Hibernate Search

Author: Francesco Areoluci

Academic Year: 2020/2021

Contents

1. Introduction

Hibernate Search is an enabling technology for indexing and full-text search on Hibernate applications. In particular, it allows to extract data from hibernate ORM entities and push it to local or remote indexes. Hibernate Search is particularly useful for applications where SQL-based searches are not suited, such as full-text and geolocation searches. The main difference with traditional search is that the stored text is not considered as a single block of text, but as a collection of tokens (words).
In order to do that, Hibernate Search consists of an **indexing component**, which associates indexes to entities, and an **index search component**, which allows to search through indexes. These services are offered by **Apache Lucene**, an high-performance, full-featured text search library written in Java.
Indexes are created and updated each time an entity is inserted, updated or removed from the database. Once the index is created, entity search can be accomplished without dealing with the underlying Lucene infrastructure.
Moreover, remote indexing and searching can be accomplished using **Elasticsearch**, a distributed search engine. These search engines are based on the concept of *inverted indexes*: a dictionary where the key is a token found in a document and the value is the list of identifiers of every document containing the token. A search involves the following steps:

- Extract tokens from the input query;
- Lookup tokens in the index to find matching documents;
- Aggregrate results of the lookup to produce a list of matching documents

In particular, Hibernate Search offers the following features:

- Declarative mapping of entity properties to index fields, either through annotations or a programmatic API.
- On-demand mass indexing of all entities in the database, to initialize the indexes with pre-existing data.
- On-the-fly automatic indexing of entities modified through a Hibernate ORM session, to always keep the indexes up-to-date.
- A Search DSL to easily build full-text search queries and retrieve the hits as Hibernate ORM entities.

2. Hibernate Search Architecture

This chapter offers a brief view of the Hibernate Search architectures and modules. In particular, Hibernate Search is composed by the following modules and functionalities:

- **Backend**: The backend module abstract the full-text search engine, by implementing indexing and searching interfaces. Hibernate Search provides two abstraction: **Lucene backend** and **Elastisearch backend**. The backend can be configured using Hibernate configuration.
- **Mapper**: This module allows to map the user model to an index model. In particular, Hibernate Search mapper offers the indexing of Hibernate ORM entities. This can be done through annotations or programmatic API.
- **Mass indexer**: Thanks to the mass indexing, Hibernate Search can rebuild indexes starting from pre-exisiting data stored on database.
- **Automatic indexer**: This allows to keeps indexes in sync with a database. Each time an entity is added, updated or removed, the mapper detects the changed and the index is updated.
- **Searching**: This is how Hibernate Search provides ways to query the index. The search involves the creation of a query, the token extraction and filtering, the search in the index and the entities retrieval.

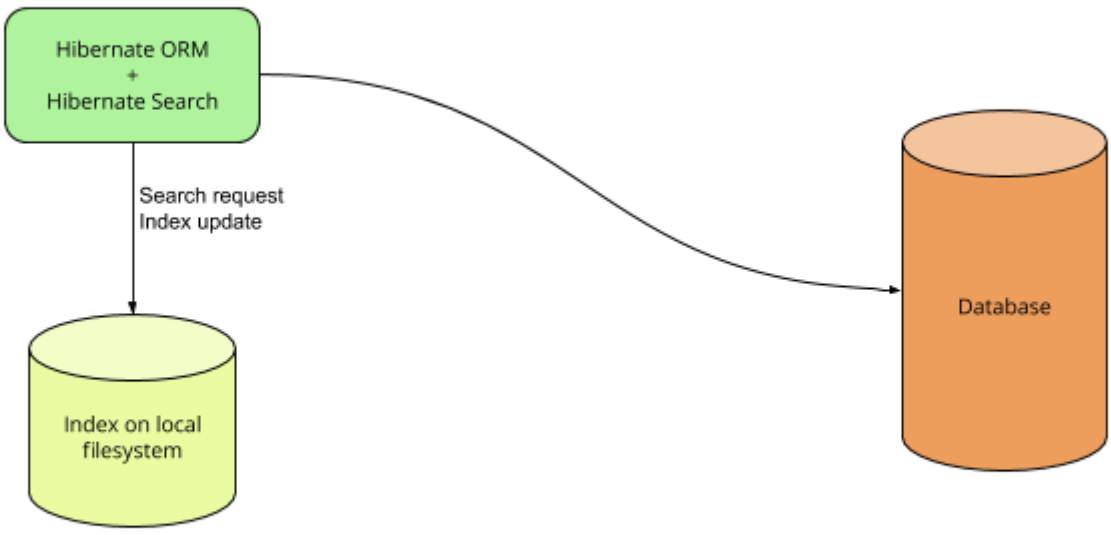

Figure 1: Application with Lucene backend

3. Hibernate Search configuration

In this chapter basic informations about Hibernate Search compatibility and configuration will be shown.

3.1 Compatibility

Hibernate Search v6 is compatible with the following technologies:

Technology	**Version**
Java Runtime	Java 8 or greater
Hibernate ORM (for the ORM mapper)	Hibernate ORM 5.4.30.Final
JPA (for the ORM mapper)	JPA 2.2
Apache Lucene (for the Lucene backend)	Lucene 8.7.0
Elasticsearch server (for the Elasticsearch backend)	Elasticsearch 5.6, 6.8 or 7.10

3.2 Dependencies

Hibernate Search artifacts can be found in Maven's Central Repository. In order to integrate Hibernate Search and Lucene Backend in an application, include the following dependencies:

```
<dependency>
   <groupId>org.hibernate.search</groupId>
   <artifactId>hibernate-search-mapper-orm</artifactId>
   <version>6.0.3.Final</version>
</dependency>
<dependency>
   <groupId>org.hibernate.search</groupId>
   <artifactId>hibernate-search-backend-lucene</artifactId>
   <version>6.0.3.Final</version>
</dependency>
```

Remote indexing and search can be obtained using Elasticsearch. In order to do that the following dependencies must be included:

```
<dependency>
   <groupId>org.hibernate.search</groupId>
   <artifactId>hibernate-search-mapper-orm</artifactId>
   <version>6.0.3.Final</version>
</dependency>
<dependency>
   <groupId>org.hibernate.search</groupId>
   <artifactId>hibernate-search-backend-elasticsearch</artifactId>
   <version>6.0.3.Final</version>
</dependency>
```

3.3 Configuration

The configuration properties of Hibernate Search are sourced from Hibernate ORM, so they can be added to any file from which Hibernate ORM takes its configuration:

- A **hibernate.properties** file in your classpath.
- The **hibernate.cfg.xml** file in your classpath, if using Hibernate ORM native bootstrapping.
- The **persistence.xml** file in your classpath, if using Hibernate ORM JPA bootstrapping.

The default parameters, such as filesystem index location and Elasticsearch remote host, can be modified by changing these files (for further informations see chapters **8** and **9**).

4. Entity mapping

Hibernate search mapping, and in particular ORM entities mapping to indexes, can be done via **annotations** and via **programmatic API**.

4.1 Indexing through annotations

As an example, lets consider the following *Book* entity:

```
import java.util.HashSet;
import java.util.Set;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.ManyToMany;

@Entity
public class Book {

    @Id
    @GeneratedValue
    private Integer id;

    private String title;
    private String isbn;
    private int pageCount;

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

    public Book() {
    }

    // ...

}
```

This entity has a many to many relationship with an *Author* entity, omitted for brevity.
Using the **annotation** approach, this entity can be indexed by adding the **@Indexed** annotation and other annotations to specify fields role in indexing. Thus, the resulting *Book* entity is the following:

```
@Entity
@Indexed
public class Book {

    @Id
    @GeneratedValue
    private Integer id;

    @FullTextField
    private String title;

    @KeywordField
    private String isbn;

    @GenericField
    private int pageCount;

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

    public Book() {
    }

    // ...

}
```

The following annotations have been added:

- **@Indexed**: marks the entity as indexed. Its index will be kept up to date;
- **@FullTextField**: maps a property to a full-text index field with the same name and type;
- **@KeywordField**: maps a property to a non-analyzed index field. Useful for identifiers;
- **@GenericField**: maps a property to an index for non-String search;
- **@IndexedEmbedded**: embeds the indexed form of associated objects into the indexed form of the embedding entity. In this case, if the *Author* entity defines an indexed field, the *authors* field will be populated automatically based on the content of the *authors* property. The books will be reindexed automatically whenever the *authors'* indexed field will change.

4.2 Indexing through programmatic API

Programmatic API allows to define the entity mapping through code that will be executed at application startup. This approach can be used instead of annotations in order to address specific needs, such as:

- Mapping a single entity differently for different deployments;
- Mapping multiple entity similarly without code duplication.

Programmatic mapping requires two steps:

- Define a class that implements the *HibernateOrmSearchMappingConfigurer* interface;
- Configure Hibernate Search to use the previously defined mapper.

The following is an example of class that implements the mapping configurer interface:

```
public class MySearchMappingConfigurer implements
HibernateOrmSearchMappingConfigurer {
    @Override
    public void configure(HibernateOrmMappingConfigurationContext context)
{
        ProgrammaticMappingConfigurationContext mapping =
context.programmaticMapping();
        TypeMappingStep bookMapping = mapping.type(Book.class);
        bookMapping.indexed();
        bookMapping.property("title")
                .fullTextField().analyzer("english");
    }
}
```

Here the programmatic mapping is accessed, the *Book* entity is defined as indexed and its *title* field is defined as an index field.
The mapper can be configured by setting the Hibernate parameter **hibernate.search.mapping.configurer** to a bean reference pointing to the defined implementation. Moreover, annotation mapping can be disabled by setting **hibernate.search.mapping.process_annotations** to *false*.

5. Application initialization

At the application startup the following initialization steps have to be accomplished:

- Indexes and schema creation
- Indexing of pre-existing data

Hibernate will take care of schema creation, through Lucene or REST API calls to Elasticsearch.
The initial indexing must be done in order to index data already present on the database which is not yet indexed. This can be done as:

```
SearchSession searchSession = Search.session(entityManager);

MassIndexer indexer = searchSession.massIndexer(Book.class)
        .threadsToLoadObjects(7);

indexer.startAndWait();
```

Here, the Hibernate Search session is created from the entity manager and the indexer is created to index the requested entity using the requested number of threads. Once the session is created, Hibernate Search will transparently index every entity persisted, updated or removed through Hibernate ORM.

6. Searching

Search can be accomplished on indexed data by preparing a query, fetching data and filtering results. The following code documents a search on *Book* entitites:

```
SearchSession searchSession = Search.session(entityManager);

SearchResult<Book> result = searchSession.search(Book.class)
        .where(f -> f.match()
                .fields("title", "authors.name")
                .matching("refactoring"))
        .fetch(20);

long totalHitCount = result.total().hitCount();
List<Book> hits = result.hits();

List<Book> hits2 =
        /* ... same DSL calls as above... */
        .fetchHits(20);
```

Here, once the search session is created, the query is initiated on *Book* entities. The lambda functions is used to create a factory *f* by specifying that only the documents that match the predicate should be returned. The *fetch()* command will build the query and fetch the result, limiting them to the top 20 results (hits).
The total number of result can be retrieved with the *hitCount()* method, while the matching entities can be retrived with the *hits()* method. If you are interested only in the hits, the *fetchHits()* method can be used. Moreover, if you wants only the total hit count, the *fetchTotalHitCount()* method can be used.

7. Analysis

Text analysis is done in search step by processing input text. An *analyzer* is made up of three components:

- **character filters** (zero or more): they allow to clean up the input text, e.g. by removing HTML tags.
- **tokenizer**: it splits the input text into a list of words, called *tokens*
- **token filters** (zero or more): allow to normalize and remove useless tokens.

In particular the default built-in analyzer will:

- split the input according to the Unicode Word Break rules;
- normalize tokens by turning uppercase letters to lowercase

Custom analyzer can be created by implementing backend-specific interfaces: *LuceneAnalysisConfigurer* or *ElasticsearchAnalysisConfigurer*. Once this is done, the backend configuration must be altered to use the newly created analyzer. Then the entity annotations should be modified in order to use the custom analyzer.

For example the following code will create a Lucene analyzer that will:

- tokenize the input

- filter the uppercase letters to lowercase
- apply a language specific stemming through a *snowball filter*
- apply *ASCII-folding*

```
public class MyLuceneAnalysisConfigurer implements
LuceneAnalysisConfigurer {
    @Override
    public void configure(LuceneAnalysisConfigurationContext context)
{
        context.analyzer("english").custom()
                .tokenizer(StandardTokenizerFactory.class)
                .tokenFilter(LowerCaseFilterFactory.class)
                .tokenFilter(SnowballPorterFilterFactory.class)
                        .param("language", "English")
                .tokenFilter(ASCIIFoldingFilterFactory.class);

        context.analyzer("name").custom()
                .tokenizer(StandardTokenizerFactory.class)
                .tokenFilter(LowerCaseFilterFactory.class)
                .tokenFilter(ASCIIFoldingFilterFactory.class);
    }
}
```

The analyzer can be applied to an entity by modifying their annotations, such as:

```
@FullTextField(analyzer = "english")
private String title;
```

Then, a query that will search for *Book* entities will use the custom analyzer when matching a possibile word for the *title* field. In this case, searching for "*refactored*" will return all entities which have "*refact**" in their title (thanks to the snowball filter).

8. More on Lucene

Hibernate Search backend for local indexing and searching is based on Lucene, a high-performance full-text search engine library. It is based on the following concepts:

- **Index**: this is used to perform fast search over the requested query. As previously said, Lucene uses an *inverted-index*: each token in a document is used as a key for the index and the document identifiers (documents that contain the token) are used as value. This type of index inverts a *page-centric* data structure to a *keyword-centric* data structure.
- **Documents**: the index is composed of one or more documents. A document is the basic unit of search and index in Lucene. Indexing involves adding a Document to an **IndexWriter** and searching involves retrieving a Document using an **IndexSearcher**.

- **Queries**: queries can be used to perform searching over an index. Using Hibernate Search, the query creation and handling through the **IndexSearcher** is abstracted. A search return a list of *Hits* (documents retrieved).

8.1 Hibernate Search Lucene backend configuration

The basic Hibernate Search Lucene configuration might not suit everyone. The next sections show some configuration options that can be used in order to configure your Hibernate Search instance.

8.1.1 Index storage

Lucene can be configured to store the index on the filesystem (default) or on the JVM's heap. The following fields have to be modified in order to change the type of storage:

```
# To configure the defaults for all indexes:
hibernate.search.backend.directory.type = <local-filesystem | local-heap>
# To configure a specific index:
hibernate.search.backend.indexes.<index name>.directory.type = <local-
filesystem | local-heap>
```

If the JVM's heap storage is used, all the indexes will be lost when the JVM shuts down. This configuration is used primarly for testing purpose on small indexes. If the local filesystem configuration is used, more configuration options are available. For example, in order to change the directory used to store the index, the following field have to be modified:

```
# To configure the defaults for all indexes:
hibernate.search.backend.directory.root = /path/to/my/root
# To configure a specific index:
hibernate.search.backend.indexes.<index name>.directory.root =
/path/to/my/root
```

8.1.2 Sharding

Sharding can be used to improve search performance when dealing with large amounts of data. It consists in splitting the index in smaller parts, called *shards*. This feature can be helpful at indexing time by spreading the stress on multiple disks (Lucene) or hosts (Elasticsearch). Moreover, it can be helpful at search time: if a certain property is often used to select documents, a routing policy can be used in order to manually route the query to a certain shard and avoid the search on the other ones.
By default sharding is disabled. It can be enabled by selecting a sharding strategy, for example:

```
# To configure the defaults for all indexes:
hibernate.search.backend.sharding.strategy = hash
hibernate.search.backend.sharding.number_of_shards = 2
```

will set up two shards. At routing time, the routing key will be hashed to assign a shard (if key is null, the document identifier will be used). This is useful when there is no explicit routing key configured in the mapping.
Explicit routing can be configured in the following way:

```
# To configure the defaults for all indexes:
hibernate.search.backend.sharding.strategy = explicit
hibernate.search.backend.sharding.shard_identifiers = fr,en,de
```

This will set up one shard for each identifier. At routing time, the key will be validated to make sure it matches the configured identifiers. If it does not, an exception will be thrown.

8.1.3 Threading

Lucene uses an internal thread pool in order to execute write operations on the index. The number of threads of the pool can be modified by changing the following field:

```
hibernate.search.backend.thread_pool.size = 4
```

8.1.4 Commit interval

Lucene's *commit* consists in pushing the changes buffered in an *IndexWriter* to the index. This is used to avoid data loss in case of system or application crash. Some operations require an immediate commit, and others not. By default, if an operation does not require an immediate commit, the commit is delayed by one second. This is particularly useful when more operations are executed in that second: the operations will be included in the same commit and the amount of commits can be drastically reduced.
The commit interval can be configured by changing the following field:

```
# To configure the defaults for all indexes:
hibernate.search.backend.io.commit_interval = 1000    # milliseconds
```

8.1.5 Refresh interval

Lucene's *refresh* consists in opening a new index reader. This is used in order to include in the search operations the latest changes of the index. Refresh is an expensive operation and by default it is done upon every search query, but only if a write has occured since the last refresh. In write-intensive contests, the throughput can be increased by manually setting a refresh interval, thus the index will be refreshed every X milliseconds (if higher than 0):

```
# To configure the defaults for all indexes:
hibernate.search.backend.io.refresh_interval = 1000    # milliseconds
```

9. More on Elasticsearch

Elasticsearch is a search engine for all types of data built on top of Lucene. As Lucene, it uses an *index* (in the form of inverted index) built of *Documents*, but these are stored as JSON documents. Elasticsearch offers REST APIs for indexing and searching operations, which are abstracted by Hibernate Search.
The main feature of Elasticsearch is its **distributed** nature: documents are distributed across different containers known as *shard*. These shard can be replicated to offer redunant copies in case of system failures and it can scale out to a multidude of servers.

9.1 Hibernate Search Elasticsearch backend

The basic Hibernate Search Elasticsearch configuration might not suit everyone. The next sections show some configuration options that can be used in order to configure your Hibernate Search instance.

9.1.1 Client configuration

As said, Elasticsearch backend communicates with an Elasticsearch cluster through REST APIs. The address of these requests can be modified by changing the following field:

```
hibernate.search.backend.hosts = localhost:9200
```

Multiple hosts can be configured by specifying a list of host:port separated by commas. Moreover, the communication protocol can be set as HTTP or HTTPS by changing the following field:

```
hibernate.search.backend.protocol = http
```

If you are using a non-standard prefix path for the Elasticsearch cluster (e.g. if a proxy is used), the default path prefix (/) can be modified by changing:

```
hibernate.search.backend.path_prefix = my/path
```

If an HTTP authentication on the cluster has been set up, credentials can be specified in the following fields:

```
hibernate.search.backend.username = ironman
hibernate.search.backend.password = j@rv1s
```

9.1.2 Connection tuning

Connection timeouts can be set up as well:

```
hibernate.search.backend.request_timeout = 30000    # Timeout when
executing a request
hibernate.search.backend.connection_timeout = 1000  # Timeout when
estabilishing a connection
hibernate.search.backend.read_timeout = 30000       # Timeout in response
reading
```

The number of simultaneous connections can be set up by modifying the following:

```
hibernate.search.backend.max_connections = 20            # Maximum number
of connections for all the hosts of a cluster
hibernate.search.backend.max_connections_per_route = 10  # Maximum number
of connections for each host in a cluster
```

9.1.3 Threading

As in Lucene, the Elasticsearch backend relies on a thread pool. The number of threads can be modified by changing the following field:

```
hibernate.search.backend.thread_pool.size = 4
```

9.1.4 Sharding

Sharding if by default disabled in Elasticsearch clusters. In order to enable it the property *index.number_of_shard* must be set up in the cluster.

9.1.5 Refresh interval

Elasticsearch relies on periodically refreshed index reader. By default it is refreshed every second, but this behaviour is customizable in the cluster by setting the *index.refresh_interval* field.

10. References

- Hibernate Search Web page - http://hibernate.org/search/
- Hibernate Search documentation - https://docs.jboss.org/hibernate/search/6.0/reference/en-US/html_single/#preface
- Apache Lucene - https://lucene.apache.org/
- Elasticsearch - https://www.elastic.co/what-is/elasticsearch