Red Hat JBoss Data Grid 6.4

Infinispan Query Guide

Using the Infinispan Query Module in Red Hat JBoss Data Grid 6.4.1

Last Updated: 2017-11-13

Red Hat JBoss Data Grid 6.4 Infinispan Query Guide

Using the Infinispan Query Module in Red Hat JBoss Data Grid 6.4.1

Misha Husnain Ali
Red Hat Engineering Content Services
mhusnain@redhat.com

Gemma Sheldon
Red Hat Engineering Content Services
gsheldon@redhat.com

Legal Notice

Abstract

This guide presents information about Infinispan Query for JBoss Data Grid and other JBoss Enterprise Platforms.

Table of Contents

CHAPTER 1. GETTING STARTED WITH INFINISPAN QUERY

1.1. INTRODUCTION

The Red Hat JBoss Data Grid Library mode Querying API enables you to search for entries in the grid using values instead of keys. It provides features such as:

- Keyword, Range, Fuzzy, Wildcard, and Phrase queries
- Combining queries
- Sorting, filtering, and pagination of query results

This API, which is based on Apache Lucene and Hibernate Search, is supported in JBoss Data Grid. Additionally, JBoss Data Grid provides an alternate mechanism that allows direct indexless searching. See Chapter 6, *The Infinispan Query DSL* for details.

Report a bug

1.2. INSTALLING QUERYING FOR RED HAT JBOSS DATA GRID

In Red Hat JBoss Data Grid, the JAR files required to perform queries are packaged within the JBoss Data Grid Library and Remote Client-Server mode downloads.

For details about downloading and installing JBoss Data Grid, see the *Getting Started Guide*'s *Download and Install JBoss Data Grid* chapter.

WARNING

The Infinispan query API directly exposes the Hibernate Search and the Lucene APIs and cannot be embedded within the `infinispan-embedded-query.jar` file. Do not include other versions of Hibernate Search and Lucene in the same deployment as `infinispan-embedded-query`. This action will cause classpath conflicts and result in unexpected behavior.

Report a bug

1.3. ABOUT QUERYING IN RED HAT JBOSS DATA GRID

1.3.1. Hibernate Search and the Query Module

Users have the ability to query the entire stored data set for specific items in Red Hat JBoss Data Grid. Applications may not always be aware of specific keys, however different parts of a value can be queried using the Query Module.

The JBoss Data Grid Query Module utilizes the capabilities of Hibernate Search and Apache Lucene to index and search objects in the cache. This allows objects to be located within the cache based on their properties, rather than requiring the keys for each object.

Objects can be searched for based on some of their properties. For example:

- Retrieve all red cars (an exact metadata match).
- Search for all books about a specific topic (full text search and relevance scoring).

An exact data match can also be implemented with the MapReduce function, however full text and relevance based scoring can only be performed via the Query Module.

Report a bug

1.3.2. Apache Lucene and the Query Module

In order to perform querying on the entire data set stored in the distributed grid, Red Hat JBoss Data Grid utilizes the capabilities of the Apache Lucene indexing tool, as well as Hibernate Search.

- Apache Lucene is a document indexing tool and search engine. JBoss Data Grid uses Apache Lucene 3.6.
- JBoss Data Grid's Query Module is a toolkit based on Hibernate Search that reduces Java objects into a format similar to a document, which is able to be indexed and queried by Apache Lucene.

In JBoss Data Grid, the Query Module indexes keys and values annotated with Hibernate Search indexing annotations, then updates the index based in Apache Lucene accordingly.

Hibernate Search intercepts changes to entries stored in the data grid to generate corresponding indexing operations

Report a bug

1.4. INDEXING

When indexing is set up, the Query module transparently indexes every added, updated, or removed cache entry. Indices improve performance of queries, though induce additional overhead during updates. For index-less querying see Chapter 6, *The Infinispan Query DSL*.

For data that already exists in the grid, create an initial Lucene index. After relevant properties and annotations are added, trigger an initial batch index of the books as shown in Section 2.4.4, "Rebuilding the Index".

Report a bug

1.4.1. Indexing with Transactional and Non-transactional Caches

In Red Hat JBoss Data Grid, the relationship between transactions and indexing is as follows:

- If the cache is transactional, index updates are applied using a listener after the commit process (after-commit listener). Index update failure does not cause the write to fail.
- If the cache is not transactional, index updates are applied using a listener that works after the event completes (post-event listener). Index update failure does not cause the write to fail.

Report a bug

1.5. SEARCHING

To execute a search, create a Lucene query (see Section 5.1.1, "Building a Lucene Query Using the Lucene-based Query API"). Wrap the query in a **`org.infinispan.query.CacheQuery`** to get the required functionality from the Lucene-based API. The following code prepares a query against the indexed fields. Executing the code returns a list of **Books**.

Example 1.1. Using Infinispan Query to Create and Execute a Search

```
QueryBuilder qb =
Search.getSearchManager(cache).buildQueryBuilderForClass(Book.class).get
();

org.apache.lucene.search.Query query = qb
  .keyword()
  .onFields("title", "author")
  .matching("Java rocks!")
  .createQuery();

// wrap Lucene query in a org.infinispan.query.CacheQuery
CacheQuery cacheQuery = Search.getSearchManager(cache).getQuery(query);

List list = cacheQuery.list();
```

Report a bug

CHAPTER 2. SET UP AND CONFIGURE INFINISPAN QUERY

2.1. SET UP INFINISPAN QUERY

2.1.1. Infinispan Query Dependencies in Library Mode

To use the JBoss Data Grid Infinispan Query via Maven, add the following dependency:

```
<dependency>
   <groupId>org.infinispan</groupId>
   <artifactId>infinispan-embedded-query</artifactId>
   <version>${infinispan.version}</version>
</dependency>
```

Non-Maven users must install all **`infinispan-embedded-query.jar`**, **`infinispan-embedded.jar`**, **`jboss-transaction-api_1.1_spec-1.0.1.Final-redhat-2.jar`** files from the JBoss Data Grid distribution.

WARNING

The Infinispan query API directly exposes the Hibernate Search and the Lucene APIs and cannot be embedded within the **`infinispan-embedded-query.jar`** file. Do not include other versions of Hibernate Search and Lucene in the same deployment as **`infinispan-embedded-query`**. This action will cause classpath conflicts and result in unexpected behavior.

Report a bug

2.2. INDEXING MODES

2.2.1. Managing Indexes

In Red Hat JBoss Data Grid's Query Module there are two options for storing indexes:

1. Each node can maintain an individual copy of the global index.
2. The index can be shared across all nodes.

When the indexes are stored locally, each write to cache must be forwarded to all other nodes so that they can update their indexes. If the index is shared, only the node where the write originates is required to update the shared index.

Lucene provides an abstraction of the directory structure called **`directory provider`**, which is used to store the index. The index can be stored, for example, as in-memory, on filesystem, or in distributed cache.

Report a bug

2.2.2. Managing the Index in Local Mode

In local mode, any Lucene Directory implementation may be used. The ***`indexLocalOnly`*** option is meaningless in local mode.

Report a bug

2.2.3. Managing the Index in Replicated Mode

In replication mode, each node can store its own local copy of the index. To store indexes locally on each node, set **`indexLocalOnly`** to **`false`**, so that each node will apply the required updates it receives from other nodes in addition to the updates started locally.

Any Directory implementation can be used. When a new node is started it must receive an up to date copy of the index. Usually this can be done via resync, however being an external operation, this may result in a slightly out of sync index, particularly where updates are frequent.

Alternatively, if a shared storage for indexes is used (see Section 2.3.3, “Infinispan Directory Provider”), **`indexLocalOnly`** must be set to **`true`** so that each node will only apply the changes originated locally. While there is no risk of having an out of sync index, this causes contention on the node used for updating the index.

The following diagram demonstrates a replicated deployment where each node has a local index.

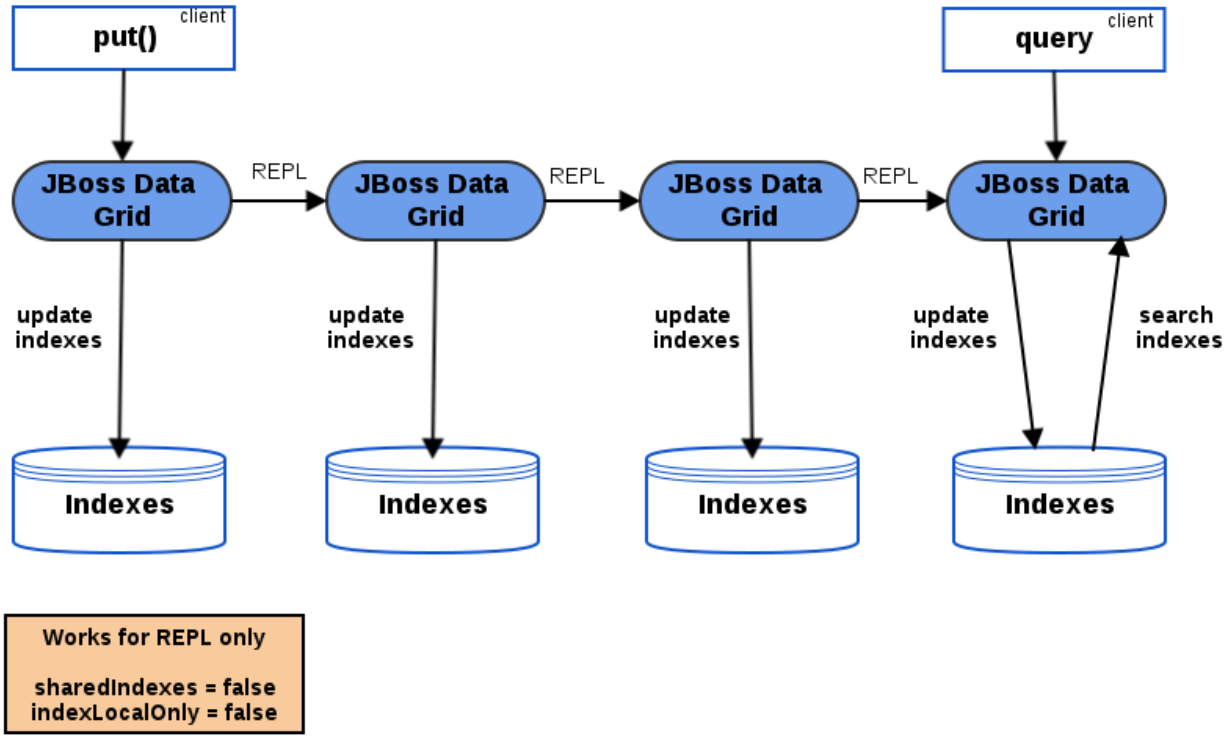


Figure 2.1. Replicated Cache Querying

Report a bug

2.2.4. Managing the Index in Distribution Mode

In both Distribution modes, the shared index must be used, with the **`indexLocalOnly`** set to **`true`**.

The following diagram shows a deployment with a shared index.

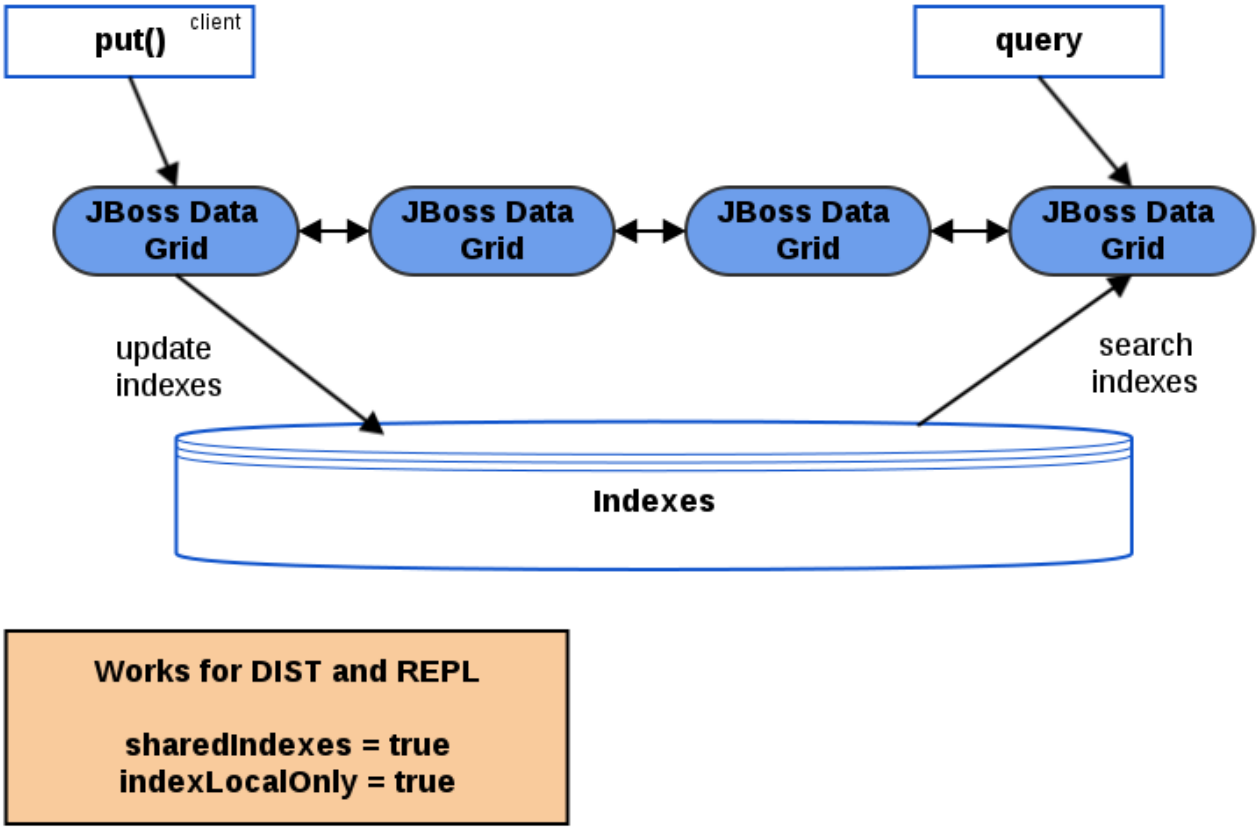


Figure 2.2. Querying with a Shared Index

Report a bug

2.2.5. Managing the Index in Invalidation Mode

Indexing and searching of elements in Invalidation mode is not supported.

Report a bug

2.3. DIRECTORY PROVIDERS

The following directory providers are supported in Infinispan Query:

- RAM Directory Provider
- Filesystem Directory Provider
- Infinispan Directory Provider

Report a bug

2.3.1. RAM Directory Provider

Storing the global index locally in Red Hat JBoss Data Grid's Query Module allows each node to

- maintain its own index.
- use **Lucene**'s in-memory or filesystem-based index directory.

The following example demonstrates an in-memory, RAM-based index store:

```
<namedCache name="indexesInMemory">
        <indexing enabled="true">
                <properties>
                        <property name="default.directory_provider"
value="ram"/>
                </properties>
        </indexing>
</namedCache>
```

Report a bug

2.3.2. Filesystem Directory Provider

To configure the storage of indexes, set the appropriate properties when enabling indexing in the JBoss Data Grid configuration.

This example shows a disk-based index store:

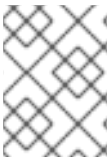
Example 2.1. Disk-based Index Store

```
<namedCache name="indexesOnDisk">
        <indexing enabled="true">
                <properties>
                        <property name="default.directory_provider"
value="filesystem"/>
                        <property name="default.indexBase"
value="/tmp/ispn_index"/>
                </properties>
        </indexing>
</namedCache>
```

Report a bug

2.3.3. Infinispan Directory Provider

In addition to the **Lucene** directory implementations, Red Hat JBoss Data Grid also ships with an **`infinispan-directory`** module.



NOTE

Red Hat JBoss Data Grid only supports **`infinispan-directory`** in the context of the Querying feature, not as a standalone feature.

The **`infinispan-directory`** allows **Lucene** to store indexes within the distributed data grid. This allows the indexes to be distributed, stored in-memory, and optionally written to disk using the cache store for durability.

Sharing the same index instance using the Infinispan **Directory Provider**, introduces a write contention point, as only one instance can write on the same index at the same time.

InfinispanIndexManager provides a default back end that sends all updates to master node which later applies the updates to the index. In case of master node failure, the update can be lost, therefore keeping the cache and index non-synchronized. Non-default back ends are not supported.

Example 2.2. Enable Shared Indexes

```
<namedCache name="globalSharedIndexes">
 <clustering mode="distributed"/>
 <indexing enabled="true">
  <properties>
   <property name="default.directory_provider" value="infinispan"/>
   <property name="default.indexmanager"
value="org.infinispan.query.indexmanager.InfinispanIndexManager" />
  </properties>
 </indexing>
</namedCache>
```

Report a bug

2.4. CONFIGURE INDEXING

2.4.1. Configure Indexing in Library Mode Using XML

Indexing can be configured in XML by adding the ***<indexing ... />*** element to the cache configuration in the Infinispan core configuration file, and optionally pass additional properties in the embedded Lucene-based Query API engine. For example:

Example 2.3. Configuring Indexing Using XML

```
<?xml version="1.0" encoding="UTF-8"?>
<infinispan
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="urn:infinispan:config:6.0
http://www.infinispan.org/schemas/infinispan-config-6.0.xsd"
      xmlns="urn:infinispan:config:6.0">
   <default>
      <indexing enabled="true">
         <properties>
            <property name="default.directory_provider" value="ram" />
         </properties>
      </indexing>
   </default>
</infinispan>
```

In this example, the index is stored in memory. As a result, when the relevant nodes shut down the index is lost. This arrangement is ideal for brief demonstration purposes, but in real world applications, use the default (store on file system) or store the index in Red Hat JBoss Data Grid to persist the index.

Report a bug

2.4.2. Configure Indexing Programmatically

Indexing can be configured programmatically, avoiding XML configuration files.

In this example, Red Hat JBoss Data Grid is started programmatically and also maps an object ***Author***, which is stored in the grid and made searchable via two properties, without annotating the class.

Example 2.4. Configure Indexing Programmatically

```
SearchMapping mapping = new SearchMapping();
mapping.entity(Author.class).indexed().providedId()
      .property("name", ElementType.METHOD).field()
      .property("surname", ElementType.METHOD).field();

Properties properties = new Properties();
properties.put(org.hibernate.search.Environment.MODEL_MAPPING, mapping);
properties.put("[other.options]", "[...]");

Configuration infinispanConfiguration = new ConfigurationBuilder()
      .indexing()
         .enable()
         .withProperties(properties)
      .build();

DefaultCacheManager cacheManager = new
DefaultCacheManager(infinispanConfiguration);

Cache<Long, Author> cache = cacheManager.getCache();
SearchManager sm = Search.getSearchManager(cache);

Author author = new Author(1, "FirstName", "Surname");
cache.put(author.getId(), author);

QueryBuilder qb = sm.buildQueryBuilderForClass(Author.class).get();
Query q =
qb.keyword().onField("name").matching("FirstName").createQuery();
CacheQuery cq = sm.getQuery(q, Author.class);
Assert.assertEquals(cq.getResultSize(), 1);
```

Report a bug

2.4.3. Configure the Index in Remote Client-Server Mode

In Remote Client-Server Mode, index configuration depends on the provider and its configuration. The indexing mode depends on the provider and whether or not it is local or distributed. The following indexing modes are supported:

- NONE
- LOCAL = indexLocalOnly="true"
- ALL = indexLocalOnly="false"

Index configuration in Remote Client-Server Mode is as follows:

Example 2.5. Configuration in Remote-Client Server Mode

```
<indexing index="LOCAL">
    <property name="default.directory_provider" value="ram" />
    <!-- Additional configuration information here -->
</indexing>
```

Report a bug

2.4.4. Rebuilding the Index

The Lucene index can be rebuilt, if required, by reconstructing it from the data store in the cache.

The index must be rebuilt if:

- The definition of what is indexed in the types has changed.
- A parameter affecting how the index is defined, such as the ***`Analyser`*** changes.
- The index is destroyed or corrupted, possibly due to a system administration error.

To rebuild the index, obtain a reference to the **`MassIndexer`** and start it as follows:

```
SearchManager searchManager = Search.getSearchManager(cache);
searchManager.getMassIndexer().start();
```

This operation reprocesses all data in the grid, and therefore may take some time.

Rebuilding the index is also available as a JMX operation.

Report a bug

2.5. TUNING THE INDEX

2.5.1. Near-Realtime Index Manager

By default, each update is immediately flushed into the index. In order to achieve better throughput, the updates can be batched. However, this can result in a lag between the update and query -- the query can see outdated data. If this is acceptable, you can use the Near-Realtime Index Manager by setting the following.

```
<property name="default.indexmanager" value="near-real-time" />
```

Report a bug

2.5.2. Tuning Infinispan Directory

Lucene directory uses three caches to store the index:

- Data cache
- Metadata cache

- Locking cache

Configuration for these caches can be set explicitly, specifying the cache names as in the example below, and configuring those caches as usual. All of these caches must be clustered unless you using Infinispan Directory in local mode.

Example 2.6. Tuning the Infinispan Directory

```
<namedCache name="indexedCache">
      <clustering mode="DIST" />
      <indexing enabled="true">
         <properties>
            <property name="default.indexmanager"
value="org.infinispan.query.indexmanager.InfinispanIndexManager" />
            <property name="default.metadata_cachename"
value="lucene_metadata_repl" />
            <property name="default.data_cachename"
value="lucene_data_dist" />
            <property name="default.locking_cachename"
value="lucene_locking_repl" />
         </properties>
      </indexing>
   </namedCache>
   <namedCache name="lucene_metadata_repl">
      <clustering mode="REPL" />
   </namedCache>
   <namedCache name="lucene_data_dist">
      <clustering mode="DIST" />
   </namedCache>
   <namedCache name="lucene_locking_repl">
      <clustering mode="REPL" />
   </namedCache>
```

Report a bug

2.5.3. Per-Index Configuration

The indexing properties in examples above apply for all indices - this is because we use the **`default.`** prefix for each property. To specify different configuration for each index, replace **`default`** with the index name. By default, this is the full class name of the indexed object, however you can override the index name in the **`@Indexed`** annotation.

Report a bug

CHAPTER 3. ANNOTATING OBJECTS AND QUERYING

Once indexing has been enabled, custom objects being stored in Red Hat JBoss Data Grid need to be assigned appropriate annotations.

As a basic requirement, all objects required to be indexed must be annotated with

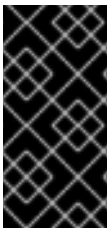
- ***@Indexed***

In addition, all fields within the object that will be searched need to be annotated with ***@Field***.

Example 3.1. Annotating Objects with @Field

```
@Indexed
public class Person
 implements Serializable {
  @Field(store = Store.YES)
  private String name;
  @Field(store = Store.YES)
  private String description;
  @Field(store = Store.YES)
  private int age;
}
```

For other annotations and options, see Chapter 4, *Mapping Domain Objects to the Index Structure*



IMPORTANT

When using JBoss EAP modules with JBoss Data Grid with the domain model as a module, add the org.infinispan.query dependency into the `module.xml` file. The custom annotations are not picked by the queries without the org.infinispan.query dependency and results in an error.

Report a bug

3.1. REGISTERING A TRANSFORMER VIA ANNOTATIONS

The key for each value must also be indexed, and the key instance must then be transformed in a String.

Red Hat JBoss Data Grid includes some default transformation routines for encoding common primitives, however to use a custom key you must provide an implementation of ***org.infinispan.query.Transformer***.

The following example shows how to annotate your key type using ***org.infinispan.query.Transformer***:

Example 3.2. Annotating the Key Type

```
@Transformable(transformer = CustomTransformer.class)
public class CustomKey {
```

```
}

public class CustomTransformer implements Transformer {
   @Override
   public Object fromString(String s) {
      return new CustomKey(...);
   }

   @Override
   public String toString(Object customType) {
      CustomKey ck = (CustomKey) customType;
      return ck.toString();
   }
}
```

The two methods must implement a biunique correspondence.

For example, for any object A the following must be true:

Example 3.3. Biunique Correspondence

```
A.equals( transformer.fromString( transformer.toString( A ) )
```

This assumes that the transformer is the appropriate Transformer implementation for objects of type A.

Report a bug

3.2. QUERYING EXAMPLE

The following provides an example of how to set up and run a query in Red Hat JBoss Data Grid.

In this example, the **Person** object has been annotated using the following:

Example 3.4. Annotating the Person Object

```
@Indexed
public class Person implements Serializable {
 @Field(store = Store.YES)
 private String name;
 @Field
 private String description;
 @Field(store = Store.YES)
 private int age;
}
```

Assuming several of these **Person** objects have been stored in JBoss DataGrid, they can be searched using querying. The following code creates a ***SearchManager*** and ***QueryBuilder*** instance:

Example 3.5. Creating the *SearchManager* and *QueryBuilder*

```
SearchManager manager = Search.getSearchManager(cache);
```

```
QueryBuilder builder = sm.buildQueryBuilderForClass(Person.class)
.get();
Query luceneQuery = builder.keyword()
 .onField("name")
 .matching("FirstName")
 .createQuery();
```

The ***SearchManager*** and ***QueryBuilder*** are used to construct a **Lucene** query. The **Lucene** query is then passed to the ***SearchManager*** to obtain a ***CacheQuery*** instance:

Example 3.6. Running the Query

```
CacheQuery query = manager.getQuery(luceneQuery);
List<Object> results = cacheQuery.list();
for (Object result : results) {
        System.out.println("Found " + result);
}
```

This ***CacheQuery*** instance contains the results of the query, and can be used to produce a list or it can be used for repeat queries.

Report a bug

CHAPTER 4. MAPPING DOMAIN OBJECTS TO THE INDEX STRUCTURE

4.1. BASIC MAPPING

In Red Hat JBoss Data Grid, the identifier for all ***@Indexed*** objects is the key used to store the value. How the key is indexed can still be customized by using a combination of ***@Transformable***, ***@ProvidedId***, custom types and custom ***FieldBridge*** implementations.

The ***@DocumentId*** identifier does not apply to JBoss Data Grid values.

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

- @Indexed
- @Field
- @NumericField

Report a bug

4.1.1. @Indexed

The **@Indexed** annotation declares a cached entry indexable. All entries not annotated with **@Indexed** are ignored.

Example 4.1. Making a class indexable with @Indexed

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

Optionally, specify the **index** attribute of the @Indexed annotation to change the default name of the index.

Report a bug

4.1.2. @Field

Each property or attribute of an entity can be indexed. Properties and attributes are not annotated by default, and therefore are ignored by the indexing process. The **@Field** annotation declares a property as indexed and allows the configuration of several aspects of the indexing process by setting one or more of the following attributes:

name

The name under which the property will be stored in the Lucene Document. By default, this attribute is the same as the property name, following the JavaBeans convention.

store

Specifies if the property is stored in the Lucene index. When a property is stored it can be retrieved in its original value from the Lucene Document. This is regardless of whether or not the element is indexed. Valid options are:

- **`Store.YES`**: Consumes more index space but allows projection. See Section 5.1.3.4, "Projection"
- **`Store.COMPRESS`**: Stores the property as compressed. This attribute consumes more CPU.
- **`Store.NO`**: No storage. This is the default setting for the store attribute.

`index`

Describes if property is indexed or not. The following values are applicable:

- **`Index.NO`**: No indexing is applied; cannot be found by querying. This setting is used for properties that are not required to be searchable, but are able to be projected.
- **`Index.YES`**: The element is indexed and is searchable. This is the default setting for the index attribute.

`analyze`

Determines if the property is analyzed. The analyze attribute allows a property to be searched by its contents. For example, it may be worthwhile to analyze a text field, whereas a date field does not need to be analyzed. Enable or disable the Analyze attribute using the following:

- **`Analyze.YES`**
- **`Analyze.NO`**

The analyze attribute is enabled by default. The **`Analyze.YES`** setting requires the property to be indexed via the **`Index.YES`** attribute.

The following attributes are used for sorting, and must not be analyzed.

`norms`

Determines whether or not to store index time boosting information. Valid settings are:

- **`Norms.YES`**
- **`Norms.NO`**

The default for this attribute is **`Norms.YES`**. Disabling norms conserves memory, however no index time boosting information will be available.

`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`**. Available settings for this attribute are:

- **`TermVector.YES`**: Stores 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`**: Does not store term vectors.

- **`TermVector.WITH_OFFSETS`**: Stores 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`**: Stores 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`**: Stores the term vector, token position and offset information. This is a combination of the **YES**, **`WITH_OFFSETS`**, and **`WITH_POSITIONS`**.

`indexNullAs`

By default, null values are ignored and not indexed. However, using ***`indexNullAs`*** permits specification of a string to be inserted as token for the null value. When using the ***`indexNullAs`*** parameter, use the same token in the search query to search for null value. Use this feature only with **`Analyze.NO`**. Valid settings for this attribute are:

- **`Field.DO_NOT_INDEX_NULL`**: This is the default value for this attribute. This setting indicates that null values will not be indexed.
- **`Field.DEFAULT_NULL_TOKEN`**: Indicates that a default null token is used. This default null token can be specified in the configuration using the default_null_token property. If this property is not set and **`Field.DEFAULT_NULL_TOKEN`** is specified, the string "_null_" will be used as default.

WARNING

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

Report a bug

4.1.3. @NumericField

The **`@NumericField`** annotation can be specified in the same scope as **`@Field`**.

The **`@NumericField`** annotation 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 optional parameters:

- ***`forField`***: Specifies the name of the related **`@Field`** that will be indexed as numeric. It is mandatory when a property contains more than a **`@Field`** declaration.

- ***precisionStep***: Changes 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 closer to the range query in normal **@Fields**. The default value for ***precisionStep*** is 4.

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

Custom **NumericFieldBridge** can also be used. Custom configurations require approximation during type transformation. The following is an example defines a custom **NumericFieldBridge**.

Example 4.2. Defining a custom NumericFieldBridge

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

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

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

}
```

Report a bug

4.2. MAPPING PROPERTIES MULTIPLE TIMES

Properties may need to be mapped multiple times per index, using different indexing strategies. For example, sorting a query by field requires that the field is not analyzed. To search by words in this property and also sort it, the property will need to be indexed it twice - once analyzed and once un-analyzed. **@Fields** can be used to perform this search. For example:

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

```
@Indexed(index = "Book" )
public class Book {
    @Fields( {
```

```
        @Field,
        @Field(name = "summary_forSort", analyze = Analyze.NO, store =
Store.YES)
    } )
    public String getSummary() {
        return summary;
    }
}
```

In the example above, the field **summary** is indexed twice - once as **summary** in a tokenized way, and once as **summary_forSort** in an untokenized way. **@Field** supports 2 attributes useful when **@Fields** is used:

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

Report a bug

4.3. EMBEDDED AND ASSOCIATED OBJECTS

Associated objects and embedded objects can be indexed as part of the root entity index. This allows searches of an entity based on properties of associated objects.

Report a bug

4.3.1. Indexing Associated Objects

The aim of the following example is to return places where the associated city is Atlanta via the Lucene query **address.city:Atlanta**. The place fields are indexed in the **Place** index. The **Place** index documents also contain the following fields:

- **address.id**
- **address.street**
- **address.city**

These fields are also able to be queried.

Example 4.4. Indexing associations

```
@Indexed
public class Place {

    @Field
    private String name;

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

public class Address {
```

```
    @Field
    private String street;

    @Field
    private String city;

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

Report a bug

4.3.2. @IndexedEmbedded

When using the **@IndexedEmbedded** technique, data is denormalized in the Lucene index. As a result, the Lucene-based Query API must be updated with any changes in the **Place** and **Address** objects to keep the index up to date. Ensure the **Place** Lucene document is updated when its **Address** changes by marking the other side of the bidirectional relationship with **@ContainedIn**. **@ContainedIn** can be used for both associations pointing to entities and on embedded objects.

The **@IndexedEmbedded** annotation can be nested. Attributes can be annotated with **@IndexedEmbedded**. The attributes of the associated class are then added to the main entity index. In the following example, the index will contain the following fields:

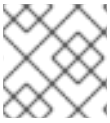
- name
- address.street
- address.city
- address.ownedBy_name

Example 4.5. Nested usage of @IndexedEmbedded and @ContainedIn

```
@Indexed
public class Place {

    @Field
    private String name;

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

public class Address {

    @Field
    private String street;

    @Field
    private String city;
```

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

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

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

The default prefix is **propertyName**, following the traditional object navigation convention. This can be overridden using the prefix attribute as it is shown on the **ownedBy** property.



NOTE

The prefix cannot be set to the empty string.

The **depth** property is used when the object graph contains a cyclic dependency of classes. For example, if **Owner** points to **Place**. the Query Module stops including attributes after reaching the expected depth, or object graph boundaries. A self-referential class is an example of cyclic dependency. In the provided example, because depth is set to 1, any **@IndexedEmbedded** attribute in **Owner** is ignored.

Using **@IndexedEmbedded** for object associations allows queries to be expressed using Lucene's query syntax. For example:

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

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

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

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

This operation is similar to the relational join operation, without data duplication. Out of the box, Lucene indexes have no notion of association; the join operation does not exist. It may be beneficial to maintain the normalized relational model while benefiting from the full text index speed and feature richness.

An associated object can be also be **@Indexed**. When **@IndexedEmbedded** points to an entity, the association must be directional and the other side must be annotated using **@ContainedIn**. If not, the Lucene-based Query API cannot update the root index when the associated entity is updated. In the provided example, a **Place** index document is updated when the associated Address instance updates.

Report a bug

4.3.3. The targetElement Property

It is possible to override the object type targeted using the ***targetElement*** parameter. This method can be used when the object type annotated by **@IndexedEmbedded** is not the object type targeted by the data grid and the Lucene-based Query API. This occurs when interfaces are used instead of their implementation.

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

```
@Indexed
public class Address {

    @Field
    private String street;

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

}

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

Report a bug

4.4. BOOSTING

Lucene uses boosting to attach more importance to specific fields or documents over others. Lucene differentiates between index and search-time boosting.

Report a bug

4.4.1. Static Index Time Boosting

The **@Boost** annotation is used to define a static boost value for an indexed class or property. This annotation can be used within **@Field**, or can be specified directly on the method or class level.

In the following example:

- the probability of Essay reaching the top of the search list will be multiplied by 1.7.
- **@Field.boost** and **@Boost** on a property are cumulative, therefore the summary field will be 3.0 (2 x 1.5), and more important than the ISBN field.
- The text field is 1.2 times more important than the ISBN field.

Example 4.7. Different ways of using @Boost

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

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

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

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

}
```

Report a bug

4.4.2. Dynamic Index Time Boosting

The **@Boost** annotation defines a static boost factor that is independent of the state of the indexed entity at runtime. However, in some cases the boost factor may depend on the actual state of the entity. In this case, use the **@DynamicBoost** annotation together with an accompanying custom ***BoostStrategy***.

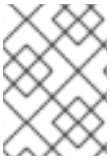
@Boost and **@DynamicBoost** annotations can both be used in relation to an entity, and all defined boost factors are cumulative. The **@DynamicBoost** can be placed at either class or field level.

In the following example, a dynamic boost is defined on class level specifying ***VIPBoostStrategy*** as implementation of the ***BoostStrategy*** interface used at indexing time. Depending on the annotation placement, either the whole entity is passed to the ***defineBoost*** method or only the annotated field/property value. The passed object must be cast to the correct type.

Example 4.8. Dynamic boost example

```
public enum PersonType {
    NORMAL,
    VIP
}

@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 provided example all indexed values of a VIP would be twice the importance of the values of a non-VIP.



NOTE

The specified ***`BoostStrategy`*** implementation must define a public no argument constructor.

Report a bug

4.5. ANALYSIS

In the Query Module, the process of converting text into single terms is called Analysis and is a key feature of the full-text search engine. Lucene uses **`Analyzer`**s to control this process.

Report a bug

4.5.1. Default Analyzer and Analyzer by Class

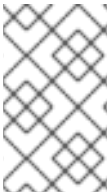
The default analyzer class is used to index tokenized fields, and is configurable through the **`default.analyzer`** property. The default value for this property is **`org.apache.lucene.analysis.standard.StandardAnalyzer`**.

The analyzer class can be defined per entity, property, and per **`@Field`**, which is useful when multiple fields are indexed from a single property.

In the following example, ***`EntityAnalyzer`*** is used to index all tokenized properties, such as name except, summary and body, which are indexed with ***`PropertyAnalyzer`*** and ***`FieldAnalyzer`*** respectively.

Example 4.9. Different ways of using @Analyzer

```
@Indexed
@Analyzer(impl = EntityAnalyzer.class)
public class MyEntity {

    @Field
    private String name;

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

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

}
```



NOTE

Avoid using different analyzers on a single entity. Doing so can create complications in building queries, and make results less predictable, particularly if using a ***QueryParser***. Use the same analyzer for indexing and querying on any field.

Report a bug

4.5.2. Named Analyzers

The Query Module uses analyzer definitions to deal with the complexity of the Analyzer function. Analyzer definitions are reusable by multiple **@Analyzer** declarations and includes the following:

- a name: the unique string used to refer to the definition.
- a list of ***CharFilters***: each ***CharFilter*** is responsible to pre-process input characters before the tokenization. ***CharFilters*** can add, change, or remove characters. One common usage is for character 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 add words into the stream provided by the ***Tokenizer***.

The **Analyzer** separates these components into multiple tasks, allowing individual components to be reused and components to be built with flexibility using the following procedure:

Procedure 4.1. The Analyzer Process

1. The ***CharFilters*** process the character input.
2. ***Tokenizer*** converts the character input into tokens.
3. The tokens are the processed by the ***TokenFilters***.

The Lucene-based Query API supports this infrastructure by utilizing the Solr analyzer framework.

Report a bug

4.5.3. Analyzer Definitions

Once defined, an analyzer definition can be reused by an **@Analyzer** annotation.

Example 4.10. Referencing an analyzer by name

```
@Indexed
@AnalyzerDef(name="customanalyzer",  )
public class Team {

    @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 useful when building queries.

```
Analyzer analyzer =
Search.getSearchManager(cache).getSearchFactory().getAnalyzer("customanaly
zer")
```

When querying, fields must use the same analyzer that has been used to index the field. The same tokens are reused between the query and the indexing process.

Report a bug

4.5.4. @AnalyzerDef for Solr

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

```
<dependency>
   <groupId>org.hibernate</groupId>
   <artifactId>hibernate-search-analyzers</artifactId>
   <version>${version.hibernate.search}</version>
<dependency>
```

In the following example, a ***CharFilter*** is defined by its factory. In this example, a mapping char filter is used, which will replace characters in the input based on the rules specified in the mapping file. Finally, a list of filters is defined by their factories. In this example, the ***StopFilter*** filter is built reading the dedicated words property file. The filter will ignore case.

Procedure 4.2. @AnalyzerDef and the Solr framework

1. **Configure the CharFilter**
   Define a ***CharFilter*** by factory. In this example, a mapping ***CharFilter*** is used, which will replace characters in the input based on the rules specified in the mapping file.

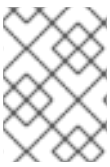
   ```
   @AnalyzerDef(name="customanalyzer",
     charFilters = {
       @CharFilterDef(factory = MappingCharFilterFactory.class, params
   = {
         @Parameter(name = "mapping",
           value = "org/hibernate/search/test/analyzer/solr/mapping-
   chars.properties")
       })
     },
   ```

2. **Define the Tokenizer**
   A ***Tokenizer*** is then defined using the **StandardTokenizerFactory.class**.

```
@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)
```

3. **List of Filters**
   Define a list of filters by their factories. In this example, the ***`StopFilter`*** filter is built reading the dedicated words property file. The filter will ignore case.

```
@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 ***`CharFilters`*** are applied in the order they are defined in the **`@AnalyzerDef`** annotation.

Report a bug

4.5.5. Loading Analyzer Resources

Tokenizers, ***TokenFilters***, and ***CharFilters*** can load resources such as configuration or metadata files using the **StopFilterFactory.class** or the synonym filter. The virtual machine default can be explicitly specified by adding a ***resource_charset*** parameter.

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

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

Report a bug

4.5.6. Dynamic Analyzer Selection

The Query Module uses the **@AnalyzerDiscriminator** annotation to enable the dynamic analyzer selection.

An analyzer can be selected based on the current state of an entity that is to be indexed. This is particularly useful in multilingual applications. For example, when using the ***BlogEntry*** class, the analyzer can depend on the language property of the entry. Depending on this property, the correct language-specific stemmer can then be chosen to index the text.

An implementation of the **Discriminator** interface must return the name of an existing Analyzer definition, or null if the default analyzer is not overridden.

The following example assumes that the language parameter is either '***de***' or '***en***', which is specified in the **@AnalyzerDefs**.

Procedure 4.3. Configure the @AnalyzerDiscriminator

1. **Predefine Dynamic Analyzers**
   The **@AnalyzerDiscriminator** requires that all analyzers that are to be used dynamically are predefined via **@AnalyzerDef**. The **@AnalyzerDiscriminator** annotation can then be placed either on the class, or on a specific property of the entity, in order to dynamically select

an analyzer. An implementation of the **`Discriminator`** interface can be specified using the **`@AnalyzerDiscriminator`** ***`impl`*** parameter.

```
@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 {

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

  @Field
  private String text;

  private Set<BlogEntry> references;

  // standard getter/setter
}
```

2. **Implement the Discriminator Interface**
   Implement the **`getAnalyzerDefinitionName()`** method, which is called for each field added to the Lucene document. The entity being indexed is also passed to the interface method.

   The ***`value`*** parameter is set if the **`@AnalyzerDiscriminator`** is placed on the property level instead of the class level. In this example, the value represents the current value of this property.

```
public class LanguageDiscriminator implements Discriminator {
    public String getAnalyzerDefinitionName(Object value, Object
entity, String field) {
        if ( value == null || !( entity instanceof Article ) ) {
            return null;
        }
        return (String) value;
    }
}
```

Report a bug

4.5.7. 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 Lucene-based Query API, which selects the correct analyzer automatically. See Section 5.1.2, “Building a Lucene Query” .

The scoped analyzer for a given entity can be retrieved using either the Lucene programmatic API or the Lucene query parser. A scoped analyzer applies the right analyzers depending on the field indexed. Multiple analyzers can be defined on a given entity, each working on an individual field. A scoped analyzer unifies these analyzers into a context-aware analyzer.

In the following example, the song title is indexed in two fields:

- Standard analyzer: used in the ***`title`*** field.
- Stemming analyzer: used in the ***`title_stemmed`*** field.

Using the analyzer provided by the search factory, the query uses the appropriate analyzer depending on the field targeted.

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

```
SearchManager manager = Search.getSearchManager(cache);

org.apache.lucene.queryParser.QueryParser parser = new QueryParser(
    org.apache.lucene.util.Version.LUCENE_36,
    "title",
    manager.getSearchFactory().getAnalyzer(Song.class)
);

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

// wrap Lucene query in a org.infinispan.query.CacheQuery
CacheQuery cacheQuery = manager.getQuery(luceneQuery, Song.class);

List result = cacheQuery.list();
//return the list of matching objects
```

NOTE

Analyzers defined via **`@AnalyzerDef`** can also be retrieved by their definition name using **`searchFactory.getAnalyzer(String)`**.

Report a bug

4.5.8. Available Analyzers

Apache Solr and Lucene ship with a number of default ***`CharFilters`***, ***`tokenizers`***, and ***`filters`***. A complete list of ***`CharFilter`***, ***`tokenizer`***, and ***`filter`*** factories is available at http://wiki.apache.org/solr/AnalyzersTokenizersTokenFilters. The following tables provide some example ***`CharFilters`***, ***`tokenizers`***, and ***`filters`***.

Table 4.1. Example of available CharFilters

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

Table 4.2. Example of available tokenizers

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

Table 4.3. Examples of available filters

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

Factory	Description	Parameters	Additional dependencies
`StopFilterFactory`	Remove words (tokens) matching a list of stop words	`words`: points to a resource file containing the stop words ignoreCase: true if `case` should be ignored when comparing stop words, `false` otherwise	`solr-core`
`SnowballPorterFilterFactory`	Reduces a word to it's root in a given language. (example: protect, protects, protection share the same root). Using such a filter allows searches matching related words.	`language`: Danish, Dutch, English, Finnish, French, German, Italian, Norwegian, Portuguese, Russian, Spanish, Swedish and a few more	`solr-core`
`ISOLatin1AccentFilterFactory`	Remove accents for languages like French	none	`solr-core`
`PhoneticFilterFactory`	Inserts phonetically similar tokens into the token stream	`encoder`: One of DoubleMetaphone, Metaphone, Soundex or RefinedSoundex inject: `true` will add tokens to the stream, `false` will replace the existing token `maxCodeLength`: sets the maximum length of the code to be generated. Supported only for Metaphone and DoubleMetaphone encodings	`solr-core` and `commons-codec`
`CollationKeyFilterFactory`	Converts each token into its `java.text.CollationKey`, and then encodes the `CollationKey` with `IndexableBinaryStringTools`, to allow it to be stored as an index term.	`custom`, `language`, `country`, `variant`, `strength`, `decomposition`see Lucene's `CollationKeyFilter` javadocs for more info	`solr-core` and `commons-io`

It is recommended that all implementations of `org.apache.solr.analysis.TokenizerFactory` and `org.apache.solr.analysis.TokenFilterFactory` are checked in your IDE to see available implementations.

Report a bug

4.6. BRIDGES

When mapping entities, Lucene represents all index fields as strings. All entity properties annotated with `@Field` are converted to strings to be indexed. Built-in bridges automatically translates properties for the Lucene-based Query API. The bridges can be customized to gain control over the translation process.

Report a bug

4.6.1. Built-in Bridges

The Lucene-based Query API includes a set of built-in bridges between a Java property type and its full text representation.

null

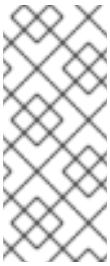
Per default `null` elements are not indexed. Lucene does not support null elements. However, in some situation it can be useful to insert a custom token representing the `null` value. See Section 4.1.2, "@Field" for more information.

java.lang.String

Strings are indexed, as are:

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

Numbers are converted into their string representation. Note that numbers cannot be compared by Lucene, or used in ranged queries out of the box, and must be padded



NOTE

Using a Range query has disadvantages. An alternative approach is to use a Filter query which will filter the result query to the appropriate range.

The Query Module supports using a custom ***StringBridge***. See Section 4.6.2, "Custom Bridges".

java.util.Date

Dates are stored as *yyyyMMddHHmmssSSS* in GMT time (200611072203012 for Nov 7th of 2006 4:03PM and 12ms EST). When using a **`TermRangeQuery`**, dates are expressed in GMT.

`@DateBridge` defines the appropriate resolution to store in the index, for example: **`@DateBridge(resolution=Resolution.DAY)`**. The date pattern will then be truncated accordingly.

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

The default **`Date`** bridge uses Lucene's **`DateTools`** to convert from and to **`String`**. All dates are expressed in GMT time. Implement a custom date bridge in order to store dates in a fixed time zone.

java.net.URI, java.net.URL

URI and URL are converted to their string representation

java.lang.Class

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

Report a bug

4.6.2. Custom Bridges

Custom bridges are available in situations where built-in bridges, or the bridge's String representation, do not sufficiently address the required property types.

Report a bug

4.6.2.1. FieldBridge

For improved flexibility, a bridge can be implemented as a **`FieldBridge`**. The **`FieldBridge`** interface provides a property value, which can then be mapped in the **`Lucene Document`**. For example, a property can be stored in two different document fields.

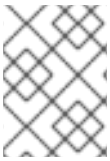
Example 4.13. Implementing the FieldBridge Interface

```
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 following example, the fields are not added directly to the **Lucene Document**. Instead the addition is delegated to the **LuceneOptions** helper. The helper will apply the options selected on **@Field**, such as ***Store*** or ***TermVector***, or apply the chosen **@Boost** value.

It is recommended that **LuceneOptions** is delegated to add fields to the **Document**, however the **Document** can also be edited directly, ignoring the **LuceneOptions**.



NOTE

LuceneOptions shields the application from changes in **Lucene API** and simplifies the code.

Report a bug

4.6.2.2. StringBridge

Use the **org.infinispan.query.bridge.StringBridge** interface to provide the Lucene-based Query API with an implementation of the expected **Object** to **String** bridge, or ***StringBridge***. All implementations are used concurrently, and therefore must be thread-safe.

Example 4.14. 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();
    }
}
```

The **@FieldBridge** annotation allows any property or field in the provided example to use the bridge:

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

Report a bug

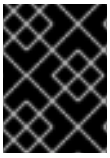
4.6.2.3. Two-Way Bridge

A **TwoWayStringBridge** is an extended version of a **StringBridge**, which can be used when the bridge implementation is used on an ID property. The Lucene-based Query API reads the string representation of the identifier and uses it to generate an object. The **@FieldBridge** annotation is used in the same way.

Example 4.15. Implementing a TwoWayStringBridge 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);
    }
}


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



IMPORTANT

The two-way process must be idempotent (ie object = stringToObject(objectToString(object))).

Report a bug

4.6.2.4. Parameterized Bridge

A **ParameterizedBridge** interface passes parameters to the bridge implementation, making it more flexible. The **ParameterizedBridge** interface can be implemented by **StringBridge**, **TwoWayStringBridge**, **FieldBridge** implementations. All implementations must be thread-safe.

The following example implements a **ParameterizedBridge** interface, with parameters passed through the **@FieldBridge** annotation.

Example 4.16. Configure the ParameterizedBridge Interface

```
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;
```

Report a bug

4.6.2.5. Type Aware Bridge

Any bridge implementing **`AppliedOnTypeAwareBridge`** will get the type the bridge is applied on injected. For example:

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

The type injected does not have any specific thread-safety requirements.

Report a bug

4.6.2.6. ClassBridge

More than one property of an entity can be combined and indexed in a specific way to the Lucene index using the **`@ClassBridge`** annotation. **`@ClassBridge`** can be defined at class level, and supports the ***`termVector`*** attribute.

In the following example, the custom **`FieldBridge`** implementation receives the entity instance as the value parameter, rather than a particular property. The particular **`CatFieldsClassBridge`** is applied to the department instance.The **`FieldBridge`** then concatenates both branch and network, and indexes the concatenation.

Example 4.17. Implementing a ClassBridge

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

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

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

    public void set( String name, Object value, Document document,
LuceneOptions luceneOptions) {

        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 );
   }
}
```

Report a bug

CHAPTER 5. QUERYING

Infinispan Query can execute Lucene queries and retrieve domain objects from a Red Hat JBoss Data Grid cache.

Procedure 5.1. Prepare and Execute a Query

1. Get **`SearchManager`** of an indexing enabled cache as follows:

```
SearchManager manager = Search.getSearchManager(cache);
```

2. Create a **`QueryBuilder`** to build queries for **`Myth.class`** as follows:

```
final org.hibernate.search.query.dsl.QueryBuilder queryBuilder =
manager.buildQueryBuilderForClass(Myth.class).get();
```

3. Create an Apache Lucene query that queries the **`Myth.class`** class' atributes as follows:

```
org.apache.lucene.search.Query query = queryBuilder.keyword()
                .onField("history").boostedTo(3)
                .matching("storm")
                .createQuery();

// wrap Lucene query in a org.infinispan.query.CacheQuery
CacheQuery cacheQuery = manager.getQuery(query);

// Get query result
List<Object> result = cacheQuery.list();
```

Report a bug

5.1. BUILDING QUERIES

Query Module queries are built on Lucene queries, allowing users to use any Lucene query type. When the query is built, Infinispan Query uses org.infinispan.query.CacheQuery as the query manipulation API for further query processing.

Report a bug

5.1.1. Building a Lucene Query Using the Lucene-based Query API

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

Report a bug

5.1.2. Building a Lucene Query

Using the Lucene programmatic API, it is possible to write full-text queries. However, when using Lucene programmatic API, the parameters must be converted to their string equivalent and must also apply the correct analyzer to the right field. A ngram analyzer for example uses several ngrams as the

tokens for a given word and should be searched as such. It is recommended to use the **`QueryBuilder`** for this task.

The Lucene-based query API is fluent. This API has a following key characteristics:

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

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

```
Search.getSearchManager(cache).buildQueryBuilderForClass(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.

Report a bug

5.1.2.1. Keyword Queries

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

Example 5.1. Keyword Search

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

Table 5.1. Keyword query parameters

Parameter	Description
keyword()	Use this parameter to find a specific word
onField()	Use this parameter to specify in which lucene field to search the word

Parameter	Description
matching()	use this parameter to specify the match for search string
createQuery()	creates the Lucene query object

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

To search a property that is not of type string.

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

}

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

NOTE

In plain Lucene, the **`Date`** object had to be converted to its string representation (in this case the year)

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

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

Example 5.2. Searching Using Ngram Analyzers

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

```
    @TokenFilterDef(factory = StopFilterFactory.class),
    @TokenFilterDef(factory = NGramFilterFactory.class,
      params = {
        @Parameter(name = "minGramSize", value = "3"),
        @Parameter(name = "maxGramSize", value = "3") } )
  }
)

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

}

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

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

NOTE

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

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

Example 5.3. Searching for Multiple Words

```
//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.

Example 5.4. Searching Multiple Fields

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

In some cases, one field must be treated differently from another field even if searching the same term. In this case, use the **`andField()`** method.

Example 5.5. Using the `andField` Method

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

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

Report a bug

5.1.2.2. Fuzzy Queries

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

Example 5.6. Fuzzy Query

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

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

Report a bug

5.1.2.3. Wildcard Queries

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

Example 5.7. Wildcard Query

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

```
    .onField("history")
    .matching("sto*")
    .createQuery();
```

NOTE

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

Report a bug

5.1.2.4. 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.

Example 5.8. Phrase Query

```
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.

Example 5.9. Adding Slop Factor

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

Report a bug

5.1.2.5. Range Queries

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

Example 5.10. Range Query

```
//look for 0 <= starred < 3
Query luceneQuery = mythQB
    .range()
    .onField("starred")
```

```
    .from(0).to(3).excludeLimit()
    .createQuery();

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

Report a bug

5.1.2.6. Combining Queries

Queries can be aggregated (combine) to create more complex queries. The following aggregation operators are available:

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

The subqueries can be any Lucene query including a boolean query itself. Following are some examples:

Example 5.11. Combining Subqueries

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

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

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

Report a bug

5.1.2.7. Query Options

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

- **`boostedTo`** (on query type and on field) boosts the query or field to a provided factor.
- **`withConstantScore`** (on query) returns all results that match the query and have a constant score equal 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 the field bridge when processing this field.

The following example illustrates how to use these options:

Example 5.12. Querying 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();
```

Report a bug

5.1.3. Build a Query with Infinispan Query

5.1.3.1. Generality

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

Example 5.13. Wrapping a Lucene Query in an Infinispan CacheQuery

```
CacheQuery cacheQuery =
Search.getSearchManager(cache).getQuery(luceneQuery);
```

For improved performance, restrict the returned types as follows:

Example 5.14. Filtering the Search Result by Entity Type

```
CacheQuery cacheQuery =
Search.getSearchManager(cache).getQuery(luceneQuery,
Customer.class);
// or
CacheQuery cacheQuery =
Search.getSearchManager(cache).getQuery(luceneQuery,
Item.class, Actor.class);
```

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

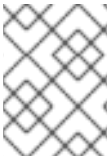
Report a bug

5.1.3.2. 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 5.15. Defining pagination for a search query

```
CacheQuery cacheQuery =
Search.getSearchManager(cache).getQuery(luceneQuery, Customer.class);
cacheQuery.firstResult(15); //start from the 15th element
cacheQuery.maxResults(10); //return 10 elements
```



NOTE

The total number of matching elements, despite the pagination, is accessible via **`cacheQuery.getResultSize()`**.

Report a bug

5.1.3.3. 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 5.16. Specifying a Lucene `Sort`

```
org.infinispan.query.CacheQuery cacheQuery =
Search.getSearchManager(cache).getQuery(luceneQuery, Book.class);
org.apache.lucene.search.Sort sort = new Sort(
    new SortField("title", SortField.STRING));
cacheQuery.setSort(sort);
List results = cacheQuery.list();
```

NOTE

Fields used for sorting must not be tokenized. For more information about tokenizing, see Section 4.1.2, “@Field”.

Report a bug

5.1.3.4. Projection

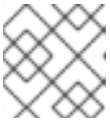
In some cases, only a small subset of the properties is required. Use Infinispan Query to return a subset of properties as follows:

Example 5.17. Using Projection Instead of Returning the Full Domain Object

```
SearchManager searchManager = Search.getSearchManager(cache);
  CacheQuery cacheQuery = searchManager.getQuery(luceneQuery,
Book.class);
  cacheQuery.projection("id", "summary", "body", "mainAuthor.name");
  List results = cacheQuery.list();
  Object[] firstResult = (Object[]) results.get(0);
  Integer id = firstResult[0];
  String summary = firstResult[1];
  String body = firstResult[2];
  String authorName = firstResult[3];
```

The Query Module 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.infinispan.query.bridge.TwoWayFieldBridge`** or **`org.infinispan.query.bridge.TwoWayStringBridge`**, the latter being the simpler version.



NOTE

All Lucene-based Query API 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 5.18. Using Projection to Retrieve Metadata

```
SearchManager searchManager = Search.getSearchManager(cache);
CacheQuery cacheQuery = searchManager.getQuery(luceneQuery, Book.class);
query.projection( FullTextQuery.SCORE, FullTextQuery.THIS,
"mainAuthor.name"
);
List results = cacheQuery.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 does.
- **`FullTextQuery.DOCUMENT`** returns the Lucene Document related to the projected object.
- **`FullTextQuery.OBJECT_CLASS`** returns the indexed entity's class.
- **`FullTextQuery.SCORE`** returns the document score in the query. Use scores to compare one result against another for a given query. However, scores are not relevant to compare the results of two different queries.
- **`FullTextQuery.ID`** is the ID property value of the projected object.
- **`FullTextQuery.DOCUMENT_ID`** is the Lucene document ID. The Lucene document ID changes between two IndexReader openings.
- **`FullTextQuery.EXPLANATION`** returns the Lucene Explanation object for the matching object/document in the query. This is not suitable for retrieving large amounts of data. Running **`FullTextQuery.EXPLANATION`** is as expensive as running a Lucene query for each matching element. As a result, projection is recommended.

Report a bug

5.1.3.5. Limiting the Time of a Query

Limit the time a query takes in Infinispan Query as follows:

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

Report a bug

5.1.3.6. Raise an Exception on Time Limit

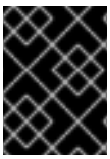
If a query uses more than the defined amount of time, a custom exception might be defined to be thrown.

To define the limit when using the CacheQuery API, use the following approach:

Example 5.19. Defining a Timeout in Query Execution

```
SearchManager searchManager = Search.getSearchManager(cache);
searchManager.setTimeoutExceptionFactory( new
MyTimeoutExceptionFactory() );
CacheQuery cacheQuery = searchManager.getQuery(luceneQuery, Book.class);

//define the timeout in seconds
cacheQuery.timeout(2, TimeUnit.SECONDS)

try {
    query.list();
}
catch (MyTimeoutException e) {
    //do something, too slow
}

private static class MyTimeoutExceptionFactory implements
TimeoutExceptionFactory {
   @Override
     public RuntimeException createTimeoutException(String message,
Query
query) {
        return new MyTimeoutException();
   }
}

public static class MyTimeoutException extends RuntimeException {
}
```

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.



IMPORTANT

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

Report a bug

5.2. RETRIEVING THE RESULTS

After building the Infinispan Query, it can be executed in the same way as a HQL or Criteria query. The same paradigm and object semantic apply to Lucene Query query and all the common operations like `list()`.

Report a bug

5.2.1. Performance Considerations

`list()` can be used to receive a reasonable number of results (for example when using pagination) and to work on them all. `list()` works best if the ***`batch-size`*** entity is correctly set up. If `list()` is used, the Query Module processes all Lucene Hits elements within the pagination.

Report a bug

5.2.2. Result Size

Some use cases require information about the total number of matching documents. Consider the following examples:

Retrieving all matching documents is costly in terms of resources. The Lucene-based Query API retrieves all matching documents regardless of pagination parameters. Since it is costly to retrieve all the matching documents, the Lucene-based Query API can retrieve the total number of matching documents regardless of the pagination parameters. All matching elements are retrieved without triggering any object loads.

Example 5.20. Determining the Result Size of a Query

```
CacheQuery cacheQuery =
Search.getSearchManager(cache).getQuery(luceneQuery,
Book.class);
//return the number of matching books without loading a single one
assert 3245 == query.getResultSize();

CacheQuery cacheQueryLimited =
Search.getSearchManager(cache).getQuery(luceneQuery, Book.class);
query.setMaxResult(10);
List results = query.list();
assert 10 == results.size()
//return the total number of matching books regardless of pagination
assert 3245 == query.getResultSize();
```

The number of results is an approximation if the index is not correctly synchronized with the database. An ansychronous cluster is an example of this scenario.

Report a bug

5.2.3. Understanding Results

Luke can be used to determine why a result appears (or does not appear) in the expected query result. The Query Module also offers the Lucene **`Explanation`** object for a given result (in a given query). This is an advanced class. Access the **`Explanation`** object as follows:

`cacheQuery.explain(int)` method

This method requires a document ID as a parameter and returns the **`Explanation`** object.

NOTE

In terms of resources, building an explanation object is as expensive as running the Lucene query. Do not build an explanation object unless it is necessary for the implementation.

Report a bug

5.3. FILTERS

Apache Lucene is able to filter query results according to a custom filtering process. This is a powerful way to apply additional data restrictions, especially since filters can be cached and reused. Applicable use cases include:

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

Report a bug

5.3.1. Defining and Implementing a Filter

The Lucene-based Query API includes transparent caches named filters which include parameters. The API is similar to the Hibernate Core filters:

Example 5.21. Enabling Fulltext Filters for a Query

```
cacheQuery = Search.getSearchManager(cache).getQuery(query,
Driver.class);
cacheQuery.enableFullTextFilter("bestDriver");
cacheQuery.enableFullTextFilter("security").setParameter( "login",
"andre" );
cacheQuery.list(); //returns only best drivers where andre has
credentials
```

In the provided example, two filters are enabled in the query. Enable or disable filters to customize the query.

Declare filters using the **`@FullTextFilterDef`** annotation. This annotation applies to **`@Indexed`** entities irrespective of the filter's query. Filter definitions are global therefore each filter must have a unique name. If two **`@FullTextFilterDef`** annotations with the same name are defined, a

`SearchException` is thrown. Each named filter must specify its filter implementation.

Example 5.22. 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 a Lucene filter that reduces the result set to drivers where the score is `5`. In the example, the filter implements the `org.apache.lucene.search.Filter` directly and contains a no-arg constructor.

Report a bug

5.3.2. The @Factory Filter

Use the following factory pattern if the filter creation requires further steps, or if the filter does not have a no-arg constructor:

Example 5.23. 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);
    }
}
```

The Lucene-based Query API 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 5.24. Passing parameters to a defined filter

```
cacheQuery = Search.getSearchManager(cache).getQuery(query,
Driver.class);
cacheQuery.enableFullTextFilter("security").setParameter( "level", 5 );
```

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

Example 5.25. 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.

Report a bug

5.3.3. Key Objects

@Key methods are needed only if:

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

The **StandardFilterKey** delegates the **equals() / hashCode()** implementation to each of the parameters equals and hashcode methods.

The defined filters are per default cached. 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 **default.filter.cache_strategy.size** (defaults to 128). For advanced use of filter caching, you can implement your own **FilterCachingStrategy**. The classname is defined by **default.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.

Report a bug

5.3.4. Full Text Filter

The Lucene-based Query API uses the ***cache*** flag of **@FullTextFilterDef**, set to **FilterCacheModeType.INSTANCE_AND_DOCIDSETRESULTS** which automatically caches the filter instance and wraps the filter around a Hibernate specific implementation of **CachingWrapperFilter**. Unlike Lucene's version of this class, **SoftReferences** are used with a hard reference count (see discussion about filter cache). The hard reference count is adjusted using ***default.filter.cache_docidresults.size*** (defaults to **5**). Wrapping is 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 the Query Module. For every filter call, a new filter instance is created. This setting addresses rapidly changing data sets or heavily memory constrained environments.

Value	Definition
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).

Report a bug

5.3.5. Using Filters in a Sharded Environment

Execute queries on a subset of the available shards in a sharded environment as follows:

1. Create a sharding strategy to select a subset of `IndexManagers` depending on filter configurations.
2. Activate the filter when running the query.

The following is an example of sharding strategy that queries a specific shard if the customer filter is activated:

Example 5.26. Querying a Specific Shard

```
public class CustomerShardingStrategy implements IndexShardingStrategy {

 // stored IndexManagers in a array indexed by customerID
 private IndexManager[] indexManagers;

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

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

 public IndexManager getIndexManagerForAddition(
     Class<?> entity, Serializable id, String idInString, Document
document) {
```

```
   Integer customerID =
Integer.parseInt(document.getFieldable("customerID").stringValue());
   return indexManagers[customerID];
 }

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

  /**
  * Optimization; don't search ALL shards and union the results; in this
case, we
  * can be certain that all the data for a particular customer Filter is
in a single
  * shard; 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;
 }
}
```

If the **customer** filter is present in the example, the query only uses the shard dedicated to the customer. The query returns all shards if the **customer** filter is not found. The sharding strategy reacts to each filter depending on the provided parameters.

Activate the filter when the query must be run. The filter is a regular filter (as defined in Section 5.3, "Filters"), which filters Lucene results after the query. As an alternate, use a special filter that is passed to the sharding strategy and then ignored for duration of the query. Use the **ShardSensitiveOnlyFilter** class to declare the filter.

Example 5.27. Using the ShardSensitiveOnlyFilter Class

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

```
CacheQuery cacheQuery = Search.getSearchManager(cache).getQuery(query,
Customer.class);
cacheQuery.enableFulltextFilter("customer").setParameter("CustomerID",
5);
@SuppressWarnings("unchecked")
List results = query.List();
```

If the **`ShardSensitiveOnlyFilter`** filter is used, Lucene filters do not need to be implemented. Use filters and sharding strategies reacting to these filters for faster query execution in a sharded environment.

Report a bug

5.4. OPTIMIZING THE QUERY PROCESS

Query performance depends on several criteria:

- The Lucene query.
- The number of objects loaded: use pagination or index projection where required.
- The way the Query Module interacts with the Lucene readers defines the appropriate reader strategy.
- Caching frequently extracted values from the index.

Report a bug

5.4.1. Caching Index Values: FieldCache

The Lucene index identifies matches to queries. Once the query is performed the results must be analyzed to extract useful information. The Lucene-based Query API is used to extract the Class type and the primary key.

Extracting the required values from the index reduces performance. In some cases this may be minor, other cases may require caching.

Requirements depend on the kind of projections being used and in some cases the Class type is not required.

The **`@CacheFromIndex`** annotation is used to perform caching on the main metadata fields required by the Lucene-based Query API.

Example 5.28. The `@CacheFromIndex` Annotation

```
import static org.infinispan.query.annotations.FieldCacheType.CLASS;
import static org.infinispan.query.annotations.FieldCacheType.ID;

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

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

- **`CLASS`**: The Query Module uses a Lucene FieldCache to improve performance of the Class type extraction from the index.

  This value is enabled by default. The Lucene-based Query API applies this value when the **`@CacheFromIndex`** annotation is not specified.

- **`ID`**: Extracting the primary identifier uses a cache. This method produces the best querying results, however it may 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 following two disadvantages:

- Memory usage: 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 is slower than when caching is disabled.

Some queries may not require a classtype, and ignore the **`CLASS`** field cache even when enabled. For example, when targeting a single class, all returned values are of that type.

The ID FieldCache requires the ids of targeted entities to be using a **`TwoWayFieldBridge`**. 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 query execution.

Report a bug

CHAPTER 6. THE INFINISPAN QUERY DSL

The Infinispan Query DSL provides an unified way of querying a cache. It can be used in Library mode for both indexed and indexless queries, as well as for Remote Querying (via the Hot Rod Java client). The Infinispan Query DSL allows queries without relying on Lucene native query API or Hibernate Search query API.

Indexless queries are only available with the Infinispan Query DSL for both the JBoss Data Grid remote and embedded mode. Indexless queries do not require a configured index (see Section 6.2, "Enabling Infinispan Query DSL-based Queries"). The Hibernate Search/Lucene-based API cannot use indexless queries.

Report a bug

6.1. CREATING QUERIES WITH INFINISPAN QUERY DSL

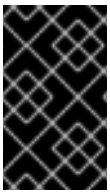
The new query API is located in the org.infinispan.query.dsl package. A query is created with the assistance of the `QueryFactory` instance, which is obtained using `Search.getQueryFactory()`. Each `QueryFactory` instance is bound to the one cache instance, and is a stateless and thread-safe object that can be used for creating multiple parallel queries.

The Infinispan Query DSL uses the following steps to perform a query.

1. A query is created by invocating the `from(Class entityType)` method, which returns a `QueryBuilder` object that is responsible for creating queries for the specified entity class from the given cache.

2. The `QueryBuilder` accumulates search criteria and configuration specified through invoking its DSL methods, and is used to build a `Query` object by invoking the `QueryBuilder.build()` method, which completes the construction. The `QueryBuilder` object cannot be used for constructing multiple queries at the same time except for nested queries, however it can be reused afterwards.

3. Invoke the `list()` method of the `Query` object to execute the query and fetch the results. Once executed, the `Query` object is not reusable. If new results must be fetched, a new instance must be obtained by calling `QueryBuilder.build()`.



IMPORTANT

A query targets a single entity type and is evaluated over the contents of a single cache. Running a query over multiple caches, or creating queries targeting several entity types is not supported.

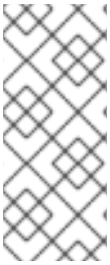
Report a bug

6.2. ENABLING INFINISPAN QUERY DSL-BASED QUERIES

In library mode, running Infinispan Query DSL-based queries is almost identical to running Lucene-based API queries. Prerequisites are:

- All libraries required for Infinispan Query (see Section 2.1.1, "Infinispan Query Dependencies in Library Mode" for details) on the classpath.

- Indexing enabled and configured for caches (optional). See Section 2.4, "Configure Indexing" for details.
- Annotated POJO cache values (optional). If indexing is not enabled, POJO annotations are also not required and are ignored if set. If indexing is not enabled, all fields that follow JavaBeans conventions are searchable instead of only the fields with Hibernate Search annotations.



NOTE

When indexing is enabled, only the indexed fields are searchable. When indexing is disabled, all the fields are searchable. Mixed-mode queries functionality is currently not available. If you query an indexed cache on non-indexed fields, an error is thrown, saying the fields are not indexed. To avoid this, either query on the intended indexed fields or use indexless cache.

Report a bug

6.3. RUNNING INFINISPAN QUERY DSL-BASED QUERIES

Once Infinispan Query DSL-based queries have been enabled, obtain a **`QueryFactory`** from the **`Search`** in order to run a DSL-based query.

Obtain a QueryFactory for a Cache

In Library mode, obtain a **`QueryFactory`** as follows:

```
QueryFactory qf = org.infinispan.query.Search.getQueryFactory(Cache<?, ?>
cache)
```

Example 6.1. Constructing a DSL-based Query

```
import org.infinispan.query.Search;
import org.infinispan.query.dsl.QueryFactory;
import org.infinispan.query.dsl.Query;

QueryFactory qf = Search.getQueryFactory(cache);
Query q = qf.from(User.class)
    .having("name").eq("John")
    .toBuilder().build();
List list = q.list();
assertEquals(1, list.size());
assertEquals("John", list.get(0).getName());
assertEquals("Doe", list.get(0).getSurname());
```

When using Remote Querying in Remote Client-Server mode, the **`Search`** object resides in package **`org.infinispan.client.hotrod`**. See the example in Section 7.2, "Performing Remote Queries via the Hot Rod Java Client" for details.

It is also possible to combine multiple conditions with boolean operators, including sub-conditions. For example:

Example 6.2. Combining Multiple Conditions

```
Query q = qf.from(User.class)
    .having("name").eq("John")
    .and().having("surname").eq("Doe")
    .and().not(qf.having("address.street").like("%Tanzania%")
    .or().having("address.postCode").in("TZ13", "TZ22"))
    .toBuilder().build();
```

This query API simplifies the way queries are written by not exposing the user to the low level details of constructing Lucene query objects. It also has the benefit of being available to remote Hot Rod clients.

The following example shows how to write a query for the **Book** entity.

Example 6.3. Querying the Book Entity

```
import org.infinispan.query.Search;
import org.infinispan.query.dsl.*;

// get the DSL query factory, to be used for constructing the Query
object:
QueryFactory qf = Search.getQueryFactory(cache);
// create a query for all the books that have a title which contains the
word "engine":
Query query = qf.from(Book.class)
      .having("title").like("%engine%")
      .toBuilder().build();
// get the results
List<Book> list = query.list();
```

Report a bug

CHAPTER 7. REMOTE QUERYING

Red Hat JBoss Data Grid's Hot Rod protocol allows remote, language neutral querying.

Two features allow this to occur:

The Infinispan Query Domain-specific Language (DSL)

JBoss Data Grid uses its own query language based on an internal DSL. The Infinispan Query DSL provides a simplified way of writing queries, and is agnostic of the underlying query mechanisms. Querying via the Hot Rod client allows remote, language-neutral querying, and is implementable in all languages currently available for the Hot Rod client.

The Infinispan Query DSL is essential for performing remote queries, but can be used in both embedded and remote mode.

Protobuf Encoding

Google's Protocol Buffers is used as an encoding format for both storing and querying data. The Infinispan Query DSL can be used remotely via the Hot Rod client that is configured to use the Protobuf marshaller. Protocol Buffers are used to adopt a common format for storing cache entries and marshalling them.

Remote clients that need to index and query their stored entities must use the Protobuf encoding format. It is also possible to store Protobuf entities for the benefit of platform independence without indexing enabled if it is not required.

Report a bug

7.1. QUERYING COMPARISON

In Library mode, both Lucene Query-based and DSL querying is available. In Remote Client-Server mode, only Remote Querying using DSL is available. The following table is a feature comparison between Lucene Query-based querying, Infinispan Query DSL and Remote Querying.

Table 7.1. Embedded querying and Remote querying

Feature	Embedded Querying and Lucene Query	Embedded DSL	Remote Querying
Indexing	Mandatory	Optional but highly recommended	Optional but highly recommended
Index contents	Selected fields	Selected fields	Selected fields
Data Storage Format	Java objects	Java objects	Protocol buffers
Keyword Queries	Yes	Yes	Yes
Range Queries	Yes	Yes	Yes
Fuzzy Queries	Yes	No	No

Feature	Embedded Querying and Lucene Query	Embedded DSL	Remote Querying
Wildcard	Yes	Limited to like queries (Matches a wildcard pattern that follows JPA rules).	Limited to like queries (Matches a wildcard pattern that follows JPA rules).
Phrase Queries	Yes	No	No
Combining Queries	AND, OR, NOT, SHOULD	AND, OR, NOT	AND, OR, NOT
Sorting Results	Yes	Yes	Yes
Filtering Results	Yes, both within the query and as appended operator	Within the query	Within the query
Pagination of Results	Yes	Yes	Yes

Report a bug

7.2. PERFORMING REMOTE QUERIES VIA THE HOT ROD JAVA CLIENT

Remote querying over Hot Rod can be enabled once the **`RemoteCacheManager`** has been configured with the Protobuf marshaller.

The following procedure describes how to enable remote querying over its caches.

Prerequisites

`RemoteCacheManager` must be configured to use the Protobuf Marshaller.

Procedure 7.1. Enabling Remote Querying via Hot Rod

1. **Add the `infinispan-remote.jar`**
   The **`infinispan-remote.jar`** is an uberjar, and therefore no other dependencies are required for this feature.

2. **Enable indexing on the cache configuration.**
   Indexing is not mandatory for Remote Queries, but it is highly recommended because it makes searches on caches that contain large amounts of data significantly faster. Indexing can be configured at any time. Enabling and configuring indexing is the same as for Library mode.

   Add the following configuration within the ***`cache-container`*** element loated inside the Infinispan subsystem element.

```
<!-- A basic example of an indexed local cache that uses the RAM
Lucene directory provider -->
 <local-cache name="an-indexed-cache" start="EAGER">
```

```
    <!-- Enable indexing using the RAM Lucene directory provider -->
    <indexing index="ALL">
       <property name="default.directory_provider">ram</property>
    </indexing>
 </local-cache>
```

3. **Register the Protobuf schema definition files**
   Register the Protobuf schema definition files by adding them in the **`___protobuf_metadata`** system cache. The cache key is a string that denotes the file name and the value is **`.proto`** file, as a string. Alternatively, protobuf schemas can also be registered by invoking the **`registerProtofile`** methods of the server's **`ProtobufMetadataManager`** MBean. There is one instance of this MBean per cache container and is backed by the **`___protobuf_metadata`**, so that the two approaches are equivalent.

   For an example of providing the protobuf schema via **`___protobuf_metadata`** system cache, see Example 7.6, “Registering a Protocol Buffers schema file” .

   The following example demonstrates how to invoke the **`registerProtofile`** methods of the **`ProtobufMetadataManager`** MBean.

   Example 7.1. Registering Protobuf schema definition files via JMX

```
import javax.management.MBeanServerConnection;
import javax.management.ObjectName;
import javax.management.remote.JMXConnector;
import javax.management.remote.JMXServiceURL;

...

String serverHost = ...          // The address of your JDG server
int serverJmxPort = ...          // The JMX port of your server
String cacheContainerName = ... // The name of your cache
container
String schemaFileName = ...      // The name of the schema file
String schemaFileContents = ... // The Protobuf schema file
contents

JMXConnector jmxConnector = JMXConnectorFactory.connect(new
JMXServiceURL("service:jmx:remoting-jmx://" + serverHost + ":" +
serverJmxPort));
MBeanServerConnection jmxConnection =
jmxConnector.getMBeanServerConnection();

ObjectName protobufMetadataManagerObjName = new
ObjectName("jboss.infinispan:type=RemoteQuery,name=" +
ObjectName.quote(cacheContainerName) +
",component=ProtobufMetadataManager");

jmxConnection.invoke(protobufMetadataManagerObjName,
"registerProtofile", new Object[]{schemaFileName,
schemaFileContents},
                     new String[]{String.class.getName(),
String.class.getName()});
jmxConnector.close();
```

Result

All data placed in the cache is immediately searchable, whether or not indexing is in use. Entries do not need to be annotated, unlike embedded queries. The entity classes are only meaningful to the Java client and do not exist on the server.

Once remote querying has been enabled, the **`QueryFactory`** can be obtained using the following:

Example 7.2. Obtaining the `QueryFactory`

```
import org.infinispan.client.hotrod.Search;
import org.infinispan.query.dsl.QueryFactory;
import org.infinispan.query.dsl.Query;
import org.infinispan.query.dsl.SortOrder;
...
remoteCache.put(2, new User("John", 33));
remoteCache.put(3, new User("Alfred", 40));
remoteCache.put(4, new User("Jack", 56));
remoteCache.put(4, new User("Jerry", 20));

QueryFactory qf = Search.getQueryFactory(remoteCache);
Query query = qf.from(User.class)
    .having("name").like("J%")
       .and().having("age").gte(33)
    .orderBy("age", SortOrder.ASC)
    .toBuilder().build();

List<User> list = query.list();
assertEquals(2, list.size());
assertEquals("John", list.get(0).getName());
assertEquals(33, list.get(0).getAge());
assertEquals("Jack", list.get(1).getName());
assertEquals(56, list.get(1).getAge());
```

Queries can now be run over Hot Rod similar to Library mode.

Report a bug

7.3. PROTOBUF ENCODING

The Infinispan Query DSL can be used remotely via the Hot Rod client. In order to do this, protocol buffers are used to adopt a common format for storing cache entries and marshalling them.

For more information, see https://developers.google.com/protocol-buffers/docs/overview

Report a bug

7.3.1. Storing Protobuf Encoded Entities

Protobuf requires data to be structured. This is achieved by declaring Protocol Buffer message types in **`.proto`** files

For example:

Example 7.3. .library.proto

```
package book_sample;
message Book {
      required string title = 1;
      required string description = 2;
      required int32 publicationYear = 3; // no native Date type
available in Protobuf

      repeated Author authors = 4;
}
message Author {
    required string name = 1;
    required string surname = 2;
}
```

The provided example:

1. An entity named ***Book*** is placed in a package named ***book_sample***.

```
package book_sample;
message Book {
```

2. The entity declares several fields of primitive types and a repeatable field named **authors**.

```
 required string title = 1;
      required string description = 2;
      required int32 publicationYear = 3; // no native Date type
available in Protobuf

      repeated Author authors = 4;
}
```

3. The **Author** message instances are embedded in the **Book** message instance.

```
message Author {
    required string name = 1;
    required string surname = 2;
}
```

Report a bug

7.3.2. About Protobuf Messages

There are a few important things to note about Protobuf messages:

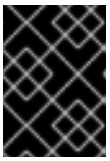
- Nesting of messages is possible, however the resulting structure is strictly a tree, and never a graph.
- There is no type inheritance.
- Collections are not supported, however arrays can be easily emulated using repeated fields.

Report a bug

7.3.3. Using Protobuf with Hot Rod

Protobuf can be used with JBoss Data Grid's Hot Rod using the following two steps:

1. Configure the client to use a dedicated marshaller, in this case, the **ProtoStreamMarshaller**. This marshaller uses the **ProtoStream** library to assist in encoding objects.



IMPORTANT

In order to use the **ProtoStreamMarshaller**, the infinispan-remote-query-client Maven dependency must be added.

2. Instruct **ProtoStream** library on how to marshall message types by registering per entity marshallers.

Example 7.4. Use the ProtoStreamMarshaller to Encode and Marshall Messages

```
import org.infinispan.client.hotrod.configuration.ConfigurationBuilder;
import org.infinispan.client.hotrod.marshall.ProtoStreamMarshaller;
import org.infinispan.protostream.SerializationContext;
ConfigurationBuilder clientBuilder = new ConfigurationBuilder();
clientBuilder.addServer()
    .host("127.0.0.1").port(11234)
    .marshaller(new ProtoStreamMarshaller());

RemoteCacheManager remoteCacheManager = new
RemoteCacheManager(clientBuilder.build());
SerializationContext srcCtx =
ProtoStreamMarshaller.getSerializationContext(remoteCacheManager);
serCtx.registerProtoFile("/library.proto");
serCtx.registerMarshaller(new BookMarshaller());
serCtx.registerMarshaller(new AuthorMarshaller());
// Book and Author classes omitted for brevity
```

In the provided example,

- The **SerializationContext** is provided by the **ProtoStream** library.
- The **SerializationContext.registerProtofile** method receives the name of a **.proto** classpath resource file that contains the message type definitions.
- The **SerializationContext** associated with the **RemoteCacheManager** is obtained, then **ProtoStream** is instructed to marshall the protobuf types.

NOTE

A **RemoteCacheManager** has no **SerializationContext** associated with it unless it was configured to use **ProtoStreamMarshaller**.

Report a bug

7.3.4. Registering Per Entity Marshallers

When using the **`ProtoStreamMarshaller`** for remote querying purposes, registration of per entity marshallers for domain model types must be provided by the user for each type or marshalling will fail. When writing marshallers, it is essential that they are stateless and threadsafe, as a single instance of them is being used.

The following example shows how to write a marshaller.

Example 7.5. BookMarshaller.java

```
import org.infinispan.protostream.MessageMarshaller;
...
public class BookMarshaller implements MessageMarshaller<Book> {
   @Override
   public String getTypeName() {
      return "book_sample.Book";
   }
   @Override
   public Class<? extends Book> getJavaClass() {
      return Book.class;
   }
   @Override
   public void writeTo(ProtoStreamWriter writer, Book book) throws
IOException {
      writer.writeString("title", book.getTitle());
      writer.writeString("description", book.getDescription());
      writer.writeCollection("authors", book.getAuthors(),
Author.class);
   }
   @Override
   public Book readFrom(ProtoStreamReader reader) throws IOException {
      String title = reader.readString("title");
      String description = reader.readString("description");
      int publicationYear = reader.readInt("publicationYear");
      Set<Author> authors = reader.readCollection("authors", new
HashSet<Author>(), Author.class);
      return new Book(title, description, publicationYear, authors);
   }
}
```

Once the client has been set up, reading and writing Java objects to the remote cache. The actual data stored in the cache will be protobuf encoded, provided that marshallers were registered with the remote client for all involved types. In the provided example, this would be ***`Book`*** and ***`Author`***.

Objects stored in protobuf format are able to be utilized with compatible clients written in different languages.

Report a bug

7.3.5. Indexing Protobuf Encoded Entities

Once the client is configured to use Protobuf, indexing can be configured for caches on the server side.

To index the entries, the server must have the knowledge of the message types defined by the Protobuf schema. A Protobuf schema file is defined in a file with a **.proto** extension. The schema is supplied to the server either by placing it in the **___protobuf_metadata** cache by a **put**, **putAll**, **putIfAbsent**, or **replace** operation, or alternatively by invoking **ProtobufMetadataManager** MBean via JMX. Both keys and values of **___protobuf_metadata** cache are Strings, the key being the file name, while the value is the schema file contents.

Example 7.6. Registering a Protocol Buffers schema file

```
import org.infinispan.client.hotrod.RemoteCache;
import org.infinispan.client.hotrod.RemoteCacheManager;
import
org.infinispan.query.remote.client.ProtobufMetadataManagerConstants;

RemoteCacheManager remoteCacheManager = ... // obtain a
RemoteCacheManager

// obtain the '__protobuf_metadata' cache
RemoteCache<String, String> metadataCache =
remoteCacheManager.getCache(ProtobufMetadataManagerConstants.PROTOBUF_ME
TADATA_CACHE_NAME);

String schemaFileContents = ... // this is the contents of the schema
file
metadataCache.put("my_protobuf_schema.proto", schemaFileContents);
```

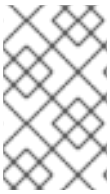
The **ProtobufMetadataManager** is a cluster-wide replicated repository of Protobuf schema definitions or **.proto** files. For each running cache manager, a separate **ProtobufMetadataManager** MBean instance exists, and is backed by the **___protobuf_metadata** cache. The **ProtobufMetadataManager** ObjectName uses the following pattern:

```
<jmx domain>:type=RemoteQuery,name=<cache manager<methodname>putAll
name>,component=ProtobufMetadataManager
```

The following signature is used by the method that registers the Protobuf schema file:

```
void registerProtofile(String name, String contents)
```

If indexing is enabled for a cache, all fields of Protobuf-encoded entries are indexed. All Protobuf-encoded entries are searchable, regardless of whether indexing is enabled.



NOTE

Indexing is recommended for improved performance but is not mandatory when using remote queries. Using indexing improves the searching speed but can also reduce the insert/update speeds due to the overhead required to maintain the index.

Report a bug

7.3.6. Custom Fields Indexing with Protobuf

All Protobuf type fields are indexed and stored by default. This behavior is acceptable in most cases but it can result in performance issues if used with too many or very large fields. To specify the fields to index and store, use the **@Indexed** and **@IndexedField** annotations directly to the Protobuf schema in the documentation comments of message type definitions and field definitions.

Example 7.7. Specifying Which Fields are Indexed

```
/*
  This type is indexed, but not all its fields are.
  @Indexed
*/
message Note {

  /*
     This field is indexed but not stored. It can be used for querying
but not for projections.
     @IndexedField(index=true, store=false)
   */
  optional string text = 1;

  /*
    A field that is both indexed and stored.
    @IndexedField
   */
  optional string author = 2;

  /* @IndexedField(index=false, store=true) */
  optional bool isRead = 3;

  /* This field is not annotated, so it is neither indexed nor stored.
*/
  optional int32 priority;
}
```

Add documentation annotations to the last line of the documentation comment that precedes the element to be annotated (message type or field definition).

The **@Indexed** annotation only applies to message types, has a boolean value (the default is **true**). As a result, using **@Indexed** is equivalent to **@Indexed(true)**. This annotation is used to selectively specify the fields of the message type which must be indexed. Using **@Indexed(false)**, however, indicates that no fields are to be indexed and so the eventual **@IndexedField** annotation at the field level is ignored.

The **@IndexedField** annotation only applies to fields, has two attributes (**index** and **store**), both of which default to **true** (using **@IndexedField** is equivalent to **@IndexedField(index=true, store=true)**). The **index** attribute indicates whether the field is indexed, and is therefore used for indexed queries. The **store** attributes indicates whether the field value must be stored in the index, so that the value is available for projections.

NOTE

The **`@IndexedField`** annotation is only effective if the message type that contains it is annotated with **`@Indexed`**.

Report a bug

7.3.7. Defining Protocol Buffers Schemas With Java Annotations

You can declare Protobuf metadata using Java annotations. Instead of providing a **`MessageMarshaller`** implementation and a **`.proto`** schema file, you can add minimal annotations to a Java class and its fields.

The objective of this method is to marshal Java objects to protobuf using the **`ProtoStream`** library. The **`ProtoStream`** library internally generates the marshallar and does not require a manually implemented one. The Java annotations require minimal information such as the Protobuf tag number. The rest is inferred based on common sense defaults (Protobuf type, Java collection type, and collection element type) and is possible to override.

The auto-generated schema is registered with the **`SerializationContext`** and is also available to the users to be used as a reference to implement domain model classes and marshallers for other languages.

The following are examples of Java annotations

Example 7.8. User.Java

```
package sample;

import org.infinispan.protostream.annotations.ProtoEnum;
import org.infinispan.protostream.annotations.ProtoEnumValue;
import org.infinispan.protostream.annotations.ProtoField;
import org.infinispan.protostream.annotations.ProtoMessage;

@ProtoMessage(name = "ApplicationUser")
public class User {

   @ProtoEnum(name = "Gender")
   public enum Gender {
    @ProtoEnumValue(number = 1, name = "M")
    MALE,

      @ProtoEnumValue(number = 2, name = "F")
      FEMALE
    }

    @ProtoField(number = 1, required = true)
    public String name;

    @ProtoField(number = 2)
    public Gender gender;
 }
```

Example 7.9. Note.Java

```
package sample;

import org.infinispan.protostream.annotations.ProtoDoc;
import org.infinispan.protostream.annotations.ProtoField;

@ProtoDoc("@Indexed")
public class Note {

   private String text;

   private User author;

   @ProtoDoc("@IndexedField(index = true, store = false)")
   @ProtoField(number = 1)
   public String getText() {
          return text;
   }

   public void setText(String text) {
      this.text = text;
   }

   @ProtoDoc("@IndexedField")
   @ProtoField(number = 2)
   public User getAuthor() {
      return author;
  }

   public void setAuthor(User author) {
      this.author = author;
   }
}
```

Example 7.10. ProtoSchemaBuilderDemo.Java

```
import org.infinispan.protostream.SerializationContext;
import org.infinispan.protostream.annotations.ProtoSchemaBuilder;
import org.infinispan.client.hotrod.RemoteCacheManager;
import org.infinispan.client.hotrod.marshall.ProtoStreamMarshaller;

...

RemoteCacheManager remoteCacheManager = ... // we have a
RemoteCacheManager
SerializationContext serCtx =
ProtoStreamMarshaller.getSerializationContext(remoteCacheManager);

// generate and register a Protobuf schema and marshallers based on Note
class and the referenced classes (User class)
ProtoSchemaBuilder protoSchemaBuilder = new ProtoSchemaBuilder();
String generatedSchema = protoSchemaBuilder
       .fileName("sample_schema.proto")
```

```
        .packageName("sample_package")
        .addClass(Note.class)
        .build(serCtx);

// the types can be marshalled now
assertTrue(serCtx.canMarshall(User.class));
assertTrue(serCtx.canMarshall(Note.class));
assertTrue(serCtx.canMarshall(User.Gender.class));

// display the schema file
System.out.println(generatedSchema);
```

The following is the **`.proto`** file that is generated by the **`ProtoSchemaBuilderDemo.java`** example.

Example 7.11. Sample_Schema.Proto

```
package sample_package;

 /* @Indexed */
message Note {

   /* @IndexedField(index = true, store = false) */
   optional string text = 1;

   /* @IndexedField */
   optional ApplicationUser author = 2;
}

message ApplicationUser {

   enum Gender {
      M = 1;
      F = 2;
   }

   required string name = 1;
   optional Gender gender = 2;
}
```

The following table lists the supported Java annotations with its application and parameters.

Table 7.2. Java Annotations

Annotation	Applies To	Purpose	Requirement	Parameters

Annotation	Applies To	Purpose	Requirement	Parameters
`@ProtoDoc`	Class/Field/Enum /Enum member	Specifies the documentation comment that will be attached to the generated Protobuf schema element (message type, field definition, enum type, enum value definition)	Optional	A single String parameter, the documentation text
`@ProtoMessage`	Class	Specifies the name of the generated message type. If missing, the class name if used instead	Optional	name (String), the name of the generated message type; if missing the Java class name is used by default

Annotation	Applies To	Purpose	Requirement	Parameters
`@ProtoField`	Field, Getter or Setter	Specifies the Protobuf field number and its Protobuf type. Also indicates if the field is repeated, optional or required and its (optional) default value. If the Java field type is an interface or an abstract class, its actual type must be indicated. If the field is repeatable and the declared collection type is abstract then the actual collection implementation type must be specified. If this annotation is missing, the field is ignored for marshalling (it is transient). A class must have at least one **`@ProtoField`** annotated field to be considered Protobuf marshallable.	Required	number (int, mandatory), the Protobuf number type (org.infinispan.protostream.descriptors.Type, optional), the Protobuf type, it can usually be inferred required (boolean, optional)name (String, optional), the Protobuf namejavaType (Class, optional), the actual type, only needed if declared type is abstract collectionImplementation (Class, optional), the actual collection type, only needed if declared type is abstract defaultValue (String, optional), the string must have the proper format according to the Java field type
`@ProtoEnum`	Enum	Specifies the name of the generated enum type. If missing, the Java enum name if used instead	Optional	name (String), the name of the generated enum type; if missing the Java enum name is used by default
`@ProtoEnumValue`	Enum member	Specifies the numeric value of the corresponding Protobuf enum value	Required	number (int, mandatory), the Protobuf number name (String), the Protobuf name; if missing the name of the Java member is used

NOTE

The **`@ProtoDoc`** annotation can be used to provide documentation comments in the generated schema and also allows to inject the **`@Indexed`** and **`@IndexedField`** annotations where needed (see Section 7.3.6, “Custom Fields Indexing with Protobuf”).

Report a bug

CHAPTER 8. MONITORING

Infinispan Query provides access to statistics and operations related to indexing. The statistics provide information about classes being indexed and entities stored in the index. Lucene query and object loading times can also be determined by specifying the `generate_statistics` property in the configuration.

Report a bug

8.1. ABOUT JAVA MANAGEMENT EXTENSIONS (JMX)

Java Management Extension (JMX) is a Java based technology that provides tools to manage and monitor applications, devices, system objects, and service oriented networks. Each of these objects is managed, and monitored by `MBeans`.

JMX is the de facto standard for middleware management and administration. As a result, JMX is used in Red Hat JBoss Data Grid to expose management and statistical information.

Report a bug

8.1.1. Using JMX with Red Hat JBoss Data Grid

Management in Red Hat JBoss Data Grid instances aims to expose as much relevant statistical information as possible. This information allows administrators to view the state of each instance. While a single installation can comprise of tens or hundreds of such instances, it is essential to expose and present the statistical information for each of them in a clear and concise manner.

In JBoss Data Grid, JMX is used in conjunction with JBoss Operations Network (JON) to expose this information and present it in an orderly and relevant manner to the administrator.

Report a bug

8.2. STATISTICSINFOMBEAN

The `StatisticsInfoMBean` MBean accesses the `Statistics` object as described in the previous section.

Report a bug

APPENDIX A. REVISION HISTORY

Revision 6.4.0-9 **Wed Apr 1 2015** **Christian Huffman**
Updated version for release.

Revision 6.4.0-8 **Tue Jan 27 2015** **Misha Husnain Ali**
Updated for release.

Revision 6.4.0-7 **Mon Jan 19 2015** **Rakesh Ghatvisave**
BZ-1168271: Added dependency information.
BZ-1182112: Added a new topic - Java annotations.
BZ-1175048: More detail in Remote Querying.
BZ-1175045: Example provided for Protobuf Encoded Entities.

Revision 6.4.0-6 **Tue Jan 06 2015** **Rakesh Ghatvisave**
BZ-1175042: Modified content as per Gustavo's feedback.
BZ-1172863: Added support caveat.
BZ-1175043: Added admonition.

Revision 6.4.0-5 **Wed Dec 24 2014** **Rakesh Ghatvisave**
BZ-1175042: Modified Infinispan DSL introduction.
BZ-1175044: Modified comparison table.

Revision 6.4.0-4 **Tue Nov 18 2014** **Misha Husnain Ali**
Final version for Beta.

Revision 6.4.0-3 **Thu Nov 13 2014** **Gemma Sheldon**
Changed admonition in 2.3.3 and 2.5.3 to indicate in 6.3.1 onwards default.exclusive_index_use should be true.

Revision 6.4.0-2 **Mon Oct 27 2014** **Rakesh Ghatvisave**
BZ-1110067: Modified sentence structure.

Revision 6.4.0-1 **Thu Oct 16 2014** **Rakesh Ghatvisave**
BZ-1110067: Updated remote querying content.

Revision 6.4.0-0 **Wed Sep 24 2014** **Misha Husnain Ali**
First draft for new release.