serviceRole` attribute of the `InjectService` should be used to explicitly name the role.

By default injected services are considered required, that is the startup will fail if a named dependent service is missing. If the service to be injected is optional, the required attribute of the `InjectService` should be declared as`false`. The default is `true`.

org.hibernate.service.spi.ServiceRegistryAwareService

The second approach is a pull approach where the service implements the optional service interface `org.hibernate.service.spi.ServiceRegistryAwareService` which declares a single `injectServices` method.
During startup, Hibernate will inject the `org.hibernate.service.ServiceRegistry` itself into services which implement this interface. The service can then use the `ServiceRegistry` reference to locate any additional services it needs.

5.3.1. The Service Registry

5.3.1.1. About the ServiceRegistry

The central service API, aside from the services themselves, is the org.hibernate.service.ServiceRegistry interface. The main purpose of a service registry is to hold, manage and provide access to services.

Service registries are hierarchical. Services in one registry can depend on and utilize services in that same registry as well as any parent registries.

Use org.hibernate.service.ServiceRegistryBuilder to build a org.hibernate.service.ServiceRegistry instance.

Example Using ServiceRegistryBuilder to Create a ServiceRegistry

```
ServiceRegistryBuilder registryBuilder =
    new ServiceRegistryBuilder( bootstrapServiceRegistry );
    ServiceRegistry serviceRegistry =
registryBuilder.buildServiceRegistry();
```

5.3.2. Custom Services

5.3.2.1. About Custom Services

Once a **`org.hibernate.service.ServiceRegistry`** is built it is considered immutable; the services themselves might accept reconfiguration, but immutability here means adding or replacing services. So another role provided by the **`org.hibernate.service.ServiceRegistryBuilder`** is to allow tweaking of the services that will be contained in the **`org.hibernate.service.ServiceRegistry`** generated from it.

There are two means to tell a **`org.hibernate.service.ServiceRegistryBuilder`** about custom services.

- Implement a **`org.hibernate.service.spi.BasicServiceInitiator`** class to control on-demand construction of the service class and add it to the **`org.hibernate.service.ServiceRegistryBuilder`** using its **`addInitiator`** method.

- Just instantiate the service class and add it to the **`org.hibernate.service.ServiceRegistryBuilder`** using its **`addService`** method.

Either approach is valid for extending a registry, such as adding new service roles, and overriding services, such as replacing service implementations.

Example: Use ServiceRegistryBuilder to Replace an Existing Service with a Custom Service

```
ServiceRegistryBuilder registryBuilder =
    new ServiceRegistryBuilder(bootstrapServiceRegistry);
registryBuilder.addService(JdbcServices.class, new MyCustomJdbcService());
ServiceRegistry serviceRegistry = registryBuilder.buildServiceRegistry();

public class MyCustomJdbcService implements JdbcServices{

   @Override
   public ConnectionProvider getConnectionProvider() {
       return null;
   }
```

```
    @Override
    public Dialect getDialect() {
        return null;
    }

    @Override
    public SqlStatementLogger getSqlStatementLogger() {
        return null;
    }

    @Override
    public SqlExceptionHelper getSqlExceptionHelper() {
        return null;
    }

    @Override
    public ExtractedDatabaseMetaData getExtractedMetaDataSupport() {
        return null;
    }

    @Override
    public LobCreator getLobCreator(LobCreationContext lobCreationContext)
{
        return null;
    }

    @Override
    public ResultSetWrapper getResultSetWrapper() {
        return null;
    }
}
```

5.3.3. The Boot-Strap Registry

5.3.3.1. About the Boot-strap Registry

The boot-strap registry holds services that absolutely have to be available for most things to work. The main service here is the **ClassLoaderService** which is a perfect example. Even resolving configuration files needs access to class loading services i.e. resource look ups. This is the root registry, no parent, in normal use.

Instances of boot-strap registries are built using the **org.hibernate.service.BootstrapServiceRegistryBuilder** class.

Using BootstrapServiceRegistryBuilder

Example: Using BootstrapServiceRegistryBuilder

```
BootstrapServiceRegistry bootstrapServiceRegistry =
    new BootstrapServiceRegistryBuilder()
    // pass in org.hibernate.integrator.spi.Integrator instances which are
not
    // auto-discovered (for whatever reason) but which should be included
    .with(anExplicitIntegrator)
    // pass in a class loader that Hibernate should use to load
application classes
    .with(anExplicitClassLoaderForApplicationClasses)
    // pass in a class loader that Hibernate should use to load resources
    .with(anExplicitClassLoaderForResources)
    // see BootstrapServiceRegistryBuilder for rest of available methods
    ...
    // finally, build the bootstrap registry with all the above options
    .build();
```

5.3.3.2. BootstrapRegistry Services

`org.hibernate.service.classloading.spi.ClassLoaderService`

Hibernate needs to interact with class loaders. However, the manner in which Hibernate, or any library, should interact with class loaders varies based on the runtime environment that is hosting the application. Application servers, OSGi containers, and other modular class loading systems impose very specific class loading requirements. This service provides Hibernate an abstraction from this environmental complexity. And just as importantly, it does so in a single-swappable-component manner.
In terms of interacting with a class loader, Hibernate needs the following capabilities:

- the ability to locate application classes
- the ability to locate integration classes
- the ability to locate resources, such as properties files and XML files
- the ability to load **`java.util.ServiceLoader`**

NOTE

Currently, the ability to load application classes and the ability to load integration classes are combined into a single **load class** capability on the service. That may change in a later release.

`org.hibernate.integrator.spi.IntegratorService`

Applications, add-ons and other modules need to integrate with Hibernate. The previous approach required a component, usually an application, to coordinate the registration of each individual module. This registration was conducted on behalf of each module’s integrator.
This service focuses on the discovery aspect. It leverages the standard Java **`java.util.ServiceLoader`** capability provided by the **`org.hibernate.service.classloading.spi.ClassLoaderService`** in order to discover implementations of the **`org.hibernate.integrator.spi.Integrator`** contract.

Integrators would simply define a file named **`/META-INF/services/org.hibernate.integrator.spi.Integrator`** and make it available on the class path.

This file is used by the **`java.util.ServiceLoader`** mechanism. It lists, one per line, the fully qualified names of classes which implement the **`org.hibernate.integrator.spi.Integrator`** interface.

5.3.4. SessionFactory Registry

While it is best practice to treat instances of all the registry types as targeting a given **`org.hibernate.SessionFactory`**, the instances of services in this group explicitly belong to a single **`org.hibernate.SessionFactory`**.

The difference is a matter of timing in when they need to be initiated. Generally they need access to the **`org.hibernate.SessionFactory`** to be initiated. This special registry is **`org.hibernate.service.spi.SessionFactoryServiceRegistry`**.

5.3.4.1. SessionFactory Services

`org.hibernate.event.service.spi.EventListenerRegistry`

Description

Service for managing event listeners.

Initiator

`org.hibernate.event.service.internal.EventListenerServiceInitiator`

Implementations

`org.hibernate.event.service.internal.EventListenerRegistryImpl`

5.3.5. Integrators

The **`org.hibernate.integrator.spi.Integrator`** is intended to provide a simple means for allowing developers to hook into the process of building a functioning **`SessionFactory`**. The **`org.hibernate.integrator.spi.Integrator`** interface defines two methods of interest:

- **`integrate`** allows us to hook into the building process
- **`disintegrate`** allows us to hook into a**`SessionFactory`** shutting down.

NOTE

There is a third method defined in **`org.hibernate.integrator.spi.Integrator`**, an overloaded form of integrate, accepting a **`org.hibernate.metamodel.source.MetadataImplementor`** instead of **`org.hibernate.cfg.Configuration`**.

In addition to the discovery approach provided by the **`IntegratorService`**, applications can manually register Integrator implementations when building the **`BootstrapServiceRegistry`**.

5.3.5.1. Integrator Use Cases

The main use cases for an **`org.hibernate.integrator.spi.Integrator`** are registering event listeners and providing services, see **`org.hibernate.integrator.spi.ServiceContributingIntegrator`**.

Example: Registering Event Listeners

```
public class MyIntegrator implements
```

```
org.hibernate.integrator.spi.Integrator {

    public void integrate(
            Configuration configuration,
            SessionFactoryImplementor sessionFactory,
            SessionFactoryServiceRegistry serviceRegistry) {
        // As you might expect, an EventListenerRegistry is the thing with
which event listeners are registered  It is a
        // service so we look it up using the service registry
        final EventListenerRegistry eventListenerRegistry =
serviceRegistry.getService(EventListenerRegistry.class);

        // If you wish to have custom determination and handling of
"duplicate" listeners, you would have to add an
        // implementation of the
org.hibernate.event.service.spi.DuplicationStrategy contract like this

eventListenerRegistry.addDuplicationStrategy(myDuplicationStrategy);

        // EventListenerRegistry defines 3 ways to register listeners:
        //     1) This form overrides any existing registrations with
        eventListenerRegistry.setListeners(EventType.AUTO_FLUSH,
myCompleteSetOfListeners);
        //     2) This form adds the specified listener(s) to the
beginning of the listener chain
        eventListenerRegistry.prependListeners(EventType.AUTO_FLUSH,
myListenersToBeCalledFirst);
        //     3) This form adds the specified listener(s) to the end of
the listener chain
        eventListenerRegistry.appendListeners(EventType.AUTO_FLUSH,
myListenersToBeCalledLast);
    }
}
```

CHAPTER 6. HIBERNATE ENVERS

6.1. ABOUT HIBERNATE ENVERS

Hibernate Envers is an auditing and versioning system, providing JBoss EAP with a means to track historical changes to persistent classes. Audit tables are created for entities annotated with **`@Audited`**, which store the history of changes made to the entity. The data can then be retrieved and queried.

Envers allows developers to:

- audit all mappings defined by the JPA specification
- audit all hibernate mappings that extend the JPA specification
- audit entities mapped by or using the native Hibernate API
- log data for each revision using a revision entity
- query historical data

6.2. ABOUT AUDITING PERSISTENT CLASSES

Auditing of persistent classes is done in JBoss EAP through Hibernate Envers and the **`@Audited`** annotation. When the annotation is applied to a class, a table is created, which stores the revision history of the entity.

Each time a change is made to the class, an entry is added to the audit table. The entry contains the changes to the class, and is given a revision number. This means that changes can be rolled back, or previous revisions can be viewed.

6.3. AUDITING STRATEGIES

6.3.1. About Auditing Strategies

Auditing strategies define how audit information is persisted, queried and stored. There are currently two audit strategies available with Hibernate Envers:

Default Audit Strategy

- This strategy persists the audit data together with a start revision. For each row that is inserted, updated or deleted in an audited table, one or more rows are inserted in the audit tables, along with the start revision of its validity.
- Rows in the audit tables are never updated after insertion. Queries of audit information use subqueries to select the applicable rows in the audit tables, which are slow and difficult to index.

Validity Audit Strategy

- This strategy stores the start revision, as well as the end revision of the audit information. For each row that is inserted, updated or deleted in an audited table, one or more rows are inserted in the audit tables, along with the start revision of its validity.

- At the same time, the end revision field of the previous audit rows (if available) is set to this revision. Queries on the audit information can then use *between start and end revision*, instead of subqueries. This means that persisting audit information is a little slower because of the extra updates, but retrieving audit information is a lot faster.

- This can also be improved by adding extra indexes.

For more information on auditing, see About Auditing Persistent Classes. To set the auditing strategy for the application, see Set the Auditing Strategy.

6.3.2. Set the Auditing Strategy

There are two audit strategies supported by JBoss EAP:

- The default audit strategy
- The validity audit strategy

Define an Auditing Strategy
Configure the **`org.hibernate.envers.audit_strategy`** property in the **`persistence.xml`** file of the application. If the property is not set in the **`persistence.xml`** file, then the default audit strategy is used.

Set the Default Audit Strategy

```
<property name="org.hibernate.envers.audit_strategy"
value="org.hibernate.envers.strategy.DefaultAuditStrategy"/>
```

Set the Validity Audit Strategy

```
<property name="org.hibernate.envers.audit_strategy"
value="org.hibernate.envers.strategy.ValidityAuditStrategy"/>
```

6.3.3. Adding Auditing Support to a JPA Entity

JBoss EAP uses entity auditing, through About Hibernate Envers, to track the historical changes of a persistent class. This section covers adding auditing support for a JPA entity.

Add Auditing Support to a JPA Entity

1. Configure the available auditing parameters to suit the deployment. See Configure Envers Parameters for details.

2. Open the JPA entity to be audited.

3. Import the **`org.hibernate.envers.Audited`** interface.

4. Apply the **`@Audited`** annotation to each field or property to be audited, or apply it once to the whole class.

   Example: Audit Two Fields

```
import org.hibernate.envers.Audited;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.Column;

@Entity
public class Person {
    @Id
    @GeneratedValue
    private int id;

    @Audited
    private String name;

    private String surname;

    @ManyToOne
    @Audited
    private Address address;

    // add getters, setters, constructors, equals and hashCode here
}
```

Example: Audit an Entire Class

```
import org.hibernate.envers.Audited;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.GeneratedValue;
import javax.persistence.Column;

@Entity
@Audited
public class Person {
    @Id
    @GeneratedValue
    private int id;

    private String name;

    private String surname;

    @ManyToOne
    private Address address;

    // add getters, setters, constructors, equals and hashCode here
}
```

Once the JPA entity has been configured for auditing, a table called `_AUD` will be created to store the historical changes.

6.4. CONFIGURATION

6.4.1. Configure Envers Parameters

JBoss EAP uses entity auditing, through Hibernate Envers, to track the historical changes of a persistent class.

Configuring the Available Envers Parameters

1. Open the **persistence.xml** file for the application.

2. Add, remove or configure Envers properties as required. For a list of available properties, see Envers Configuration Properties.

   Example: Envers Parameters

```
<persistence-unit name="mypc">
  <description>Persistence Unit.</description>
  <jta-data-source>java:jboss/datasources/ExampleDS</jta-data-
source>
  <shared-cache-mode>ENABLE_SELECTIVE</shared-cache-mode>
  <properties>
    <property name="hibernate.hbm2ddl.auto" value="create-drop" />
    <property name="hibernate.show_sql" value="true" />
    <property name="hibernate.cache.use_second_level_cache"
value="true" />
    <property name="hibernate.cache.use_query_cache" value="true" />
    <property name="hibernate.generate_statistics" value="true" />
    <property name="org.hibernate.envers.versionsTableSuffix"
value="_V" />
    <property name="org.hibernate.envers.revisionFieldName"
value="ver_rev" />
  </properties>
</persistence-unit>
```

6.4.2. Enable or Disable Auditing at Runtime

Enable or Disable Entity Version Auditing at Runtime

1. Subclass the **AuditEventListener** class.

2. Override the following methods that are called on Hibernate events:

   - **onPostInsert**
   - **onPostUpdate**
   - **onPostDelete**
   - **onPreUpdateCollection**
   - **onPreRemoveCollection**
   - **onPostRecreateCollection**

3. Specify the subclass as the listener for the events.
4. Determine if the change should be audited.
5. Pass the call to the superclass if the change should be audited.

6.4.3. Configure Conditional Auditing

Hibernate Envers persists audit data in reaction to various Hibernate events, using a series of event listeners. These listeners are registered automatically if the Envers jar is in the class path.

Implement Conditional Auditing

1. Set the **`hibernate.listeners.envers.autoRegister`** Hibernate property to false in the **`persistence.xml`** file.
2. Subclass each event listener to be overridden. Place the conditional auditing logic in the subclass, and call the super method if auditing should be performed.
3. Create a custom implementation of **`org.hibernate.integrator.spi.Integrator`**, similar to **`org.hibernate.envers.event.EnversIntegrator`**. Use the event listener subclasses created in step two, rather than the default classes.
4. Add a **`META-INF/services/org.hibernate.integrator.spi.Integrator`** file to the jar. This file should contain the fully qualified name of the class implementing the interface.

6.4.4. Envers Configuration Properties

Table 6.1. Entity Data Versioning Configuration Parameters

Property Name	Default Value	Description
`org.hibernate.envers.audit_table_prefix`		A string that is prepended to the name of an audited entity, to create the name of the entity that will hold the audit information.
`org.hibernate.envers.audit_table_suffix`	_AUD	A string that is appended to the name of an audited entity to create the name of the entity that will hold the audit information. For example, if an entity with a table name of **`Person`** is audited, Envers will generate a table called **`Person_AUD`** to store the historical data.
`org.hibernate.envers.revision_field_name`	REV	The name of the field in the audit entity that holds the revision number.

Property Name	Default Value	Description
`org.hibernate.envers.revision_type_field_name`	REVTYPE	The name of the field in the audit entity that holds the type of revision. The current types of revisions possible are: **`add`**, **`mod`** and **`del`** for inserting, modifying or deleting respectively.
`org.hibernate.envers.revision_on_collection_change`	true	This property determines if a revision should be generated if a relation field that is not owned changes. This can either be a collection in a one-to-many relation, or the field using the **`mappedBy`** attribute in a one-to-one relation.
`org.hibernate.envers.do_not_audit_optimistic_locking_field`	true	When true, properties used for optimistic locking (annotated with **`@Version`**) will automatically be excluded from auditing.
`org.hibernate.envers.store_data_at_delete`	false	This property defines whether or not entity data should be stored in the revision when the entity is deleted, instead of only the ID, with all other properties marked as null. This is not usually necessary, as the data is present in the last-but-one revision. Sometimes, however, it is easier and more efficient to access it in the last revision. However, this means the data the entity contained before deletion is stored twice.
`org.hibernate.envers.default_schema`	null (same as normal tables)	The default schema name used for audit tables. Can be overridden using the **`@AuditTable(schema="…")`** annotation. If not present, the schema will be the same as the schema of the normal tables.
`org.hibernate.envers.default_catalog`	null (same as normal tables)	The default catalog name that should be used for audit tables. Can be overridden using the **`@AuditTable(catalog="…")`** annotation. If not present, the catalog will be the same as the catalog of the normal tables.

Property Name	Default Value	Description
`org.hibernate.envers.audit_strategy`	**`org.hibernate.envers.strategy.DefaultAuditStrategy`**	This property defines the audit strategy that should be used when persisting audit data. By default, only the revision where an entity was modified is stored. Alternatively, **`org.hibernate.envers.strategy.ValidityAuditStrategy`** stores both the start revision and the end revision. Together, these define when an audit row was valid.
`org.hibernate.envers.audit_strategy_validity_end_rev_field_name`	REVEND	The column name that will hold the end revision number in audit entities. This property is only valid if the validity audit strategy is used.
`org.hibernate.envers.audit_strategy_validity_store_revend_timestamp`	false	This property defines whether the timestamp of the end revision, where the data was last valid, should be stored in addition to the end revision itself. This is useful to be able to purge old audit records out of a relational database by using table partitioning. Partitioning requires a column that exists within the table. This property is only evaluated if the **`ValidityAuditStrategy`** is used.
`org.hibernate.envers.audit_strategy_validity_revend_timestamp_field_name`	REVEND_TSTMP	Column name of the timestamp of the end revision at which point the data was still valid. Only used if the **`ValidityAuditStrategy`** is used, and **`org.hibernate.envers.audit_strategy_validity_store_revend_timestamp`** evaluates to true.

6.5. QUERYING AUDIT INFORMATION

6.5.1. Retrieve Auditing Information Through Queries

Hibernate Envers provides the functionality to retrieve audit information through queries.

NOTE

Queries on the audited data will be, in many cases, much slower than corresponding queries on **`live`** data, as they involve correlated subselects.

Querying for Entities of a Class at a Given Revision

The entry point for this type of query is:

```
AuditQuery query = getAuditReader()
    .createQuery()
    .forEntitiesAtRevision(MyEntity.class, revisionNumber);
```

Constraints can then be specified, using the **`AuditEntity`** factory class. The query below only selects entities where the **`name`** property is equal to**`John`**:

```
query.add(AuditEntity.property("name").eq("John"));
```

The queries below only select entities that are related to a given entity:

```
query.add(AuditEntity.property("address").eq(relatedEntityInstance));
// or
query.add(AuditEntity.relatedId("address").eq(relatedEntityId));
```

The results can then be ordered, limited, and have aggregations and projections (except grouping) set. The example below is a full query.

```
List personsAtAddress = getAuditReader().createQuery()
    .forEntitiesAtRevision(Person.class, 12)
    .addOrder(AuditEntity.property("surname").desc())
    .add(AuditEntity.relatedId("address").eq(addressId))
    .setFirstResult(4)
    .setMaxResults(2)
    .getResultList();
```

Query Revisions where Entities of a Given Class Changed

The entry point for this type of query is:

```
AuditQuery query = getAuditReader().createQuery()
    .forRevisionsOfEntity(MyEntity.class, false, true);
```

Constraints can be added to this query in the same way as the previous example. There are additional possibilities for this query:

`AuditEntity.revisionNumber()`

Specify constraints, projections and order on the revision number in which the audited entity was modified.

`AuditEntity.revisionProperty(propertyName)`

Specify constraints, projections and order on a property of the revision entity, corresponding to the revision in which the audited entity was modified.

`AuditEntity.revisionType()`

Provides accesses to the type of the revision (ADD, MOD, DEL).

The query results can then be adjusted as necessary. The query below selects the smallest revision number at which the entity of the **`MyEntity`** class, with the**`entityId`** ID has changed, after revision number 42:

```
Number revision = (Number) getAuditReader().createQuery()
    .forRevisionsOfEntity(MyEntity.class, false, true)
    .setProjection(AuditEntity.revisionNumber().min())
    .add(AuditEntity.id().eq(entityId))
    .add(AuditEntity.revisionNumber().gt(42))
    .getSingleResult();
```

Queries for revisions can also minimize/maximize a property. The query below selects the revision at which the value of the **actualDate** for a given entity was larger than a given value, but as small as possible:

```
Number revision = (Number) getAuditReader().createQuery()
    .forRevisionsOfEntity(MyEntity.class, false, true)
    // We are only interested in the first revision
    .setProjection(AuditEntity.revisionNumber().min())
    .add(AuditEntity.property("actualDate").minimize()
        .add(AuditEntity.property("actualDate").ge(givenDate))
        .add(AuditEntity.id().eq(givenEntityId)))
    .getSingleResult();
```

The **minimize()** and **maximize()** methods return a criteria, to which constraints can be added, which must be met by the entities with the maximized/minimized properties.

There are two boolean parameters passed when creating the query.

selectEntitiesOnly

This parameter is only valid when an explicit projection is not set.
If **true**, the result of the query will be a list of entities that changed at revisions satisfying the specified constraints.
If **false**, the result will be a list of three element arrays. The first element will be the changed entity instance. The second will be an entity containing revision data. If no custom entity is used, this will be an instance of **DefaultRevisionEntity**. The third element array will be the type of the revision (ADD, MOD, DEL).

selectDeletedEntities

This parameter specifies if revisions in which the entity was deleted must be included in the results. If true, the entities will have the revision type **DEL**, and all fields, except id, will have the value **null**.

Query Revisions of an Entity that Modified a Given Property

The query below will return all revisions of **MyEntity** with a given id, where the**actualDate** property has been changed.

```
AuditQuery query = getAuditReader().createQuery()
  .forRevisionsOfEntity(MyEntity.class, false, true)
  .add(AuditEntity.id().eq(id));
  .add(AuditEntity.property("actualDate").hasChanged())
```

The **hasChanged** condition can be combined with additional criteria. The query below will return a horizontal slice for **MyEntity** at the time the revisionNumber was generated. It will be limited to the revisions that modified **prop1**, but not**prop2**.

```
AuditQuery query = getAuditReader().createQuery()
```

```
  .forEntitiesAtRevision(MyEntity.class, revisionNumber)
  .add(AuditEntity.property("prop1").hasChanged())
  .add(AuditEntity.property("prop2").hasNotChanged());
```

The result set will also contain revisions with numbers lower than the revisionNumber. This means that this query cannot be read as "Return all **MyEntities** changed in revisionNumber with **prop1** modified and **prop2** untouched."

The query below shows how this result can be returned, using the **forEntitiesModifiedAtRevision** query:

```
AuditQuery query = getAuditReader().createQuery()
  .forEntitiesModifiedAtRevision(MyEntity.class, revisionNumber)
  .add(AuditEntity.property("prop1").hasChanged())
  .add(AuditEntity.property("prop2").hasNotChanged());
```

Query Entities Modified in a Given Revision

The example below shows the basic query for entities modified in a given revision. It allows entity names and corresponding Java classes changed in a specified revision to be retrieved:

```
Set<Pair<String, Class>> modifiedEntityTypes = getAuditReader()
    .getCrossTypeRevisionChangesReader().findEntityTypes(revisionNumber);
```

There are a number of other queries that are also accessible from org.hibernate.envers.CrossTypeRevisionChangesReader:

List<Object> findEntities(Number)

Returns snapshots of all audited entities changed (added, updated and removed) in a given revision. Executes **n+1** SQL queries, where **n** is a number of different entity classes modified within the specified revision.

List<Object> findEntities(Number, RevisionType)

Returns snapshots of all audited entities changed (added, updated or removed) in a given revision filtered by modification type. Executes **n+1** SQL queries, where **n** is a number of different entity classes modified within specified revision. Map<RevisionType, List<Object>>

findEntitiesGroupByRevisionType(Number)

Returns a map containing lists of entity snapshots grouped by modification operation, for example, addition, update or removal. Executes **3n+1** SQL queries, where **n** is a number of different entity classes modified within specified revision.

6.5.2. Traversing Entity Associations Using Properties of Referenced Entities

You can use the properties of a referenced entity to traverse entities in a query. This enables you to query for one-to-one and many-to-one associations.

The examples below demonstrate some of the ways you can traverse entities in a query.

- In revision number 1, find cars where the owner is age 20 or lives at address number 30, then order the result set by car make.

```
List<Car> resultList = auditReader.createQuery()
                .forEntitiesAtRevision( Car.class, 1 )
                .traverseRelation( "owner", JoinType.INNER, "p" )
                .traverseRelation( "address", JoinType.INNER, "a" )
                .up().up().add(
AuditEntity.disjunction().add(AuditEntity.property( "p", "age" )
                       .eq( 20 ) ).add( AuditEntity.property( "a",
"number" ).eq( 30 ) ) )
                .addOrder( AuditEntity.property( "make" ).asc()
).getResultList();
```

- In revision number 1, find the car where the owner age is equal to the owner address number.

```
Car result = (Car) auditReader.createQuery()
                .forEntitiesAtRevision( Car.class, 1 )
                .traverseRelation( "owner", JoinType.INNER, "p" )
                .traverseRelation( "address", JoinType.INNER, "a" )
                .up().up().add(AuditEntity.property( "p", "age" )
                        .eqProperty( "a", "number" )
).getSingleResult();
```

- In revision number 1, find all cars where the owner is age 20 or where there is no owner.

```
List<Car> resultList = auditReader.createQuery()
                .forEntitiesAtRevision( Car.class, 1 )
                .traverseRelation( "owner", JoinType.LEFT, "p" )
                .up().add( AuditEntity.or( AuditEntity.property(
"p", "age").eq( 20 ),
                        AuditEntity.relatedId( "owner" ).eq( null )
) )
                .addOrder( AuditEntity.property( "make" ).asc()
).getResultList();
```

- In revision number 1, find all cars where the make equals "car3", and where the owner is age 30 or there is no no owner.

```
List<Car> resultList = auditReader.createQuery()
                .forEntitiesAtRevision( Car.class, 1 )
                .traverseRelation( "owner", JoinType.LEFT, "p" )
                .up().add( AuditEntity.and( AuditEntity.property(
"make" ).eq( "car3" ), AuditEntity.property( "p", "age" ).eq( 30 ) )
)
                .getResultList();
```

- In revision number 1, find all cars where the make equals "car3" or where or the owner is age 10 or where there is no owner.

```
List<Car> resultList = auditReader.createQuery()
                .forEntitiesAtRevision( Car.class, 1 )
                .traverseRelation( "owner", JoinType.LEFT, "p" )
                .up().add( AuditEntity.or( AuditEntity.property(
"make" ).eq( "car3" ), AuditEntity.property( "p", "age" ).eq( 10 ) )
)
                .getResultList();
```

6.6. PERFORMANCE TUNING

6.6.1. Alternative Batch Loading Algorithms

Hibernate allows you to load data for associations using one of four fetching strategies: join, select, subselect and batch. Out of these four strategies, batch loading allows for the biggest performance gains as it is an optimization strategy for select fetching. In this strategy, Hibernate retrieves a batch of entity instances or collections in a single SELECT statement by specifying a list of primary or foreign keys. Batch fetching is an optimization of the lazy select fetching strategy.

There are two ways to configure batch fetching: per-class level or per-collection level.

- Per-class Level
  When Hibernate loads data on a per-class level, it requires the batch size of the association to pre-load when queried. For example, consider that at runtime you have 30 instances of a **car** object loaded in session. Each**car** object belongs to an **owner** object. If you were to iterate through all the**car** objects and request their owners, with **lazy** loading, Hibernate will issue 30 select statements - one for each owner. This is a performance bottleneck.

  You can instead, tell Hibernate to pre-load the data for the next batch of owners before they have been sought via a query. When an **owner** object has been queried, Hibernate will query many more of these objects in the same SELECT statement.

  The number of **owner** objects to query in advance depends upon the**batch-size** parameter specified at configuration time:

  ```
  <class name="owner" batch-size="10"></class>
  ```

  This tells Hibernate to query at least 10 more **owner** objects in expectation of them being needed in the near future. When a user queries the **owner** of **car A**, the **owner** of **car B** may already have been loaded as part of batch loading. When the user actually needs the **owner** of **car B**, instead of going to the database (and issuing a SELECT statement), the value can be retrieved from the current session.

  In addition to the **batch-size** parameter, Hibernate 4.2.0 has introduced a new configuration item to improve in batch loading performance. The configuration item is called **Batch Fetch Style** configuration and specified by the **hibernate.batch_fetch_style** parameter.

  Three different batch fetch styles are supported: LEGACY, PADDED and DYNAMIC. To specify which style to use, use **org.hibernate.cfg.AvailableSettings#BATCH_FETCH_STYLE**.

  - LEGACY: In the legacy style of loading, a set of pre-built batch sizes based on **ArrayHelper.getBatchSizes(int)** are utilized. Batches are loaded using the next-smaller pre-built batch size from the number of existing batchable identifiers.
    Continuing with the above example, with a **batch-size** setting of 30, the pre-built batch sizes would be [30, 15, 10, 9, 8, 7, .., 1]. An attempt to batch load 29

identifiers would result in batches of 15, 10, and 4. There will be 3 corresponding SQL queries, each loading 15, 10 and 4 owners from the database.

  - PADDED - Padded is similar to LEGACY style of batch loading. It still utilizes pre-built batch sizes, but uses the next-bigger batch size and pads the extra identifier placeholders.
    As with the example above, if 30 owner objects are to be initialized, there will only be one query executed against the database.

    However, if 29 owner objects are to be initialized, Hibernate will still execute only one SQL select statement of batch size 30, with the extra space padded with a repeated identifier.

  - Dynamic - While still conforming to batch-size restrictions, this style of batch loading dynamically builds its SQL SELECT statement using the actual number of objects to be loaded.
    For example, for 30 owner objects, and a maximum batch size of 30, a call to retrieve 30 owner objects will result in one SQL SELECT statement. A call to retrieve 35 will result in two SQL statements, of batch sizes 30 and 5 respectively. Hibernate will dynamically alter the second SQL statement to keep at 5, the required number, while still remaining under the restriction of 30 as the batch-size. This is different to the PADDED version, as the second SQL will not get PADDED, and unlike the LEGACY style, there is no fixed size for the second SQL statement - the second SQL is created dynamically.

    For a query of less than 30 identifiers, this style will dynamically only load the number of identifiers requested.

- Per-collection Level
  Hibernate can also batch load collections honoring the batch fetch size and styles as listed in the per-class section above.

  To reverse the example used in the previous section, consider that you need to load all the **`car`** objects owned by each **`owner`** object. If 10 **`owner`** objects are loaded in the current session iterating through all owners will generate 10 SELECT statements, one for every call to **`getCars()`** method. If you enable batch fetching for the cars collection in the mapping of Owner, Hibernate can pre-fetch these collections, as shown below.

  ```
  <class name="Owner"><set name="cars" batch-size="5"></set></class>
  ```

  Thus, with a batch size of five and using legacy batch style to load 10 collections, Hibernate will execute two SELECT statements, each retrieving five collections.

6.6.2. Second Level Caching of Object References for Non-mutable Data

Hibernate automatically caches data within memory for improved performance. This is accomplished by an in-memory cache which reduces the number of times that database lookups are required, especially for data that rarely changes.

Hibernate maintains two types of caches. The primary cache, also called the first-level cache, is mandatory. This cache is associated with the current session and all requests must pass through it. The secondary cache, also called the second-level cache, is optional, and is only consulted after the primary cache has been consulted.

Data is stored in the second-level cache by first disassembling it into a state array. This array is deep copied, and that deep copy is put into the cache. The reverse is done for reading from the cache. This works well for data that changes (mutable data), but is inefficient for immutable data.

Deep copying data is an expensive operation in terms of memory usage and processing speed. For large data sets, memory and processing speed become a performance-limiting factor. Hibernate allows you to specify that immutable data be referenced rather than copied. Instead of copying entire data sets, Hibernate can now store the reference to the data in the cache.

This can be done by changing the value of the configuration setting **`hibernate.cache.use_reference_entries`** to **`true`**. By default, **`hibernate.cache.use_reference_entries`** is set to **`false`**.

When **`hibernate.cache.use_reference_entries`** is set to **`true`**, an immutable data object that does not have any associations is not copied into the second-level cache, and only a reference to it is stored.

WARNING

When **`hibernate.cache.use_reference_entries`** is set to **`true`**, immutable data objects with associations are still deep copied into the second-level cache.

CHAPTER 7. HIBERNATE SEARCH

7.1. GETTING STARTED WITH HIBERNATE SEARCH

7.1.1. About Hibernate Search

Hibernate Search provides full-text search capability to Hibernate applications. It is especially suited to search applications for which SQL-based solutions are not suited, including: full-text, fuzzy and geolocation searches. Hibernate Search uses Apache Lucene as its full-text search engine, but is designed to minimize the maintenance overhead. Once it is configured, indexing, clustering and data synchronization is maintained transparently, allowing you to focus on meeting your business requirements.

NOTE

The prior release of JBoss EAP included Hibernate 4.2 and Hibernate Search 4.6. JBoss EAP 7 includes Hibernate 5 and Hibernate Search 5.5.

Hibernate Search 5.5 works with Java 7 and now builds upon Lucene 5.3.x. If you are using any native Lucene APIs make sure to align with this version.

7.1.2. 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 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 having to interact with the Apache Lucene APIs directly. Instead, interaction with the underlying Lucene indexes is handled via an **`IndexManager`**. By default there is a one-to-one relationship between IndexManager and Lucene index. The IndexManager abstracts the specific index configuration, including the selected *back end*, *reader strategy* and the *DirectoryProvider*.

Once the index is created, you can search for entities and return lists of managed entities instead of dealing with the underlying Lucene infrastructure. The same persistence context is shared between Hibernate and Hibernate Search. The **`FullTextSession`** class is built on top of the Hibernate **`Session`** class so that the application code can use the unified **`org.hibernate.Query`** or **`javax.persistence.Query`** APIs exactly the same way an HQL, JPA-QL, or native query would.

Transactional batching mode is recommended for all operations, whether or not they are JDBC-based.

NOTE

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

NOTE

Hibernate Search works perfectly fine in the Hibernate or EntityManager long conversation pattern, known as atomic conversation.

7.1.3. About the Directory Provider

Apache Lucene, which is part of the Hibernate Search infrastructure, has the concept of a Directory for storage of indexes. Hibernate Search handles the initialization and configuration of a Lucene Directory instance via a *Directory Provider*.

The **`directory_provider`** property specifies the directory provider to be used to store the indexes. The default file system directory provider is **`filesystem`**, which uses the local file system to store indexes.

7.1.4. About the Worker

Updates to Lucene indexes are handled by the Hibernate Search *Worker*, which receives all entity changes, queues them by context and applies them once a context ends. The most common context is the transaction, but may be dependent on the number of entity changes or some other application events.

For better efficiency, interactions are batched and generally applied once the context ends. Outside a transaction, the index update operation is executed right after the actual database operation. 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. A worker may be configured with a specific batch size limit, after which indexing occurs regardless of the context.

There are two immediate benefits to this method of handling index updates:

- 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, but ACID behavior is rarely useful for full text search indexes since they can be rebuilt from the source at any time.

The two batch modes, no scope vs transactional, are the equivalent of autocommit versus transactional behavior. From a performance perspective, the *transactional* mode is recommended. The scoping choice is made transparently. Hibernate Search detects the presence of a transaction and adjust the scoping.

7.1.5. Back End Setup and Operations

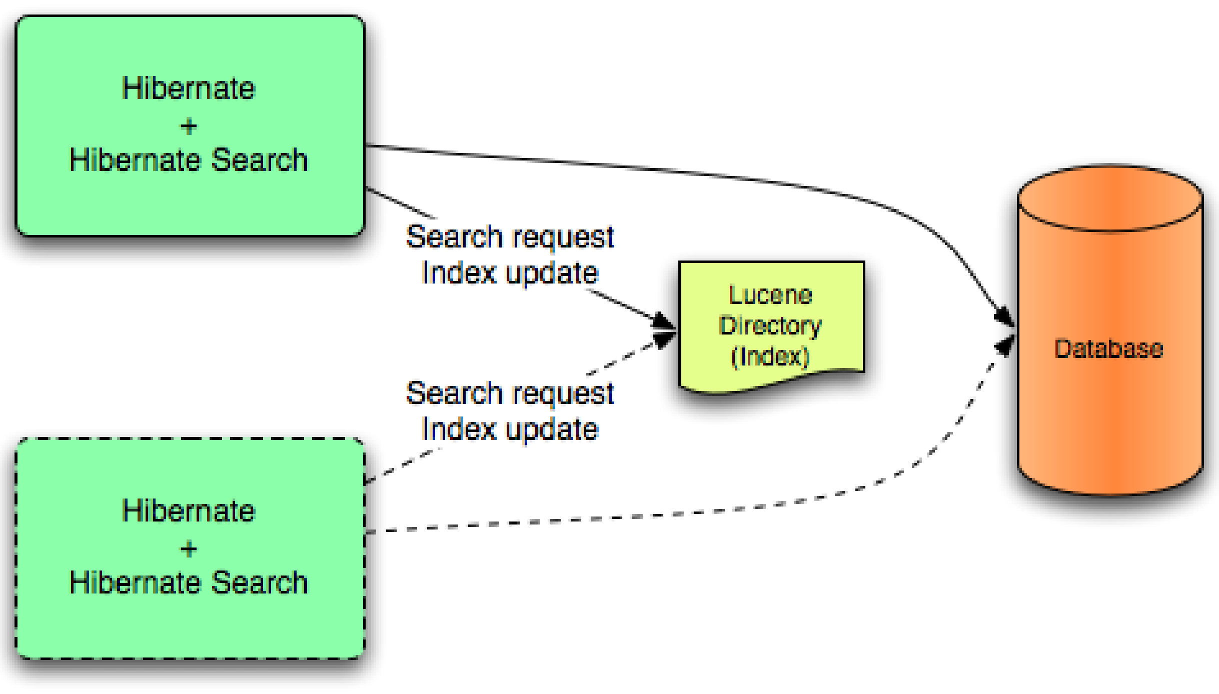
7.1.5.1. Back End

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

7.1.5.2. Lucene

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

Figure 7.1. Lucene Back-end Configuration

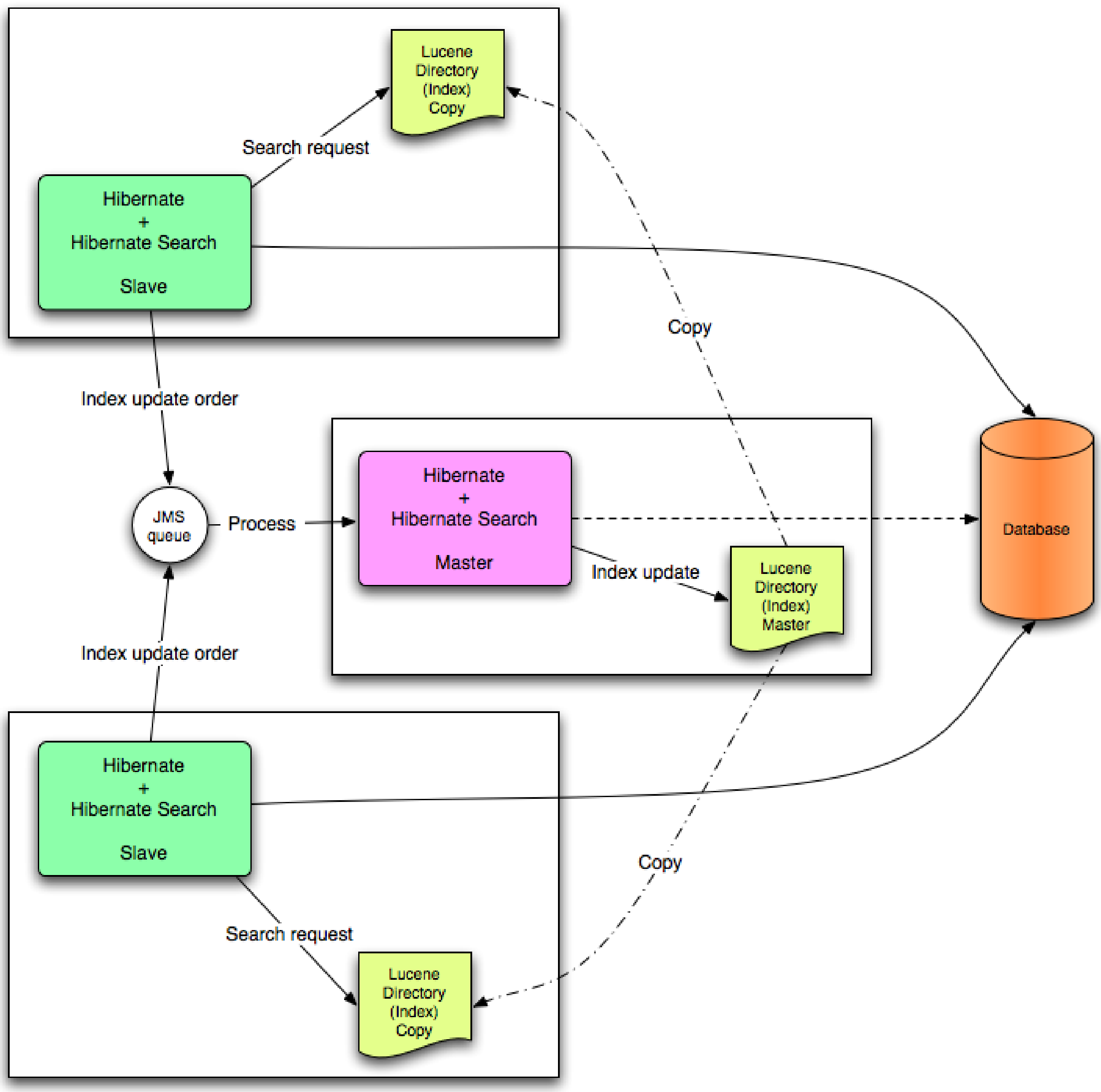


Lucene mode targets non-clustered or clustered applications where the directory manages the locking strategy. The primary advantage of Lucene mode is simplicity and immediate visibility of changes in Lucene queries. The Near Real Time (NRT) back end is an alternative back end for non-clustered and non-shared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

7.1.5.3. JMS

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

Figure 7.2. JMS Back-end Configuration


Lucene Directory (Index) Copy
Search request
Hibernate + Hibernate Search Slave
Copy
Index update order
JMS queue
Process
Hibernate + Hibernate Search Master
Index update
Lucene Directory (Index) Master
Database
Index update order
Hibernate + Hibernate Search Slave
Copy
Search request
Lucene Directory (Index) Copy

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

7.1.6. Reader Strategies

When executing a query, Hibernate Search uses a reader strategy to interact with the Apache Lucene indexes. Choose a reader strategy based on the profile of the application like frequent updates, read mostly, asynchronous index update.

7.1.6.1. The Shared Strategy

Using the **shared** strategy, Hibernate Search shares the same**IndexReader** for a given Lucene index across multiple queries and threads provided that the **IndexReader** remains updated. If the **IndexReader** is not updated, a new one is opened and provided. Each

IndexReader is made of several **SegmentReaders**. The shared strategy reopens segments that have been modified or created after the last opening and shares the already loaded segments from the previous instance. This is the default strategy.

7.1.6.2. The Not-shared Strategy

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

7.1.6.3. Custom Reader Strategies

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

7.2. CONFIGURATION

7.2.1. Minimum Configuration

Hibernate Search has been designed to provide flexibility in its configuration and operation, with default values carefully chosen to suit the majority of use cases. At a minimum a Directory Provider must be configured, along with its properties. The default Directory Provider is **filesystem**, which uses the local file system for index storage. For details of available Directory Providers and their configuration, see DirectoryProvider Configuration.

If you are using Hibernate directly, settings such as the DirectoryProvider must be set in the configuration file, either **hibernate.properties** or **hibernate.cfg.xml**. If you are using Hibernate via JPA, the configuration file is **persistence.xml**.

7.2.2. Configuring the IndexManager

Hibernate Search offers several implementations for this interface:

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

To specify an IndexManager other than the default, specify the following property:

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

7.2.2.1. Directory-based

The **Directory-based** implementation is the default **IndexManager** implementation. It is highly configurable and allows separate configurations for the reader strategy, back ends, and directory providers.

7.2.2.2. Near Real Time

The **NRTIndexManager** is an extension of the default **IndexManager** and leverages the

Lucene NRT, Near Real Time, feature for low latency index writes. However, it ignores configuration settings for alternative back ends other than **lucene** and acquires exclusive write locks on the **Directory**.

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

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

7.2.2.3. Custom

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

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

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

7.2.3. DirectoryProvider Configuration

A **DirectoryProvider** is the Hibernate Search abstraction around a Lucene**Directory** and handles the configuration and the initialization of the underlying Lucene resources. Directory Providers and Their Properties shows the list of the directory providers available in Hibernate Search together with their corresponding options.

Each indexed entity is associated with a Lucene index (except of the case where multiple entities share the same index). 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).

The DirectoryProvider and any additional options can be configured 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. 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**.

Specifying the Index Name

```
package org.hibernate.example;
```

```
@Indexed
public class Status { ... }

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

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

Configuring Directory Providers

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

NOTE

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

Directory Providers and Their Properties

ram

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 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)`**. See the Javadocs for 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).

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.

- **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 amount of MegaBytes to move in a single low level copy instruction; defaults to 16MB.
- **locking_strategy** : optional, see 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)`**. See the Javadocs for these Directory implementations before changing this setting. Even though **`NIOFSDirectory`** or **`MMapDirectory`** can bring substantial performance boosts, there are also issues of which you need to be aware.

`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 LockFactory Configuration
- **retry_marker_lookup** : optional, default to 0. Defines how many times Hibernate Search checks 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 cannot find the master index it will try again until it is found in background, without preventing the application to start: fullText queries performed before the index is initialized are not blocked but will return empty results. When not enabling the option or explicitly setting it to zero it will fail with an exception instead of scheduling a retry timer. To prevent the application from starting without an invalid index but still control an initialization timeout, see `retry_marker_lookup` instead.
- **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)`. See the Javadocs for these Directory implementations before changing this setting. Even though `NIOFSDirectory` or `MMapDirectory` can bring substantial performance boosts you need also to be aware of the issues.

NOTE

If the built-in directory providers do not fit your needs, you can write your own directory provider by implementing the `org.hibernate.store.DirectoryProvider` interface. In this case, pass the fully qualified class name of your provider into the `directory_provider` property. You can pass any additional properties using the prefix `hibernate.search.`*`<indexname>`*.

7.2.4. 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.

Use the worker configuration to refine how Infinispan Query interacts with Lucene. Several architectural components and possible extension points are available for this configuration.

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.

Table 7.1. Scope Configuration

Property	Description
`hibernate.search.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.worker.*`	All configuration properties prefixed with **`hibernate.search.worker`** are passed to the Worker during initialization. This allows adding custom, worker specific parameters.
`hibernate.search.worker.batch_size`	Defines the maximum number of indexing operation batched per context. Once the limit is reached indexing will be triggered even though the context has not ended yet. This property only works if the **`Worker`** implementation delegates the queued work to BatchedQueueingProcessor, which is what the **`TransactionalWorker`** does.

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

NOTE

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

Table 7.2. Execution Configuration

Property	Description
`hibernate.search.<indexName>.worker.execution`	**`sync`**: synchronous execution (default) **`async`**: asynchronous execution
`hibernate.search.<indexName>.worker.thread_pool.size`	The back end can apply updates from the same transaction context (or batch) in parallel, using a thread pool. The default value is 1. You can experiment with larger values if you have many operations per transaction.

Property	Description
`hibernate.search.<indexName>.worker.buffer_queue.max`	Defines the maximal number of work queue if the thread pool is starved. Useful only for asynchronous execution. Default to infinite. If the limit is reached, the work is done by the main thread.

So far all work is done within the same virtual machine (VM), no matter which execution mode. The total amount of work has not changed for the single VM. Luckily there is a better approach, namely delegation. It is possible to send the indexing work to a different server by configuring **`hibernate.search.default.worker.backend`**. Again this option can be configured differently for each index.

Table 7.3. Back-end Configuration

Property	Description
`hibernate.search.<indexName>.worker.backend`	**`lucene`**: The default back end which runs index updates in the same VM. Also used when the property is undefined or empty. **`jms`**: JMS back end. Index updates are send to a JMS queue to be processed by an indexing master. See JMS Back-end Configuration for additional configuration options and 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 **`BackendQueueProcessor`**. This way you can implement your own communication layer. The implementation is responsible for returning a **`Runnable`** instance which on execution will process the index work.

Table 7.4. JMS Back-end Configuration

Property	Description
`hibernate.search.<indexName>.worker.jndi.*`	Defines the JNDI properties to initiate the InitialContext, if necessary. JNDI is only used by the JMS back end.

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

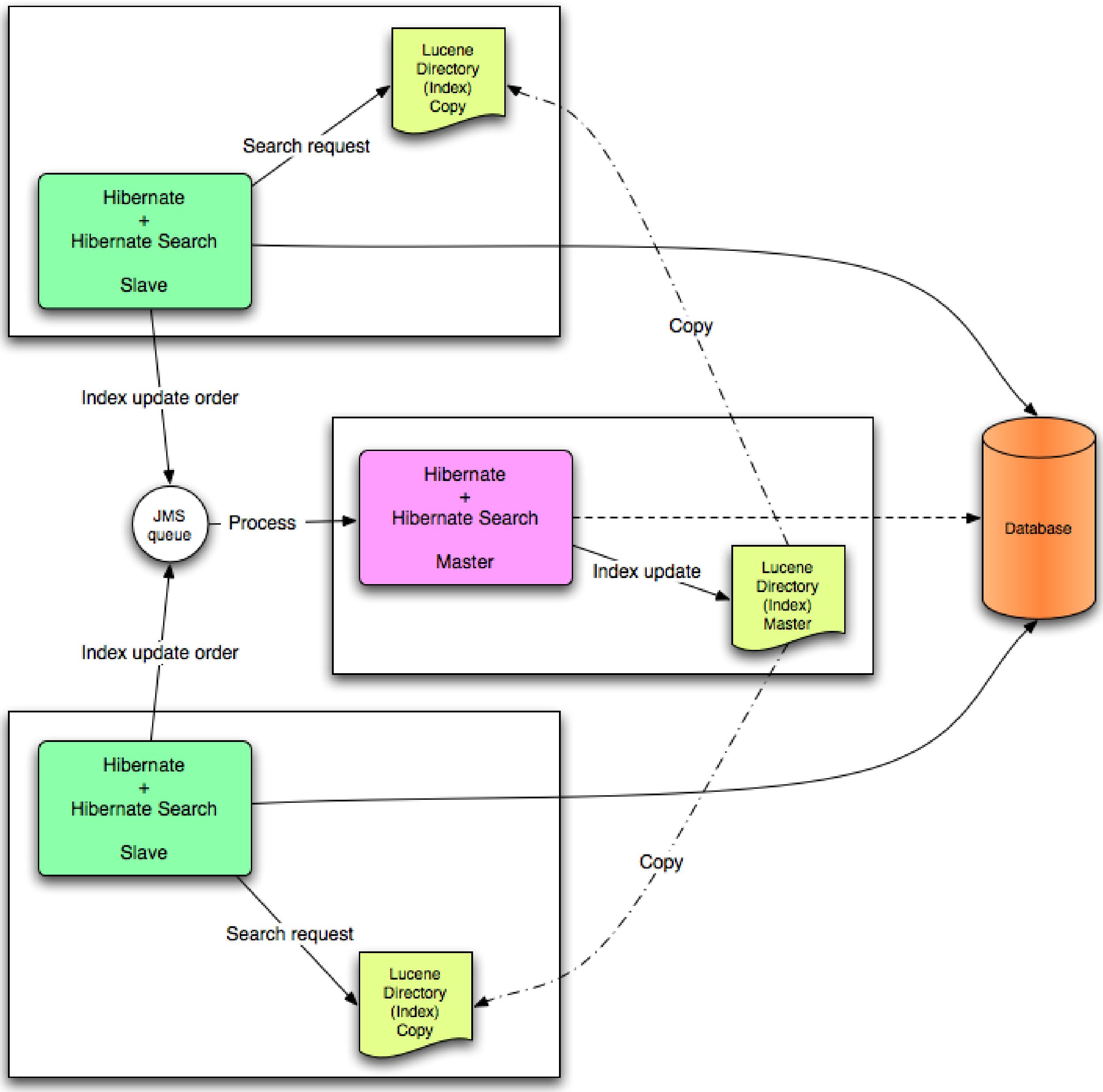
WARNING

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

7.2.4.1. JMS Master/Slave Back End

This section describes in greater detail how to configure the master/slave Hibernate Search architecture.

Figure 7.3. JMS Backend Configuration


Lucene Directory (Index) Copy
Search request
Hibernate + Hibernate Search Slave
Copy
Index update order
JMS queue
Process
Hibernate + Hibernate Search Master
Index update
Lucene Directory (Index) Master
Database
Index update order
Hibernate + Hibernate Search Slave
Copy
Search request
Lucene Directory (Index) Copy

7.2.4.2. Slave Nodes

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

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

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

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

NOTE

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

7.2.4.3. Master Node

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

Index update operations in the JMS queue are executed and the master index is copied regularly.

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

## Back-end configuration
#Back-end is the default for Lucene
```

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.

Message-driven Bean Processing the Indexing Queue

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

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

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

This example inherits from the abstract JMS controller class available in the Hibernate Search source code and implements a Java EE MDB. This implementation is given as an example and can be adjusted to make use of non Java EE message-driven beans.

7.2.5. Tuning Lucene Indexing

7.2.5.1. Tuning Lucene Indexing Performance

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

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

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

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

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

- **max_merge_docs** = 10
- **merge_factor** = 20
- **ram_buffer_size** = 64MB

- **term_index_interval** = Lucene default

All other values will use the defaults defined in Lucene.

The default for all values is to leave them at Lucene's own default. The values listed in 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**.

NOTE

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

Table 7.5. Indexing Performance and Behavior Properties

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

Property	Description	Default Value
`hibernate.search.[default|<indexname>].indexwriter.merge_factor`	Controls segment merge frequency and size. Determines how often segment indexes are merged when insertion occurs. With smaller values, less RAM is used while indexing, and searches on unoptimized indexes are faster, but indexing speed is slower. With larger values, more RAM is used during indexing, and while searches on unoptimized indexes are slower, indexing is faster. Thus larger values (> 10) are best for batch index creation, and smaller values (< 10) for indexes that are interactively maintained. The value must not be lower than 2.	10
`hibernate.search.[default|<indexname>].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 though 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.maxMergeSizeForOptimize`**.	Unlimited

Property	Description	Default Value
`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`**
`hibernate.search.[default|<indexname>].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 is 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`	Set the interval between indexed terms. Large values cause less memory to be used by IndexReader, but slow random-access to terms.Small values cause more memory to be used by an IndexReader, and speed random-access to terms. See Lucene documentation for more details.	128
`hibernate.search.[default|<indexname>].indexwriter.use_compound_file`	The advantage of using the compound file format is that less file descriptors are used. The disadvantage is that indexing takes more time and temporary disk space. You can set this parameter to **`false`** in an attempt to improve the indexing time, but you could run out of file descriptors if **`mergeFactor`** is also large. Boolean parameter, use **`true`** or **`false`**. The default value for this option is **`true`**.	true

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

WARNING

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

7.2.5.2. The Lucene IndexWriter

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

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

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

7.2.5.3. Performance Option Configuration

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

Example performance option configuration

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

- **`max_merge_docs`** = 10

- **`merge_factor`** = 20
- **`ram_buffer_size`** = 64MB
- **`term_index_interval`** = Lucene default

All other values will use the defaults defined in Lucene.

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

NOTE

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

Table 7.6. Indexing Performance and Behavior Properties

Property	Description	Default Value
`default.<indexname>.exclusive_index_use`	Set to **`true`** when no other process will need to write to the same index. This enables Hibernate Search to work in exclusive mode on the index and improve performance when writing changes to the index.	**`true`** (improved performance, releases locks only at shutdown)
`default.<indexname>.max_queue_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 does not make much sense unless the **`worker.execution`** is configured as **`async`**.	**`1000`**
`default.<indexname>.indexwriter.max_buffered_delete_terms`	Determines the minimal number of delete terms required before the buffered in-memory delete terms are applied and flushed. If there are documents buffered in memory at the time, they are merged and a new segment is created.	Disabled (flushes by RAM usage)
`default.<indexname>.indexwriter.max_buffered_docs`	Controls the amount of documents buffered in memory during indexing. The bigger the more RAM is consumed.	Disabled (flushes by RAM usage)
`default.<indexname>.indexwriter.max_merge_docs`	Defines the largest number of documents allowed in a segment. Smaller values perform better on frequently changing indexes, larger values provide better search performance if the index does not change often.	Unlimited (Integer.MAX_VALUE)

Property	Description	Default Value
`default.<indexname>.indexwriter.merge_factor`	Controls segment merge frequency and size. Determines how often segment indexes are merged when insertion occurs. With smaller values, less RAM is used while indexing, and searches on unoptimized indexes are faster, but indexing speed is slower. With larger values, more RAM is used during indexing, and while searches on unoptimized indexes are slower, indexing is faster. Thus larger values (> 10) are best for batch index creation, and smaller values (< 10) for indexes that are interactively maintained. The value must not be lower than 2.	10
`default.<indexname>.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 though they are less frequent. See also **`org.apache.lucene.index.LogDocMergePolicy.minMergeSize`**.	0 MB (actually ~1K)
`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
`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.maxMergeSizeForOptimize`**.	Unlimited

Property	Description	Default Value
`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`**
`default.<indexname>.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 is best to flush by RAM usage instead of document count and use as large a RAM buffer as you can.	16 MB
`default.<indexname>.indexwriter.term_index_interval`	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
`default.<indexname>.indexwriter.use_compound_file`	The advantage of using the compound file format is that less file descriptors are used. The disadvantage is that indexing takes more time and temporary disk space. You can set this parameter to **`false`** in an attempt to improve the indexing time, but you could run out of file descriptors if **`mergeFactor`** is also large. Boolean parameter, use **`true`** or **`false`**. The default value for this option is **`true`**.	true

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

7.2.5.4. Tuning the Indexing Speed

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

When tuning indexing speed 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. Set the **blackhole** as worker back end and start your indexing routines. This back end does not disable Hibernate Search. It generates the required change sets to the index, but discards them instead of flushing them to the index. In contrast to setting the **hibernate.search.indexing_strategy** to **manual**, using **blackhole** will possibly load more data from the database because associated entities are re-indexed as well.

```
hibernate.search.[default|<indexname>].worker.backend blackhole
```

WARNING

The **blackhole** back end is not to be used in production, only as a diagnostic tool to identify indexing bottlenecks.

7.2.5.5. Control Segment Size

The following options configure the maximum size of segments created:

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

Control Segment Size

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

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

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

7.2.6. LockFactory Configuration

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

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

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

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

Table 7.7. List of Available LockFactory Implementations

Name	Class	Description
LockFactory Configuration **`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.

Name	Class	Description
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 does not use a file marker but is a Java object lock held in memory; therefore it is possible to use it only when you are sure the index is not going to be shared by any other process. This is the default implementation for the **ram** directory provider.
none	org.apache.lucene.store.NoLockFactory	Changes to this index are not coordinated by a lock.

The following is an example of locking strategy configuration:

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

7.2.7. Index Format Compatibility

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

WARNING

Back up indexes before attempting to update the index format.

Hibernate Search exposes the **hibernate.search.lucene_version** configuration property.

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

Force Analyzers to Be Compatible with a Lucene 3.0 Created Index

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

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

7.3. HIBERNATE SEARCH FOR YOUR APPLICATION

7.3.1. First Steps with Hibernate Search

To get started with Hibernate Search for your application, follow these topics.

- Enable Hibernate Search Using Maven
- Indexing
- Searching
- Analyzer

7.3.2. Enable Hibernate Search Using Maven

Use the following configuration in your Maven project to add **`hibernate-search-orm`** dependencies:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-search-orm</artifactId>
      <version>5.5.1.Final-redhat-1</version>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-search-orm</artifactId>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

7.3.3. Add Annotations

For this section, consider the example in which you have a database containing details of books. Your application contains the Hibernate managed classes **example.Book** and **example.Author** and you want to add free text search capabilities to your application to enable searching for books.

Example: Entities Book and Author Before Adding Hibernate Search Specific Annotations

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

  @Id
  @GeneratedValue
  private Integer id;

  private String title;

  private String subtitle;

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

  private Date publicationDate;

  public Book() {}

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

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

  @Id
  @GeneratedValue
  private Integer id;

  private String name;

  public Author() {}

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

To achieve this you have to add a few annotations to the Book and Author class. The first annotation **@Indexed** marks Book as indexable. By design Hibernate Search stores an untokenized ID in the index to ensure index unicity for a given entity. **@DocumentId** marks the property to use for this purpose and is in most cases the same as the database primary key. The **@DocumentId** annotation is optional in the case where an**@Id** annotation exists.

Next the fields you want to make searchable must be marked as such. In this example, start with **`title`** and **`subtitle`** and annotate both with **`@Field`**. The parameter **`index=Index.YES`** will ensure that the text will be indexed, while **`analyze=Analyze.YES`** ensures that the text will be analyzed using the default Lucene analyzer. Usually, analyzing means chunking a sentence into individual words and potentially excluding common words like **`'a'`** or ‘the’. We will talk more about analyzers a little later on. The third parameter we specify within **`@Field`**, **`store=Store.NO`**, ensures that the actual data will not be stored in the index. Whether this data is stored in the index or not has nothing to do with the ability to search for it. From Lucene’s perspective it is not necessary to keep the data once the index is created. The benefit of storing it is the ability to retrieve it via projections.

Without projections, Hibernate Search will per default execute a Lucene query in order to find the database identifiers of the entities matching the query criteria and use these identifiers to retrieve managed objects from the database. The decision for or against projection has to be made on a case to case basis. The default behavior is recommended since it returns managed objects whereas projections only return object arrays. Note that **`index=Index.YES`**, **`analyze=Analyze.YES`** and **`store=Store.NO`** are the default values for these parameters and could be omitted.

Another annotation not yet discussed is **`@DateBridge`**. This annotation is one of the built-in field bridges in Hibernate Search. The Lucene index is purely string based. For this reason Hibernate Search must convert the data types of the indexed fields to strings and vice-versa. A range of predefined bridges are provided, including the DateBridge which will convert a java.util.Date into a String with the specified resolution. For more details see Bridges.

This leaves us with **`@IndexedEmbedded`**. This annotation is used to index associated entities (**`@ManyToMany`**, **`@*ToOne`**, **`@Embedded`** and **`@ElementCollection`**) as part of the owning entity. This is needed since a Lucene index document is a flat data structure which does not know anything about object relations. To ensure that the authors' name will be searchable you have to ensure 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 Embedded and Associated Objects.

These settings should be sufficient for now. For more details on entity mapping see Mapping an Entity.

Example: Entities After Adding Hibernate Search Annotations

```
package example;
...
@Entity

public class Book {

  @Id
  @GeneratedValue
  private Integer id;

  private String title;

  private String subtitle;

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

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

  public Book() {
  }

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

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

  @Id
  @GeneratedValue
  private Integer id;

  private String name;

  public Author() {
  }

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

7.3.4. Indexing

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

Example: Using the Hibernate Session to Index Data

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

Example: Using JPA to Index Data

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

After executing the above code, you should be able to see a Lucene index under **`/var/lucene/indexes/example.Book`**. Inspect this index withLuke to help you to understand how Hibernate Search works.

7.3.5. Searching

To execute a search, create a Lucene query using either the Lucene API or the Hibernate Search query DSL. Wrap the query in a org.hibernate.Query to get the required functionality from the Hibernate API. The following code prepares a query against the indexed fields. Executing the code returns a list of Books.

Example: Using a Hibernate Search Session to Create and Execute a Search

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

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

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

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

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

Example: Using JPA to Create and Execute a Search

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

// create native Lucene query 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", "publicationDate")
  .matching("Java rocks!")
  .createQuery();
```

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

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

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

7.3.6. Analyzer

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

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

Specify the fully qualified class name or the analyzer to use, or see an analyzer defined by the **`@AnalyzerDef`** annotation with the**`@Analyzer`** annotation. The Solr analyzer framework with its factories are utilized for the latter option. For more information about factory classes, see the Solr JavaDoc or read the corresponding section on the Solr Wiki.

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

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

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

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

  @Field
  @Analyzer(definition = "customanalyzer")
```

```
    private String title;

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

    @IndexedEmbedded
    private Set authors = new HashSet();

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

    public Book() {
    }

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

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

7.4. MAPPING ENTITIES TO THE INDEX STRUCTURE

7.4.1. Mapping an Entity

All the metadata information required to index entities is described through annotations, so 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.

7.4.1.1. Basic Mapping

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

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

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

7.4.1.2. @Indexed

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

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

You can optionally specify the **index** attribute of the @Indexed annotation to change the default name of the index.

7.4.1.3. @Field

For each property (or attribute) of your entity, you have the ability to describe how it will be indexed. The default (no annotation present) means that the property is ignored by the indexing process.

NOTE

Prior to Hibernate Search 5, numeric field encoding was only chosen if explicitly requested via **@NumericField**. As of Hibernate 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**.

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

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

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

- **index**: describe whether the property is indexed or not. The different values are **Index.NO**, meaning that it is not indexed and cannot be found by a query and **Index.YES**, meaning that the element gets indexed and is searchable. The default value is **Index.YES**. **Index.NO** can be useful for cases where a property is not required to be searchable, but should be available for projection.

  NOTE

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

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

NOTE

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

NOTE

Fields used for sorting *must not* be analyzed.

- **`norms`**: describes whether index time boosting information should be stored (**`Norms.YES`**) or not (**`Norms.NO`**). Not storing it can save a considerable amount of memory, but there will not be any index time boosting information available. The default value is **`Norms.YES`**.

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

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

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

NOTE

When the **indexNullAs** parameter is used it is important to use the same token in the search query to search for **null** values. It is also advisable to use this feature only with un-analyzed fields (**Analyze.NO**).

WARNING

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

7.4.1.4. @NumericField

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

Value	Definition
forField	(Optional) Specify the name of the related @Field that will be indexed as numeric. It is 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 in normal @Fields. Default value is 4.

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

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

Example: Defining a Custom NumericFieldBridge

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

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

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

}
```

7.4.1.5. @Id

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

Infinispan Query uses the entity's **id** property to ensure the index is uniquely identified. By design, an ID is stored and must not be converted into a token. To mark a property as index ID, use the **@DocumentId** annotation.

Example: 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; }
}
```

The example above defines an index with four fields: **id** , **Abstract**, **text** and **grade** . Note that by default the field name is not capitalized, following the JavaBean specification. The **grade** field is annotated as numeric with a slightly larger precision step than the default.

7.4.1.6. Mapping Properties Multiple Times

Sometimes you need 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. To search by words on this property and still sort it, it needs to be indexed - once analyzed and once un-analyzed. @Fields allows you to achieve this goal.

Example: 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 this example the field **summary** is indexed twice, once as **summary** in a tokenized way, and once as **summary_forSort** in an untokenized way.

7.4.1.7. Embedded and Associated Objects

Associated objects as well as embedded objects can be indexed as part of the root entity index. This is useful if you expect to search a given entity based on properties of associated objects. The aim is to return places where the associated city is Atlanta (In the Lucene query parser language, it would translate into **address.city:Atlanta**). The place fields will be indexed in the **Place** index. The **Place** index documents will also contain the fields **address.id**, **address.street**, and **address.city** which you will be able to query.

Example: Indexing Associations

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

    @Field
    private String name;

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

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

    @Field
    private String street;

    @Field
    private String city;

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

Because the data is denormalized in the Lucene index when using the **`@IndexedEmbedded`** technique, Hibernate Search must be aware of any change in the Place object and any change in the Address object to keep the index up to date. To ensure the Lucene document is updated when it is Address changes, mark the other side of the bidirectional relationship with **`@ContainedIn`**.

NOTE

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

To expand upon this, the following example demonstrates nesting **`@IndexedEmbedded`**.

Example: Nested Usage of @IndexedEmbedded and @ContainedIn

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

    @Field
    private String name;

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

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

    @Field
    private String street;
    @Field
    private String city;

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

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

@Embeddable
public class Owner {
    @Field
    private String name;
   ...
}
```

Any **@*ToMany**, **@*ToOne** and **@Embedded** attribute can be annotated with**@IndexedEmbedded**. The attributes of the associated class will then be added to the main entity index. 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 an 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 will be ignored.

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

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

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

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

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

This behavior 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 does not exist. It might help to keep the relational model normalized while benefiting from the full text index speed and feature richness.

NOTE

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

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

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

Example: Using the targetElement Property of @IndexedEmbedded

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

    @Field
    private String street;

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

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

7.4.1.8. Limiting Object Embedding to Specific Paths

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

When using only depth all indexed fields of the embedded type will be added recursively at the same depth. This makes it harder to select 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 the example below:

Example: 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 the example above, you would be able to search on a Person by **`name`** and/or **`surname`**, and/or the **`name`** of the parent. It will not index the**`surname`** of the parent, so searching on parent's surnames will not be possible but speeds up indexing, saves space and improve overall performance.

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

Example: Using the includePaths Property of @IndexedEmbedded

```
@Entity
```

```
@Indexed
public class Human {

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

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

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

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

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

    ...//other fields omitted
```

In the example above, every human will have its name and surname attributes indexed. The name and surname of parents will also be indexed, 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

Having explicit control of the indexed paths might be easier if you are 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.

7.4.2. 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.

7.4.2.1. 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: Different Ways of Using @Boost

```
@Entity
@Indexed

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 the example above, 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.

7.4.2.2. Dynamic Index Time Boosting

The **`@Boost`** annotation used inStatic Index Time Boostingdefines a static boost factor which is independent of the state 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: Dynamic Boost

```
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 the example above, 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 is 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.

7.4.3. Analysis

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

7.4.3.1. 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: Different Ways of Using @Analyzer

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

    @Field
    private String name;

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

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

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

WARNING

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

7.4.3.2. 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 lets you build your customized analyzer in a very flexible way (like Lego). Generally speaking the char filters do some pre-processing in the character input, then the Tokenizer starts the tokenizing process by turning the character input into tokens which are then further processed by the TokenFilters. Hibernate Search supports this infrastructure by utilizing the Solr analyzer framework.

Let us review a concrete example stated below. 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: @AnalyzerDef and the Solr Framework

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

NOTE

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

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

Example: Use a Specific Charset to Load the Property File

```
@AnalyzerDef(name="customanalyzer",
  charFilters = {
    @CharFilterDef(factory = MappingCharFilterFactory.class, params = {
      @Parameter(name = "mapping",
        value = "org/hibernate/search/test/analyzer/solr/mapping-
chars.properties")
    })
  },
  tokenizer = @TokenizerDef(factory = StandardTokenizerFactory.class),
  filters = {
    @TokenFilterDef(factory = ISOLatin1AccentFilterFactory.class),
    @TokenFilterDef(factory = LowerCaseFilterFactory.class),
    @TokenFilterDef(factory = StopFilterFactory.class, params = {
      @Parameter(name="words",
        value=
"org/hibernate/search/test/analyzer/solr/stoplist.properties" ),
      @Parameter(name="resource_charset", value = "UTF-16BE"),
      @Parameter(name="ignoreCase", value="true")
  })
})
public class Team {
    ...
}
```

Once defined, an analyzer definition can be reused by an @Analyzer declaration as seen in the following example.

Example: Referencing an Analyzer by Name

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

    @Field
    private String name;

    @Field
    private String location;

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

Analyzer instances declared by @AnalyzerDef are also available by their name in the SearchFactory which is quite useful when building queries.

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

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

7.4.3.3. Available Analyzers

Solr and Lucene come with many 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 us check a few of them.

Table 7.8. Available Char Filters

Factory	Description	Parameters
MappingCharFilterFactory	Replaces one or more characters with one or more characters, based on mappings specified in the resource file	**`mapping`**: points to a resource file containing the mappings using the format: "á" ⇒ "a"; "ñ" ⇒ "n"; "ø" ⇒ "o"
HTMLStripCharFilterFactory	Remove HTML standard tags, keeping the text	none

Table 7.9. Available Tokenizers

Factory	Description	Parameters
StandardTokenizerFactory	Use the Lucene StandardTokenizer	none
HTMLStripCharFilterFactory	Remove HTML tags, keep the text and pass it to a StandardTokenizer.	none
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

Table 7.10. Available Filters

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

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

7.4.3.4. Dynamic Analyzer Selection

So far all the introduced ways to specify an analyzer were static. However, there are use cases where it is useful to select an analyzer depending on the current state of the entity to be indexed, for example in a multilingual applications. For an BlogEntry class for example the analyzer could depend on the language property of the entry. Depending on this property the correct language specific stemmer should be chosen to index the actual text.

To enable this dynamic analyzer selection Hibernate Search introduces the AnalyzerDiscriminator annotation. Following example demonstrates the usage of this annotation.

Example: 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 BlogEntry ) ) {
            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. The example above assumes that the language parameter is either 'de' or 'en' which matches the specified names in the **`@AnalyzerDefs`**.

7.4.3.5. Retrieving an Analyzer

Retrieving an analyzer can be used when multiple analyzers have been used in a domain model, in order to benefit from stemming or phonetic approximation, etc. In this case, use the same analyzers to building a query. Alternatively, use the Hibernate Search query DSL, which selects the correct analyzer automatically. See

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

NOTE

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

Example: Using the Scoped Analyzer When Building a Full-text Query

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

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

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

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

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

NOTE

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

7.4.4. Bridges

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

7.4.4.1. 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 for more information.

java.lang.String

Strings are indexed as are short, Short, integer, Integer, long, Long, float, Float, double,

Double, BigInteger, BigDecimal

Numbers are converted into their string representation. Note that numbers cannot be compared by Lucene (that is, used in ranged queries) out of the box: they have to be padded.

NOTE

Using a Range query has drawbacks, an alternative approach is to use a Filter query which will filter the result query to the appropriate range. Hibernate Search also supports the use of a custom StringBridge as described in Custom Bridges.

java.util.Date

Dates are stored as yyyyMMddHHmmssSSS in GMT time (200611072203012 for Nov 7th of 2006 4:03PM and 12ms EST). You should not really bother with the internal format. What is important is that when using a TermRangeQuery, you should know that the dates have to be expressed in GMT time.
Usually, storing the date up to the millisecond is not necessary. **`@DateBridge`** defines the appropriate resolution you are willing to store in the index (**`@DateBridge(resolution=Resolution.DAY)`**). The date pattern will then be truncated accordingly.

```
@Entity
@Indexed
public class Meeting {
    @Field(analyze=Analyze.NO)

    private Date date;
    ...
```

WARNING

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

IMPORTANT

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

java.net.URI, java.net.URL

URI and URL are converted to their string representation.

java.lang.Class

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

7.4.4.2. Custom Bridges

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

7.4.4.2.1. StringBridge

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

Example: Custom StringBridge Implementation

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

    private int PADDING = 5;

    public String objectToString(Object object) {
        String rawInteger = ( (Integer) object ).toString();
        if (rawInteger.length() > PADDING)
            throw new IllegalArgumentException( "Try to pad on a number
too big" );
```

```
        StringBuilder paddedInteger = new StringBuilder( );
        for ( int padIndex = rawInteger.length() ; padIndex < PADDING ;
padIndex++ ) {
            paddedInteger.append('0');
        }
        return paddedInteger.append( rawInteger ).toString();
    }
}
```

Given the string bridge defined in the previous example, any property or field can use this bridge thanks to the **`@FieldBridge`** annotation:

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

7.4.4.2.2. Parameterized Bridge

Parameters can also be passed to the bridge implementation making it more flexible. Following example implements a ParameterizedBridge interface and parameters are passed through the **`@FieldBridge`** annotation.

Example: Passing Parameters to Your Bridge Implementation

```
public class PaddedIntegerBridge implements StringBridge,
ParameterizedBridge {

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

    public void setParameterValues(Map<String,String> parameters) {
        String padding = parameters.get( PADDING_PROPERTY );
        if (padding != null) this.padding = Integer.parseInt( padding );
    }

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


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

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

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

7.4.4.2.3. Type Aware Bridge

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

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

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

7.4.4.2.4. Two-Way Bridge

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

Example: Implementing a TwoWayStringBridge Usable for id Properties

```
public class PaddedIntegerBridge implements TwoWayStringBridge,
ParameterizedBridge {

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

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

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

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

```
}

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

IMPORTANT

It is important for the two-way process to be idempotent (i.e., object = stringToObject(objectToString(object))).

7.4.4.2.5. FieldBridge

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

Example: Implementing the FieldBridge Interface

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

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

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

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

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

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

NOTE

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

7.4.4.2.6. ClassBridge

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

Example: Implementing a Class Bridge

```
@Entity
@Indexed
(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.

7.5. QUERYING

Hibernate Search can execute Lucene queries and retrieve domain objects managed by an InfinispanHibernate session. The search provides the power of Lucene without leaving the Hibernate paradigm, giving another dimension to the Hibernate classic search mechanisms (HQL, Criteria query, native SQL query).

Preparing and executing a query consists of following four steps:

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

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

Example: Creating a FullTextSession

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

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

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

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

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

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

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

Example: Creating a Lucene Query Using the QueryParser

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

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

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

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

Example: Creating a Search Query Using the JPA API

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

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

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

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

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

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

NOTE

In these examples, the Hibernate API has been used. The same examples can also be written with the Java Persistence API by adjusting the way the **`FullTextQuery`** is retrieved.

7.5.1. Building Queries

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

7.5.1.1. Building a Lucene Query Using the Lucene API

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

7.5.1.2. Building a Lucene Query

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

The Hibernate Search query API is fluent, with the following key characteristics:

- Method names are in English. As a result, API operations can be read and understood as a series of English phrases and instructions.

- It uses IDE autocompletion which helps possible completions for the current input prefix and allows the user to choose the right option.

- It often uses the chaining method pattern.

- It is easy to use and read the API operations.

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

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

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

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

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

7.5.1.3. Keyword Queries

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

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

Table 7.11. Keyword Query Parameters

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

- The value "storm" is passed through the **history** FieldBridge. This is useful when numbers or dates are involved.
- The field bridge value is then passed to the analyzer used to index the field **history**. This ensures that the query uses the same term transformation than the indexing (lower case, ngram, stemming and so on). If the analyzing process generates several terms for a given word, a boolean query is used with the **SHOULD** logic (roughly an **OR** logic).

To search a property that is not of type string.

```
@Indexed
public class Myth {
  @Field(analyze = Analyze.NO)
  @DateBridge(resolution = Resolution.YEAR)
  public Date getCreationDate() { return creationDate; }
  public Date setCreationDate(Date creationDate) { this.creationDate =
creationDate; }
  private Date creationDate;

  ...
}

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

NOTE

In plain Lucene, the Date object had to be converted to its string representation, which in this case is the year.

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

The next example searches a field that uses ngram analyzers. The ngram analyzers index succession of ngrams of words, which helps to avoid user typos. For example, the 3-grams of the word hibernate are hib, ibe, ber, ern, rna, nat, ate.

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

public class Myth {
  @Field(analyzer=@Analyzer(definition="ngram")
  public String getName() { return name; }
  public String setName(String name) { this.name = name; }
  private String name;

  ...
}

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

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

NOTE

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

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

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

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

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

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

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

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

7.5.1.4. Fuzzy Queries

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

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

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

7.5.1.5. Wildcard Queries

Wildcard queries are useful in circumstances where only part of the word is known. The **?** represents a single character and * represents multiple characters. 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. The risk of ***** or **?** being mangled is too high.

7.5.1.6. Phrase Queries

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

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

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

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

7.5.1.7. Range Queries

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

```
//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();
```

7.5.1.8. Combining Queries

Queries can be combined to create more complex queries. The following aggregation operators are available:

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

The subqueries can be any Lucene query including a boolean query itself.

Example: SHOULD Query

```
//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();
```

Example: MUST Query

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

Example: MUST NOT Query

```
//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();
```

7.5.1.9. Query Options

The Hibernate Search query DSL is an easy-to-use and easy-to-read query API. In accepting and producing Lucene queries, you can incorporate query types not yet supported by the DSL.

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

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

Example: Combination of Query 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();
```

7.5.1.10. Build a Hibernate Search Query

7.5.1.10.1. Generality

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

Example: Wrapping a Lucene Query in a Hibernate Query

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

For improved performance, restrict the returned types as follows:

Example: Filtering the Search Result by Entity Type

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

// or

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

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

7.5.1.10.2. Pagination

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

Example: Defining Pagination for a Search Query

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

NOTE

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

7.5.1.10.3. Sorting

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

Example: Specifying a Lucene Sort

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

List results = query.list();
```

NOTE

Fields used for sorting must not be tokenized. For more information about tokenizing, see @Field.

7.5.1.10.4. Fetching Strategy

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

Example: Specifying FetchMode on a Query

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

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

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

IMPORTANT

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

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

7.5.1.10.5. Projection

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

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

- The properties projected must be stored in the index (**`@Field(store=Store.YES)`**), which increases the index size.
- The properties projected must use a **`FieldBridge`** implementing org.hibernate.search.bridge.TwoWayFieldBridge or **`org.hibernate.search.bridge.TwoWayStringBridge`**, the latter being the simpler version.

NOTE

All Hibernate Search built-in types are two-way.

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

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

Example: Using Projection to Retrieve Metadata

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

Fields can be mixed with the following projection constants:

- **FullTextQuery.THIS:** returns the initialized and managed entity (as a non projected query would have done).
- **FullTextQuery.DOCUMENT:** returns the Lucene Document related to the object projected.
- **FullTextQuery.OBJECT_CLASS:** returns the class of the indexed entity.
- **FullTextQuery.SCORE:** returns the document score in the query. Scores are handy to compare one result against an other for a given query but are useless when comparing the result of different queries.
- **FullTextQuery.ID:** the ID property value of the projected object.
- **FullTextQuery.DOCUMENT_ID:** the Lucene document ID. Be careful in using this value as a Lucene document ID can change over time between two different IndexReader opening.
- **FullTextQuery.EXPLANATION:** returns the Lucene Explanation object for the matching object/document in the given query. This is not suitable for retrieving large amounts of data. Running explanation typically is as costly as running the whole Lucene query per matching element. As a result, projection is recommended.

7.5.1.10.6. Customizing Object Initialization Strategies

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

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

Example: Check the Second-level Cache Before Using a Query

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

`ObjectLookupMethod` defines the strategy to check if an object is easily accessible (without fetching it from the database). Other options are:

- **`ObjectLookupMethod.PERSISTENCE_CONTEXT`** is used if many matching entities are already loaded into the persistence context (loaded in the Session or EntityManager).

- **`ObjectLookupMethod.SECOND_LEVEL_CACHE`** checks the persistence context and then the second-level cache.

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

- Correctly configure and activate the second-level cache.

- Enable the second-level cache for the relevant entity. This is done using annotations such as @Cacheable.

- Enable second-level cache read access for either Session, EntityManager or Query. Use **`CacheMode.NORMAL`** in Hibernate native APIs or**`CacheRetrieveMode.USE`** in Java Persistence APIs.

WARNING

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

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

- **QUERY** (default) uses a set of queries to load several objects in each batch. This approach is recommended.

- **FIND_BY_ID** loads one object at a time using the**`Session.get`** or **`EntityManager.find`** semantic. This is recommended if the batch size is set for the entity, which allows Hibernate Core to load entities in batches.

7.5.1.10.7. Limiting the Time of a Query

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

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

7.5.1.10.8. Raise an Exception on Time Limit

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

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

Example: Defining a Timeout in Query Execution

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

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

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

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

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

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

Example: Defining a Timeout in Query Execution

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

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

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

IMPORTANT

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

7.5.2. Retrieving the Results

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

7.5.2.1. 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. Do not forget to close the ScrollableResults object when you are 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.

7.5.2.2. Result Size

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

- to provide a total search results feature, as provided by Google searches. For example, "1-10 of about 888,000,000 results"
- to implement a fast pagination navigation
- to implement a multi-step search engine that adds approximation if the restricted query returns zero or not enough results

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

Example: 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 == ;
```

```
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 == ;
```

NOTE

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

7.5.2.3. ResultTransformer

Projection results are returned as Object arrays. If the data structure used for the object does not match the requirements of the application, apply a ResultTransformer. The ResultTransformer builds the required data structure after the query execution.

Example: Using ResultTransformer with Projections

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

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

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

7.5.2.4. Understanding Results

If the results of a query are not what you expected, the **Luke** tool is useful in understanding the outcome. 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 is unrelated to the entity ID. Be careful not to confuse these concepts.

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

Example: Retrieving the Lucene Explanation Object Using Projection

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

Use the Explanation object only when required as it is roughly as expensive as running the Lucene query again.

7.5.2.5. Filters

Apache Lucene has a powerful feature that allows you 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. Use cases include:

- security
- temporal data (example, view only last month’s data)
- population filter (example, search limited to a given category)

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: Enabling Fulltext Filters for a Query

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

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

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

Example: Defining and Implementing a Filter

```
@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: Creating a Filter Using the Factory Pattern

```
@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.

Infinispan Query uses a @Factory annotated method to build the filter instance. The factory must have a no argument 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: Passing Parameters to a Defined Filter

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

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

Example: Using Parameters in the Actual Filter Implementation

```
public class SecurityFilterFactory {
    private Integer level;

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

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

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

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

@Key methods are needed only if:

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

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

As mentioned before the defined filters are per default cached and the cache uses a combination of hard and soft references to allow disposal of memory when needed. The hard reference cache keeps track of the most recently used filters and transforms the ones least used to SoftReferences when needed. Once the limit of the hard reference cache is reached additional filters are cached as SoftReferences. To adjust the size of the hard reference cache, use **`hibernate.search.filter.cache_strategy.size`** (defaults to 128). For advanced use of filter caching, implement your own FilterCachingStrategy. The classname is defined by **`hibernate.search.filter.cache_strategy`**.

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

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

Value	Definition
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.

Filters should be cached in the following situations:

- 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)

7.5.2.6. Using Filters in a Sharded Environment

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

Query a Subset of Index Shards

1. Create a sharding strategy that does select a subset of IndexManagers depending on a filter configuration.
2. Activate the filter at query time.

Example: Query a Subset of Index Shards

In this example the query is run against a specific customer shard if the **`customer`** filter is activated.

```
public class CustomerShardingStrategy implements IndexShardingStrategy {

     // stored IndexManagers in an 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, only the shard dedicated to this customer is queried, otherwise, all shards are returned. A given sharding strategy can react to one or more filters and depends on their parameters.

The second step is to activate the filter at query time. While the filter can be a regular filter (as defined in) 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 is otherwise ignored).

To use this feature, specify the ShardSensitiveOnlyFilter class when declaring your filter.

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

FullTextQuery query = ftEm.createFullTextQuery(luceneQuery,
Customer.class);
query.enableFulltextFilter("customer").setParameter("CustomerID", 5);
@SuppressWarnings("unchecked")
List<Customer> results = query.getResultList();
```

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

7.5.3. Faceting

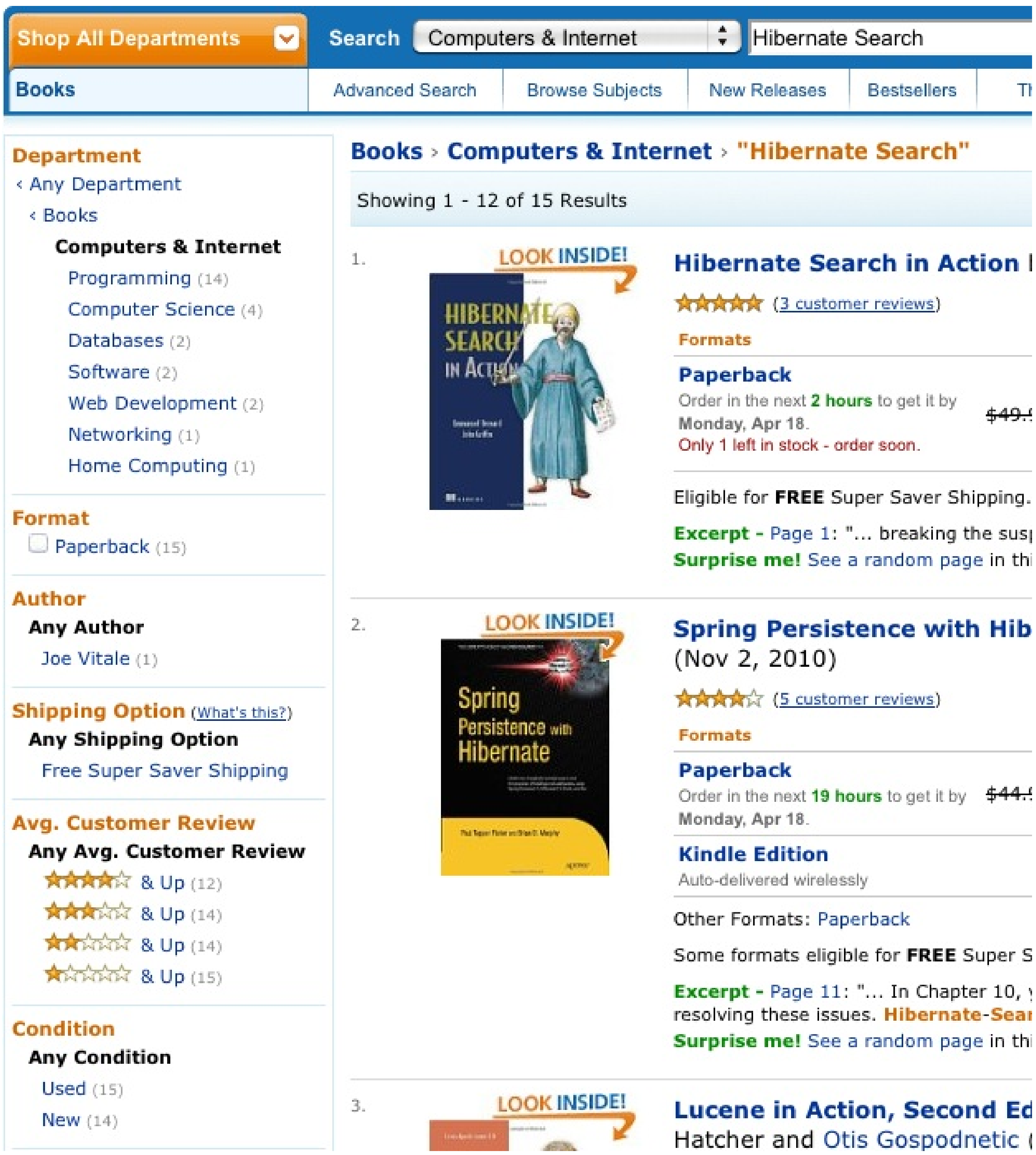
Faceted search is a technique which allows the results of a query to be divided into multiple categories. This categorization includes the calculation of hit counts for each category and the ability to further restrict search results based on these facets (categories). The example below shows a faceting example. The search results in fifteen hits which are displayed on the main part of the page. The navigation bar on the left, however, shows the category *Computers & Internet* with its subcategories*Programming*, *Computer Science*, *Databases*, *Software*, *Web Development*, *Networking* and *Home Computing*. For each of these subcategories the number of books is shown matching the main search criteria and belonging to the respective subcategory. This division of the category *Computers & Internet* is one concrete search facet. Another one is for example the average customer review.

Faceted search divides the results of a query into categories. The categorization includes the calculation of hit counts for each category and the further restricts search results based on these facets (categories). The following example displays a faceting search results in fifteen hits displayed on the main page.

The left side navigation bar displays the categories and subcategories. For each of these subcategories the number of books matches the main search criteria and belongs to the respective subcategory. This division of the category Computers & Internet is one concrete search facet. Another example is the average customer review.

Example: Search for Hibernate Search on Amazon

In Hibernate Search, the classes QueryBuilder and FullTextQuery are the entry point into the faceting API. The former creates faceting requests and the latter accesses the FacetManager. The FacetManager applies faceting requests on a query and selects facets that are added to an existing query to refine search results. The examples use the entity Cd as shown in the example below:



Shop All Departments | Search | Computers & Internet | Hibernate Search

Books | Advanced Search | Browse Subjects | New Releases | Bestsellers | Th

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

Format
Paperback (15)

Author
Any Author
Joe Vitale (1)

Shipping Option (What's this?)
Any Shipping Option
Free Super Saver Shipping

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

Condition
Any Condition
Used (15)
New (14)

Books › Computers & Internet › "Hibernate Search"

Showing 1 - 12 of 15 Results

1. LOOK INSIDE!

Hibernate Search in Action
★★★★★ (3 customer reviews)
Formats
Paperback
Order in the next **2 hours** to get it by **Monday, Apr 18**.
Only 1 left in stock - order soon.
~~$49.~~
Eligible for **FREE** Super Saver Shipping.
Excerpt - Page 1: "... breaking the sus
Surprise me! See a random page in thi

2. LOOK INSIDE!

Spring Persistence with Hib
(Nov 2, 2010)
★★★★☆ (5 customer reviews)
Formats
Paperback
Order in the next **19 hours** to get it by **Monday, Apr 18**.
~~$44.~~
Kindle Edition
Auto-delivered wirelessly
Other Formats: Paperback
Some formats eligible for **FREE** Super S
Excerpt - Page 11: "... In Chapter 10,
resolving these issues. **Hibernate-Sea**
Surprise me! See a random page in thi

3. LOOK INSIDE!

Lucene in Action, Second Ed
Hatcher and Otis Gospodnetic

Example: Entity Cd

```
@Indexed
public class Cd {

    private int id;

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

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

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

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


// setter/getter
...
```

NOTE

Prior to Hibernate Search 5.2, there was no need to explicitly use a @Facet annotation. In Hibernate Search 5.2 it became necessary in order to use Lucene's native faceting API.

7.5.3.1. Creating a Faceting Request

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

Example: Creating a Discrete Faceting Request

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

When executing this faceting request a Facet instance will be created for each discrete value for the indexed field **`label`**. The Facet instance will record the actual field value including how often this particular field value occurs within the original query results. orderedBy, includeZeroCounts and maxFacetCount are optional parameters which can be applied on any faceting request. 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 or the order in which ranges were specified. includeZeroCount determines whether facets with a count of 0 will be included in the result (by default they are) and maxFacetCount allows to limit the maximum amount of facets returned.

NOTE

At the moment there are several preconditions an indexed field has to meet in order to apply faceting on it. The indexed property must be of type String, Date or a subtype of Number and **`null`** values should be avoided. Furthermore the property has to be indexed with **`Analyze.NO`** and in case of a numeric property @NumericField needs to be specified.

The creation of a range faceting request is quite similar except that we have to specify ranges for the field values we are faceting on. A range faceting request can be seen below where three different price ranges are specified. The **`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: Creating a Range Faceting Request

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

7.5.3.2. Applying a Faceting Request

A faceting request is applied to a query via the FacetManager class which can be retrieved via the FullTextQuery class.

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

A faceting request can be applied on a query using the FacetManager, which can be retrieved via the FullTextQuery.

Example: 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" );
...
```

Multiple faceting requests can be retrieved using **`getFacets()`** and specifying the faceting request name.

The **`disableFaceting()`** method disables a faceting request by specifying its name.

7.5.3.3. Restricting Query Results

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

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

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

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

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

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

// re-execute the query
cds = fullTextQuery.list();
assertTrue(cds.size() == 2);
```

7.5.4. Optimizing the Query Process

Query performance depends on several criteria:

- The Lucene query.
- The number of objects loaded: use pagination (always) or index projection (if needed).
- The way Hibernate Search interacts with the Lucene readers: defines the appropriate reader strategy.

- Caching frequently extracted values from the index. See Caching Index Values: FieldCache for more information.

7.5.4.1. Caching Index Values: FieldCache

The primary function of a Lucene index is to identify matches to your queries. After the query is performed the results must be analyzed to extract useful information. Hibernate Search would typically 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.

The requirements depend on the kind of Projections being used, as 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 with different kinds of caching of the main metadata fields required by Hibernate Search:

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

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

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

- **`CLASS`**: Hibernate Search will use a Lucene FieldCache to improve performance of the class type extraction from the index.
  This value is enabled by default, and is what Hibernate Search will apply if you do not specify the @CacheFromIndex annotation.

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

NOTE

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

Using a FieldCache has two downsides to consider:

- Memory usage: these caches can be quite memory hungry. Typically the CLASS cache has lower requirements than the ID cache.

- Index warmup: when using field caches, the first query on a new index or segment will be slower than when you do not have caching enabled.

With some queries, the class type will not 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 builting bridges), and all types being loaded in a specific query must use the fieldname for the id, and have IDs of the same type (this is evaluated at each query execution).

7.6. 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. In these cases Hibernate Search exposes the Manual Index APIs to explicitly update or remove a single entity from the index, 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.

7.6.1. 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.

Directly add an object or instance to the index using **`FullTextSession.index(T entity)`**. The index is updated when the entity is indexed. Infinispan Query applies changes to the index during the transaction commit.

Example: Indexing an Entity Using 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 for more details.

Use a MassIndexer to add all instances for a type (or for all indexed types). See Using a MassIndexer for more information.

7.6.2. Deleting Instances from the Index

It is 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`**.

The purging operation permits the removal of a single entity or all entities of a given type from a Lucene index without physically removing them from the database. This operation is performed using the FullTextSession.

Example: Purging a Specific Instance of an Entity from the Index

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

```
Transaction tx = fullTextSession.beginTransaction();
for (Customer customer : customers) {
fullTextSession.purgeAll( Customer.class );
//optionally optimize the index
//fullTextSession.getSearchFactory().optimize( Customer.class );
tx.commit(); //index is updated at commit time
```

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

NOTE

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

NOTE

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

7.6.3. Rebuilding the Index

If you change the entity mapping to the index, chances are that the whole Index needs to be updated; For example if you decide to index 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:

Changing the entity mapping in the indexer may require the entire index to be updated. For example, if an existing field is to be indexed using a different analyzer, the index will need to be rebuilt for affected types.

Additionally, if the database is replaced by restoring from a backup or being imported from a legacy system, the index will need to be rebuilt from existing data. Infinispan Query provides two main strategies:

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

7.6.3.1. Using flushToIndexes()

This strategy consists of 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 do not empty the queue periodically; to do this 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: Index Rebuilding Using index() and flushToIndexes()

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

NOTE

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

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

7.6.3.2. 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. Querying the index is not recommended when a MassIndexer is busy.

Example: Rebuild the Index Using a MassIndexer

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

This will rebuild the index, deleting it and then reloading all entities from the database. Although it is simple to use, some tweaking is recommended to speed up the process.

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: Using a Tuned MassIndexer

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

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

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

NOTE

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

NOTE

The MassIndexer is unaware of transactions, therefore there is no need to begin one or commit afterward. Because it is not transactional it is not recommended to let users use the system during its processing, as it is unlikely people will be able to find results and the system load might be too high anyway.

Other parameters that 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 is possible to obtain a similar effect by using a **`LimitTokenCountAnalyzer`**.

All **`.indexwriter`** parameters are Lucene specific and Hibernate Search passes these parameters through.

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

7.7. INDEX OPTIMIZATION

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

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

Optimizing the Lucene index speeds up searches, but has no effect on the index update performance. Searches can be performed during an optimization process, however they will be slower than expected. All index updates are on hold during the optimization. It is therefore recommended to schedule optimization:

- On an idle system or when searches are least frequent.
- After a large number of index modifications are applied.

MassIndexer optimizes indexes by default at the start and at the end of processing. Use **`MassIndexer.optimizeAfterPurge`** and **`MassIndexer.optimizeOnFinish`** to change this default behavior. See Using a MassIndexer for more information.

7.7.1. Automatic Optimization

Hibernate Search can automatically optimize an index after either:

Infinispan Query automatically optimizes the index after:

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

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

Example: Defining Automatic Optimization Parameters

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

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

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

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

The default implementation of OptimizerStrategy can be overridden by implementing **org.hibernate.search.store.optimization.OptimizerStrategy** and setting the **optimizer.implementation** property to the fully qualified name of your implementation. This implementation must implement the interface, be a public class and have a public constructor taking no arguments.

Example: 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.

7.7.2. Manual Optimization

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

Example: Programmatic Index Optimization

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

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

The first example optimizes the Lucene index holding Orders and the second optimizes all indexes.

NOTE

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

searchFactory.optimize() is applied to the master node because it does not affect the JMC back end.

7.7.3. Adjusting Optimization

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

Further index optimization parameters include:

- **hibernate.search.[default|<indexname>].indexwriter.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**

7.8. ADVANCED FEATURES

7.8.1. 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: Accessing the SearchFactory

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

7.8.2. Using an IndexReader

Queries in Lucene are executed on an IndexReader. Hibernate Search might cache index readers to maximize performance, or provide other efficient strategies to retrieve an updated IndexReader minimizing I/O operations. Your code can access these cached resources, but there are several requirements.

Example: 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 determines which indexes are needed to query this entity (considering a sharding strategy). Using the configured ReaderProvider on each index, it returns a compound **`IndexReader`** on top of all involved indexes. Because this IndexReader is shared amongst several clients, you must adhere to the following rules:

- Never call indexReader.close(), instead use readerProvider.closeReader(reader) when necessary, preferably in a finally block.
- Don not use this IndexReader for modification operations (it is a readonly IndexReader, and any such attempt will result in an exception).

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

As an alternative to the method open(Class… types) you can use open(String… indexNames), allowing you to 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: Accessing an IndexReader by Index Names

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

7.8.3. Accessing a Lucene Directory

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

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

7.8.4. Sharding Indexes

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

WARNING

Sharding should only be implemented if the advantages outweigh the disadvantages. Searching sharded indexes will typically be slower as all shards have to be opened for a single search.

Possible use cases for sharding are:

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

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

By default sharding is not enabled unless the number of shards is configured. To do this use the **`hibernate.search.<indexName>.sharding_strategy.nbr_of_shards`** property.

Example: Enabling Index Sharding

In this example, five shards are enabled.

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

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

Example: Specifying a Custom Sharding Strategy

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

The IndexShardingStrategy property also allows for optimizing searches by selecting which shard to run the query against. By activating a filter a sharding strategy can select a subset of the shards used to answer a query (IndexShardingStrategy.getIndexManagersForQuery) and thus speed up the query execution.

Each shard has an independent IndexManager and so can be configured to use a different directory provider and back-end configuration. The IndexManager index names for the Animal entity in the example below are **`Animal.0`** to **`Animal.4`**. In other words, each shard has the name of its owning index followed by **`.`** (dot) and its index number.

Example: Sharding Configuration for Entity Animal

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

In the example above, the configuration uses the default id string hashing strategy and shards the Animal index into 5 sub-indexes. All sub-indexes are filesystem instances and the directory where each sub-index is stored is as followed:

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

- for sub-index 3: **`/usr/lucene/shared/Animal03`** (overridden indexBase, overridden indexName)
- for sub-index 4: **`/usr/lucene/indexes/Animal.4`** (shared indexBase, default indexName)

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

7.8.5. Customizing Lucene's Scoring Formula

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

Extend org.apache.lucene.search.Similarity to customize Lucene's scoring formula. The abstract methods match the formula used to calculate the score of query **`q`** for document **`d`** as follows:

```
*score(q,d) = coord(q,d) · queryNorm(q) · ∑ ~t in q~ ( tf(t in d) ·
idf(t) ^2^ · t.getBoost() · norm(t,d) )*
```

Factor	Description
tf(t ind)	Term frequency factor for the term (t) in the document (d).
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. See Similarity's Javadocs for more information.

Hibernate Search provides three ways to modify Lucene's similarity calculation.

First you can set the default similarity by specifying the fully specified class name of your Similarity implementation using the property **`hibernate.search.similarity`**. The default value is org.apache.lucene.search.DefaultSimilarity.

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

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

Finally you can override the default similarity on class level using the **@Similarity** annotation.

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

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

WARNING

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

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

7.8.6. Exception Handling Configuration

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

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

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

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

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

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

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

7.8.7. Disable Hibernate Search

Hibernate Search can be partially or completely disabled as required. Hibernate Search's indexing can be disabled, for example, if the index is read-only, or you prefer to perform indexing manually, rather than automatically. It is also possible to completely disable Hibernate Search, preventing indexing and searching.

Disable Indexing

To disable Hibernate Search indexing, change the **`indexing_strategy`** configuration option to **`manual`**, then restart JBoss EAP.

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

Disable Hibernate Search Completely

To disable Hibernate Search completely, disable all listeners by changing the **`autoregister_listeners`** configuration option to **`false`**, then restart JBoss EAP.

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

7.9. 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.

Access to Statistics via JMX

To enable access to statistics via JMX, set the property **`hibernate.search.jmx_enabled`** to **`true`**. This will automatically register the**`StatisticsInfoMBean`** bean, providing access to statistics using the **`Statistics`** object. Depending on your configuration the **`IndexingProgressMonitorMBean`** bean may also be registered.

Monitoring Indexing

If the mass indexer API is used, you can monitor indexing progress using the **`IndexingProgressMonitorMBean`** bean. The bean is only bound to JMX while indexing is in progress.

NOTE

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

APPENDIX A. REFERENCE MATERIAL

A.1. HIBERNATE PROPERTIES

Table A.1. Connection Properties Configurable in the `persistence.xml` File

Property Name	Value	Description
`javax.persistence.jdbc.driver`	**`org.hsqldb.jdbcDriver`**	The class name of the JDBC driver to be used.
`javax.persistence.jdbc.user`	sa	The username.
`javax.persistence.jdbc.password`		The password.
`javax.persistence.jdbc.url`	**`jdbc:hsqldb:.`**	The JDBC connection URL.

Table A.2. Hibernate Configuration Properties

Property Name	Description
hibernate.dialect	The class name of a Hibernate **`org.hibernate.dialect.Dialect`**. Allows Hibernate to generate SQL optimized for a particular relational database. In most cases Hibernate will be able to choose the correct **`org.hibernate.dialect.Dialect`** implementation, based on the JDBC metadata returned by the JDBC driver.
hibernate.show_sql	Boolean. Writes all SQL statements to console. This is an alternative to setting the log category **`org.hibernate.SQL`** to **`debug`**.
hibernate.format_sql	Boolean. Pretty print the SQL in the log and console.
hibernate.default_schema	Qualify unqualified table names with the given schema/tablespace in generated SQL.
hibernate.default_catalog	Qualifies unqualified table names with the given catalog in generated SQL.
hibernate.session_factory_name	The org.hibernate.SessionFactory will be automatically bound to this name in JNDI after it has been created. For example, **`jndi/composite/name`**.

Property Name	Description
hibernate.max_fetch_depth	Sets a maximum depth for the outer join fetch tree for single-ended associations (one-to-one, many-to-one). A **`0`** disables default outer join fetching. The recommended value is between **`0`** and **`3`**.
hibernate.default_batch_fetch_size	Sets a default size for Hibernate batch fetching of associations. The recommended values are **`4`**, **`8`**, and **`16`**.
hibernate.default_entity_mode	Sets a default mode for entity representation for all sessions opened from this **`SessionFactory`**. Values include: **`dynamic-map`**, **`dom4j`**, **`pojo`**.
hibernate.order_updates	Boolean. Forces Hibernate to order SQL updates by the primary key value of the items being updated. This will result in fewer transaction deadlocks in highly concurrent systems.
hibernate.generate_statistics	Boolean. If enabled, Hibernate will collect statistics useful for performance tuning.
hibernate.use_identifier_rollback	Boolean. If enabled, generated identifier properties will be reset to default values when objects are deleted.
hibernate.use_sql_comments	Boolean. If turned on, Hibernate will generate comments inside the SQL, for easier debugging. Default value is **`false`**.
hibernate.id.new_generator_mappings	Boolean. This property is relevant when using @GeneratedValue. It indicates whether or not the new IdentifierGenerator implementations are used for javax.persistence.GenerationType.AUTO, javax.persistence.GenerationType.TABLE and javax.persistence.GenerationType.SEQUENCE. Default value is **`true`**.

Property Name	Description
hibernate.ejb.naming_strategy	Chooses the org.hibernate.cfg.NamingStrategy implementation when using Hibernate EntityManager. **`hibernate.ejb.naming_strategy`** is no longer supported in Hibernate 5.0. If used, a deprecation message will be logged indicating that it is no longer supported and has been removed in favor of the split ImplicitNamingStrategy and PhysicalNamingStrategy. If the application does not use EntityManager, follow the instructions here to configure the NamingStrategy: Hibernate Reference Documentation - Naming Strategies. For an example on native bootstrapping using MetadataBuilder and applying the implicit naming strategy, see http://docs.jboss.org/hibernate/orm/5.0/userguide/html_single/Hibernate_User_Guide.html#bootstrap-native-metadata in the Hibernate 5.0 documentation. The physical naming strategy can be applied by using **`MetadataBuilder.applyPhysicalNamingStrategy()`**. For further details on **`org.hibernate.boot.MetadataBuilder`**, see https://docs.jboss.org/hibernate/orm/5.0/javadocs/.

Property Name	Description
hibernate.implicit_naming_strategy	Specifies the **org.hibernate.boot.model.naming.ImplicitNamingStrategy** class to be used. **hibernate.implicit_naming_strategy** can also be used to configure a custom class that implements ImplicitNamingStrategy. Following short names are defined for this setting: • **default** - **ImplicitNamingStrategyJpaCompliantImpl** • **jpa** - **ImplicitNamingStrategyJpaCompliantImpl** • **legacy-jpa** - **ImplicitNamingStrategyLegacyJpaImpl** • **legacy-hbm** - **ImplicitNamingStrategyLegacyHbmImpl** • **component-path** - **ImplicitNamingStrategyComponentPathImpl** The default setting is defined by the **ImplicitNamingStrategy** in the **default** short name. If the default setting is empty, the fallback is to use **ImplicitNamingStrategyJpaCompliantImpl**.
hibernate.physical_naming_strategy	Pluggable strategy contract for applying physical naming rules for database object names. Specifies the PhysicalNamingStrategy class to be used. **PhysicalNamingStrategyStandardImpl** is used by default. **hibernate.physical_naming_strategy** can also be used to configure a custom class that implements PhysicalNamingStrategy.

IMPORTANT

For **hibernate.id.new_generator_mappings**, new applications should keep the default value of **true**. Existing applications that used Hibernate 3.3.x may need to change it to **false** to continue using a sequence object or table based generator, and maintain backward compatibility.

Table A.3. Hibernate JDBC and Connection Properties

Property Name	Description

Property Name	Description
hibernate.jdbc.fetch_size	A non-zero value that determines the JDBC fetch size (calls **`Statement.setFetchSize()`**).
hibernate.jdbc.batch_size	A non-zero value enables use of JDBC2 batch updates by Hibernate. The recommended values are between **`5`** and **`30`**.
hibernate.jdbc.batch_versioned_data	Boolean. Set this property to **`true`** if the JDBC driver returns correct row counts from **`executeBatch()`**. Hibernate will then use batched DML for automatically versioned data. Default value is to **`false`**.
hibernate.jdbc.factory_class	Select a custom org.hibernate.jdbc.Batcher. Most applications will not need this configuration property.
hibernate.jdbc.use_scrollable_resultset	Boolean. Enables use of JDBC2 scrollable resultsets by Hibernate. This property is only necessary when using user-supplied JDBC connections. Hibernate uses connection metadata otherwise.
hibernate.jdbc.use_streams_for_binary	Boolean. This is a system-level property. Use streams when writing/reading **`binary`** or **`serializable`** types to/from JDBC.
hibernate.jdbc.use_get_generated_keys	Boolean. Enables use of JDBC3 **`PreparedStatement.getGeneratedKeys()`** to retrieve natively generated keys after insert. Requires JDBC3+ driver and JRE1.4+. Set to false if JDBC driver has problems with the Hibernate identifier generators. By default, it tries to determine the driver capabilities using connection metadata.
hibernate.connection.provider_class	The class name of a custom org.hibernate.connection.ConnectionProvider which provides JDBC connections to Hibernate.
hibernate.connection.isolation	Sets the JDBC transaction isolation level. Check java.sql.Connection for meaningful values, but note that most databases do not support all isolation levels and some define additional, non-standard isolations. Standard values are **`1, 2, 4, 8`**.

Property Name	Description
hibernate.connection.autocommit	Boolean. This property is not recommended for use. Enables autocommit for JDBC pooled connections.
hibernate.connection.release_mode	Specifies when Hibernate should release JDBC connections. By default, a JDBC connection is held until the session is explicitly closed or disconnected. The default value auto will choose **`after_statement`** for the JTA and CMT transaction strategies, and **`after_transaction`** for the JDBC transaction strategy. Available values are auto (default), on_close, **`after_transaction`**, **`after_statement`**. This setting only affects the session returned from **`SessionFactory.openSession`**. For the session obtained through **`SessionFactory.getCurrentSession`**, the **`CurrentSessionContext`** implementation configured for use controls the connection release mode for that session.
hibernate.connection.*<propertyName>*	Pass the JDBC property *<propertyName>* to **`DriverManager.getConnection()`**.
hibernate.jndi.*<propertyName>*	Pass the property *<propertyName>* to the JNDI **`InitialContextFactory`**.

Table A.4. Hibernate Cache Properties

Property Name	Description
`hibernate.cache.region.factory_class`	The class name of a custom **`CacheProvider`**.
`hibernate.cache.use_minimal_puts`	Boolean. Optimizes second-level cache operation to minimize writes, at the cost of more frequent reads. This setting is most useful for clustered caches and, in Hibernate3, is enabled by default for clustered cache implementations.
`hibernate.cache.use_query_cache`	Boolean. Enables the query cache. Individual queries still have to be set cacheable.

Property Name	Description
hibernate.cache.use_second_level_cache	Boolean. Used to completely disable the second level cache, which is enabled by default for classes that specify a **<cache>** mapping.
hibernate.cache.query_cache_factory	The class name of a custom **QueryCache** interface. The default value is the built-in **StandardQueryCache**.
hibernate.cache.region_prefix	A prefix to use for second-level cache region names.
hibernate.cache.use_structured_entries	Boolean. Forces Hibernate to store data in the second-level cache in a more human-friendly format.
hibernate.cache.default_cache_concurrency_strategy	Setting used to give the name of the default org.hibernate.annotations.CacheConcurrencyStrategy to use when either @Cacheable or @Cache is used. **@Cache(strategy="..")** is used to override this default.

Table A.5. Hibernate Transaction Properties

Property Name	Description
hibernate.transaction.factory_class	The classname of a **TransactionFactory** to use with Hibernate **Transaction** API. Defaults to **JDBCTransactionFactory**).
jta.UserTransaction	A JNDI name used by **JTATransactionFactory** to obtain the JTA **UserTransaction** from the application server.
hibernate.transaction.manager_lookup_class	The classname of a **TransactionManagerLookup**. It is required when JVM-level caching is enabled or when using hilo generator in a JTA environment.
hibernate.transaction.flush_before_completion	Boolean. If enabled, the session will be automatically flushed during the before completion phase of the transaction. Built-in and automatic session context management is preferred.

Property Name	Description
`hibernate.transaction.auto_close_session`	Boolean. If enabled, the session will be automatically closed during the after completion phase of the transaction. Built-in and automatic session context management is preferred.

Table A.6. Miscellaneous Hibernate Properties

Property Name	Description
`hibernate.current_session_context_class`	Supply a custom strategy for the scoping of the "current" **`Session`**. Values include **`jta`**, **`thread`**, **`managed`**, **`custom.Class`**.
`hibernate.query.factory_class`	Chooses the HQL parser implementation: **`org.hibernate.hql.internal.ast.ASTQueryTranslatorFactory`** or **`org.hibernate.hql.internal.classic.ClassicQueryTranslatorFactory`**.
`hibernate.query.substitutions`	Used to map from tokens in Hibernate queries to SQL tokens (tokens might be function or literal names). For example, **`hqlLiteral=SQL_LITERAL, hqlFunction=SQLFUNC`**.
`hibernate.query.conventional_java_constants`	Indicates whether the Java constants follow the Java naming conventions or not. Default is **`false`**. Existing applications may set it to **`true`** only if conventional Java constants are being used in the applications. Setting this to **`true`** has significant performance improvement because then Hibernate can determine if an alias should be treated as a Java constant simply by checking if the alias follows the Java naming conventions. When this property is set to **`false`**, Hibernate determines an alias should be treated as a Java constant by attempting to load the alias as a class, which is an overhead for the application. If alias fails to load as a class, then Hibernate treats the alias as a Java constant.

Property Name	Description
`hibernate.hbm2ddl.auto`	Automatically validates or exports schema DDL to the database when the **`SessionFactory`** is created. With **`create-drop`**, the database schema will be dropped when the **`SessionFactory`** is closed explicitly. Property value options are **`validate`**, **`update`**, **`create`**, **`create-drop`**
`hibernate.hbm2ddl.import_files`	Comma-separated names of the optional files containing SQL DML statements executed during the SessionFactory creation. This is useful for testing or demonstrating. For example, by adding INSERT statements, the database can be populated with a minimal set of data when it is deployed. An example value is **`/humans.sql,/dogs.sql`**. File order matters, as the statements of a given file are executed before the statements of the following files. These statements are only executed if the schema is created, for example if **`hibernate.hbm2ddl.auto`** is set to **`create`** or **`create-drop`**.
`hibernate.hbm2ddl.import_files_sql_extractor`	The classname of a custom ImportSqlCommandExtractor. Defaults to the built-in SingleLineSqlCommandExtractor. This is useful for implementing a dedicated parser that extracts a single SQL statement from each import file. Hibernate also provides MultipleLinesSqlCommandExtractor, which supports instructions/comments and quoted strings spread over multiple lines (mandatory semicolon at the end of each statement).
`hibernate.bytecode.use_reflection_optimizer`	Boolean. This is a system-level property, which cannot be set in the **`hibernate.cfg.xml`** file. Enables the use of bytecode manipulation instead of runtime reflection. Reflection can sometimes be useful when troubleshooting. Hibernate always requires either cglib or javassist even if the optimizer is turned off.
`hibernate.bytecode.provider`	Both javassist or cglib can be used as byte manipulation engines. The default is **`javassist`**. The value is either **`javassist`** or **`cglib`**.

Table A.7. Hibernate SQL Dialects (`hibernate.dialect`)

RDBMS	Dialect
DB2	**`org.hibernate.dialect.DB2Dialect`**
DB2 AS/400	**`org.hibernate.dialect.DB2400Dialect`**
DB2 OS390	**`org.hibernate.dialect.DB2390Dialect`**
Firebird	**`org.hibernate.dialect.FirebirdDialect`**
FrontBase	**`org.hibernate.dialect.FrontbaseDialect`**
H2 Database	**`org.hibernate.dialect.H2Dialect`**
HypersonicSQL	**`org.hibernate.dialect.HSQLDialect`**
Informix	**`org.hibernate.dialect.InformixDialect`**
Ingres	**`org.hibernate.dialect.IngresDialect`**
Interbase	**`org.hibernate.dialect.InterbaseDialect`**
MariaDB 10	**`org.hibernate.dialect.MySQL57InnoDBDialect`**
MariaDB Galera Cluster 10	**`org.hibernate.dialect.MySQL57InnoDBDialect`**
Mckoi SQL	**`org.hibernate.dialect.MckoiDialect`**
Microsoft SQL Server 2000	**`org.hibernate.dialect.SQLServerDialect`**
Microsoft SQL Server 2005	**`org.hibernate.dialect.SQLServer2005Dialect`**
Microsoft SQL Server 2008	**`org.hibernate.dialect.SQLServer2008Dialect`**
Microsoft SQL Server 2012	**`org.hibernate.dialect.SQLServer2012Dialect`**
Microsoft SQL Server 2014	**`org.hibernate.dialect.SQLServer2012Dialect`**
Microsoft SQL Server 2016	**`org.hibernate.dialect.SQLServer2012Dialect`**
MySQL5	**`org.hibernate.dialect.MySQL5Dialect`**
MySQL5.7	**`org.hibernate.dialect.MySQL57InnoDBDialect`**
MySQL5 with InnoDB	**`org.hibernate.dialect.MySQL5InnoDBDialect`**
MySQL with MyISAM	**`org.hibernate.dialect.MySQLMyISAMDialect`**

RDBMS	Dialect
Oracle (any version)	**`org.hibernate.dialect.OracleDialect`**
Oracle 9i	**`org.hibernate.dialect.Oracle9iDialect`**
Oracle 10g	**`org.hibernate.dialect.Oracle10gDialect`**
Oracle 11g	**`org.hibernate.dialect.Oracle10gDialect`**
Oracle 12c	**`org.hibernate.dialect.Oracle12cDialect`**
Pointbase	**`org.hibernate.dialect.PointbaseDialect`**
PostgreSQL	**`org.hibernate.dialect.PostgreSQLDialect`**
PostgreSQL 9.2	**`org.hibernate.dialect.PostgreSQL9Dialect`**
PostgreSQL 9.3	**`org.hibernate.dialect.PostgreSQL9Dialect`**
PostgreSQL 9.4	**`org.hibernate.dialect.PostgreSQL94Dialect`**
Postgres Plus Advanced Server	**`org.hibernate.dialect.PostgresPlusDialect`**
Progress	**`org.hibernate.dialect.ProgressDialect`**
SAP DB	**`org.hibernate.dialect.SAPDBDialect`**
Sybase	**`org.hibernate.dialect.SybaseASE15Dialect`**
Sybase 15.7	**`org.hibernate.dialect.SybaseASE157Dialect`**
Sybase 16	**`org.hibernate.dialect.SybaseASE157Dialect`**
Sybase Anywhere	**`org.hibernate.dialect.SybaseAnywhereDialect`**

IMPORTANT

The **`hibernate.dialect`** property should be set to the correct **`org.hibernate.dialect.Dialect`** subclass for the application database. If a dialect is specified, Hibernate will use sensible defaults for some of the other properties. This means that they do not have to be specified manually.

Revised on 2018-10-11 12:31:18 UTC