Hibernate Search

Full-text search for entities

Hibernate Search

Hibernate Search automatically extracts data from Hibernate ORM entities to push it to local Apache Lucene indexes or remote Elasticsearch/OpenSearch indexes.

- **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.
- And much more!

Full-text search

Full-text search is a set of techniques for searching, **in a corpus of text documents**, the documents that best match a given query.

The main difference with traditional search — for example in an SQL database — is that the stored **text is not considered as a single block of text**, but as a collection of tokens (words).

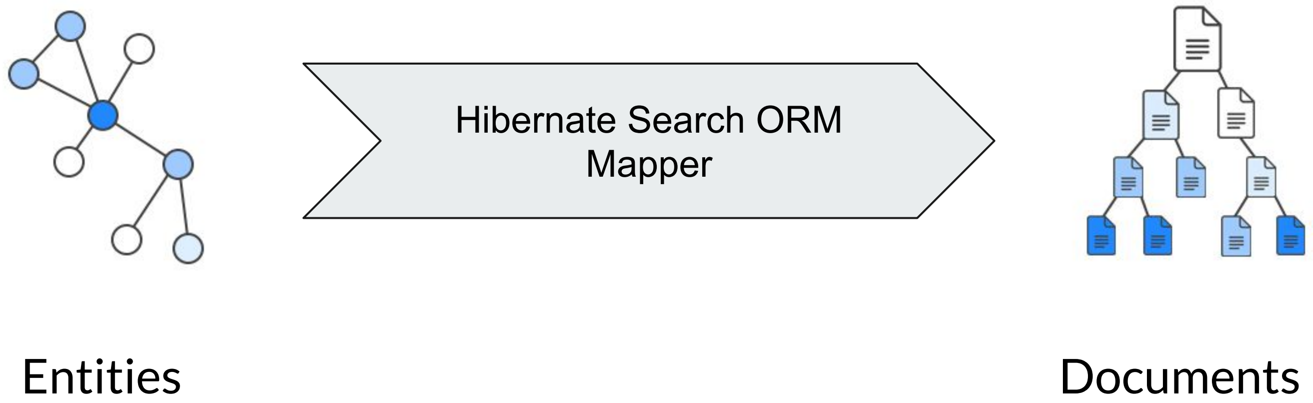
To simplify, these search engines are based on the concept of **inverted indexes**: a dictionary where the key is a token (word) found in a document, and the value is the list of identifiers of every document containing this token.

Full-text search

Still simplifying, once all documents are indexed, searching for documents involves three steps:

1. extracting tokens (words) from the query;
2. looking up these tokens in the index to find matching documents;
3. aggregating the results of the lookups to produce a list of matching documents.

Mapping



Mapping

```
TypeMappingStep bookMapping = mapping.type( Book.class );
bookMapping.indexed();
TypeMappingStep authorMapping = mapping.type( Author.class );
authorMapping.indexed().index( "AuthorIndex" );
TypeMappingStep userMapping = mapping.type( User.class );
userMapping.indexed().backend( "backend2" );
```

```java
@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<>();

}
```

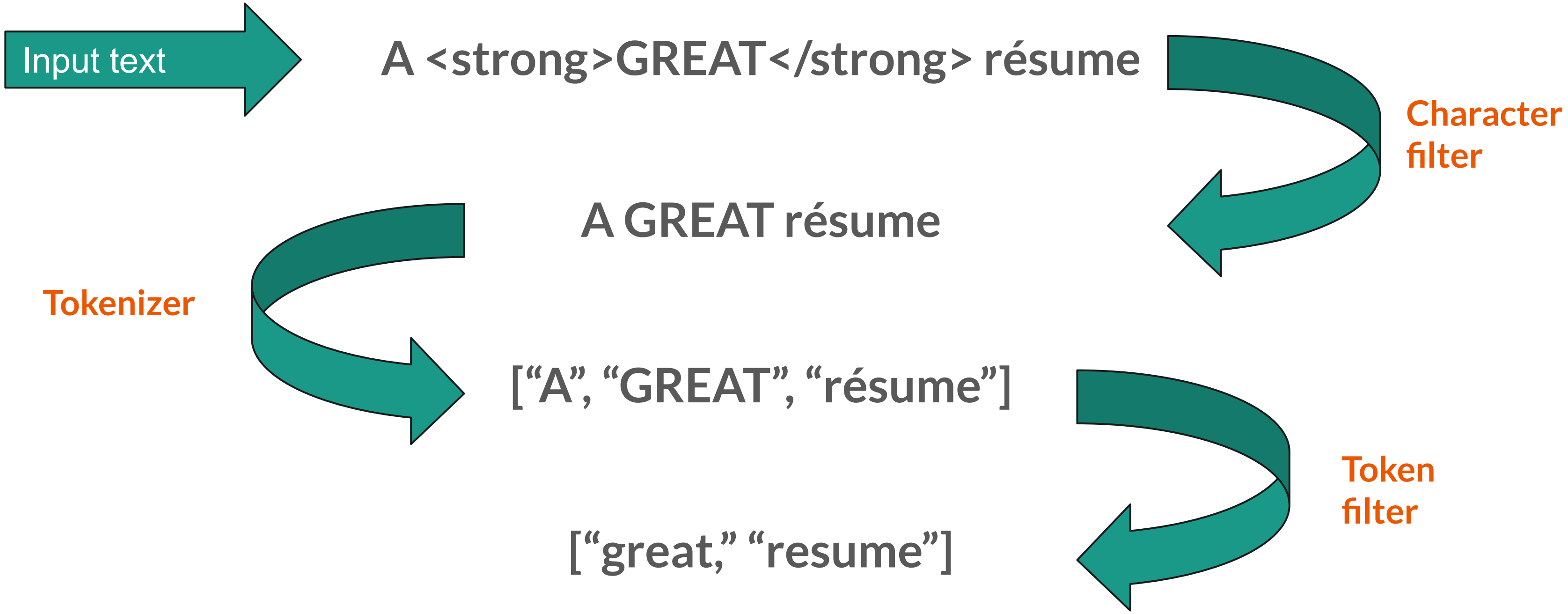
Analysis

As mentioned in Full-text search, the full-text engine works on tokens, which means text has to be processed both when indexing (document processing, to build the token → document index) and when searching (query processing, to generate a list of tokens to look up).

The analyzer is made up of three types of components, which will each process the text successively in the following order:

1. Character filter: transforms the input characters. Replaces, adds or removes characters.
2. Tokenizer: splits the text into several words, called "tokens".
3. Token filter: transforms the tokens.

Analysis


Input text
A <strong>GREAT</strong> résume
Character filter
A GREAT résume
Tokenizer
[“A”, “GREAT”, “résume”]
Token filter
[“great,” “resume”]

Backend

The backend is the abstraction over the full-text engines, where "things get done". It implements generic indexing and searching interfaces for use by the mapper through "index managers", each providing access to one index.

elasticsearch

APACHE LUCENE™

Automatic indexing

By default, every time an entity is changed through a Hibernate ORM Session, if that entity is mapped to an index, Hibernate Search updates the relevant index.

Automatic indexing only considers changes applied directly to entity instances in Hibernate ORM sessions.

Reindexing large volumes of data with the `MassIndexer`

There are cases where automatic indexing is not enough:

- when restoring a database backup;
- when indexes had to be wiped, for example because the Hibernate Search mapping or some core settings changed;
- when automatic indexing had to be disabled for performance reasons

Reindexing large volumes of data with the `MassIndexer`

The MassIndexer takes the following approach to provide a reasonably high throughput:

- Indexes are purged completely when mass indexing starts.
- Mass indexing is performed by several parallel threads.

Searching

Beyond simply indexing, Hibernate Search also exposes high-level APIs to search these indexes without having to resort to native APIs.

One key feature of these search APIs is the ability to use indexes to perform the search, but to return entities loaded **from the database**, effectively offering a new type of query for Hibernate ORM entities.

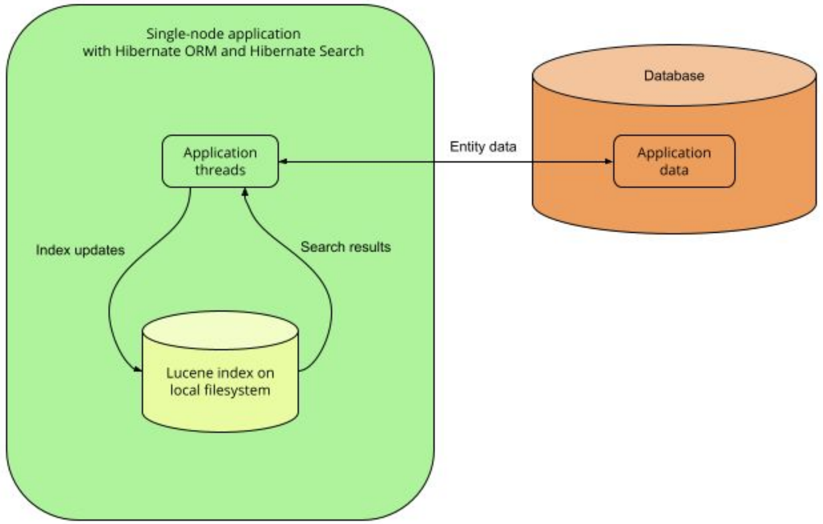
Searching

```
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();
```

Architecture	**Single-node with Lucene**	**No coordination with Elasticsearch**	**Outbox polling with Elasticsearch**
Application topology	Single-node	Single-node or multi-node	
Extra bits to maintain	Indexes on filesystem	Elasticsearch cluster	
Guarantee of index updates	When the commit returns (non-transactional)		On commit (transactional)
Visibility of index updates	Configurable: immediate or eventual	Configurable: immediate (poor performance) or eventual	Eventual
Native features	Mostly for experts	For anyone	
Overhead for application threads	Low to medium		Very low
Overhead for the database	Low		Low to medium
Impact on database schema	None		Extra tables
Limitations	Automatic indexing ignores: JPQL/SQL queries, asymmetric association updates		
	Out-of-sync indexes in rare situations: concurrent `@IndexedEmbedded`, backend I/O errors		No other known limitation

Single-node application with the Lucene backend

With the Lucene backend, indexes are local to a given application node (JVM). They are accessed through direct calls to the Lucene library, without going through the network.



Single-node application with the Lucene backend

Pros:

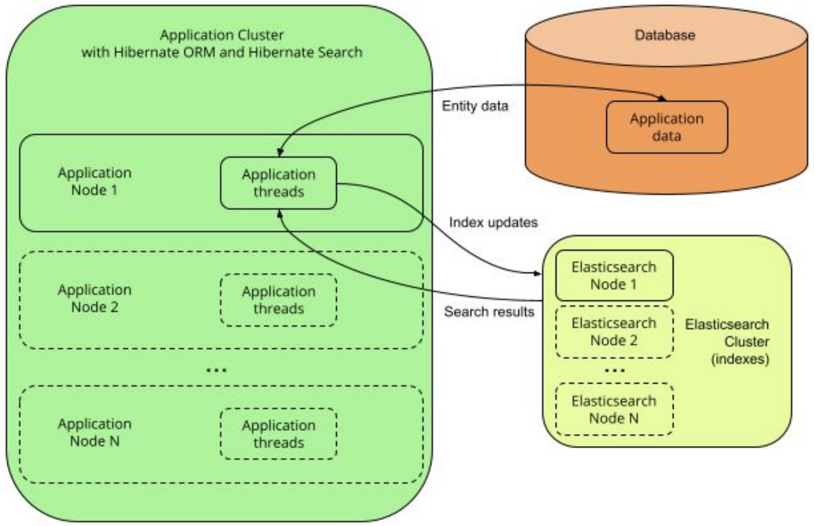
- Simplicity: no external services are required, everything lives on the same server.
- Immediate visibility (~milliseconds) of index updates.

Cons:

- Backend errors during automatic indexing may lead to out-of sync indexes.
- In rare cases, automatic indexing involving `@IndexedEmbedded` may lead to out-of sync indexes.
- Not so easy to extend.
- **No horizontal scalability.**

Application with the Elasticsearch backend

With the Elasticsearch backend, indexes are not tied to the application node. They are managed by a separate cluster of Elasticsearch nodes, and accessed through calls to REST APIs.



Application with the Elasticsearch backend

Pros:

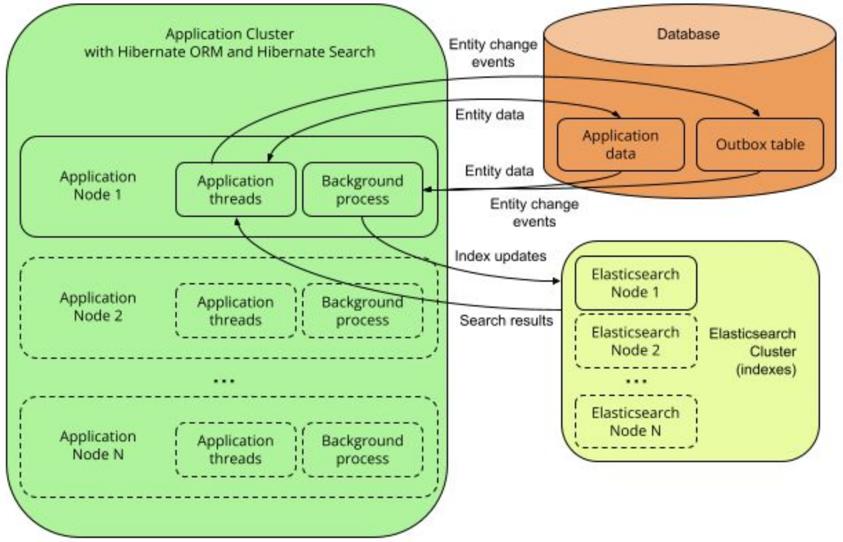
- Easy to extend.
- Horizontal scalability of the indexes.
- Horizontal scalability of the application.

Cons:

- Backend errors during automatic indexing may lead to out-of sync indexes.
- In rare cases, automatic indexing involving `@IndexedEmbedded` may lead to out-of sync indexes.
- Need to manage an additional service.
- Delayed visibility (~1 second) of index updates (near-real-time).

Multi-node application with outbox polling and Elasticsearch backend

With Hibernate Search’s outbox-polling coordination strategy, entity change events are not processed immediately in the ORM session where they arise, but are pushed to an outbox table in the database.



Multi-node application with outbox polling and Elasticsearch backend

Pros:

- **Safest.**
- Easy to extend.
- Horizontal scalability of the indexes.
- Horizontal scalability of the application.

Cons:

- Need to manage an additional service.
- Delayed visibility (~1 second) of index updates (near-real-time).
- Impact on the database schema.
- Overhead for the database.

Compatibility

	Version
Java Runtime	8, 11, 17
Hibernate	5.6.11.Final
Apache Lucene (for Lucene backend)	8.11.1
Elasticsearch (for Elasticsearch backend)	5.6, 6.8, 7.10, 7.16

Framework support

springboot

QUARKUS