# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preface</td>
<td>1</td>
</tr>
<tr>
<td>1. Getting started</td>
<td>2</td>
</tr>
<tr>
<td>1.1. Compatibility</td>
<td>2</td>
</tr>
<tr>
<td>1.2. Migration notes</td>
<td>2</td>
</tr>
<tr>
<td>1.3. Dependencies</td>
<td>3</td>
</tr>
<tr>
<td>1.4. Configuration</td>
<td>4</td>
</tr>
<tr>
<td>1.5. Mapping</td>
<td>5</td>
</tr>
<tr>
<td>1.6. Indexing</td>
<td>9</td>
</tr>
<tr>
<td>1.7. Searching</td>
<td>10</td>
</tr>
<tr>
<td>1.8. Analysis</td>
<td>13</td>
</tr>
<tr>
<td>1.9. What’s next</td>
<td>17</td>
</tr>
<tr>
<td>2. Concepts</td>
<td>18</td>
</tr>
<tr>
<td>2.1. Full-text search</td>
<td>18</td>
</tr>
<tr>
<td>2.2. Mapping</td>
<td>18</td>
</tr>
<tr>
<td>2.3. Analysis</td>
<td>18</td>
</tr>
<tr>
<td>3. Architecture</td>
<td>19</td>
</tr>
<tr>
<td>4. Configuration</td>
<td>20</td>
</tr>
<tr>
<td>4.1. Configuration sources</td>
<td>20</td>
</tr>
<tr>
<td>4.2. Structure of configuration properties</td>
<td>20</td>
</tr>
<tr>
<td>4.3. Type of configuration properties</td>
<td>21</td>
</tr>
<tr>
<td>4.4. Configuration property tracking</td>
<td>22</td>
</tr>
<tr>
<td>4.5. Bean resolution</td>
<td>22</td>
</tr>
<tr>
<td>5. Hibernate ORM integration</td>
<td>24</td>
</tr>
<tr>
<td>5.1. General configuration</td>
<td>24</td>
</tr>
<tr>
<td>5.2. Mapping ORM entities to indexes</td>
<td>24</td>
</tr>
<tr>
<td>5.3. Indexing</td>
<td>36</td>
</tr>
<tr>
<td>5.4. Search query</td>
<td>40</td>
</tr>
<tr>
<td>6. Search DSLs</td>
<td>43</td>
</tr>
<tr>
<td>6.1. Predicate DSL</td>
<td>43</td>
</tr>
<tr>
<td>6.2. Sort DSL</td>
<td>43</td>
</tr>
<tr>
<td>6.3. Projection DSL</td>
<td>43</td>
</tr>
<tr>
<td>6.4. Type of arguments passed to the DSL</td>
<td>43</td>
</tr>
<tr>
<td>6.5. Type of projected values</td>
<td>45</td>
</tr>
<tr>
<td>6.6. Targeting multiple fields</td>
<td>46</td>
</tr>
<tr>
<td>7. Lucene backend</td>
<td>49</td>
</tr>
<tr>
<td>7.1. General configuration</td>
<td>49</td>
</tr>
<tr>
<td>7.2. Field types</td>
<td>50</td>
</tr>
<tr>
<td>7.3. Analysis</td>
<td>51</td>
</tr>
<tr>
<td>7.4. Multi-tenancy</td>
<td>52</td>
</tr>
</tbody>
</table>
Preface

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.
Chapter 1. Getting started

This section will guide you through the initial steps required to integrate Hibernate Search into your application.

Hibernate Search 6.0.0.Alpha6 is a **technology preview** and is **not ready for production**.

Use it to have a sneak peak at the APIs, make suggestions or warn us of what you consider blocking early so we can fix it, but **do not** use it to address business needs!

Read the dedicated page on our website for more detailed and up-to-date information.

1.1. Compatibility

<table>
<thead>
<tr>
<th>Table 1. Compatibility</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Java Runtime</strong></td>
</tr>
<tr>
<td><strong>Hibernate ORM (for the ORM mapper)</strong></td>
</tr>
<tr>
<td><strong>JPA (for the ORM mapper)</strong></td>
</tr>
</tbody>
</table>

1.2. Migration notes

If you are upgrading an existing application from an earlier version of Hibernate Search to the latest release, make sure to check out the migration guide.

**To Hibernate Search 5 users**

If you pull our artifacts from a Maven repository and you come from Hibernate Search 5, be aware that just bumping the version number will not be enough.

In particular, the group IDs changed from `org.hibernate` to `org.hibernate.search`, most of the artifact IDs changed to reflect the new mapper-backend design, and the Lucene integration now requires an explicit dependency instead of being available by default. Read Dependencies for more information.

Additionally, be aware that a lot of APIs changed, some only because of a package change, others because of more fundamental changes (like moving away from using Lucene types in Hibernate Search APIs).
1.3. Dependencies

The Hibernate Search artifacts can be found in Maven's Central Repository.

If you do not want to, or cannot, fetch the JARs from a Maven repository, you can get them from the distribution bundle hosted at Sourceforge.

In order to use Hibernate Search, you will need at least two direct dependencies:

- a dependency to the "mapper", which extracts data from your domain model and maps it to indexable documents;
- and a dependency to the "backend", which allows to index and search these documents.

Below are the most common setups and matching dependencies for a quick start; read Architecture for more information.

Hibernate ORM + Lucene

Allows indexing of ORM entities in a single application node, storing the index on the local filesystem.

If you get Hibernate Search from Maven, use these dependencies:

```xml
<dependency>
  <groupId>org.hibernate.search</groupId>
  <artifactId>hibernate-search-mapper-orm</artifactId>
  <version>6.0.0.Alpha6</version>
</dependency>
<dependency>
  <groupId>org.hibernate.search</groupId>
  <artifactId>hibernate-search-backend-lucene</artifactId>
  <version>6.0.0.Alpha6</version>
</dependency>
```

If you get Hibernate Search from the distribution bundle, copy the JARs from dist/engine, dist/mapper/orm, dist/backend/lucene, and their respective lib subdirectories.

Hibernate ORM + Elasticsearch

Allows indexing of ORM entities on multiple application nodes, storing the index on a remote Elasticsearch cluster (to be configured separately).

If you get Hibernate Search from Maven, use these dependencies:
If you get Hibernate Search from the distribution bundle, copy the JARs from `dist/engine`, `dist/mapper/orm`, `dist/backend/elasticsearch`, and their respective `lib` subdirectories.

### 1.4. Configuration

Once you have added all required dependencies to your application you have to add a couple of properties to your Hibernate ORM configuration file.

In case you are a Hibernate ORM new timer we recommend you start there to implement entity persistence in your application, and only then come back here to add Hibernate Search indexing.

The properties 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 minimal working configuration is short, but depends on your setup:
Example 1. Hibernate Search properties in persistence.xml for a "Hibernate ORM + Lucene" setup

```xml
<property name="hibernate.search.backends.myBackend.type" value="lucene"/>
<property name="hibernate.search.backends.myBackend.directory_provider" value="local_directory"/>
<!--
<property name="hibernate.search.backends.myBackend.root_directory" value="some/filesystem/path"/>
--> ③
<property name="hibernate.search.default_backend" value="myBackend"/>
```
① Define a backend named "myBackend" relying on Lucene technology.
② Define the storage for that backend as a local filesystem directory.
③ The backend will store indexes in the current working directory by default. If you want to store the indexes elsewhere, uncomment this line and set the value of the property.
④ Make sure to use the backend we just defined for all indexes.

Example 2. Hibernate Search properties in persistence.xml for a "Hibernate ORM + Elasticsearch" setup

```xml
<property name="hibernate.search.backends.myBackend.type" value="elasticsearch"/>
<property name="hibernate.search.backends.myBackend.hosts" value="https://elasticsearch.mycompany.com"/>
<property name="hibernate.search.backends.myBackend.username" value="ironman"/>
<property name="hibernate.search.backends.myBackend.password" value="j@rV1s"/>
--> ②
<property name="hibernate.search.default_backend" value="myBackend"/>
```
① Define a backend named "myBackend" relying on Elasticsearch technology.
② The backend will attempt to connect to http://localhost:9200 by default. If you want to connect to another URL, uncomment these lines and set the value for the "hosts" property, and optionally the username and password.
③ Make sure to use the backend we just defined for all indexes.

1.5. Mapping

Let's assume that your application contains the Hibernate ORM managed classes Book and Author and you want to index them in order to search the books contained in your database.
To make these entities searchable, you will need to map them to an index structure. The mapping can be defined using annotations, or using a programmatic API; this getting started guide will show you a simple annotation mapping. For more details, refer to Hibernate ORM integration.
Below is an example of how the model above can be mapped.

**Example 4. Book and Author entities AFTER adding Hibernate Search specific annotations**

```java
import java.util.HashSet;
import java.util.Set;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import org.hibernate.search.mapper.pojo.mapping.definition.annotation.GenericField;
import org.hibernate.search.mapper.pojo.mapping.definition.annotation.Indexed;
import org.hibernate.search.mapper.pojo.mapping.definition.annotation.IndexedEmbedded;

@Entity
@Indexed
public class Book {
    @Id
    @GeneratedValue
    private Integer id;

    @GenericField
    private String title;

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

    public Book() {
    }

    // Getters and setters
    // ...
}
```
import java.util.HashSet;
import java.util.Set;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import org.hibernate.search.mapper.pojo.mapping.definition.annotation.GenericField;

@Entity
public class Author {

@Id
@GeneratedValue
private Integer id;

@GenericField
private String name;

@ManyToMany(mappedBy = "authors")
private Set<Book> books = new HashSet<>();

public Author() {
    // Getters and setters
    // ...
}

@Indexed marks Book as indexed, i.e. an index will be created for that entity, and that index will be kept up to date.

By default, the JPA @Id is used to generate a document identifier.

@GenericField maps a property to an index field with the same name and type. As such, the field is indexed in a way that only allows exact matches; full-text matches will be discussed in a moment.

@IndexedEmbedded allows to "embed" the indexed form of associated objects (entities or embeddables) into the indexed form of the embedding entity. Here, the Author class defines a single indexed field, name. Thus adding @IndexedEmbedded to the authors property of Book will add a single authors.name field to the Book index. This field will be populated automatically based on the content of the authors property, and the books will be reindexed automatically whenever the name property of their author changes. See Indexed-embedded for more information.

Entities that are only @IndexedEmbedded in other entities, but do not require to be searchable by themselves, do not need to be annotated with @Indexed.

This is a very simple example, but is enough to get started. Just remember that Hibernate Search allows more complex mappings:

- Other @*Field annotations exist, some of them allowing full-text search, some of them allowing finer-grained configuration for field of a certain type. You can find out more about @*Field annotations in Direct field mapping.
• Properties, or even types, can be mapped with finer-grained control using "bridges". See Bridges for more information.

1.6. Indexing

Hibernate Search will transparently index every entity persisted, updated or removed through Hibernate ORM. Thus this code would transparently populate your index:

Example 5. Using Hibernate ORM to persist data, and implicitly indexing it through Hibernate Search

```java
// Not shown: get the entity manager and open a transaction
Author author = new Author();
author.setName( "John Doe" );

Book book = new Book();
book.setTitle( "Refactoring: Improving the Design of Existing Code" );
book.getAuthors().add( author );
author.getBooks().add( book );

entityManager.persist( author );
entityManager.persist( book );
// Not shown: commit the transaction and close the entity manager
```

By default, in particular when using the Elasticsearch backend, changes will not be visible right after the transaction is committed. A slight delay (by default one second) will be necessary for Elasticsearch to process the changes.

For that reason, if you modify entities in a transaction, and then a execute search query right after that transaction, the search results may not be consistent with the changes you just performed.

See Synchronization with the indexes for more information about this behavior and how to tune it.

However, keep in mind that data already present in your database when you add the Hibernate Search integration is unknown to Hibernate Search, and thus has to be indexed through a batch process. To that end, you can use the mass indexer API, as shown in the following code:
Example 6. Using Hibernate Search MassIndexer API to manually (re)index the already persisted data

```java
SearchSession searchSession = Search.getSearchSession( entityManager ); ①
MassIndexer indexer = searchSession.createIndexer( Book.class ) ②
    .threadsToLoadObjects( 7 ); ③
indexer.startAndWait(); ④
```

① Get a Hibernate Search session, called `SearchSession`, from the `EntityManager`.

② Create an "indexer", passing the entity types you want to index. Pass no type to index all of them.

③ It is possible to set the number of threads to be used. For the complete option list see [manual-index-changes].

④ Invoke the batch indexing process.

1.7. Searching

Once the data is indexed, you can perform search queries.

The following code will prepare a search query targeting the index for the `Book` entity, filtering the results so that at least one field among `title` and `authors.name` matches the string *Refactoring: Improving the Design of Existing Code* exactly.
Example 7. Using Hibernate Search to query the indexes

```java
// Not shown: get the entity manager and open a transaction
SearchSession searchSession = Search.getSearchSession( entityManager ); ①

SearchResult<Book> result = searchSession.search( Book.class ) ②
    .predicate( f -> f.match() ③
        .onFields( "title", "authors.name" )
        .matching( "Refactoring: Improving the Design of Existing Code" )
    )
    .fetch(); ④

long totalHitCount = result.getTotalHitCount(); ⑤
List<Book> hits = result.getHits(); ⑥

List<Book> hits2 =
    /* ... same DSL calls as above... */
    .fetchHits(); ⑦

// Not shown: commit the transaction and close the entity manager
```

① Get a Hibernate Search session, called `SearchSession`, from the `EntityManager`.

② Initiate a search query on the index mapped to the `Book` entity.

③ Define that only documents matching the given predicate should be returned. The predicate is created using a factory `f` passed as an argument to the lambda expression.

④ Build the query and fetch the results.

⑤ Retrieve the total number of matching entities.

⑥ Retrieve matching entities.

⑦ In case you’re not interested in the whole result, but only in the hits, you can also call `fetchHits()` directly.

If for some reason you don’t want to use lambdas, you can use an alternative, object-based syntax, but it will be a bit more verbose:
Example 8. Using Hibernate Search to query the indexes - object-based syntax

```java
// Not shown: get the entity manager and open a transaction
SearchSession searchSession = Search.getSearchSession( entityManager ); ①

SearchScope<Book> scope = searchSession.scope( Book.class ); ②

SearchResult<Book> result = scope.search() ③
    .predicate( scope.predicate().match() ④
        .onFields( "title", "authors.name" )
        .matching( "Refactoring: Improving the Design of Existing Code" )
        .toPredicate() )
    .fetch(); ⑤

long totalHitCount = result.getTotalHitCount(); ⑥
List<Book> hits = result.getHits(); ⑦
List<Book> hits2 =
    /* ... same DSL calls as above... */
    .fetchHits(); ⑧

// Not shown: commit the transaction and close the entity manager
```

① Get a Hibernate Search session, called SearchSession, from the EntityManager.
② Create a "search scope", representing the indexed types that will be queried.
③ Initiate a search query targeting the search scope.
④ Define that only documents matching the given predicate should be returned. The predicate is created using the same search scope as the query.
⑤ Build the query and fetch the results.
⑥ Retrieve the total number of matching entities.
⑦ Retrieve matching entities.
⑧ In case you're not interested in the whole result, but only in the hits, you can also call fetchHits() directly.

It is possible to get just the total hit count, using fetchTotalHitCount() method.

Example 9. Using Hibernate Search to count the matches

```java
// Not shown: get the entity manager and open a transaction
SearchSession searchSession = Search.getSearchSession( entityManager );

long totalHitCount = searchSession.search( Book.class )
    .predicate( f -> f.match() ⑦
        .onFields( "title", "authors.name" )
        .matching( "Refactoring: Improving the Design of Existing Code" )
    )
    .fetchTotalHitCount(); ①

// Not shown: commit the transaction and close the entity manager
```

① Fetch the total hit count.

⑦ Fetch the total hit count.
1.8. Analysis

Exact matches are well and good, but obviously not what you would expect from a full-text search engine.

For non-exact matches, you will need to configure analysis.

1.8.1. Concept

In the Lucene world (Lucene, Elasticsearch, Solr, …), non-exact matches can be achieved by applying what is called an "analyzer" to both documents (when indexing) and search terms (when querying).

The analyzer will perform three steps, delegated to the following components, in the following order:

1. Character filter: transforms the input text: replaces, adds or removes characters. This step is rarely used, generally text is transformed in the third step.
2. Tokenizer: splits the text into several words, called "tokens".
3. Token filter: transforms the tokens: replaces, add or removes characters in a token, derives new tokens from the existing ones, removes tokens based on some condition, …

In order to perform non-exact matches, you will need to either pick a pre-defined analyzer, or define your own by combining character filters, a tokenizer, and token filters.

The following section will give a reasonable example of a general-purpose analyzer. For more advanced use cases, refer to the Analysis section.

1.8.2. Configuration

Once you know what analysis is and which analyzer you want to apply, you will need to define it, or at least give it a name in Hibernate Search. This is done though analysis configurers, which are defined per backend:

1. First, you need to implement an analysis configurer, a Java class that implements a backend-specific interface: LuceneAnalysisConfigurer or ElasticsearchAnalysisConfigurer.
2. Second, you need to alter the configuration of your backend to actually use your analysis configurer.

As an example, let's assume that one of your indexed Book entities has the title "Refactoring: Improving the Design of Existing Code", and you want to get hits for any of the following search terms: "Refactor", "refactors", "refactored" and "refactoring". One way to achieve this is to use an analyzer
with the following components:

- A "standard" tokenizer, which splits words at whitespaces, punctuation characters and hyphens. It is a good general purpose tokenizer.
- A "lowercase" filter, which converts every character to lowercase.
- A "snowball" filter, which applies language-specific stemming.

The examples below show how to define an analyzer with these components, depending on the backend you picked.

**Example 10. Analysis configurer implementation and configuration in persistence.xml for a "Hibernate ORM + Lucene" setup**

```java
package org.hibernate.search.documentation.gettingstarted.withhsearch.withanalysis;
import org.hibernate.search.backend.lucene.analysis.LuceneAnalysisConfigurer;
import org.hibernate.search.backend.lucene.analysis.model.dsl.LuceneAnalysisDefinitionContainerContext;
import org.apache.lucene.analysis.core.LowerCaseFilterFactory;
import org.apache.lucene.analysis.miscellaneous.ASCIIFoldingFilterFactory;
import org.apache.lucene.analysis.snowball.SnowballPorterFilterFactory;
import org.apache.lucene.analysis.standard.StandardTokenizerFactory;

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

<property name="hibernate.search.backends.myBackend.analysis_configurer" value="org.hibernate.search.documentation.gettingstarted.withhsearch.withanalysis.MyLuceneAnalysisConfigurer"/>

1. Define a custom analyzer named "myAnalyzer".
2. Set the tokenizer to a standard tokenizer. You need to pass factory classes to refer to components.
3. Set the token filters. Token filters are applied in the order they are given.
4. Set the value of a parameter for the last added char filter/tokenizer/token filter.
5. Assign the configurer to the backend "myBackend" in the Hibernate Search configuration (here in persistence.xml).
Example 11. Analysis configurer implementation and configuration in persistence.xml for a "Hibernate ORM + Elasticsearch" setup

```
package org.hibernate.search.documentation.gettingstarted.withhsearch.withanalysis;

import org.hibernate.search.backend.elasticsearch.analysis.ElasticsearchAnalysisConfigurer;
import org.hibernate.search.backend.elasticsearch.analysis.model.dsl.ElasticsearchAnalysisDefinitionContainerContext;

public class MyElasticsearchAnalysisConfigurer implements ElasticsearchAnalysisConfigurer {
    @Override
    public void configure(ElasticsearchAnalysisDefinitionContainerContext context) {
        context.analyzer("myAnalyzer").custom( ①  
            .withTokenizer("standard") ②  
            .withTokenFilters("asciifolding", "lowercase", "mySnowballFilter" ); ③  
        );

        context.tokenFilter("mySnowballFilter") ④  
            .type("snowball")  
            .param("language", "English"); ⑤  
    }
}
```

```
<property name="hibernate.search.backends.myBackend.analysis_configurer"  
    value="org.hibernate.search.documentation.gettingstarted.withhsearch.withanalysis.MyElasticsearchAnalysisConfigurer"/> ⑥
```

① Define a custom analyzer named "myAnalyzer".
② Set the tokenizer to a standard tokenizer.
③ Set the token filters. Token filters are applied in the order they are given.
④ Note that, for Elasticsearch, any parameterized char filter, tokenizer or token filter must be defined separately and given a name.
⑤ Set the value of a parameter for the char filter/tokenizer/token filter being defined.
⑥ Assign the configurer to the backend "myBackend" in the Hibernate Search configuration (here in persistence.xml).

Once analysis is configured, the mapping must be adapted to assign the relevant analyzer to each field:

Example 12. Book and Author entities after adding Hibernate Search specific annotations
import java.util.HashSet;
import java.util.Set;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import org.hibernate.search.mapper.pojo.mapping.definition.annotation.FullTextField;
import org.hibernate.search.mapper.pojo.mapping.definition.annotation.Indexed;
import org.hibernate.search.mapper.pojo.mapping.definition.annotation.IndexedEmbedded;

@Entity
@Indexed
public class Book {

@Id
@GeneratedValue
private Integer id;

@FullTextField(analyzer = "myAnalyzer") ①
private String title;

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

public Book() {
    // Getters and setters
    // ...
}
}

import org.hibernate.search.mapper.pojo.mapping.definition.annotation.FullTextField;

@Entity
public class Author {

@Id
@GeneratedValue
private Integer id;

@FullTextField(analyzer = "myAnalyzer") ①
private String name;

@ManyToMany(mappedBy = "authors")
private Set<Book> books = new HashSet<>();

public Author() {
    // Getters and setters
    // ...
}
}

① Replace the @GenericField annotation with @FullTextField, and set the analyzer
parameter to the name of the custom analyzer configured earlier.

That's it! Now, once the entities will be reindexed, you will be able to search for the terms "Refactor", "refactors", "refactored" or "refactoring", and the book with the title "Refactoring: Improving the Design of Existing Code" will show up in the results.

Mapping changes are not auto-magically applied to already-indexed data. Unless you know what you are doing, you should remember to reindex your data after you changed the Hibernate Search mapping of your entities.

Example 13. Using Hibernate Search to query the indexes after analysis was configured

```java
// Not shown: get the entity manager and open a transaction
SearchSession searchSession = Search.getSearchSession( entityManager );

SearchResult<Book> result = searchSession.search( Book.class )
    .predicate( factory -> factory.match()
        .onFields( "title", "authors.name" )
        .matching( "refactor" )
    )
    .fetch();
// Not shown: commit the transaction and close the entity manager
```

1.9. What’s next

The above paragraphs helped you getting an overview of Hibernate Search. The next step after this tutorial is to get more familiar with the overall architecture of Hibernate Search (Architecture) and explore the basic features in more detail.

Two topics which were only briefly touched in this tutorial were analysis configuration (Analysis) and bridges (Bridges). Both are important features required for more fine-grained indexing.

Other features that you will probably want to use include sorts and projections.

If you want to see an example project using Hibernate Search, you can also have a look at the "Library" showcase, a sample application using Hibernate Search in a Spring Boot environment.
Chapter 2. Concepts

2.1. Full-text search

2.2. Mapping

2.3. Analysis

This section is currently incomplete. A decent introduction is included in the getting started guide: see Analysis.

For more information about how to configure analysis, see the documentation of each backend:

• Analysis for the Lucene backend
• Analysis for the Elasticsearch backend
Chapter 3. Architecture

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.
Chapter 4. Configuration

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.

4.1. Configuration sources

When using Hibernate Search within Hibernate ORM, configuration properties are retrieved from Hibernate ORM.

This means that wherever you set Hibernate ORM properties, you can set Hibernate Search properties:

- In a `hibernate.properties` file at the root of your classpath.
- In `persistence.xml`, if you bootstrap Hibernate ORM with the JPA APIs
- In JVM system properties (`-DmyProperty=myValue` passed to the `java` command)
- In the configuration file of your framework, for example `application.yaml` / `application.properties` for Spring Boot.

4.2. Structure of configuration properties

Configuration properties are all grouped under a common root. In the ORM integration, this root is `hibernate.search`, but other integrations (Infinispan, ...) may use a different one. This documentation will use `hibernate.search` in all examples.

Under that root, we can distinguish between four categories of properties.

Global properties

These properties potentially affect all Hibernate Search. They are generally located just under the `hibernate.search` root.

Notable properties:

- `hibernate.search.default_backend`: defines the name of the backend used by default on all indexes.

Other global properties are explained in the relevant parts of this documentation:

- Hibernate ORM integration

Backend properties

These properties affect a single backend. They are grouped under a common root that includes the backend name: `hibernate.search.backends.<backend name>`. The backend name is
arbitrarily defined by the user: just pick a string, such as myBackend or elasticsearch, and make sure to use it consistently.

Notable properties:

- `hibernate.search.backends.<backend name>.type`: the type of the backend. Set this to either lucene or elasticsearch.

Other backend properties are explained in the relevant parts of this documentation:

- Lucene backend
- Elasticsearch backend

Index properties

These properties affect either one or multiple indexes, depending on the root.

With the root `hibernate.search.backends.<backend name>.index_defaults`, they set defaults for all indexes of the referenced backend. The backend name must match the name defined in the mapping.

With the root `hibernate.search.backends.<backend name>.indexes.<index name>`, they set the value for a specific index, overriding the defaults (if any). The backend and index names must match the names defined in the mapping. For ORM entities, the default index name is the name of the indexed class, without the package: org.mycompany.Book will have Book as its default index name. Index names can be customized in the mapping.

Examples:

- `hibernate.search.backends.myBackend.index_defaults.lifecycle.strategy = validate` sets the lifecycle.strategy property for all indexes of the backend myBackend.
- `hibernate.search.backends.myBackend.indexes.Product.lifecycle.strategy = none` sets the lifecycle.strategy property for the Product index of the backend myBackend.

Other index properties are explained in the relevant parts of this documentation:

- Lucene backend
- Elasticsearch backend

4.3. Type of configuration properties

Property values can be set programmatically as Java objects, or through a configuration file as a string that will have to be parsed.
Each configuration property in Hibernate Search has an assigned type, and this type defines the accepted values in both cases.

Here are the definitions of all property types.

<table>
<thead>
<tr>
<th>Designation</th>
<th>Accepted Java objects</th>
<th>Accepted String format</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>java.lang.String</td>
<td>Any string</td>
</tr>
<tr>
<td>Boolean</td>
<td>java.lang.Boolean</td>
<td>true or false (case-insensitive)</td>
</tr>
<tr>
<td>Integer</td>
<td>java.lang.Number (will call .intValue())</td>
<td>Any string that can be parsed by Integer.parseInt</td>
</tr>
<tr>
<td>Long</td>
<td>java.lang.Number (will call .longValue())</td>
<td>Any string that can be parsed by Long.parseLong</td>
</tr>
<tr>
<td>Bean reference of type T</td>
<td>An instance of T or org.hibernate.search.engine.environment.bean.BeanReference or a reference by type as a java.lang.Class (see Bean resolution)</td>
<td>A reference by name as a java.lang.String (this can be a fully-qualified class name, see Bean resolution)</td>
</tr>
<tr>
<td>Multi-valued bean reference of type T</td>
<td>A java.util.Collection containing bean references (see above)</td>
<td>Whitespace separated string containing bean references (see above)</td>
</tr>
</tbody>
</table>

### 4.4. Configuration property tracking

When using the ORM integration, Hibernate Search will track the parts of the provided configuration that are actually used and will log a warning if any configuration property is never used, because that might indicate a configuration issue.

To disable this warning, set the hibernate.search.enable_configuration_property_tracking boolean property to false.

### 4.5. Bean resolution

Hibernate Search allows to plug in references to custom beans in various places: configuration properties, mapping annotations, arguments to APIs, ...

Everywhere a custom bean reference is expected, three types of references are accepted:

- A reference by type, as a java.lang.Class.
- A reference by name, as a java.lang.String.
- A reference by type and name (through a BeanReference, see below).

Bean resolution (i.e. the process of turning this reference into an object instance) happens as follows:
• If a dependency injection framework is integrated into Hibernate ORM, the reference is first requested to the DI framework. Currently CDI and recent versions of Spring are supported.

• Otherwise, or if the DI framework cannot find a matching bean definition, reflection is used to resolve the bean. References by name are turned into a reference by type by interpreting the bean name as the fully-qualified class name Reference by type are resolved by calling the public, no-argument constructor of the given type. References by type and name are resolved as a reference by name, then the resulting object is checked to be an instance of the given type.

For experienced users, Hibernate Search also provides the org.hibernate.search.engine.environment.bean.BeanReference type, which is accepted in configuration properties and APIs. This interface allows to plug in custom instantiation and cleanup code. See the javadoc of this interface for details.
Chapter 5. Hibernate ORM integration

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.

5.1. General configuration

5.1.1. Enabling the integration

The Hibernate ORM integration is enabled by default as soon as it is present in the classpath.

If for some reason you need to disable it, set the `hibernate.search.autoregister_listeners boolean property` to `false`.

5.1.2. Other configuration properties

Other configuration properties are mentioned in the relevant parts of this documentation. You can find a full reference of available properties in the Hibernate Search javadoc: `org.hibernate.search.mapper.orm.cfg.HibernateOrmMapperSettings`.

5.2. Mapping ORM entities to indexes

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.

5.2.1. Configuration

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.

By default, Hibernate Search will automatically process mapping annotations for entity types, as well as nested types in those entity types, for instance embedded types. If you want to ignore these annotations, set `hibernate.search.enable_annotation_mapping` to `false`.

To configure the mapping manually, you can set a mapping configurer. By setting `hibernate.search.mapping_configurer` to a bean reference of type `org.hibernate.search.mapper.orm.mapping.HibernateOrmSearchMappingConfigurer`, you can use a programmatic API to define the mapping.

See Programmatic mapping for more information about the programmatic mapping API.
5.2.2. Entity/index mapping

In order to index an entity, it must be annotated with @Indexed. All entities not annotated with @Indexed will be ignored by the indexing process.

Example 14. Marking a class for indexing with @Indexed

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

    // Fields and methods...
}
```

By default:

- The index name will be the fully qualified name of the entity.
- The index will be created in the default backend. See the getting started guide or Structure of configuration properties for more information about how to configure backends.
- The identifier of indexed documents will be generated from the entity identifier. Most types commonly used for entity identifiers are supported out of the box, but for more exotic types you may need specific configuration. See Identifier mapping for details.
- The index won't have any field. Fields must be mapped to properties explicitly. See Direct field mapping for details.

You can change the name of the index by setting @Indexed(index = ...). Note that index names must be unique in a given application.

Example 15. Explicit index name with @Indexed.index

```java
@Entity
@Indexed(index = "Author")
public class Author {

    // Fields and methods...
}
```

If you defined multiple backends, you can map entities to another backend than the default one. By setting @Indexed(backend = "backend2") you inform Hibernate Search that the index for your entity must be created in the backend named "backend2". This may be useful if your model has clearly defined sub-parts with very different indexing requirements.

Example 16. Explicit backend with @Indexed.backend

```java
@Entity
@Indexed(backend = "backend2")
public class User {

    // Fields and methods...
}
```
Entities indexed in different backends cannot be targeted by the same query. For example, with the mappings defined above, and assuming "backend2" is not the default backend, the following code will throw an exception, because Author and User are indexed in different backends:

```java
// This will fail
SearchQuery<Object> query = searchSession.search(
    Arrays.asList( Author.class, User.class )
).predicate( f -> f.matchAll() )
.toQuery();
```

### 5.2.3. Identifier mapping

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.

### 5.2.4. Direct field mapping

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.

Direct field mapping allows to map a property to an index field directly: you just need to add an annotation, configure the field through the annotation attributes, and Hibernate Search will take care of extracting the property value and populating the index field when necessary.

Direct field mapping looks like this:
Example 17. Mapping properties to fields directly

```java
@FullTextField(analyzer = "myAnalyzer", projectable = Projectable.YES) ①
@KeywordField(name = "title_sort", normalizer = "myNormalizer", sortable = Sortable.YES) ②
private String title;
@GenericField(projectable = Projectable.YES, sortable = Sortable.YES) ③
private Integer pageCount;
```

① Map the `title` property to a full-text field with the same name. Some options can be set to customize the fields' behavior, in this case the analyzer (for full-text indexing) and the fact that this field is projectable (its value can be retrieved from the index).

② Map the `title` property to another field, configured differently: it is not analyzed, but simply normalized (i.e. it’s not split into multiple tokens), and it is stored in such a way that it can be used in sorts.

Mapping a single property to multiple fields is particularly useful when doing full-text search: at query time, you can use a different field depending on what you need. You can map a property to as many fields as you want, but each must have a unique name.

③ Map another property to its own field.

Before you map a property, you must consider two things:

The `@Field` annotation

In its simplest form, direct field mapping is achieved by applying the `@GenericField` annotation to a property. This annotation will work for every supported property type, but is rather limited: it does not allow full-text search in particular. To go further, you will need to rely on different, more specific annotations, which offer specific attributes. The available annotations are described in details in Available field annotations.

The type of the property

In order for the `@Field` annotation to work correctly, the type of the mapped property must be supported by Hibernate Search. See Built-in value bridges for a list of all types that are supported out of the box, and Mapping custom property types for indications on how to handle more complex types, be it simply containers (`List<String>`, `Map<String, Integer>`, …) or custom types.

Each field annotation has its own attributes, but the following ones are common to most annotations:

name

The name of the index field. By default, it is the same as the property name. You may want to change it in particular when mapping a single property to multiple fields.

Value: `String`. Defaults to the name of the property.
**sortable**

Whether the field can be **sorted on**, i.e. whether a specific data structure is added to the index to allow efficient sorts when querying.

Value: `Sortable.YES, Sortable.NO, Sortable.DEFAULT`.

⚠️ This option is not available for `@FullTextField`. See [here](#) for an explanation and some solutions.

**projectable**

Whether the field can be **projected on**, i.e. whether the field value is stored in the index to allow later retrieval when querying.

Value: `Projectable.YES, Projectable.NO, Projectable.DEFAULT`.

**indexNullAs**

Disabled by default. When set, it defines the value to use as a replacement anytime the property value is null.

⚠️ The replacement is defined as a String. Thus its value has to be parsed. Look up the column Parsing method for 'indexNullAs' in [Built-in value bridges](#) to find out the format used when parsing.

**Available field annotations**

Various direct field mapping annotations exist, each offering its own set of customization options:

- **@GenericField**
  
  A good default choice that will work for every supported property type.
  
  Fields mapped using this annotation do not provide any advanced features such as full-text search: matches on a generic field are exact matches.

- **@FullTextField**
  
  A text field whose value is considered as multiple words. Only works for `String` fields.
  
  Matches on a full-text field can be more subtle than exact matches: match fields which contains a given word, match fields regardless of case, match fields ignoring diacritics, ...
  
  Full-text fields must be assigned an `analyzer`, referenced by its name. See [Analysis](#) for more details about analyzers and full-text analysis.
Full-text fields cannot be sorted on. If you need to sort on the value of a property, it is recommended to use `@KeywordField`, with a normalizer if necessary (see below). Note that multiple fields can be added to the same property, so you can use both `@FullTextField` and `@KeywordField` if you need both full-text search and sorting.

@KeywordField

A text field whose value is considered as a single keyword. Only works for `String` fields.

Keyword fields allow subtle matches, similarly to full-text fields, with the limitation that keyword fields only contain one token. On the other hand, this limitation allows keyword fields to be sorted on.

Keyword fields may be assigned a normalizer, referenced by its name. See `Analysis` for more details about normalizers and full-text analysis.

@ScaledNumberField

A numeric field for integer or floating-point values that require a higher precision than doubles but always have roughly the same scale. Only works for either `java.math.BigDecimal` or `java.math.BigInteger` fields.

Scaled numbers are indexed as integers, typically a long (64 bits), with a fixed scale that is consistent for all values of the field across all documents. Because scaled numbers are indexed with a fixed precision, they cannot represent all `BigDecimal` or `BigInteger` values. Values that are too large to be indexed will trigger a runtime exception. Values that have trailing decimal digits will be rounded to the nearest integer.

To index numbers that have significant digits after the decimal point, set the `decimalScale` to the number of digits you need indexed. The decimal point will be shifted that many times to the right before indexing, preserving that many digits from the decimal part. To index very large numbers that cannot fit in a long, set the decimal point to a negative value. The decimal point will shifted that many times to the left before indexing, dropping all digits from the decimal part.

`decimalScale` with strictly positive values is allowed only for `BigDecimal`, since `BigInteger` values have no decimal digits.

Note that shifting of the decimal points is completely transparent and will not affect how you use the search DSL: you be expected to provide "normal" `BigDecimal` or `BigInteger` values, and Hibernate Search will apply the `decimalScale` and rounding transparently.

As a result of the rounding, search predicates and sorts will only be as precise as what the `decimalScale` allows.

Note that rounding does not affect projections, which will return the original value without any loss of precision.
A typical use case is monetary amounts, with a decimal scale of 2 because only two digits are generally needed beyond the decimal point.

Using Hibernate ORM mapping, a default `decimalScale` is taken automatically from the underlying `scale` value of the relative SQL `@Column`, using the Hibernate ORM metadata. The value could be overridden explicitly using the `decimalScale` attribute.

**Mapping spatial types**

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.

**Mapping custom property types**

Even types that are not supported out of the box can be mapped. There are various solutions, some simple and some more powerful, but they all come down to extracting data from the unsupported type and convert it to types that are supported by the backend.

There are two cases to distinguish:

1. If the unsupported type is simply a container (`List<String>`) or multiple nested containers (`Map<Integer, List<String>>`) whose elements have a supported type, then what you need is a container value extractor.

   By default, built-in extractors are transparently applied to standard container types: `Iterable` and subtypes, `Map` (extracting the value), `Optional`, `OptionalInt`, ... If that is all you need, then no extra configuration is necessary.

   If your container is a custom one, or you need a different behavior than the default (extract keys instead of values from a `Map`, for example), then you will need to set a custom extractor chain on the `@*Field` annotation. All `@*Field` annotations expose an `extraction` attribute to that end. See Container value extractors for more information on available extractors and custom extractors.

2. Otherwise, you will have to rely on a custom component, called a bridge, to extract data from your type. See Bridges for more information on custom bridges.

**5.2.5. Bridges**

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.
Starting with Hibernate Search 6, there are five separate interfaces for bridges:

- **ValueBridge** can be used for simple use cases when mapping an object’s property.

  The ValueBridge is applied at the property level using one of the pre-defined @*Field* annotations: @GenericField, @FullTextField, ...

  **ValueBridge** is a suitable interface for your custom bridge if:
  
  - The property value should be mapped to a single index field.
  
  - The bridge should be applied to a property whose type is effectively immutable. For example *Integer*, or a custom *enum* type, or a custom bean type whose content never changes would be suitable candidates, but a custom bean type with setters would most definitely not.

- **PropertyBridge** can be used for more complex uses cases when mapping an object’s property.

  The PropertyBridge is applied at the property level using a custom annotation.

  **PropertyBridge** can be used even if the property being mapped has a mutable type, or if its value should be mapped to multiple index fields.

- **TypeBridge** should be used when mapping multiple properties of an object, potentially combining them in the process.

  The TypeBridge is applied at the type level using a custom annotation.

  Similarly to PropertyBridge, TypeBridge can be used even if the properties being mapped have a mutable type, or if their values should be mapped to multiple index fields.

- **IdentifierBridge** can be used together with @DocumentId to map an unusual entity identifier to a document identifier.

- **RoutingKeyBridge** can be used to define a "routing key", i.e. a key that will be used to determine the shard where corresponding documents must be stored in the index.

You can find example of custom bridges in the Hibernate Search source code:

- `org.hibernate.search.integrationtest.showcase.library.bridge.ISBNBridge` implements ValueBridge.


- `org.hibernate.search.integrationtest.showcase.library.bridge.AccountBorrowalSummaryBridge` implements TypeBridge. The corresponding annotation is `org.hibernate.search.integrationtest.showcase.library.bridge.annotation.AccountBorrowalSummaryBridge`. 
Value bridges

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.

Built-in value bridges

Some types have built-in value bridges, meaning they are supported out-of-the-box for direct field mapping using @*Field* annotations.

Below is a table listing all types with built-in value bridges, along with the value assigned to the "raw" fields, i.e. the value passed to the underlying backend.

For information about the underlying indexing and storage used by the backend, see Lucene field types or Elasticsearch field types depending on your backend.

<table>
<thead>
<tr>
<th>Property type</th>
<th>Value of &quot;raw&quot; fields (if different)</th>
<th>Parsing method for 'indexNullAs'</th>
</tr>
</thead>
<tbody>
<tr>
<td>All enum types</td>
<td>name() as a java.lang.String</td>
<td>Enum.valueOf(String)</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.Byte, byte</td>
<td>java.lang.String</td>
<td>Byte.parseByte(String)</td>
</tr>
<tr>
<td>java.lang.Short, short</td>
<td>-</td>
<td>Short.parseShort(String)</td>
</tr>
<tr>
<td>java.lang.Integer, int</td>
<td>-</td>
<td>Integer.parseInt(String)</td>
</tr>
<tr>
<td>java.lang.Long, long</td>
<td>-</td>
<td>Long.parseLong(String)</td>
</tr>
<tr>
<td>java.lang.Double, double</td>
<td>-</td>
<td>Double.parseDouble(String)</td>
</tr>
<tr>
<td>java.lang.Float, float</td>
<td>-</td>
<td>Float.parseFloat(String)</td>
</tr>
<tr>
<td>java.lang.Boolean, boolean</td>
<td>-</td>
<td>Accepts the strings true or false, ignoring case</td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td>-</td>
<td>new BigDecimal(String)</td>
</tr>
<tr>
<td>java.math.BigInteger</td>
<td>-</td>
<td>new BigInteger(String)</td>
</tr>
<tr>
<td>java.net.URI</td>
<td>toString() as a java.lang.String</td>
<td>new URI(String)</td>
</tr>
<tr>
<td>java.net.URL</td>
<td>toExternalForm() as a java.lang.String</td>
<td>new URL(String)</td>
</tr>
<tr>
<td>java.time.Instant</td>
<td>-</td>
<td>Instant.parse(String)</td>
</tr>
<tr>
<td>java.time.LocalDate</td>
<td>-</td>
<td>LocalDate.parse(String)</td>
</tr>
<tr>
<td>Property type</td>
<td>Value of &quot;raw&quot; fields (if different)</td>
<td>Parsing method for 'indexNullAs'</td>
</tr>
<tr>
<td>-------------------------------------</td>
<td>--------------------------------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>java.time.LocalTime</td>
<td>-</td>
<td>LocalDateTime.parse(String)</td>
</tr>
<tr>
<td>java.time.LocalDateTime</td>
<td>-</td>
<td>LocalDateTime.parse(String)</td>
</tr>
<tr>
<td>java.time.OffsetDateTime</td>
<td>-</td>
<td>OffsetDateTime.parse(String)</td>
</tr>
<tr>
<td>java.time.OffsetTime</td>
<td>-</td>
<td>OffsetTime.parse(String)</td>
</tr>
<tr>
<td>java.time.ZonedDateTime</td>
<td>-</td>
<td>ZonedDateTime.parse(String)</td>
</tr>
<tr>
<td>java.time.ZoneId</td>
<td>getId() as a java.lang.String</td>
<td>ZoneId.of(String)</td>
</tr>
<tr>
<td>java.time.ZoneOffset</td>
<td>getTotalSeconds() as a java.lang.Integer</td>
<td>ZoneOffset.of(String)</td>
</tr>
<tr>
<td>java.time.Period</td>
<td>A formatted java.lang.String: &lt;years on 11 characters&gt; &lt;months on 11 characters&gt; &lt;days on 11 characters&gt;</td>
<td>Period.parse(String)</td>
</tr>
<tr>
<td>java.time.Duration</td>
<td>toNanos() as a java.lang.Long</td>
<td>Duration.parse(String)</td>
</tr>
<tr>
<td>java.time.Year</td>
<td>-</td>
<td>Year.parse(String)</td>
</tr>
<tr>
<td>java.time.YearMonth</td>
<td>-</td>
<td>YearMonth.parse(String)</td>
</tr>
<tr>
<td>java.time.MonthDay</td>
<td>-</td>
<td>MonthDay.parse(String)</td>
</tr>
<tr>
<td>java.util.UUID</td>
<td>toString() as a java.lang.String</td>
<td>UUID.fromString(String)</td>
</tr>
<tr>
<td>java.util.Calendar</td>
<td>A java.time.ZonedDateTime representing the same date/time and timezone; see Support for legacy java.util date/time APIs</td>
<td>ZonedDateTime.parse(String)</td>
</tr>
<tr>
<td>java.util.Date</td>
<td>toInstant() as a java.time.Instant; see Support for legacy java.util date/time APIs</td>
<td>Instant.parse(String)</td>
</tr>
<tr>
<td>java.sql.Timestamp</td>
<td>toInstant() as a java.time.Instant; see Support for legacy java.util date/time APIs</td>
<td>Instant.parse(String)</td>
</tr>
<tr>
<td>java.sql.Date</td>
<td>toInstant() as a java.time.Instant; see Support for legacy java.util date/time APIs</td>
<td>Instant.parse(String)</td>
</tr>
<tr>
<td>java.sql.Time</td>
<td>toInstant() as a java.time.Instant; see Support for legacy java.util date/time APIs</td>
<td>Instant.parse(String)</td>
</tr>
</tbody>
</table>
Type bridges and property bridges

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.

Identifier bridges

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.

Document identifiers have slightly different requirements than index fields, which is why they are mapped using a different type of bridge.

Built-in identifier bridges

Some types have built-in identifier bridges, meaning they are supported out-of-the-box for document ID mapping.

Below is a table listing all types with built-in identifier bridges, along with the value of the document identifier, i.e. the value passed to the underlying backend.

<table>
<thead>
<tr>
<th>Property type</th>
<th>Value of document identifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.String</td>
<td>Same</td>
</tr>
<tr>
<td>java.lang.Short, short</td>
<td>toString()</td>
</tr>
<tr>
<td>java.lang.Integer, int</td>
<td>toString()</td>
</tr>
<tr>
<td>java.lang.Long, long</td>
<td>toString()</td>
</tr>
<tr>
<td>java.math.BigInteger</td>
<td>toString()</td>
</tr>
<tr>
<td>All enum types</td>
<td>name()</td>
</tr>
<tr>
<td>java.util.UUID</td>
<td>toString()</td>
</tr>
</tbody>
</table>

Routing key bridges

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.

Support for legacy java.util date/time APIs

Using legacy date/time types such as java.util.Calendar, java.util.Date, java.sql.Timestamp, java.sql.Date, java.sql.Time is not recommended, due to their numerous quirks and shortcomings. The java.time package introduced in Java 8 should generally be preferred.
That being said, integration constraints may force you to rely on the legacy date/time APIs, which is why Hibernate Search still attempts to support them on a best effort basis.

Since Hibernate Search uses the `java.time` APIs to represent date/time internally, the legacy date/time types need to be converted before they can be indexed. Hibernate Search keeps things simple: `java.util.Date`, `java.util.Calendar`, etc. will be converted using their time-value (number of milliseconds since the epoch), which will be assumed to represent the same date/time in Java 8 APIs. In the case of `java.util.Calendar`, timezone information will be preserved for projections.

For all dates after 1900, this will work exactly as expected.

Before 1900, indexing and searching through Hibernate Search APIs will also work as expected, but if you need to access the index natively, for example through direct HTTP calls to an Elasticsearch server, you will notice that the indexed values are slightly "off". This is caused by differences in the implementation of `java.time` and legacy date/time APIs which lead to slight differences in the interpretation of time-values (number of milliseconds since the epoch).

The "drifts" are consistent: they will also happen when building a predicate, and they will happen in the opposite direction when projecting. As a result, the differences will not be visible from an application relying on the Hibernate Search APIs exclusively. They will, however, be visible when accessing indexes natively.

For the large majority of use cases, this will not be a problem. If this behavior is not acceptable for your application, you should look into implementing custom value bridges and instructing Hibernate Search to use them by default for `java.util.Date`, `java.util.Calendar`, etc.: see Default bridge resolver.

Technically, conversions are difficult because the `java.time` APIs and the legacy date/time APIs do not have the same internal calendar.

In particular:

- `java.time` assumes a "Local Mean Time" before 1900, while legacy date/time APIs do not support it (JDK-6281408). As a result, time values (number of milliseconds since the epoch) reported by the two APIs will be different for dates before 1900.

- `java.time` uses a proleptic Gregorian calendar before October 15, 1582, meaning it acts as if the Gregorian calendar, along with its system of leap years, had always existed. Legacy date/time APIs, on the other hand, use the Julian calendar before that date (by default), meaning the leap years are not exactly the same ones. As a result, some dates that are deemed valid by one API will be deemed invalid by the other, for example February 29, 1500.

Those are the two main problems, but there may be others.
Default bridge resolver

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.

5.2.6. Indexed-embedded

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.

5.2.7. Container value extractors

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.

5.2.8. Programmatic mapping

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.

5.3. Indexing

5.3.1. 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.

Below are a few facts to explain how automatic indexing works.

*Changes have to occur in the ORM session in order to be detected*

Hibernate Search uses internal events of Hibernate ORM in order to detect changes: these events will only be triggered if you actually manipulate managed entity objects in your code, updating them by setting their properties or deleting them by calling the appropriate method on the Hibernate ORM session.

Conversely, changes resulting from `insert/delete/update` queries, be it SQL or JPQL/HQL queries, are not detected by Hibernate Search. This is because queries are executed on the database side, without Hibernate having any knowledge of which entities are actually created, deleted or updated. One workaround is to explicitly reindex after you run such queries.

*Indexing happens after transactions are committed*

Indexes are not updated immediately, but only after transactions are successfully committed. That
way, if a transaction is rolled back, the indexes will be left in a state consistent with the database.

**Index changes may not be visible immediately**

By default, the transaction commit will return after index changes are committed to the indexes. This means index changes are safely stored to disk, but this does not mean a search query ran immediately after the transaction commit will take the changes into account: when using the Elasticsearch backend in particular, changes may take some time to be visible from search queries.

See *Synchronization with the indexes* for details.

**Only relevant changes trigger indexing**

Hibernate Search is aware of the properties that are accessed when building indexed documents. Thanks to that knowledge, it is able to skip reindexing when a property is modified, but does not affect the indexed document.

You can control this "dirty checking" by setting the boolean property `hibernate.search.automatic_indexing.enable_dirty_check`:

- by default, or when set to `true`, Hibernate Search will consider whether modified properties are relevant before triggering reindexing.
- when set to `false`, Hibernate Search will trigger reindexing upon any change, regardless of the entity properties that changed.

**Indexing may fetch extra data from the database**

Even when you change only a single property of an indexed entity, if that property is indexed, Hibernate Search needs to rebuild the corresponding document in full.

Even if Hibernate Search tries to only load what is necessary for indexing, depending on your mapping, this may lead to lazy associations being loaded just to reindex entities, even if you didn’t need them in your business code.

This extra cost can be mitigated to some extent by leveraging Hibernate ORM’s batch fetching; see the `batch_fetch_size` property and the `@BatchSize` annotation.

Automatic indexing may be unnecessary if your index is read-only or if you update it regularly through explicit indexing. You can enable or disable automatic indexing by setting the configuration property `hibernate.search.automatic_indexing.strategy`:

- when set to `session`, each change to an indexed entity (persist, update, delete) through a Hibernate ORM Session/EntityManager will automatically lead to a similar modification to the index.
- when set to `none`, changes to entities are ignored, and indexing requires an explicit action (see *Explicit indexing*).
Synchronization with the indexes

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.

Hibernate Search offers multiple strategies to control synchronization with the indexes during automatic indexing, i.e. to control the minimum progress of indexing before the transaction commit returns.

You can define a default strategy for all sessions by setting the configuration property `hibernate.search.automatic_indexing.synchronization_strategy`:

- when set to `queued`, the transaction commit will return as soon as the index changes are queued in the backend.

  This strategy offers no guarantee as to whether indexing will be performed successfully, or even whether indexing will be performed at all: the local JVM may crash before the works are executed, in which case the indexing requests will be forgotten, or indexing may simply fail.

- by default or when set to `committed`, the transaction commit will return as soon as the index changes are committed to disk.

  This generally means that at the very least that the backend validated the index changes, took appropriate measures to be able to recover the changes in the event of a crash, and confirmed to Hibernate Search it did so (e.g. for Elasticsearch, Hibernate Search received a successful response to the HTTP request).

  This strategy offers no guarantee as to whether indexed documents are searchable: the backend may delay indexing in order to improve performance, meaning a search query executed immediately after Hibernate ORM returns from the transaction may return outdated information.

  This is true in particular with the Elasticsearch backend, which is near-real-time by default.

- when set to `searchable`, the transaction commit will return as soon as the index changes are committed to disk and the relevant documents are searchable. The backend will be told to make the documents searchable as soon as possible.

  Depending on the backend and its configuration, this strategy may lead to poor indexing throughput, because the backend may not be optimized for frequent, on-demand index refreshes.

  That is why this strategy is only recommended if you know your backend is optimized for it (for example this is true for the default configuration of the Lucene backend, but not for the Elasticsearch backend), or for integration tests.

While the above configuration property defines a default, it is possible to override this default on a
particular session by calling `SearchSession#setAutomaticIndexingSynchronizationStrategy` and passing a different strategy. The built-in strategies can be retrieved by calling `AutomaticIndexingSynchronizationStrategy.queued()`, `AutomaticIndexingSynchronizationStrategy.committed()` or `AutomaticIndexingSynchronizationStrategy.searchable()`, but you can also define a custom strategy.

**Example 18. Overriding the automatic indexing synchronization strategy**

```java
SearchSession searchSession = Search.getSearchSession(entityManager); ①
searchSession.setAutomaticIndexingSynchronizationStrategy(
    AutomaticIndexingSynchronizationStrategy.searchable()
); ②
entityManager.getTransaction().begin();
try {
    Book book = entityManager.find(Book.class, 1);
    book.setTitle(book.getTitle() + " (2nd edition)" ); ③
    entityManager.getTransaction().commit(); ④
} catch (RuntimeException e) {
    entityManager.getTransaction().rollback();
}
List<Book> result = searchSession.search(Book.class)
    .predicate(f -> f.match().onField("title" ).matching("2nd edition") )
    .fetchHits(); ⑤
```

① Obtain the search session, which by default uses the synchronization strategy configured in properties.

② Override the synchronization strategy.

③ Change an entity.

④ Commit the changes, triggering reindexing.

⑤ The overridden strategy guarantees that the modified book will be present in these results, even though the query was executed just after the transaction commit.

### 5.3.2. Explicit indexing

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.

### Rebuilding the whole index

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.
5.4. Search query

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.

5.4.1. Concept

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.

5.4.2. Sort

By default, query results are sorted by relevance. Other sorts, including the sort by field value, can be configured when building the search query:

Example 19. Using custom sorts

```java
SearchSession searchSession = Search.getSearchSession( entityManager );
List<Book> result = searchSession.search( Book.class )
    .predicate( f -> f.matchAll() )
    .sort( f -> f.byField( "pageCount" ).desc() )
    .then().byField( "title_sort" )
    .fetchHits();
```

① Start building the query as usual.

② Mention that the results of the query are expected to be sorted on field "pageCount" in descending order, then (for those with the same page count) on field "title_sort" in ascending order. If the field does not exist or cannot be sorted on, an exception will be thrown.

③ The results are sorted according to instructions.

Or alternatively, if you don't want to use lambdas:
**Example 20. Using custom sorts - object-based syntax**

```java
SearchSession searchSession = Search.getSearchSession( entityManager );
SearchScope<Book> scope = searchSession.scope( Book.class );

List<Book> result = scope.search()
    .predicate( scope.predicate().matchAll().toPredicate() )
    .sort()
        .byField( "pageCount" ).desc()
        .then().byField( "title_sort" )
        .toSort()
    )
    .fetchHits();
```

There are a few constraints regarding sorts by field. In particular, in order for a field to be "sortable", it must be marked as such in the mapping, so that the correct data structures are available in the index.

The sort DSL offers more sort types, and multiple options for each type of sort. To learn more about the field sort, and all the other types of sort, refer to Sort DSL.

**5.4.3. Projection**

For some use cases, you only need the query to return a small subset of the data contained in your domain object. In these cases, returning managed entities and extracting data from these entities may be overkill: extracting the data from the index itself would avoid the database round-trip.

Projections do just that: they allow the query to return something more precise than just "the matching entities". Projections can be configured when building the search query:

**Example 21. Using projections to extract data from the index**

```java
SearchSession searchSession = Search.getSearchSession( entityManager );

List<String> result = searchSession.search( Book.class )
    .asProjection( f -> f.field( "title", String.class ) )
    .predicate( f -> f.matchAll() )
    .fetchHits();
```

1. Start building the query as usual.
2. Mention that the expected result of the query is a projection on field "title", of type String. If that type is not appropriate or if the field does not exist, an exception will be thrown.
3. The query is type-safe and will return results of the expected type.

Or alternatively, if you don't want to use lambdas:
Example 22. Using projections to extract data from the index - lambda syntax

```java
SearchSession searchSession = Search.getSearchSession(entityManager);
SearchScope<Book> scope = searchSession.scope(Book.class);
List<String> result = scope.search()
    .asProjection( scope.projection().field("title", String.class).toProjection() )
    .predicate( scope.predicate().matchAll().toPredicate() )
    .fetchHits();
```

There are a few constraints regarding field projections. In particular, in order for a field to be "projectable", it must be marked as such in the mapping, so that it is correctly stored in the index.

While field projections are certainly the most common, they are not the only type of projection. Other projections allow to compose custom beans containing extracted data, get references to the extracted documents or the corresponding entities, or get information about the search query itself (score, ...).

The following example shows how to retrieve the managed entity corresponding to each matched document along with the score of that document, and wraps this information into a custom bean:

Example 23. Using advanced projection types

```java
public class MyEntityAndScoreBean<T> {
    public final T entity;
    public final float score;
    public MyEntityAndScoreBean(T entity, float score) {
        this.entity = entity;
        this.score = score;
    }
}
```

```java
SearchSession searchSession = Search.getSearchSession(entityManager);
List<MyEntityAndScoreBean<Book>> result = searchSession.search(Book.class)
    .asProjection( f -> f.composite(
        MyEntityAndScoreBean::new,
        f.entity(),
        f.score()
    ) )
    .predicate( f -> f.matchAll() )
    .fetchHits();
```

The sort DSL offers more projection types, and multiple options for each type of projection. To learn more about the field projection, and all the other types of projection, refer to Projection DSL.
Chapter 6. Search DSLs

6.1. Predicate DSL

6.2. Sort DSL

6.3. Projection DSL

6.4. Type of arguments passed to the DSL

Some predicates, such as the `match` predicate or the `range` predicate, require a parameter of type `Object` at some point (`matching(Object)`, `above(Object)`, ...). Similarly, it is possible to pass an argument of type `Object` in the sort DSL when defining the behavior for missing values (`onMissingValue().use(Object)`).

These methods do not actually accept any object, and will throw an exception when passed an argument with the wrong type.

Generally the expected type of this argument should be rather obvious: for example if you created a field by mapping an `Integer` property, then an `Integer` value will be expected when building a predicate; if you mapped a `java.time.LocalDate`, then a `java.time.LocalDate` will be expected, etc.
Things get a little more complex if you start defining and using custom bridges. You will then have properties of type A mapped to an index field of type B. What should you pass to the DSL? To answer that question, we need to understand DSL converters.

DSL converters are a feature of Hibernate Search that allows the DSL to accept arguments that match the type of the indexed property, instead of the type of the underlying index field.

Each custom bridge has the possibility to define a DSL converter for the index fields it populates. When it does, every time that field is mentioned in the predicate DSL, Hibernate Search will use that DSL converter to convert the value passed to the DSL to a value that the backend understands.

For example, let’s imagine an AuthenticationEvent entity with an outcome property of type AuthenticationOutcome. This AuthenticationOutcome type is an enum. We index the AuthenticationEvent entity and its outcome property in order to allow users to find events by their outcome.

The default bridge for enums puts the result of Enum.name() into a String field. However, this default bridge also defines a DSL converter under the hood. As a result, any call to the DSL will be expected to pass an AuthenticationOutcome instance:

```
List<AuthenticationEvent> result = searchSession.search( AuthenticationEvent.class )
    .predicate( f -> f.match().onField( "outcome" )
        .matching( AuthenticationOutcome.INVALID_PASSWORD ) )
    .fetchHits();
```

This is handy, and especially appropriate if users are asked to select an outcome in a list of choices. But what if we want users to type in some words instead, i.e. what if we want full-text search on the outcome field? Then we will not have an AuthenticationOutcome instance to pass to the DSL, only a String...

In that case, we will first need to assign some text to each enum. This can be achieved by defining a custom ValueBridge<AuthenticationOutcome, String> and applying it to the outcome property so as to index a textual description of the outcome, instead of the default Enum#name().

Then, we will need to tell Hibernate Search that the value passed to the DSL should not be passed to the DSL converter, but should be assumed to match the type of the index field directly (in this case, String). To that end, one can simply use the variant of the matching method that accepts a DslConverter parameter, and pass DslConverter.DISABLED:
Example 25. Disabling the DSL converter

```java
List<AuthenticationEvent> result = searchSession.search(AuthenticationEvent.class)
    .predicate(f -> f.match().onField("outcome")
        .matching("Invalid password", DslConverter.DISABLED))
    .fetchHits();
```

All methods that apply DSL converters offer a variant that accepts a `DslConverter` parameter: `matching, from, to, above, below,...`

A DSL converter is always automatically generated for value bridges. However, more complex bridges will require explicit configuration.

See Type bridges and property bridges for more information.

6.5. Type of projected values

Generally the type of values returned by projections argument should be rather obvious: for example if you created a field by mapping an `Integer` property, then an `Integer` value will be returned when projecting; if you mapped a `java.time.LocalDate`, then a `java.time.LocalDate` will be returned, etc.

Things get a little more complex if you start defining and using custom bridges. You will then have properties of type `A` mapped to an index field of type `B`. What will be returned by projections? To answer that question, we need to understand projection converters.

Projection converters are a feature of Hibernate Search that allows the projections to return values that match the type of the indexed property, instead of the type of the underlying index field.

Each custom bridge has the possibility to define a projection converter for the index fields it populates. When it does, every time that field is projected on, Hibernate Search will use that projection converter to convert the projected value returned by the index.

For example, let's imagine an `Order` entity with a `status` property of type `OrderStatus`. This `OrderStatus` type is an enum. We index the `Order` entity and its `status` property.

The default bridge for enums puts the result of `Enum.name()` into a `String` field. However, this default bridge also defines a projection converter. As a result, any projection on the `status` field will return an `OrderStatus` instance:
Example 26. Transparent conversion of projections

```java
List<OrderStatus> result = searchSession.search(Order.class)
    .asProjection(f -> f.field("status", OrderStatus.class))
    .predicate(f -> f.matchAll())
    .fetchHits();
```

This is probably what you want in general. But in some cases, you may want to disable this conversion and return the index value instead (i.e. the value of `Enum.name()`).

In that case, we will need to tell Hibernate Search that the value returned by the backend should not be passed to the projection converter. To that end, one can simply use the variant of the `field` method that accepts a `ProjectionConverter` parameter, and pass `ProjectionConverter.DISABLED`:

Example 27. Disabling the projection converter

```java
List<String> result = searchSession.search(Order.class)
    .asProjection(f -> f.field("status", String.class, ProjectionConverter.DISABLED))
    .predicate(f -> f.matchAll())
    .fetchHits();
```

Projection converters must be configured explicitly in custom bridges.

See Value bridges and Type bridges and property bridges for more information.

6.6. Targeting multiple fields

Sometimes a predicate/sort/projection targets multiple field, which may have conflicting definitions:

- when multiple field names are passed to the `onFields` method in the predicate DSL (each field has its own definition);
- or when the search query targets multiple indexes (each index has its own definition of each field).

In such cases, the definition of the targeted fields is expected to be compatible. For example targeting an `Integer` field and a `java.time.LocalDate` field in the same `match` predicate will not work, because you won't be able to pass a non-null argument to the `matching(Object)` method that is both an `Integer` and a `java.time.LocalDate`.

If you are looking for a simple rule of thumb, here it is: if the indexed properties do not have the same type, or are mapped differently, the corresponding fields are probably not going to be compatible.

However, if you're interested in the details, Hibernate Search is a bit more flexible than that.
There are three different constraints when it comes to field compatibility:

1. The fields must be "encoded" in a compatible way. This means the backend must use the same representation for the two fields, for example they are both `Integer`, or they are both `BigDecimal` with the same decimal scale, or they are both `LocalDate` with the same date format, etc.

2. The fields must have a compatible DSL converter (for predicates and sorts) or projection converter (for projections).

3. For full-text predicates, the fields must have a compatible analyzer.

The following sections describe all the possible incompatibilities, and how to solve them.

### 6.6.1. Incompatible codec

In a search query targeting multiple indexes, if a field is encoded differently in each index, you cannot apply predicates, sorts or projections on that field.

Encoding is not only about the field type, such as `LocalDate` or `BigDecimal`. Some codecs are parameterized and two codecs with different parameters will often be considered incompatible. Examples of parameters include the format for temporal types or the decimal scale for `BigDecimal` and `BigInteger`.

In that case, your only option is to change your mapping to avoid the conflict:

1. rename the field in one index
2. OR change the field type in one index
3. OR if the problem is simply different codec parameters (date format, decimal scale, …), align the value of these parameters in one index with the other index.

If you choose to rename the field in one index, you will still be able to apply a similar predicate to the two fields in a single query: you will have to create one predicate per field and combine them with a boolean junction.

### 6.6.2. Incompatible DSL converters

Incompatible DSL converters are only a problem when you need to pass an argument to the DSL in certain methods: `matching(Object)`/`to(Object)`/`above(Object)`/`below(Object)`/etc. in the predicate DSL, or `onMissingValue().use(Object)` in the sort DSL.

If two fields encoded in a compatible way (for example both as `String`), but that have different DSL converters (for example the first one converts from `String` to `String`, but the second one converts from `Integer` to `String`), you can still use these methods, but you will need to disable the DSL converter as explained in Type of arguments passed to the DSL: you will just pass the "index" value to
the DSL (using the same example, a String).

6.6.3. Incompatible projection converters

If, in a search query targeting multiple indexes, a field is encoded in a compatible way in every indexes (for example both as String), but that has a different projection converters (for example the first one converts from String to String, but the second one converts from String to Integer), you can still project on this field, but you will need to disable the projection converter as explained in Type of projected values: the projection will return the "index", unconverted value (using the same example, a String).

6.6.4. Incompatible analyzer

Incompatible analyzers are only a problem with full-text predicates: match predicate on a text field, phrase predicate, simple query string predicate, ...

If two fields encoded in a compatible way (for example both as String), but that have different analyzers, you can still use these predicates, but you will need to explicitly configure the predicate to either set the search analyzer to an analyzer of your choosing with .analyzer(analyzerName), or skip analysis completely with .skipAnalysis().

See Predicate DSL for more information about how to create predicates and about the available options.
Chapter 7. Lucene backend

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.

7.1. General configuration

In order to define a Lucene backend, the `hibernate.search.backends.<backend name>.type` property must be set to `lucene`.

All other configuration properties are optional, but the defaults might not suit everyone. In particular, you might want to set the location of your indexes in the filesystem. See below for the details of every configuration property.

7.1.1. Directory provider

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.

7.1.2. Index format compatibility

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

While an index format incompatibility is a rare event, it can happen more often that Lucene’s Analyzer implementations might slightly change its behavior. This can lead to some documents not matching anymore, even though they used to.

To avoid this analyzer incompatibility, Hibernate Search allows to configure to which version of Lucene the analyzers and other Lucene classes should conform their behavior.

This configuration property is set at the backend level:

```
hibernate.search.backends.<backend-name>.lucene_version = LUCENE_47
```

Depending on the specific version of Lucene you’re using, you might have different options available: see [org.apache.lucene.util.Version](http://lucene.apache.org/core/) contained in [lucene-core.jar](http://lucene.apache.org/core/) for a list of allowed options.
When this option is not set, Hibernate Search will instruct Lucene to use the latest version, which is usually the best option for new projects. Still, it’s recommended to define the version you’re using explicitly in the configuration, so that when you happen to upgrade, Lucene the analyzers will not change behavior. You can then choose to update this value at a later time, for example when you have the chance to rebuild the index from scratch.

The setting will be applied consistently when using Hibernate Search APIs, but if you are also making use of Lucene bypassing Hibernate Search (for example when instantiating an Analyzer yourself), make sure to use the same value.

7.1.3. Other configuration properties

Other configuration properties are mentioned in the relevant parts of this documentation. You can find a full reference of available properties in the Hibernate Search javadoc:

- org.hibernate.search.backend.lucene.cfg.LuceneBackendSettings.
- org.hibernate.search.backend.lucene.cfg.LuceneIndexSettings.

7.2. Field types

Some types are not supported directly by the Elasticsearch backend, but will work anyway because they are "bridged" by the mapper. For example a java.util.Date in your entity model is "bridged" to java.time.Instant, which is supported by the Elasticsearch backend. See Built-in value bridges for more information.

Table 4. Field types supported by the Lucene backend

<table>
<thead>
<tr>
<th>Field type</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.String</td>
</tr>
<tr>
<td>java.lang.Byte</td>
</tr>
<tr>
<td>java.lang.Short</td>
</tr>
<tr>
<td>java.lang.Integer</td>
</tr>
<tr>
<td>java.lang.Long</td>
</tr>
<tr>
<td>java.lang.Double</td>
</tr>
<tr>
<td>java.lang.Float</td>
</tr>
<tr>
<td>java.lang.Boolean</td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
</tr>
<tr>
<td>java.math.BigInteger</td>
</tr>
<tr>
<td>java.time.Instant</td>
</tr>
<tr>
<td>java.time.LocalDate</td>
</tr>
<tr>
<td>java.time.LocalTime</td>
</tr>
</tbody>
</table>

50
Date/time types do not support the whole range of years that can be represented in `java.time` types:

- `java.time` can represent years ranging from \(-999.999.999\) to \(999.999.999\).
- The Lucene backend supports dates ranging from year \(-292.275.054\) to year \(292.278.993\).

### 7.3. Analysis

This section is currently incomplete. A decent introduction is included in the getting started guide: see [Analysis](#).

To configure analysis in a Lucene backend, you will need to:

- Implement a bean that implements the `org.hibernate.search.backend.lucene.analysis.LuceneAnalysisConfigurer` interface.
- Configure your backend to use that bean by setting the configuration property `hibernate.search.backends.<backend name>.analysis_configurer` to a `bean reference` pointing to your bean.

To know which character filters, tokenizers and token filters are available, either browse the Lucene Javadoc or read the corresponding section on the [Solr Wiki](#).
Why the reference to the Apache Solr wiki for Lucene?

The analyzer factory framework was originally created in the Apache Solr project. Most of these implementations have been moved to Apache Lucene, but the documentation for these additional analyzers can still be found in the Solr Wiki. You might find other documentation referring to the "Solr Analyzer Framework"; just remember you don't need to depend on Apache Solr anymore: the required classes are part of the core Lucene distribution.

7.4. Multi-tenancy

Multi-tenancy is supported and handled transparently, according to the tenant ID defined in the current session:

- documents will be indexed with the appropriate values, allowing later filtering;
- queries will filter results appropriately.

However, multi-tenancy must be enabled explicitly. To do so, set the `hibernate.search.backends.<backend name>.multi_tenancy_strategy` property:

- to none for single-tenancy;
- to discriminator for discriminator-based multi-tenancy: adds a "tenant ID" field to every document.
Chapter 8. Elasticsearch backend

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.

8.1. Compatibility

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.

Hibernate Search expects an Elasticsearch cluster running version 5.6, 6.x or 7.x. The version running on your cluster will be automatically detected on startup, and Hibernate Search will adapt based on the detected version; see Version for details.

The targeted version is mostly transparent to Hibernate Search users, but there are a few differences in how Hibernate Search behaves depending on the Elasticsearch version that may affect you. The following table details those differences.

<table>
<thead>
<tr>
<th>Feature</th>
<th>5.6</th>
<th>6.x</th>
<th>7.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formats for date fields in the Elasticsearch schema</td>
<td>Formats changed in ES 7, see Field types</td>
<td></td>
<td></td>
</tr>
<tr>
<td>indexNullAs on geo_point fields</td>
<td>Not available</td>
<td>Available</td>
<td></td>
</tr>
</tbody>
</table>

8.1.1. Upgrading Elasticsearch

When upgrading your Elasticsearch cluster, some administrative tasks are still required on your cluster: Hibernate Search will not take care of those.

On top of that, there are some fundamental differences between some versions of Elasticsearch: for example date formats changed in Elasticsearch 7, meaning the schema defined in Elasticsearch 6 may not be compatible with the one expected by Hibernate Search for Elasticsearch 7.

In such cases, the easiest way to upgrade is to delete your indexes manually, make Hibernate Search re-create the indexes along with their schema, and reindex your data.

8.2. General configuration

In order to define an Elasticsearch backend, the hibernate.search.backends.<backend name>.type property must be set to elasticsearch.
All other configuration properties are optional, but the defaults might not suit everyone. In particular your production Elasticsearch cluster is probably not reachable at http://localhost:9200. See below for the details of every configuration property.

8.2.1. Client properties

**Hosts**

```properties
hibernate.search.backends.<backend name>.hosts = http://localhost:9200 (default)
```

The Elasticsearch host (or hosts) to send indexing requests and search queries to. Also defines the scheme (http or https) and port for each host.

Expects a String representing an URI such as http://localhost or https://es.mycompany.com:4400, or a String containing multiple such URIs separated by whitespace characters, or a Collection<String> containing such URIs.

**HTTP authentication**

```properties
hibernate.search.backends.<backend name>.username = ironman (default is empty)
hibernate.search.backends.<backend name>.password = j@rv1s (default is empty)
```

The username and password to send when connecting to the Elasticsearch servers.

⚠️ If you use HTTP instead of HTTPS in any of the Elasticsearch host URLs (see above), your password will be transmitted in clear text over the network.

**Timeouts**

```properties
hibernate.search.backends.<backend name>.request_timeout = 60000 (default)
hibernate.search.backends.<backend name>.connection_timeout = 3000 (default)
hibernate.search.backends.<backend name>.read_timeout = 60000 (default)
```

- **request_timeout** defines the timeout when executing a request. This includes the time needed to establish a connection, send the request and read the response.
- **connection_timeout** defines the timeout when establishing a connection.
- **read_timeout** defines the timeout when reading a response.

These properties expect a positive integer value in milliseconds, such as 3000.

**Connections**

```properties
hibernate.search.backends.<backend name>.max_connections = 20 (default)
hibernate.search.backends.<backend name>.max_connections_per_route = 10 (default)
```

- **max_connections** defines maximum number of simultaneous connections to the
Elasticsearch cluster, all hosts taken together.

- **max_connections_per_route** defines maximum number of simultaneous connections to each host of the Elasticsearch cluster.

These properties expect a positive **integer value**, such as **20**.

### 8.2.2. Discovery

When using automatic discovery, the Elasticsearch client will periodically probe for new nodes in the cluster, and will add those to the host list (see **hosts** in Client properties).

Automatic discovery is controlled by the following properties:

- **discovery.enabled** defines whether the feature is enabled. Expects a boolean value.
- **discovery.refresh_interval** defines the interval between two executions of the automatic discovery. Expects a positive integer, in seconds.
- **discovery.default_scheme** defines the default scheme to use when connecting to automatically discovered nodes. Expects a String: either "http" or "https".

### 8.2.3. Version

Different versions of Elasticsearch expose slightly different APIs. As a result, Hibernate Search needs to be aware of the version of Elasticsearch it is talking to in order to generate correct HTTP requests.

By default, Hibernate Search will query the Elasticsearch cluster at boot time to know its version, and will infer the correct behavior to adopt.

Alternatively, you can tell Hibernate Search the Elasticsearch version to target. Hibernate Search will still query the Elasticsearch cluster to check that the configured version matches the actual version, but only after most of the metadata has been validated. This can be helpful when developing, in particular.

To configure the version, set the **hibernate.search.backends.<backend name>.version** property to a string following the format **x.y.z-qualifier**, where x, y and z are integers and **qualifier** is an optional string of word characters (alphabetic or _).

Incomplete versions are allowed in some cases, for example **7.0** or just **7**. This is not possible for all versions, however. For example, Elasticsearch 5 is only supported from 5.6.0 onwards, so **5** is not an acceptable value and will trigger an exception: **5.6** must be used instead.
8.2.4. Logging

The `hibernate.search.backends.<backend name>.log.json_pretty_printing` boolean property defines whether JSON included in logs should be pretty-printed (indented, with line breaks). It defaults to `false`.

8.2.5. Authentication on Amazon Web Services

The Hibernate Search Elasticsearch backend, once configured, will work just fine in most setups. However, if you need to use Amazon’s managed Elasticsearch service, you will find it requires a proprietary authentication method: request signing.

While request signing is not supported by default, you can enable it with an additional dependency and a little bit of configuration.

You will need to add this dependency:

```xml
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-search-elasticsearch-aws</artifactId>
  <version>6.0.0.Alpha6</version>
</dependency>
```

With that dependency in your classpath, Hibernate Search will be able to understand the following configuration properties.

```
hibernate.search.backends.<backend name>.aws.signing.enabled = false (default)
hibernate.search.backends.<backend name>.aws.signing.access_key = AKIDEXAMPLE (no default)
hibernate.search.backends.<backend name>.aws.signing.secret_key = wJalrXUtnFEMI/K7MDENG+bPxRfiCYEXAMPLEKEY (no default)
hibernate.search.backends.<backend name>.aws.signing.region = us-east-1 (no default)
```

- `aws.signing.enabled` defines whether request signing is enabled. Expects a boolean value.
- `aws.signing.access_key` defines the access key. Expects a string value. This property has no default and must be provided for the AWS authentication to work.
- `aws.signing.secret_key` defines the secret key. Expects a string value. This property has no default and must be provided for the AWS authentication to work.
- `aws.signing.region` defines the AWS region. Expects a string value. This property has no default and must be provided for the AWS authentication to work.

Should you need help with finding the correct values for these properties, please refer to the AWS documentation related to security credentials and regions.
8.2.6. Other configuration properties

Other configuration properties are mentioned in the relevant parts of this documentation. You can find a full reference of available properties in the Hibernate Search javadoc:

- `org.hibernate.search.backend.elasticsearch.cfg.ElasticsearchBackendSettings`
- `org.hibernate.search.backend.elasticsearch.cfg.ElasticsearchIndexSettings`

8.2.7. Configuration of the Elasticsearch cluster

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.

8.3. Index lifecycle

Hibernate Search includes a feature named "index lifecycle management", where it will automatically create, validate, update, or drop an index on startup or shutdown.

`hibernate.search.default.elasticsearch.index_schema_management_strategy create` (default)

The following strategies are available:

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>The index, its mappings and the analyzer definitions will not be created, deleted nor altered. Hibernate Search will not even check that the index already exists.</td>
</tr>
<tr>
<td>validate</td>
<td>The index, its existing mappings and analyzer definitions will be checked to be compatible with the mapping defined in your application. The index, its mappings and analyzer definitions will not be created, deleted nor altered.</td>
</tr>
<tr>
<td>update</td>
<td>The index, its mappings and analyzer definitions will be created, existing mappings will be updated if there are no conflicts. Caution: if analyzer definitions have to be updated, the index will be closed automatically during the update.</td>
</tr>
<tr>
<td>create</td>
<td><strong>The default</strong>: an existing index will not be altered, a missing index will be created along with their mappings and analyzer definitions.</td>
</tr>
<tr>
<td>drop-and-create</td>
<td>Indexes will be deleted if existing and then created along with their mappings and analyzer definitions to match the mapping defined in your application. This will delete all content from the indexes! Useful during development.</td>
</tr>
<tr>
<td>Value</td>
<td>Definition</td>
</tr>
<tr>
<td>--------------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>drop-and-create-and-drop</td>
<td>Similar to drop-and-create but will also delete the index at shutdown. Commonly used for tests.</td>
</tr>
</tbody>
</table>

Mapping validation is as permissive as possible. Fields or mappings that are unknown to Hibernate Search will be ignored, and settings that are more powerful than required will be deemed valid. For example, a field that is not marked as sortable in Hibernate Search but marked as "docvalues": true in Elasticsearch will be deemed valid.

One exception: date formats must match exactly the formats specified by Hibernate Search, due to implementation constraints.

You can fine-tune the strategy using the following properties:

```bash
hibernate.search.backends.<backend name>.indexes.<index name>.lifecycle.strategy create (default)
hibernate.search.backends.<backend name>.indexes.<index name>.lifecycle.minimal_required_status green (default)
hibernate.search.backends.<backend name>.indexes.<index name>.lifecycle.minimal_required_status_wait_timeout 10000 (default)
# OR
hibernate.search.backends.<backend name>.index_defaults.lifecycle.strategy create (default)
hibernate.search.backends.<backend name>.index_defaults.lifecycle.minimal_required_status green (default)
hibernate.search.backends.<backend name>.index_defaults.lifecycle.minimal_required_status_wait_timeout 10000 (default)
```

The properties minimal_required_status and minimal_required_status_wait_timeout define the minimal required status of the index on startup, before Hibernate Search can start using it, and the maximum time to wait for this status, as an integer value in milliseconds. These properties are ignored when the none strategy is selected, because the index will not be checked on startup (see above).

Since Elasticsearch on Amazon Web Services (AWS) does not support the _close /_open operations, the update strategy will fail when trying to update analyzer definitions on an AWS Elasticsearch cluster.

The only workaround is to avoid the update strategy on AWS.
Strategies in production environments

It is strongly recommended to use either none or validate in a production environment.

The alternatives drop-and-create and drop-and-create-and-drop are obviously unsuitable in this context unless you want to reindex everything upon every startup, and update may leave your mapping half-updated in case of conflict.

To be precise, if your mapping changed in an incompatible way, such as a field having its type changed, updating the mapping may be impossible without manual intervention. In this case, the update strategy will prevent Hibernate Search from starting, but it may already have successfully updated the mappings for another index, making a rollback difficult.

When updating analyzer definitions Hibernate Search will temporarily stop the affected indexes during the update. This means the update strategy should be used with caution when multiple clients use Elasticsearch indexes managed by Hibernate Search: those clients should be synchronized in such a way that while Hibernate Search is starting, no other client needs to access the index.

For these reasons, migrating your mapping on a live cluster should be carefully planned as part of the deployment process.

8.4. Field types

Some types are not supported directly by the Elasticsearch backend, but will work anyway because they are "bridged" by the mapper. For example a java.util.Date in your entity model is "bridged" to java.time.Instant, which is supported by the Elasticsearch backend. See Built-in value bridges for more information.

<table>
<thead>
<tr>
<th>Field type</th>
<th>Data type in Elasticsearch</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.String</td>
<td>text if an analyzer is defined, keyword otherwise</td>
</tr>
<tr>
<td>java.lang.Byte</td>
<td>byte</td>
</tr>
<tr>
<td>java.lang.Short</td>
<td>short</td>
</tr>
<tr>
<td>java.lang.Integer</td>
<td>integer</td>
</tr>
<tr>
<td>java.lang.Long</td>
<td>long</td>
</tr>
<tr>
<td>java.lang.Double</td>
<td>double</td>
</tr>
<tr>
<td>java.lang.Float</td>
<td>float</td>
</tr>
<tr>
<td>Field type</td>
<td>Data type in Elasticsearch</td>
</tr>
<tr>
<td>--------------------------------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>java.lang.Boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td>Not supported yet; see HSEARCH-3487</td>
</tr>
<tr>
<td>java.math.BigInteger</td>
<td>Not supported yet; see HSEARCH-3487</td>
</tr>
<tr>
<td>java.time.Instant</td>
<td>date with format uuuu-MM-dd’T’HH:mm:ss.SSSSSSSSSZZZZZ (ES7 and above) or yyyy-MM-dd’T’HH:mm:ss.SSS'Z'</td>
</tr>
<tr>
<td>java.time.LocalDate</td>
<td>date with format uuuu-MM-dd (ES7 and above) or yyyy-MM-dd</td>
</tr>
<tr>
<td>java.time.LocalTime</td>
<td>date with format HH:mm:ss.SSSSSSSSS (ES7 and above) or HH:mm:ss.SSS</td>
</tr>
<tr>
<td>java.time.ZonedDateTime</td>
<td>date with format uuuu-MM-dd’T’HH:mm:ss.SSSSSSSSS (ES7 and above) or yyyy-MM-dd’T’HH:mm:ss.SSS</td>
</tr>
<tr>
<td>java.time.ZonedDateTime</td>
<td>date with format uuuu-MM-dd’T’HH:mm:ss.SSSSSSSSSZZZZ'[‘VV’]’ (ES7 and above) or yyyy-MM-dd’T’HH:mm:ss.SSSZZ'[‘ZZZ’]’</td>
</tr>
<tr>
<td>java.time.OffsetDateTime</td>
<td>date with format uuuu-MM-dd’T’HH:mm:ss.SSSSSSSSSZZZZ (ES7 and above) or yyyy-MM-dd’T’HH:mm:ss.SSSZZ (ES6 and below)</td>
</tr>
<tr>
<td>java.time.OffsetTime</td>
<td>date with format HH:mm:ss.SSSSSSSSSZZZZ (ES7 and above) or HH:mm:ss.SSSZZ (ES6 and below)</td>
</tr>
<tr>
<td>java.time.Year</td>
<td>date with format uuuu (ES7 and above) or yyyy</td>
</tr>
<tr>
<td>java.time.YearMonth</td>
<td>date with format uuuu-MM (ES7 and above) or yyyy-MM</td>
</tr>
<tr>
<td>java.time.MonthDay</td>
<td>date with format uuuu-MM-dd (ES7 and above) or yyyy-MM-dd (ES6 and below). The year is always set to 0.</td>
</tr>
<tr>
<td>org.hibernate.search.engine.spatial.GeoPoint</td>
<td>geo_point</td>
</tr>
</tbody>
</table>
The Elasticsearch date type does not support the whole range of years that can be modeled in java.time types:

- java.time supports years ranging from \(-999.999.999\) to \(999.999.999\).
- Elasticsearch supports years ranging from \(-292.275.054\) to \(292.278.993\).

8.4.1. Analysis

This section is currently incomplete. A decent introduction is included in the getting started guide: see Analysis.

To configure analysis in an Elasticsearch backend, you will need to:

- Implement a bean that implements the org.hibernate.search.backend.elasticsearch.analysis.ElasticsearchAnalysisConfigurer interface.
- Configure your backend to use that bean by setting the configuration property hibernate.search.backends.<backend name>.analysis_configurer to a bean reference pointing to your bean.

To know which character filters, tokenizers and token filters are available, refer to the documentation:

- If you want to use a built-in analyzer and not create your own: analyzers;
- If you want to define your own analyzer: character filters, tokenizers, token filters.

8.5. Multi-tenancy

Multi-tenancy is supported and handled transparently, according to the tenant ID defined in the current session:

- documents will be indexed with the appropriate values, allowing later filtering;
- queries will filter results appropriately.

However, multi-tenancy must be enabled explicitly. To do so, set the hibernate.search.backends.<backend name>.multi_tenancy_strategy property:

- to none (the default) for single-tenancy;
- to discriminator for discriminator-based multi-tenancy: adds a "tenant ID" field to every document.
Chapter 9. Index Optimization

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.
Chapter 10. Monitoring

⚠️ This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.
Chapter 11. Advanced features

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.
Chapter 12. Internals of Hibernate Search

This section is intended for new Hibernate Search contributors looking for an introduction to how Hibernate Search works.

Knowledge of the Hibernate Search APIs and how to use them is a requirement to understand this section.

12.1. General overview

This section focuses on describing what the different parts of Hibernate Search are at a high level and how they interact with each other.

Hibernate Search internals are split into three parts:

**Backends**

The backends are where "things get done". They implement common indexing and searching interfaces for use by the mappers through "index managers", each providing access to one index. Examples include the Lucene backend, delegating to the Lucene library, and the Elasticsearch backend, delegating to a remote Elasticsearch cluster.

The word "backend" may refer either to a whole Maven module (e.g. "the Elasticsearch backend") or to a single, central class in this module (e.g. the ElasticsearchBackend class implementing the Backend interface), depending on context.

**Mappers**

Mappers are what users see. They "map" the user model to an index, and provide APIs consistent with the user model to perform indexing and searching. For instance the POJO mapper provides APIs that allow to index getters and fields of Java objects according to a configuration provided at boot time.

The word "mapper" may refer either to a whole Maven module (e.g. "the POJO mapper") or to a single, central class in this module (e.g. the PojoMapper class implementing the Mapper interface), depending on context.

**Engine**

The engine defines some APIs, a lot of SPIs, and implements the code needed to start and stop Hibernate Search, and to "glue" mappers and backends together during bootstrap.

Those parts are strictly separated in order to allow to use them interchangeably. For instance the Elasticsearch backend could be used indifferently with a POJO mapper or a JSON mapper, and we will
only have to implement the backend once.

Here is an example of what Hibernate Search would look like at runtime, from a high level perspective:

A "mapping" is a very coarse-grained term, here. A single POJO mapping, for instance, may support many indexed entities.
The mapping was provided, during bootstrap, with several "index managers", each exposing SPIS allowing to search and index. The purpose of the mapping is to transform calls to their APIs into call to the index manager SPIS. This requires to perform conversions of:

- indexed data: the data manipulated by the mapping may take any form, but it has to be converted to a document accepted by the index manager.
- index references, e.g. a search query targeting classes MyEntity and MyOtherEntity must instead target index manager 1 and index manager 2.
- document references, e.g. a search query executed at the index manager level may return "document 1 in index 1 matched the query", but the user wants to see "entity 1 of type MyEntity matched the query".

The purpose of the SearchIntegration is mainly to keep track of every resource (mapping or backend) created at bootstrap, and allow to close it all from a single call.

Finally, the purpose of the backend and its index managers is to execute the actual work and return results when relevant.

The architecture is able to support more complex user configurations. The example below shows a Hibernate Search instance with two mappings: a POJO mapping and a JSON mapping.

The example is deliberately a bit contrived, in order to demonstrate some subtleties:
• There are two mappings in this example. Most setups will only configure one mapping, but it is
important to keep in mind there may be more. In particular, we anticipate that Infinispan may
need multiple different mappings in a single Hibernate Search instance, in order to handle the
multiple input types it accepts from its users.

• There are multiple backends in this example. Again, most setups will only ever configure one, but
there may be good reasons to use more. For instance if someone wants to index part of the
entities in one Elasticsearch cluster, and the other part in another cluster.

• Here, the two mappings each use one index manager from the same Elasticsearch backend. This is
currently possible, though whether there are valid uses cases for this remains to be determined,
mainly based on the Infinispan needs.

### 12.1.1. Bootstrap

Bootstrap starts by creating at least two components:

• The `SearchIntegrationBuilder`, which allows to setup all the mapper-independent
configuration: bean resolver, configuration property sources for the backends, ...

• At least one `MappingInitiator` instance, of a type provided by the mapper module, which will
register itself to the `SearchIntegrationBuilder`. From the point of view of the engine, it is a
callback that will come into play later.

The idea is that the `SearchIntegrationBuilder` will allow one or more initiators to provide
configuration about their mapping, in particular metadata about various "mappable" types (in short,
the types manipulated by the user). Then the builder will organize this metadata, check the
consistency to some extent, create backends and index manager builders as necessary, and then
provide the (organized) metadata back to the mapper module along with handles to index manager
builders so that it can start its own bootstrapping.

To sum up: the `SearchIntegrationBuilder` is a facilitator, allowing to start mapper bootstrapping
with everything that is necessary:

• engine services and components (`BuildContext`);

• configuration properties (`ConfigurationPropertySource`);

• organized metadata (`TypeMetadataContributorProvider`);

• one handle to the backend layer (`IndexManagerBuildingState`) for each indexed type.

All this is provided to the mapper through the `MappingInitiator` and `Mapper` interfaces.

Mapper bootstrapping is really up to the mapper module, but one thing that won’t change is what
mappers can do with the handles to the backend layer. These handles are instances of
`IndexManagerBuildingState` and each one represents an index manager being built. As the
mapper inspects the metadata, it will infer the required fields in the index, and will contribute this
information to the backend using the dedicated SPI: `IndexModelBindingContext`, `IndexSchemaElement`, `IndexSchemaFieldContext` are the most important parts.

All this information about the required fields and their options (field type, whether it's stored, how it is analyzed, ...) will be validated and will allow the backend to build an internal representation of the index schema, which will be used for various, backend-specific purposes, for example initializing a remote Elasticsearch index or inferring the required type of parameters to a range query on a given field.

### 12.1.2. Indexing

The entry point for indexing is specific to each mapper, and so are the upper levels of each mapper implementation. But at the lower levels, indexing in a mapper comes down to using the backend SPIs.

When indexing, the mapper must build a document that will be passed to the backend. This is done using document elements and index field references. During bootstrap, whenever the mapper declared a field, the backend returned a reference (see `IndexSchemaFieldTerminalContext#getReference`). In order to build a document, the mapper extracts data from an object to index, retrieves a document element from the backend, and pass the field reference along with the value to the document element, so that the value is added to the field.

The other part of indexing (or altering the index in any way) is to give an order to the index manager: "add this document", "delete this document", ... This is done through the `IndexWorkPlan` class. The mapper should create a work plan whenever it needs to execute a series of works.

`IndexWorkPlan` carries some context usually associated to a "session" in the JPA world, including the tenant identifier when using multi-tenancy, in particular. Thus the mapper should instantiate a new work plan whenever this context changes.

For now index-scoped operations such as flush, optimize, etc. are unavailable from work plans. HSEARCH-3305 will introduce APIs and SPIs for these.

### 12.1.3. Searching

Searching is a bit different from indexing, in that users are presented with APIs focused on the index rather than the mapped objects. The idea is that when you search, you will mainly target index fields, not properties of mapped objects (though they may happen to have the same name).

As a result, mapper APIs only define entry points for searching so as to offer more natural ways of defining the search scope and to provide additional settings. For example `PojoSearchManager#search` allows to define the search scope using the Java classes of mapped types instead of index names. But somewhere along the API calls, mappers end up exposing generic APIs, for instance `SearchQueryResultDefinitionContext` or `SearchPredicateContainerContext`. 
Those generic APIs are mostly implemented in the engine. The implementation itself relies on lower-level, less "user-focused" SPIs implemented by backends, such as SearchPredicateFactory or FieldSortBuilder.

Also, the SPIs implemented by backends allow mappers to inject a "loading context" (see SearchQueryBuilderFactory.asEntity) that will essentially transform document references into the entity that was initially indexed.

12.2. POJO mapper

What we call the POJO mapper is in fact an abstract basis for implementing mappers from Java objects to a full-text index. This module implements most of the necessary logic, and defines SPIs to implement the bits that are specific to each mapper.

There are currently only two implementations: the Hibernate ORM mapper, and the JavaBean mapper. The second one is mostly here to demonstrate that implementing a mapper that doesn't rely on Hibernate ORM is possible: we do not expect much real-life usage.

The following sections do not address everything in the POJO mapper, but instead focus on the more complex parts.

12.2.1. Representation of the POJO metamodel

The bootstrapping process of the POJO mapper relies heavily on the POJO metamodel to infer what will have to be done at runtime. Multiple constructs are used to represent this metamodel.

Models

PojoTypeModel, PojoPropertyModel and similar are at the root of everything. They are SPIs, to be implemented by the Hibernate ORM mapper for instance, and they provide basic information about mapped types: Java annotations, list of properties, type of each property, "handle" to access each property on an instance of this type, ...

Container value extractor paths

ContainerExtractorPath and BoundContainerExtractorPath both represent a list of ContainerExtractor to be applied to a property. They allow to represent what will have to be done to get from a property of type Map<String, List<MyEntity>> to a sequence of MyEntity, for example. The difference between the "bound" version and the other is that the "bound" version was applied to a POJO model, allowing to guarantee that it will work when applied to that model, and allowing to infer the type of extracted values. See ContainerExtractorBinder for more information.

Paths

POJO paths come in two flavors: PojoModelPath and BoundPojoModelPath. Each has a number of subtypes representing "nodes" in a path. The POJO paths represent how to get from a given
type to a given value, by accessing properties, extracting container values (see container value extractor paths above), and casting types. As for container value extractor paths, the difference between the "bound" version and the other is that the "bound" version was applied to a POJO model, allowing to guarantee that it will work when applied to that model (except for casts, obviously), and allowing to infer the type of extracted values.

Additional metadata

PojoTypeAdditionalMetadata, PojoPropertyAdditionalMetadata and PojoValueAdditionalMetadata allow to represent POJO metadata that would not typically be found in a "plain old Java object" without annotations. The metadata may come from various sources: Hibernate Search’s annotations, Hibernate Search’s programmatic API, or even from other metamodels such as Hibernate ORM’s. The "additional metadata" objects are a way to represent this metadata the same way, wherever it comes from. Examples of "additional metadata" include whether a given type is an entity type, property markers ("this property represents a latitude"), or information about inter-entity associations.

Model elements

PojoModelElement, PojoModelProperty and similar are representations of the POJO metamodel for use by Hibernate Search users in bridges. They are API, on contrary to PojoTypeModel et. al. which are SPI, but their implementation relies on both the POJO model and additional metadata. Their main purpose is to shield users from eventual changes in our SPIs, and to allow users to get "accessors" so that they can extract information from the bridge elements at runtime.

When retrieving accessors, users indirectly declare what parts of the POJO model they will extract and use in their bridge, and Hibernate Search actually makes use of this information (see Implicit reindexing resolvers).

12.2.2. Indexing processors

Indexing processors are the objects responsible for extracting data from a POJO and pushing it to a document.

Index processors are organized as trees, each node being an implementation of PojoIndexingProcessor. The POJO mapper assigns one tree to each indexed entity type.

Here are the main types of nodes:

- **PojoIndexingProcessorTypeNode**: A node representing a POJO type (a Java class).
- **PojoIndexingProcessorPropertyNode**: A node representing a POJO property.
- **PojoIndexingProcessorContainerElementNode**: A node representing elements in a container (List, Optional, ...).
At runtime, the root node will be passed the entity to index and a handle to the document being built. Then each node will "process" its input, i.e. perform one (or more) of the following:

- extract data from the Java object passed as input: extract the value of a property, the elements of a list, ...
- pass the extracted data along with the handle to the document being built to a user-configured bridge, which will add fields to the document.
- pass the extracted data along with the handle to the document being built to a nested node, which will in turn "process" its input.

For nodes representing an indexed embedded, some more work is involved to add an object field to the document and ensure nested nodes add fields to that object field instead of the root document. But this is specific to indexed embedded: manipulation of the document is generally only performed by bridges.

This representation is flexible enough to allow it to represent almost any mapping, simply by defining the appropriate node types and ensuring the indexing processor tree is built correctly, yet explicit enough to not require any metadata lookup at runtime.

Indexing processors are logged at the debug level during bootstrap. Enable this level of logging for the Hibernate Search classes if you want to understand the indexing processor tree that was generated for a given mapping.

**Bootstrap**

For each indexed type, the building process consists in creating a root `PojoIndexingProcessorTypeNode` builder, and applying metadata contributors to this builder (see Bootstrap), creating nested builders as the need arises (when a metadata contributor mentions a POJO property, for instance). Whenever an `@IndexedEmbedded` is found, the process is simply applied recursively on a type node created as a child of the `@IndexedEmbedded` property node.

As an example, let's consider the following mapped model:
The class `IndexedEntityClass` is indexed. It has two mapped fields, plus an indexed-embedded on a property named `embedded` of type `EmbeddedEntityClass`. The class `EmbeddedEntityClass` has one mapped field, plus an indexed-embedded on a property named `secondLevelEmbedded` of type `SecondLevelEmbeddedEntityClass`. The class `SecondLevelEmbeddedEntityClass`, finally, has one mapped field, plus an indexed-embedded on a property named `thirdLevelEmbedded`, of type `IndexedEntityClass`. To avoid any infinite recursion, the indexed-embedded is bounded to a maximum depth of 1, meaning it will embed fields mapped directly in the `IndexedEntityClass` type, but will not transitively include any of its indexed-embedded.

This model is converted using the process described above into this node builder tree:
While the mapped model was originally organized as a cyclic graph, the indexing processor nodes are organized as a tree, which means among others it is acyclic. This is necessary to be able to process entities in a straightforward way at runtime, without relying on complex logic, mutable states or metadata lookups.

This transformation from a potentially cyclic graph into a tree results from the fact we "unroll" the indexed-embedded definitions, breaking cycles by creating multiple indexing processor nodes for the same type if the type appears at different levels of embedding.

In our example, `IndexedEntityClass` is exactly in this case: the root node represents this type, but the type node near the bottom also represents the same type, only at a different level of embedding.

If you want to learn more about how `@IndexedEmbedded` path filtering, depth filtering, cycles, and prefixes are handled, a good starting point is `IndexModelBindingContextImpl#addIndexedEmbeddedIfIncluded`.

Ultimately, the created indexing process tree will follow approximately the same structure as the builder tree. The indexing processor tree may be a bit different from the builder tree, due to optimizations. In particular, some nodes may be trimmed down if we detect that the node will not
contribute anything to documents at runtime, which may happen for some property nodes when using @IndexedEmbedded with path filtering (includePaths) or depth filtering (maxDepth).

This is the case in our example for the "embedded" node near the bottom. The builder node was created when applying and interpreting metadata, but it turns out the node does not have any child nor any bridge. As a result, this node will be ignored when creating the indexing processor.

12.2.3. Implicit reindexing resolvers

Reindexing resolvers are the objects responsible for determining, whenever an entity changes, which other entities include that changed entity in their indexed form and should thus be reindexed.

Similarly to indexing processors, the PojoImplicitReindexingResolver contains nodes organized as a tree, each node being an implementation of PojoImplicitReindexingResolverNode. The POJO mapper assigns one PojoImplicitReindexingResolver containing one tree to each indexed or contained entity type. Indexed entity types are those mapped to an index (using @Indexed or similar), while "contained" entity types are those being the target of an @IndexedEmbedded or being manipulated in a bridge using the PojoModelElement API.

Here are the main types of nodes:

- PojoImplicitReindexingResolverOriginalTypeNode: A node representing a POJO type (a Java class).
- PojoImplicitReindexingResolverCastedTypeNode: A node representing a POJO type (a Java class) to be casted to a supertype or subtype, applying nested nodes only if the cast succeeds.
- PojoImplicitReindexingResolverPropertyNode: A node representing a POJO property.
- PojoImplicitReindexingResolverContainerElementNode: A node representing elements in a container (List, Optional, ...).
- PojoImplicitReindexingResolverDirtinessFilterNode: A node representing a filter, delegating to its nested nodes only if some precise paths are considered dirty.
- PojoImplicitReindexingResolverMarkingNode: A node representing a value to be marked as "to reindex".

At runtime, the root node will be passed the changed entity, the "dirtiness state" of that entity (in short, a list of properties that changed in that entity), and a collector of entities to re-index. Then each node will "resolve" entities to reindex according to its input, i.e. perform one (or more) of the following:

- check that the "dirtiness state" contains specific dirty paths that make reindexing relevant for this node
- extract data from the Java object passed as input: extract the value of a property, the elements of a list, try to cast the object to a given type, ...
• pass the extracted data to the collector
• pass the extracted data along with the collector to a nested node, which will in turn "resolve" entities to reindex according to its input.

As with indexing processor, this representation is very flexible, yet explicit enough to not require any metadata lookup at runtime.

Reindexing resolvers are logged at the debug level during bootstrap. Enable this level of logging for the Hibernate Search classes if you want to understand the reindexing resolver tree that was generated for a given mapping.

Bootstrap

One reindexing resolver tree is built during bootstrap for each indexed or contained type. The entry point to building these resolvers may not be obvious: it is the indexing resolver building process. Indeed, as we build the indexing processor for a given indexed type, we discover all the paths that will be walked through in the entity graph when indexing this type, and thus what the indexed type's indexing process definitely depends on. Which is all the information we need to build the reindexing resolvers.

In order to understand how reindexing resolvers are built, it is important to keep in mind that reindexing resolvers mirror indexing processors: if the indexing processor for entity A references entity B at some point, then you can be sure that the reindexing resolver for entity B will reference entity A at some point.

As an example, let's consider the indexing processor builder tree from the previous section (Indexing processors):
As we build the indexing processors, we will also build another tree to represent dependencies from the root type (IndexedEntityClass) to each dependency. This is where dependency collectors come into play.

Dependency collectors are organized approximately the same way as the indexing processor builders, as a tree. A root node is provided to the root builder, then one node will be created for each of his children, and so on. Along the way, each builder will be able to notify its dependency collector that it will actually build an indexing processor (it wasn't trimmed down due to some optimization), which means the node needs to be taken into account in the dependency tree. This is done through the PojoIndexingDependencyCollectorValueNode#collectDependency method, which triggers some additional steps.
TypeBridge and PropertyBridge implementations are allowed to go through associations and access properties from different entities. For this reason, when such bridges appear in an indexing processor, we create dependency collector nodes as necessary to model the bridge’s dependencies. For more information, see PojoModelTypeRootElement#contributeDependencies (type bridges) and PojoModelPropertyRootElement#contributeDependencies (property bridges).

Let’s see what our dependency collector tree will ultimately look like:

![Dependency Collector Tree Diagram]

The value nodes in red are those that we will mark as a dependency using PojoIndexingDependencyCollectorValueNode#collectDependency. The embedded property at the bottom will be detected as not being used during indexing, so the corresponding value node will not be marked as a dependency, but all the other value nodes will.

The actual reindexing resolver building happens when PojoIndexingDependencyCollectorValueNode#collectDependency is called for each value node. To understand how it works, let us use the value node for longField as an example.

When collectDependency is called on this node, the dependency collector will first backtrack to the
last encountered entity type, because that is the type for which "change events" will be received by
the POJO mapper. Once this entity type is found, the dependency collector type node will retrieve the
reindexing resolver builder for this type from a common pool, shared among all dependency collectors
for all indexed types.

Reindexing resolver builders follow the same structure as the reindexing resolvers they build: they are
nodes in a tree, and there is one type of builder for each type of reindexing resolver node:
PojoImplicitReindexingResolverOriginalTypeNodeBuilder,
PojoImplicitReindexingResolverPropertyNodeBuilder,...

Back to our example, when `collectDependency` is called on the value node for `longField`, we
backtrack to the last encountered entity type, and the dependency collector type node retrieves what
will be the builder of our "root" reindexing resolver node:

From there, the reindexing resolver builder is passed to the next dependency collector value node
using the `PojoIndexingDependencyCollectorValueNode#markForReindexing` method. This
method also takes as a parameter the path to the property that is depended on, in this case
`longField`.

The value node will then use its knowledge of the dependency tree (using its ancestors in the
dependency collector tree) to build a `BoundPojoModelPath` from the previous entity type to that
value. In our case, this path is `Type EmbeddedEntityClass ⇒ Property
"secondLevelEmbedded" ⇒ No container value extractor.`
This path represents an association between two entity types: `EmbeddedEntityClass` on the containing side, and `SecondLevelEmbeddedEntityClass` on the contained side. In order to complete the reindexing resolver tree, we need to invert this association, i.e. find out the inverse path from `SecondLevelEmbeddedEntityClass` to `EmbeddedEntityClass`. This is done in `PojoAssociationPathInverter` using the "additional metadata" mentioned in Representation of the POJO metamodel.

Once the path is successfully inverted, the dependency collector value node can add new children to the reindexing resolver builder:

The resulting reindexing resolver builder is then passed to the next dependency collector value node, and the process repeats:
Once we reach the dependency collector root, we are almost done. The reindexing resolver builder tree has been populated with every node needed to reindex `IndexedEntityClass` whenever a change occurs in the `longField` property of `SecondLevelEmbeddedEntityClass`.

The only thing left to do is register the path that is depended on (in our example, `longField`). With this path registered, we will be able to build a `PojoPathFilter`, so that whenever `SecondLevelEmbeddedEntityClass` changes, we will walk through the tree, but not all the tree: if at some point we notice that a node is relevant only if `longField` changed, but the "dirtiness state" tells us that `longField` did not change, we can skip a whole branch of the tree, avoiding useless lazy loading and reindexing.

The example above was deliberately simple, to give a general idea of how reindexing resolvers are built. In the actual algorithm, we have to handle several circumstances that make the whole process significantly more complex:

**Polymorphism**

Due to polymorphism, the target of an association at runtime may not be of the exact type declared in the model. Also because of polymorphism, an association may be defined on an abstract entity type, but have different inverse sides, and even different target types, depending on the concrete entity subtype.

There are all sorts of intricate corner cases to take into account, but they are for the main part addressed this way:
• Whenever we create a type node in the reindexing resolver building tree, we take care to determine all the possible concrete entity types for the considered type, and create one reindexing resolver type node builder per possible entity type.

• Whenever we resolve the inverse side of an association, take care to resolve it for every concrete "source" entity type, and to apply all of the resulting inverse paths.

If you want to observe the algorithm handling this live, try debugging `AutomaticIndexingPolymorphicOriginalSideAssociationIT` or `AutomaticIndexingPolymorphicInverseSideAssociationIT`, and put breakpoints in the `collectDependency/markForReindexing` methods of dependency collectors.

**Embedded types**

Types in the dependency collector tree may not always be entity types. Thus, the path of associations (both the ones to invert and the inverse paths) may be more complex than just one property plus one container value extractor.

If you want to observe the algorithm handling this live, try debugging `AutomaticIndexingEmbeddableIT`, and put breakpoints in the `collectDependency/markForReindexing` methods of dependency collectors.

**Fine-grained dirty checking**

Fine-grained dirty checking consists in keeping track of which properties are dirty in a given entity, so as to only reindex "containing" entities that actually use at least one of the dirty properties. Without this, Hibernate Search could trigger unnecessary reindexing from time to time, which could have a very bad impact on performance depending on the user model.

In order to implement fine-grained dirty checking, each reindexing resolver node builder not only stores the information that the corresponding node should be reindexed whenever the root entity changes, but it also keeps track of which properties of the root entity should trigger reindexing of this particular node. Each builder keeps this state in a `PojoImplicitReindexingResolverMarkingNodeBuilder` instance it delegates to.

If you want to observe the algorithm handling this live, try debugging `AutomaticIndexingBasicIT.directValueUpdate_nonIndexedField`, and put breakpoints in the `collectDependency/markForReindexing` methods of dependency collectors (to see what happens at bootstrap), and in the `resolveEntitiesToReindex` method of `PojoImplicitReindexingResolverDirtinessFilterNode` (to see what happens at runtime).

### 12.3. JSON mapper

The JSON mapper does not currently exist, but there are plans to work on it.
Chapter 13. Further reading

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.
Chapter 14. Credits

The full list of contributors to Hibernate Search can be found in the copyright.txt file in the Hibernate Search sources, available in particular in our git repository.

The following contributors have been involved in this documentation:

- Emmanuel Bernard
- Hardy Ferentschik
- Gustavo Fernandes
- Sanne Grinovero
- Mincong Huang
- Nabeel Ali Memon
- Gunnar Morling
- Yoann Rodière
- Guillaume Smet