# Table of Contents

Preface .................................................................................................................. 1

1. Getting started .................................................................................................... 2
   1.1. Compatibility ................................................................................................. 2
   1.2. Migration notes ............................................................................................ 2
   1.3. Dependencies ............................................................................................... 3
   1.4. Configuration ............................................................................................... 4
   1.5. Mapping ........................................................................................................ 5
   1.6. Initialization .................................................................................................. 9
      1.6.1. Schema management ............................................................................. 9
      1.6.2. Initial indexing ...................................................................................... 9
   1.7. Indexing ........................................................................................................ 10
   1.8. Searching ..................................................................................................... 11
   1.9. Analysis ......................................................................................................... 14
   1.10. What's next ................................................................................................. 18

2. Concepts .............................................................................................................. 19
   2.1. Full-text search ............................................................................................ 19
   2.2. Mapping ........................................................................................................ 19
   2.3. Analysis ......................................................................................................... 20
   2.4. Commit and refresh ..................................................................................... 21
   2.5. Sharding and routing ................................................................................... 22

3. Architecture ......................................................................................................... 25

4. Configuration ....................................................................................................... 26
   4.1. Configuration sources ................................................................................ 26
   4.2. Structure of configuration properties ......................................................... 26
   4.3. Type of configuration properties ................................................................ 27
      4.3.1. Configuration Builders ......................................................................... 28
   4.4. Configuration property checking ................................................................. 29
   4.5. Beans ........................................................................................................... 30
      4.5.1. Supported frameworks ......................................................................... 30
      4.5.2. Bean resolution .................................................................................... 30
      4.5.3. Bean injection ...................................................................................... 31
      4.5.4. Bean lifecycle ....................................................................................... 31
   4.6. Background failure handling ........................................................................ 32

5. Mapping Hibernate ORM entities to indexes ..................................................... 35
   5.1. Configuration ............................................................................................... 35
      5.1.1. Enabling/disabling Hibernate Search .................................................... 35
      5.1.2. Configuring the mapping ...................................................................... 35
      5.1.3. Other configuration properties .............................................................. 35
   5.2. Entity/index mapping .................................................................................. 36
6.8.2. Type objects ......................................................... 118
6.8.3. Multi-valued fields ............................................. 119
6.8.4. Object fields ...................................................... 120
6.8.5. Object field storage ............................................ 122
6.8.6. Dynamic fields with field templates ...................... 123
6.9. Defining index field types ....................................... 127
  6.9.1. Basics .......................................................... 127
  6.9.2. Available data types ......................................... 128
  6.9.3. Available type options ...................................... 128
  6.9.4. DSL converter ................................................ 129
  6.9.5. Projection converter ........................................ 130
  6.9.6. Backend-specific types ..................................... 131
6.10. Assigning default-specific bridges with the bridge resolver .......................... 131
7. Managing the index schema ......................................... 132
  7.1. Basics ............................................................. 132
  7.2. Automatic schema management on startup/shutdown .......... 132
  7.3. Manual schema management .................................. 134
  7.4. How schema management works ................................ 137
8. Indexing Hibernate ORM entities ..................................... 140
  8.1. Automatic indexing ............................................ 140
    8.1.1. Configuration .............................................. 140
    8.1.2. How automatic indexing works ......................... 140
    8.1.3. Synchronization with the indexes ...................... 142
      Basics .......................................................... 142
      Per-session override ......................................... 143
      Custom strategy .............................................. 144
  8.2. Reindexing large volumes of data with the MassIndexer .......... 145
    8.2.1. Basics ...................................................... 145
    8.2.2. MassIndexer parameters ................................ 147
    8.2.3. Tuning the MassIndexer for best performance ...... 152
      Basics .......................................................... 152
      Threads and JDBC connections ................................ 153
  8.3. Reindexing large volumes of data with the JSR-352 integration ............... 154
  8.4. Manual indexing ................................................ 154
    8.4.1. Basics ...................................................... 154
    8.4.2. Controlling entity reads and index writes with SearchIndexingPlan ...... 154
    8.4.3. Explicitly indexing and deleting specific documents .... 158
    8.4.4. Explicitly altering a whole index ...................... 160
9. Searching .............................................................. 164
  9.1. Query DSL ....................................................... 164
    9.1.1. Basics ...................................................... 164
    9.1.2. Advanced entity types targeting ....................... 165
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.Beta8 is currently a technology preview.

If you encounter problems, be sure to report them: new preview versions are released regularly.

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

1.1. Compatibility

Table 1. Compatibility

<table>
<thead>
<tr>
<th>Java Runtime</th>
<th>Java 8 or greater.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hibernate ORM (for the ORM mapper)</td>
<td>Hibernate ORM 5.4.17.Final.</td>
</tr>
<tr>
<td>JPA (for the ORM mapper)</td>
<td>JPA 2.2.</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.Beta8</version>
</dependency>

<dependency>
  <groupId>org.hibernate.search</groupId>
  <artifactId>hibernate-search-backend-lucene</artifactId>
  <version>6.0.0.Beta8</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.root" value="some/filesystem/path"/>
-->①
<property name="hibernate.search.default_backend" value="myBackend"/>
②
③
```

① Define a backend named "myBackend" relying on Lucene technology.
② 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.
③ 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="elasticsearch.mycompany.com"/>
<property name="hibernate.search.backends.myBackend.protocol" value="https"/>
<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"/>
③
```

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 Mapping Hibernate ORM entities to indexes.
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 Mapping associated elements with @IndexedEmbedded 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 Mapping a property to an index field with @GenericField, @FullTextField, ....
- Properties, or even types, can be mapped with finer-grained control using "bridges". See Bridges for more information.

### 1.6. Initialization

Before the application is started for the first time, some initialization may be required:

- The indexes and their schema need to be created.
- Data already present in the database (if any) needs to be indexed.

#### 1.6.1. Schema management

Before indexing can take place, indexes and their schema need to be created, either on disk (Lucene) or through REST API calls (Elasticsearch).

Fortunately, by default, Hibernate Search will take care of creating indexes on the first startup: you don't have to do anything.

The next time the application is started, existing indexes will be re-used.

Any change to your mapping (adding new fields, changing the type of existing fields, ...) between two restarts of the application will require an update to the index schema.

This will require some special handling, though it can easily be solved by dropping and re-creating the index. See Changing the mapping of an existing application for more information.

#### 1.6.2. Initial indexing

As we'll see later, Hibernate Search takes care of triggering indexing every time an entity changes in the application.

However, data already present in the 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 5. Using Hibernate Search MassIndexer API to manually (re)index the already persisted data

```java
SearchSession searchSession = Search.session( entityManager ); ①
MassIndexer indexer = searchSession.massIndexer( 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 Reindexing large volumes of data with the MassIndexer.

④ Invoke the batch indexing process.

If no data is initially present in the database, mass indexing is not necessary.

1.7. Indexing

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

Example 6. 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 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.

1.8. 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.session(entityManager); ①

SearchResult<Book> result = searchSession.search(Book.class) ②
    .where( f -> f.match( ③
        .fields("title", "authors.name")
        .matching("Refactoring: Improving the Design of Existing Code")
    )
    .fetch(20); ④

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

List<Book> hits2 =
    /* ... same DSL calls as above... */
    .fetchHits(20); 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, limiting to the top 20 hits.

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

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

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

SearchResult<Book> result = searchSession.search(scope) ③
    .where(scope.predicate().match()) ④
    .fields("title", "authors.name")
    .matching("Refactoring: Improving the Design of Existing Code")
    .toPredicate()
    .fetch(20); ⑤

long totalHitCount = result.totalHitCount(); ⑥
List<Book> hits = result.hits(); ⑦

List<Book> hits2 =
    /* ... same DSL calls as above... */
    .fetchHits(20); 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, limiting to the top 20 hits.
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

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

long totalHitCount = searchSession.search(Book.class)
    .where(f -> f.match())
    .fields("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.
Note that, while the examples above retrieved hits as managed entities, it is just one of the possible hit types. See Query DSL for more information.

1.9. 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, i.e. how text is supposed to be processed when indexing and searching. This involves analyzers, which are made up of three types of components, applied one after the other:

- (rarely) zero or more character filters, to clean up the input text: A <strong>GREAT</strong> résume ⇒ A GREAT résume.
- a tokenizer, to split the input text into words, called "tokens": A GREAT résume ⇒ [A, GREAT, résume].
- zero or more token filters, to normalize the tokens and remove meaningless tokens. [A, GREAT, résume] ⇒ [great, resume].

There are built-in analyzers, but it is generally better to build your own by picking the filters and tokenizer most suited to your specific needs.

The following paragraphs will explain how to configure and use a simple yet reasonably useful analyzer. For more information about analysis and how to configure it, refer to the Analysis section.

Each custom analyzer needs to be given a name in Hibernate Search. This is done through 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.LuceneAnalysisConfigurationContext;
import org.apache.lucene.analysis.core.LowerCaseFilterFactory;
import org.apache.lucene.analysis.miscellaneous.ASCII FoldingFilterFactory;
import org.apache.lucene.analysis.snowball.SnowballPorterFilterFactory;
import org.apache.lucene.analysis.standard.StandardTokenizerFactory;

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

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

1. Define a custom analyzer named "english", to analyze English text such as book titles.

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. Define another custom analyzer, named "name", to analyze author names. On contrary to the first one, do not use enable stemming, as it is unlikely to lead to useful results on proper nouns.

6. 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

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

import org.hibernate.search.backend.elasticsearch.analysis.ElasticsearchAnalysisConfigurer;
import org.hibernate.search.backend.elasticsearch.analysis.ElasticsearchAnalysisConfigurationContext;

public class MyElasticsearchAnalysisConfigurer implements ElasticsearchAnalysisConfigurer {

    @Override
    public void configure(ElasticsearchAnalysisConfigurationContext context) {
        context.analyzer("english").custom()
            .tokenizer("standard")
            .tokenFilters("lowercase", "snowball_english", "asciifolding");

        context.tokenFilter("snowball_english")
            .type("snowball")
            .param("language", "English");

        context.analyzer("name").custom()
            .tokenizer("standard")
            .tokenFilters("lowercase", "asciifolding");
    }
}
```

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

1. Define a custom analyzer named "english", to analyze English text such as book titles.
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 assigned a name.
5. Set the value of a parameter for the char filter/tokenizer/token filter being defined.
6. Define another custom analyzer, named "name", to analyze author names. On contrary to the first one, do not use enable stemming, as it is unlikely to lead to useful results on proper nouns.
7. 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 = "english")
    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.FullTextField;

@Entity
public class Author {
    @Id
    @GeneratedValue
    private Integer id;

    @FullTextField(analyzer = "name")
    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.session( entityManager );

SearchResult<Book> result = searchSession.search( Book.class )
    .where( f -> f.match() )
    .fields( "title", "authors.name" )
    .matching( "refactor" )
    .fetch( 20 );
// Not shown: commit the transaction and close the entity manager
```

1.10. 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, projections, aggregations, and schema management.

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

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

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

Hibernate Search relies on either Apache Lucene or Elasticsearch to implement full-text search. Since Elasticsearch uses Lucene internally, they share a lot of characteristics and their general approach to full-text search.

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

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

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

Lucene and Elasticsearch are not limited to just text search: numeric data is also supported, enabling support for integers, doubles, longs, dates, etc. These types are indexed and queried using a slightly different approach, which obviously does not involve text processing.

2.2. Mapping

Applications targeted by Hibernate search generally use an entity-based model to represent data. In this model, each entity is a single object with a few properties of atomic type (String, Integer, LocalDate, ...). Each entity can have multiple associations to one or even many other entities.

Entities are thus organized as a graph, where each node is an entity and each association is an edge.

By contrast, Lucene and Elasticsearch work with documents. Each document is a collection of "fields", each field being assigned a name—a unique string—and a value—which can be text, but also numeric data such as an integer or a date. Fields also have a type, which not only determines the type of values (text/numeric), but more importantly the way this value will be stored: indexed, stored, with doc values, etc. It is possible to introduce nested documents, but not real associations.
Documents are thus organized, at best, as a collection of trees, where each tree is a document, optionally with nested documents.

There are multiple mismatches between the entity model and the document model: properties vs. fields, associations vs. nested documents, graph vs. collection of trees.

The goal of mapping, in Hibernate search, is to resolve these mismatches by defining how to transform one or more entities into a document, and how to resolve a search hit back into the original entity. This is the main added value of Hibernate Search, the basis for everything else from automatic indexing to the various search DSLs.

Mapping is usually configured using annotations in the entity model, but this can also be achieved using a programmatic API. To learn more about how to configure mapping, see Mapping Hibernate ORM entities to indexes.

To learn how to index the resulting documents, see Indexing Hibernate ORM entities (hint: it's automatic).

To learn how to search with an API that takes advantage of the mapping to be closer to the entity model, in particular by returning hits as entities instead of just document identifiers, see Searching.

2.3. Analysis

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

However, the processing is not just about "tokenizing". Index lookups are exact lookups, which means that looking up Great (capitalized) will not return documents containing only great (all lowercase). An extra step is performed when processing text to address this caveat: token filtering, which normalizes tokens. Thanks to that "normalization", Great will be indexed as great, so that an index lookup for the query great will match as expected.

In the Lucene world (Lucene, Elasticsearch, Solr, ...), text processing during both the indexing and searching phases is called "analysis" and is performed by an "analyzer".

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

1. Character filter: transforms the input characters. Replaces, adds or removes characters.
2. Tokenizer: splits the text into several words, called "tokens".
3. Token filter: transforms the tokens. Replaces, add or removes characters in a token, derives new tokens from the existing ones, removes tokens based on some condition, ...
The tokenizer usually splits on whitespaces (though there are other options). Token filters are usually where customization takes place. They can remove accented characters, remove meaningless suffixes (−ing, −s, …) or tokens (a, the, …), replace tokens with a chosen spelling (wi-fi ⇒ wifi), etc.

Character filters, though useful, are rarely used, because they have no knowledge of token boundaries.

Unless you know what you are doing, you should generally favor token filters.

In some cases, it is necessary to index text in one block, without any tokenization:

- For some types of text, such as SKUs or other business codes, tokenization simply does not make sense: the text is a single "keyword".
- For sorts by field value, tokenization is not necessary. It is also forbidden in Hibernate Search due to performance issues; only non-tokenized fields can be sorted on.

To address these use cases, a special type of analyzer, called "normalizer", is available. Normalizers are simply analyzers that are guaranteed not to use a tokenizer: they can only use character filters and token filters.

In Hibernate Search, analyzers and normalizers are referenced by their name, for example when defining a full-text field. Analyzers and normalizers have two separate namespaces.

Some names are already assigned to built-in analyzers (in Elasticsearch in particular), but it is possible (and recommended) to assign names to custom analyzers and normalizers, assembled using built-in components (tokenizers, filters) to address your specific needs.

Each backend exposes its own APIs to define analyzers and normalizers, and generally to configure analysis. See the documentation of each backend for more information:

- Analysis for the Lucene backend
- Analysis for the Elasticsearch backend

### 2.4. Commit and refresh

In order to get the best throughput when indexing and when searching, both Elasticsearch and Lucene rely on "buffers" when writing to and reading from the index:

- When writing, changes are not directly written to the index, but to an "index writer" that buffers changes in-memory or in temporary files.

  The changes are "pushed" to the actual index when the writer is committed. Until the commit happens, uncommitted changes are in an "unsafe" state: if the application crashes or if the server suffers from a power loss, uncommitted changes will be lost.
• When reading, e.g. when executing a search query, data is not read directly from the index, but from an "index reader" that exposes a view of the index as it was at some point in the past.

The view is updated when the reader is refreshed. Until the refresh happens, results of search queries might be slightly out of date: documents added since the last refresh will be missing, documents delete since the last refresh will still be there, etc.

Unsafe changes and out-of-date indexes are obviously undesirable, but they are a trade-off that improves performance.

Different factors influence when refreshes and commit happen:

- **Automatic indexing** will, by default, require that a commit of the index writer is performed after each set of changes, meaning the changes are safe after the Hibernate ORM transaction commit returns. However, no refresh is requested by default, meaning the changes may only be visible at a later time, when the backend decides to refresh the index reader. This behavior can be customized by setting a different synchronization strategy.

- The **mass indexer** will not require any commit or refresh until the very end of mass indexing, so as to maximize indexing throughput.

- Whenever there are no particular commit or refresh requirements, backend defaults will apply:
  - See here for Elasticsearch.
  - See here for Lucene.

- A commit may be forced explicitly through the **flush() API**.

- A refresh may be forced explicitly through the **refresh() API**.

Even though we use the word "commit", this is not the same concept as a commit in relational database transactions: there is no transaction and no "rollback" is possible.

There is no concept of isolation, either. After a refresh, all changes to the index are taken into account: those committed to the index, but also those that are still buffered in the index writer.

For this reason, commits and refreshes can be treated as completely orthogonal concepts: certain setups will occasionally lead to committed changes not being visible in search queries, while others will allow even uncommitted changes to be visible in search queries.

### 2.5. Sharding and routing

Sharding consists in splitting index data into multiple "smaller indexes", called shards, in order to
improve performance when dealing with large amounts of data.

In Hibernate Search, similarly to Elasticsearch, another concept is closely related to sharding: routing. Routing consists in resolving a document identifier, or generally any string called a "routing key", into the corresponding shard.

When indexing:

- A document identifier and optionally a routing key are generated from the indexed entity.
- The document, along with its identifier and optionally its routing key, is passed to the backend.
- The backend "routes" the document to the correct shard, and adds the routing key (if any) to a special field in the document (so that it’s indexed).
- The document is indexed in that shard.

When searching:

- The search query can optionally be passed one or more routing keys.
- If no routing key is passed, the query will be executed on all shards.
- If one or more routing keys are passed:
  - The backend resolves these routing keys into a set of shards, and the query will only be executed on all shards, ignoring the other shards.
  - A filter is added to the query so that only documents indexed with one of the given routing keys are matched.

Sharding, then, can be leveraged to boost performance in two ways:

- When indexing: a sharded index can spread the "stress" onto multiple shards, which can be handled by separate threads and located on different disks (Lucene) or be located on different servers (Elasticsearch).
- When searching: if one property, let’s call it category, is often used to select a subset of documents, this property can be defined as a routing key in the mapping, so that it’s used to route documents instead of the document ID. As a result, documents with the same value for category will be indexed in the same shard. Then when searching, if a query already filters documents so that it is known that the hits will all have the same value for category, the query can be manually routed to the shards containing documents with this value, and the other shards can be ignored.

To enable sharding, some configuration is required:

- The backends require explicit configuration: see here for Lucene and here for Elasticsearch.
- In most cases, document IDs are used to route documents to shards by default. This does not allow taking advantage of routing when searching, which requires multiple documents to share the same routing key. Applying routing to a search query in that case will not return any result. To
Sharding is static by nature:

- Each entity, and its corresponding document, is expected to stay in the same shard from its creation to its deletion. Modifying an entity in such a way that its routing key, and thus its corresponding shard, changes, will lead to duplicate documents. Thus, properties used to generate routing keys must be immutable.

- Each index is expected to have the same shards, with the same identifiers, from one boot to the other. Changing the number of shards or their identifiers will require full reindexing.
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 three 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 mapping

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.io.commit_interval = 500` sets the `io.commit_interval` property for all indexes 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>Comma-separated string containing bean references (see above)</td>
</tr>
</tbody>
</table>

4.3.1. Configuration Builders

Both BackendSettings and IndexSettings provide tools to help build the configuration property keys.

BackendSettings

BackendSettings.backendKey("myBackend", ElasticsearchBackendSettings.HOSTS) is equivalent to hibernate.search.backends.myBackend.host.

For a list of available property keys, see ElasticsearchBackendSettings or LuceneBackendSettings.

IndexSettings

IndexSettings.indexDefaultsKey("myBackend", ElasticsearchIndexSettings.LIFECYCLE_MINIMAL_REQUIRED_STATUS) is equivalent to hibernate.search.backends.signature.index_defaults.lifecycle.minimal_required_status.

For a list of available property keys, see ElasticsearchIndexSettings or LuceneIndexSettings.
You can also use the `IndexSettings.indexKey("myBackend", "myIndex", ElasticsearchIndexSettings.LIFECYCLE_MINIMAL_REQUIRED_STATUS)` to apply a configuration to a specific index.

**Example 14. Using the helper to build hibernate configuration**

```java
private Properties buildHibernateConfiguration() {
    Properties config = new Properties();
    // add hibernate configuration
    String myBackend = "myBackend";
    // backend configuration
    config.put( BackendSettings.backendKey( myBackend, ElasticsearchBackendSettings.HOSTS ), "127.0.0.1:9200" );
    config.put( BackendSettings.backendKey( myBackend, ElasticsearchBackendSettings.PROTOCOL ), "http" );
    config.put( BackendSettings.backendKey( myBackend, ElasticsearchBackendSettings.TYPE ),
               ElasticsearchBackendSettings.TYPE_NAME );
    config.put( BackendSettings.backendKey( myBackend, ElasticsearchBackendSettings.MULTI_TENANCY_STRATEGY ),
               MultiTenancyStrategyName.DISCRIMINATOR.externalRepresentation() );
    config.put( BackendSettings.backendKey( myBackend, ElasticsearchBackendSettings.VERSION ), "7.7" );
    config.put( BackendSettings.backendKey( myBackend, ElasticsearchBackendSettings.VERSION_CHECK_ENABLED ), "false" );
    // index configuration
    config.put( IndexSettings.indexDefaultsKey( myBackend, ElasticsearchIndexSettings.SCHEMA_MANAGEMENT_MINIMAL_REQUIRED_STATUS ),
                IndexStatus.YELLOW.externalRepresentation() );
    // orm configuration
    config.put( HibernateOrmMapperSettings.AUTOMATIC_INDEXING_SYNCHRONIZATION_STRATEGY,
                AutomaticIndexingSynchronizationStrategyNames.ASYNC );
    // engine configuration
    config.put( EngineSettings.BACKGROUND_FAILURE_HANDLER, "myFailureHandler" );
    config.put( EngineSettings.DEFAULT_BACKEND, myBackend );
    return config;
}
```

### 4.4. Configuration property checking

Hibernate Search will track the parts of the provided configuration that are actually used and will log a warning if any configuration property starting with "hibernate.search." is never used, because that might indicate a configuration issue.

To disable this warning, set the `hibernate.search.configuration_property_checking.strategy` property to `ignore`. 
4.5. Beans

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

This section describes the supported frameworks, how the beans are resolved and how the beans can be injected with other beans.

4.5.1. Supported frameworks

When using the Hibernate Search integration into Hibernate ORM, all dependency injection frameworks supported by Hibernate ORM are supported.

This includes, but may not be limited to:

- all CDI-compliant frameworks, including WildFly and Quarkus;
- the Spring framework.

When the framework is not supported, or when using Hibernate Search without Hibernate ORM, beans can only be retrieved using reflection by calling the public, no-arg constructor of the referenced type.

4.5.2. Bean resolution

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.

### 4.5.3. Bean injection

All beans resolved by Hibernate Search using a supported framework can take advantage of injection features of this framework.

For example a bean can be injected with another bean by annotating one of its fields in the bridge with `@Inject`.

Lifecycle annotations such as `@PostConstruct` should also work as expected.

Even when not using any framework, it is still possible to take advantage of the `BeanResolver`. This component, passed to several methods during bootstrap, exposes several methods to resolve a reference into a bean, exposing programmatically what would usually be achieved with an `@Inject` annotation. See the javadoc of `BeanResolver` for more information.

### 4.5.4. Bean lifecycle

As soon as beans are no longer needed, Hibernate Search will release them and let the dependency injection framework call the appropriate methods (`@PreDestroy`, ...).

Some beans are only necessary during bootstrap, such as `ElasticsearchAnalysisConfigurer`s, so they will be released just after bootstrap.

Other beans are necessary at runtime, such as `ValueBridge`s, so they will be released on shutdown.

Be careful to define the scope of your beans as appropriate.

Immutable beans or beans used only once such as `ElasticsearchAnalysisConfigurer` may safely most scopes, but some beans are expected to be mutable and instantiated multiple times, such as for example `PropertyBinder`.

For these beans, it is recommended to use the "dependent" scope (CDI terminology) or the "prototype" scope (Spring terminology). When in doubt, this is also generally the safest choice for beans injected into Hibernate Search.

Beans resolved by Hibernate Search using a supported framework can take advantage of injection features of this framework.
4.6. Background failure handling

Hibernate Search generally propagates exceptions occurring in background threads to the user thread, but in some cases, such as Lucene segment merging failures, or some failures during automatic indexing, the exception in background threads cannot be propagated. By default, when that happens, the failure is logged at the ERROR level.

To customize background failure handling, you will need to:

- Define a class that implements the `org.hibernate.search.engine.reporting.FailureHandler` interface.

- Configure the backend to use that implementation by setting the configuration property `hibernate.search.background_failure_handler` to a bean reference pointing to the implementation.

Hibernate Search will call the `handle` methods whenever a failure occurs.
Example 15. Implementing and using a **FailureHandler**

```java
package org.hibernate.search.documentation.reporting.failurehandler;
import java.util.ArrayList;
import java.util.List;
import org.hibernate.search.engine.reporting.EntityIndexingFailureContext;
import org.hibernate.search.engine.reporting.FailureContext;
import org.hibernate.search.engine.reporting.FailureHandler;
import org.hibernate.search.util.impl.test.rule.StaticCounters;

public class MyFailureHandler implements FailureHandler {
    @Override
    public void handle(FailureContext context) {
        String failingOperationDescription = context.failingOperation().toString();
        Throwable throwable = context.throwable()
        // ... report the failure ...
    }

    @Override
    public void handle(EntityIndexingFailureContext context) {
        String failingOperationDescription = context.failingOperation().toString();
        Throwable throwable = context.throwable();
        List<String> entityReferencesAsString = new ArrayList<>();
        for (Object entityReference : context.entityReferences()) {
            entityReferencesAsString.add(entityReference.toString());
        }
        // ... report the failure ...
    }
}
```

1. **handle(FailureContext)** is called for generic failures that do not fit any other specialized **handle** method.
2. Get a description of the failing operation from the context.
3. Get the throwable thrown when the operation failed from the context.
4. Use the context-provided information to report the failure in any relevant way.
5. **handle(EntityIndexingFailureContext)** is called for failures occurring when indexing entities.
6. On top of the failing operation and throwable, the context also lists references to entities that could not be indexed correctly because of the failure.
7. Use the context-provided information to report the failure in any relevant way.

1. Assign the background failure handler using a Hibernate Search configuration property.
When a failure handler’s `handle` method throws an error or exception, Hibernate Search will catch it and log it at the ERROR level. It will not be propagated.
Chapter 5. Mapping Hibernate ORM entities to indexes

5.1. Configuration

5.1.1. Enabling/disabling Hibernate Search

The Hibernate Search integration into Hibernate ORM 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.enabled` boolean property to `false`.

5.1.2. Configuring the mapping

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. See Entity/index mapping and Mapping a property to an index field with `@GenericField`, `@FullTextField`, ... to get started with annotation-based mapping.

If you want to ignore these annotations, set `hibernate.search.mapping.process_annotations` 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.

Example 16. Implementing a mapping configurer

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

See Programmatic mapping for more information about the programmatic mapping API.

5.1.3. Other configuration properties

Other configuration properties are mentioned in the relevant parts of this documentation. You can find

5.2. Entity/index mapping

In order to index an entity, it must be annotated with @Indexed.

Example 17. Marking a class for indexing with @Indexed

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

Subclasses inherit the @Indexed annotation and will also be indexed by default. Each indexed subclass will have its own index, though this will be transparent when searching (all targeted indexes will be queried simultaneously).

If the fact that @Indexed is inherited is a problem for your application, you can annotate subclasses with @Indexed(enabled = false).

By default:

- The index name will be equal to the entity name, which in Hibernate ORM is set using the @Entity annotation and defaults to the simple class name.
- The index will be created in the default backend. See the getting stated 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 Mapping the document identifier for details.
- The index won’t have any field. Fields must be mapped to properties explicitly. See Mapping a property to an index field with @GenericField, @FullTextField, ... 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 18. Explicit index name with @Indexed.index

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

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 19. Explicit backend with `@Indexed.backend`**

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

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 because Author and User are indexed in different backends
List<Object> hits = searchSession.search(
    Arrays.asList( Author.class, User.class )
).where( f -> f.matchAll() ).fetchHits( 20 );
```

5.3. Mapping the document identifier

5.3.1. Basics

Index documents, much like entities, need to be assigned an identifier so that Hibernate Search can handle updates and deletion.

When indexing Hibernate ORM entities, the entity identifier is used as a document identifier by default.

Provided the entity identifier has a supported type, identifier mapping will work out of the box and no explicit mapping is necessary.

5.3.2. Explicit identifier mapping

Explicit identifier mapping is required in the following cases:

- The document identifier is not the entity identifier.
- OR the entity identifier has a type that is not supported by default. This is the case of composite identifiers, in particular.

To select a property to map to the document identifier, just apply the `@DocumentId` annotation to that property:
Example 20. Mapping a property to the document identifier explicitly with @DocumentId

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

@Id
@GeneratedValue
private Integer id;

@NaturalId
@DocumentId
private String isbn;

public Book() {
}

// Getters and setters
// ...
}
```

When the property type is not supported, it is also necessary to implement a custom identifier bridge, then refer to it in the @DocumentId annotation:

Example 21. Mapping a property with unsupported type to the document identifier with @DocumentId

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

@Id
@Convert(converter = ISBNAttributeConverter.class)
@DocumentId(identifierBridge = @IdentifierBridgeRef(type = ISBNIdentifierBridge.class))
private ISBN isbn;

public Book() {
}

// Getters and setters
// ...
}
```

5.3.3. Supported identifier property types

Below is a table listing all types with built-in identifier bridges, i.e. property types that are supported out of the box when mapping a property to a document identifier.

The table also explains the value assigned to the document identifier, i.e. the value passed to the underlying backend.

Table 2. Property types with built-in identifier bridges
### 5.4. Mapping a property to an index field with @GenericField, @FullTextField, ...

#### 5.4.1. Basics

Properties of an entity can be mapped 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.

Mapping a property to an index field looks like this:

**Example 22. Mapping properties to fields directly**

```java
@FullTextField(analyzer = "english", projectable = Projectable.YES) ①
@KeywordField(name = "title_sort", normalizer = "english", 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, property/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 Supported property types 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.

### 5.4.2. Available field annotations

Various field annotations exist, each offering its own set of attributes.

This section lists the different annotations and their use. For more details about available attributes, see Field annotation attributes.

**@GenericField**

A good default choice that will work for every property type with built-in support.

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. Moreover, you can define a specific analyzer a search analyzer to analyze searched terms differently.

| 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.

This annotation allows to set the decimalScale attribute.

@NonStandardField
An annotation for advanced use cases where a value binder is used and that binder is expected to define an index field type that does not support any of the standard options: searchable, sortable,...

This annotation is very useful for cases when a field type native to the backend is necessary: defining the mapping directly as JSON for Elasticsearch, or manipulating IndexableField directly for Lucene.

Fields mapped using this annotation have very limited configuration options from the annotation (no searchable sortable etc.), but the value binder will be able to pick a non-standard field type, which generally gives much more flexibility.

5.4.3. Field annotation attributes

Various field mapping annotations exist, each offering its own set of attributes.

This section lists the different annotation attributes and their use. For more details about available annotations, see Available field 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**. The name must not contain the dot character (\.). 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 retrieval later when querying.

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

**aggregable**

Whether the field can be **aggregated**, i.e. whether the field value is stored in a specific data structure in the index to allow aggregations later when querying.

Value: **Aggregable.YES, Aggregable.NO, Aggregable.DEFAULT.**

**searchable**

Whether the field can be searched on. i.e. whether the field is indexed in order to allow applying predicates later when querying.

Value: **Searchable.YES, Searchable.NO, Searchable.DEFAULT.**

**indexNullAs**

The value to use as a replacement anytime the property value is null.

Disabled by default.

The replacement is defined as a String. Thus its value has to be parsed. Look up the column **Parsing method for 'indexNullAs'** in **Supported property types** to find out the format used when parsing.

**extraction**

How elements to index should be extracted from the property in the case of container types (**List, Optional, Map, ...**).
By default, for properties that have a container type, the innermost elements will be indexed. For example for a property of type `List<String>`, elements of type `String` will be indexed.

This default behavior and ways to override it are described in the section Mapping container types with container extractors.

**analyzer**

The analyzer to apply to field values when indexing and querying. Only available on `@FullTextField`.

See Analysis for more details about analyzers and full-text analysis.

**searchAnalyzer**

An optional different analyzer, overriding the one defined with the `analyzer` attribute, to use only when analyzing searched terms. If not defined, the same `analyzer` will be used.

See Analysis for more details about analyzers and full-text analysis.

**normalizer**

The normalizer to apply to field values when indexing and querying. Only available on `@KeywordField`.

See Analysis for more details about normalizers and full-text analysis.

**norms**

Whether index-time scoring information for the field should be stored or not. Only available on `@KeywordField` and `@FullTextField`.

Enabling norms will improve the quality of scoring. Disabling norms will reduce the disk space used by the index.

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

**termVector**

The term vector storing strategy. Only available on `@FullTextField`.

The different values of this attribute are:

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>TermVector.YES</td>
<td>Store the term vectors of each document. This produces two synchronized arrays, one contains document terms and the other contains the term's frequency.</td>
</tr>
<tr>
<td>Value</td>
<td>Definition</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TermVector.NO</td>
<td>Do not store term vectors.</td>
</tr>
<tr>
<td>TermVector.WITH_POSITIONS</td>
<td>Store the term vector and token position information. This is the same as TermVector.YES plus it contains the ordinal positions of each occurrence of a term in a document.</td>
</tr>
<tr>
<td>TermVector.WITH_OFFSETS</td>
<td>Store the term vector and token offset information. This is the same as TermVector.YES plus it contains the starting and ending offset position information for the terms.</td>
</tr>
<tr>
<td>TermVector.WITH_POSITION_OFFSETS</td>
<td>Store the term vector, token position and offset information. This is a combination of the YES, WITH_OFFSETS and WITH_POSITIONS.</td>
</tr>
<tr>
<td>TermVector.WITH_POSITIONS_PAYLOADS</td>
<td>Store the term vector, token position and token payloads. This is the same as TermVector.WITH_POSITIONS plus it contains the payload of each occurrence of a term in a document.</td>
</tr>
<tr>
<td>TermVector.WITH_POSITIONS_OFFSETS_PAYLOADS</td>
<td>Store the term vector, token position, offset information and token payloads. This is the same as TermVector.WITH_POSITION_OFFSETS plus it contains the payload of each occurrence of a term in a document.</td>
</tr>
</tbody>
</table>

**decimalScale**

How the scale of a large number (BigInteger or BigDecimal) should be adjusted before it is indexed as a fixed-precision integer. Only available on @ScaledNumberField.

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.

5.4.4. Supported property types

Below is a table listing all types with built-in value bridges, i.e. property types that are supported out of the box when mapping a property to an index field.

The table also explains the value assigned to the index field, i.e. the value passed to the underlying backend for indexing.

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 3. Property types with built-in value bridges

<table>
<thead>
<tr>
<th>Property type</th>
<th>Value of index field (if different)</th>
<th>Limitations</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>-</td>
<td>Enum.valueOf(String)</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.Byte, byte</td>
<td>-</td>
<td>-</td>
<td>Byte.parseByte(String)</td>
</tr>
<tr>
<td>java.lang.Short, short</td>
<td>-</td>
<td>-</td>
<td>Short.parseShort(String)</td>
</tr>
<tr>
<td>Property type</td>
<td>Value of index field (if different)</td>
<td>Limitations</td>
<td>Parsing method for 'indexNullAs'</td>
</tr>
<tr>
<td>---------------</td>
<td>-------------------------------------</td>
<td>-------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>java.lang.Integer, int</td>
<td>-</td>
<td>-</td>
<td>Integer.parseInt(String)</td>
</tr>
<tr>
<td>java.lang.Long, long</td>
<td>-</td>
<td>-</td>
<td>Long.parseLong(String)</td>
</tr>
<tr>
<td>java.lang.Double, double</td>
<td>-</td>
<td>-</td>
<td>Double.parseDouble(String)</td>
</tr>
<tr>
<td>java.lang.Float, float</td>
<td>-</td>
<td>-</td>
<td>Float.parseFloat(String)</td>
</tr>
<tr>
<td>java.lang.Boolean, boolean</td>
<td>-</td>
<td>-</td>
<td>Accepts the strings true or false, ignoring case</td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td>-</td>
<td>-</td>
<td>new BigDecimal(String)</td>
</tr>
<tr>
<td>java.math.BigInteger</td>
<td>-</td>
<td>-</td>
<td>new BigInteger(String)</td>
</tr>
<tr>
<td>java.net.URI</td>
<td>toString() as a java.lang.String</td>
<td>-</td>
<td>new URI(String)</td>
</tr>
<tr>
<td>java.net.URL</td>
<td>toExternalForm() as a java.lang.String</td>
<td>-</td>
<td>new URL(String)</td>
</tr>
<tr>
<td>java.time.Instant</td>
<td>-</td>
<td>Possibly lower range/resolution</td>
<td>Instant.parse(String)</td>
</tr>
<tr>
<td>java.time.LocalDate</td>
<td>-</td>
<td>Possibly lower range/resolution</td>
<td>LocalDate.parse(String)</td>
</tr>
<tr>
<td>java.time.LocalDateTime</td>
<td>-</td>
<td>Possibly lower range/resolution</td>
<td>LocalDateTime.parse(String)</td>
</tr>
<tr>
<td>java.time.OffsetDateTime</td>
<td>-</td>
<td>Possibly lower range/resolution</td>
<td>OffsetDateTime.parse(String)</td>
</tr>
<tr>
<td>java.time.OffsetTime</td>
<td>-</td>
<td>Possibly lower range/resolution</td>
<td>OffsetTime.parse(String)</td>
</tr>
<tr>
<td>java.time.ZonedDateTime</td>
<td>-</td>
<td>Possibly lower range/resolution</td>
<td>ZonedDateTime.parse(String)</td>
</tr>
<tr>
<td>Property type</td>
<td>Value of index field (if different)</td>
<td>Limitations</td>
<td>Parsing method for 'indexNullAs'</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>java.time.ZoneId</td>
<td>getId() as a java.lang.String</td>
<td>-</td>
<td>ZoneId.of(String)</td>
</tr>
<tr>
<td>java.time.ZoneOffset</td>
<td>getTotalSeconds() as a java.lang.Integer</td>
<td>-</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>-</td>
<td>Period.parse(String)</td>
</tr>
<tr>
<td>java.time.Duration</td>
<td>toNanos() as a java.lang.Long</td>
<td>Possibly lower range/resolution</td>
<td>Duration.parse(String)</td>
</tr>
<tr>
<td>java.time.Year</td>
<td>-</td>
<td>Possibly lower range/resolution</td>
<td>Year.parse(String)</td>
</tr>
<tr>
<td>java.time.YearMonth</td>
<td>-</td>
<td>Possibly lower range/resolution</td>
<td>YearMonth.parse(String)</td>
</tr>
<tr>
<td>java.time.MonthDay</td>
<td>-</td>
<td>-</td>
<td>MonthDay.parse(String)</td>
</tr>
<tr>
<td>java.util.UUID</td>
<td>toString() as a java.lang.String</td>
<td>-</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.</td>
<td>See Support for legacy java.util date/time APIs.</td>
<td>ZonedDateTime.parse(String)</td>
</tr>
<tr>
<td>java.util.Date</td>
<td>Instant.ofEpochMilli(long) as a java.time.Instant.</td>
<td>See Support for legacy java.util date/time APIs.</td>
<td>Instant.parse(String)</td>
</tr>
<tr>
<td>java.sql.Timestamp</td>
<td>Instant.ofEpochMilli(long) as a java.time.Instant.</td>
<td>See Support for legacy java.util date/time APIs.</td>
<td>Instant.parse(String)</td>
</tr>
<tr>
<td>java.sql.Date</td>
<td>Instant.ofEpochMilli(long) as a java.time.Instant.</td>
<td>See Support for legacy java.util date/time APIs.</td>
<td>Instant.parse(String)</td>
</tr>
<tr>
<td>Property type</td>
<td>Value of index field (if different)</td>
<td>Limitations</td>
<td>Parsing method for 'indexNullAs'</td>
</tr>
<tr>
<td>---------------</td>
<td>------------------------------------</td>
<td>-------------</td>
<td>---------------------------------</td>
</tr>
<tr>
<td>java.sql.Time</td>
<td>Instant.ofEpochMilli(long) as a java.time.Instant.</td>
<td>See Support for legacy java.util date/time APIs.</td>
<td>Instant.parse(String)</td>
</tr>
</tbody>
</table>

**Range and resolution of date/time fields**

With a few exceptions, most date and time values are passed as-is to the backend; e.g. a `LocalDateTime` property would be passed as a `LocalDateTime` to the backend.

Internally, however, the Lucene and Elasticsearch backend use a different representation of date/time types. As a result, date and time fields stored in the index may have a smaller range and resolution than the corresponding Java type.

The documentation of each backend provides more information: see [here for Lucene](#) and [here for Elasticsearch](#).

5.4.5. 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 Assigning default bridges with the 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.

5.4.6. 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 converting it to types that are supported by the backend.

There are two cases to distinguish between:

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 extractor. See Mapping container types with container extractors for more
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.

5.5. Mapping associated elements with @IndexedEmbedded

5.5.1. Basics

Using only @Indexed combined with @*Field annotations allows indexing an entity and its direct properties, which is nice but simplistic. A real-world model will include multiple object types holding references to one another, like the authors association in the example below.
Example 23. A multi-entity model with associations

This mapping will declare the following fields in the Book index:

- **title**
- ... and nothing else.

```java
@Entity
@Indexed ①
public class Book {
    @Id
    private Integer id;

    @FullTextField(analyzer = "english") ②
    private String title;

    @ManyToMany
    private List<Author> authors = new ArrayList<>(); ③

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

```java
@Entity
public class Author {
    @Id
    private Integer id;

    private String name;

    @ManyToMany
    @MappedBy = "authors"
    private List<Book> books = new ArrayList<>();

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

① The Book entity is indexed.

② The title of the book is mapped to an index field.

③ But how to index the Author name into the Book index?

When searching for a book, users will likely need to search by author name. In the world of high-performance indexes, cross-index joins are costly and usually not an option. The best way to address such use cases is generally to copy data: when indexing a Book, just copy the name of all its authors into the Book document.
That's what `@IndexedEmbedded` does: it instructs Hibernate Search to *embed* the fields of an associated object into the main object. In the example below, it will instruct Hibernate Search to embed the `name` field defined in `Author` into `Book`, creating the field `authors.name`.

`@IndexedEmbedded` can be used on Hibernate ORM's `@Embedded` properties as well as associations (`@OneToOne`, `@OneToMany`, `@ManyToMany`, ...).
Example 24. Using `@IndexedEmbedded` to index associated elements

This mapping will declare the following fields in the `Book` index:

- `title`
- `authors.name`

```java
@Entity
@Indexed
public class Book {
    @Id
    private Integer id;
    @FullTextField(analyzer = "english")
    private String title;
    @ManyToMany
    @IndexedEmbedded
    private List<Author> authors = new ArrayList<>();
    public Book() {
    }
    // Getters and setters
    // ...
}
```

```java
@Entity
public class Author {
    @Id
    private Integer id;
    @FullTextField(analyzer = "name")
    private String name;
    @ManyToMany(mappedBy = "authors")
    private List<Book> books = new ArrayList<>();
    public Author() {
    }
    // Getters and setters
    // ...
}
```

1. Add an `@IndexedEmbedded` to the `authors` property.
2. Map `Author.name` to an index field, even though `Author` is not directly mapped to an index (no `@Indexed`).
Document identifiers are not index fields. Consequently, they will be ignored by @IndexedEmbedded.

To embed another entity’s identifier with @IndexedEmbedded, map that identifier to a field explicitly using @GenericField or another @*Field annotation.

When @IndexedEmbedded is applied to an association, i.e. to a property that refers to entities (like the example above), the association must be bi-directional. Otherwise, Hibernate Search will throw an exception on startup.

See Reindexing when embedded elements change for the reasons behind this restriction and ways to circumvent it.

Index-embedding can be nested on multiple levels; for example you can decide to index-embed the place of birth of authors, so as to be able to search for books written by Russian authors exclusively:

Example 25. Nesting multiple @IndexedEmbedded

This mapping will declare the following fields in the Book index:

- title
- authors.name
- authors.placeOfBirth.country

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

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

    @ManyToMany
    @IndexedEmbedded
    private List<Author> authors = new ArrayList<>();

    public Book() {
    }

    // Getters and setters
    // ...
}
```
@Entity
class Author {
    @Id
    private Integer id;

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

    @Embedded
    @IndexedEmbedded
    private Address placeOfBirth;

    @ManyToMany(mappedBy = "authors")
    private List<Book> books = new ArrayList<>();

    public Author() {
    }

    // Getters and setters
    // ...
}

@Embeddable
class Address {
    @FullTextField(analyzer = "name")
    private String country;

    private String city;

    private String street;

    public Address() {
    }

    // Getters and setters
    // ...
}

1. Add an @IndexedEmbedded to the authors property.
2. Map Author.name to an index field, even though Author is not directly mapped to an index (no @Indexed).
3. Add an @IndexedEmbedded to the placeOfBirth property.
4. Map Address.country to an index field, even though Address is not directly mapped to an index (no @Indexed).

By default, @IndexedEmbedded will nest other @IndexedEmbedded encountered in the indexed-embedded type recursively, without any sort of limit, which can cause infinite recursion.

To address this, see Filtering embedded fields and breaking @IndexedEmbedded cycles.
5.5.2. @IndexedEmbedded and null values

When properties targeted by an @IndexedEmbedded contain null elements, these elements are simply not indexed.

On contrary to Mapping a property to an index field with @GenericField, @FullTextField, ..., there is no indexNullAs feature to index a specific value for null objects, but you can take advantage of the exists predicate in search queries to look for documents where a given @IndexedEmbedded has or doesn’t have a value: simply pass the name of the object field to the exists predicate, for example authors in the example above.

5.5.3. @IndexedEmbedded on container types

When properties targeted by an @IndexedEmbedded have a container type (List, Optional, Map, ... ), the innermost elements will be embedded. For example for a property of type List<MyEntity>, elements of type MyEntity will be embedded.

This default behavior and ways to override it are described in the section Mapping container types with container extractors.

5.5.4. Setting the object field name with name

By default, @IndexedEmbedded will create an object field with the same name as the annotated property, and will add embedded fields to that object field. So if @IndexedEmbedded is applied to a property named authors in a Book entity, the index field name of the authors will be copied to the index field authors.name when Book is indexed.

It is possible to change the name of the object field by setting the name attribute; for example using @IndexedEmbedded(name = "allAuthors") in the example above will result in the name of authors being copied to the index field allAuthors.name instead of authors.name.

The name must not contain the dot character (.).

5.5.5. Setting the field name prefix with prefix

The prefix attribute in @IndexedEmbedded is deprecated and will ultimately be removed. Use name instead.

By default, @IndexedEmbedded will prepend the name of embedded fields with the name of the property it is applied to followed by a dot. So if @IndexedEmbedded is applied to a property named authors in a Book entity, the name field of the authors will be copied to the authors.name field when Book is indexed.

It is possible to change this prefix by setting the prefix attribute, for example
The prefix should generally be a sequence of non-dots ending with a single dot, for example `my_Property..`

Changing the prefix to a string that does not include any dot at the end (`my_Property`), or that includes a dot anywhere but at the very end (`my.Property.`), will lead to complex, undocumented, legacy behavior. Do this at your own risk.

In particular, a prefix that does not end with a dot will lead to incorrect behavior in some APIs exposed to custom bridges: the `addValue/addObject` methods that accept a field name.

### 5.5.6. Casting the target of `@IndexedEmbedded` with `targetType`

By default, the type of indexed-embedded values is detected automatically using reflection, taking into account container extraction if relevant; for example `@IndexedEmbedded List<MyEntity>` will be detected as having values of type `MyEntity`. Fields to be embedded will be inferred from the mapping of the value type and its supertypes; in the example, `@GenericField` annotations present on `MyEntity` and its superclasses will be taken into account, but annotations defined in its subclasses will be ignored.

If for some reason a schema does not expose the correct type for a property (e.g. a raw `List`, or `List<MyEntityInterface>` instead of `List<MyEntityImpl>`) it is possible to define the expected type of values by setting the `targetType` attribute in `@IndexedEmbedded`. On bootstrap, Hibernate Search will then resolve fields to be embedded based on the given target type, and at runtime it will cast values to the given target type.

Failures to cast indexed-embedded values to the designated type will be propagated and lead to indexing failure.

### 5.5.7. Reindexing when embedded elements change

When the "embedded" entity changes, Hibernate Search will handle reindexing of the "embedding" entity.

This will work transparently most of the time, as long as the association `@IndexedEmbedded` is applied to is bi-directional (uses Hibernate ORM's `mappedBy`).

When Hibernate Search is unable to handle an association, it will throw an exception on bootstrap. If this happens, refer to `Basics` to know more.
5.5.8. Filtering embedded fields and breaking `@IndexedEmbedded` cycles

By default, `@IndexedEmbedded` will "embed" everything: every field encountered in the indexed-embedded element, and every `@IndexedEmbedded` encountered in the indexed-embedded element, recursively.

This will work just fine for simpler use cases, but may lead to problems for more complex models:

- If the indexed-embedded element declares many index fields (Hibernate Search fields), only some of which are actually useful to the "index-embedding" type, the extra fields will decrease indexing performance needlessly.

- If there is a cycle of `@IndexedEmbedded` (e.g. A index-embeds b of type B, which index-embeds a of type A) the index-embedding type will end up with an infinite amount of fields (a.b.someField, a.b.a.b.someField, a.b.a.b.a.b.someField, ...), which Hibernate Search will detect and reject with an exception.

To address these problems, it is possible to filter the fields to embed, to only include those that are actually useful. Two filtering attributes are available on `@IndexedEmbedded` and may be combined:

- **includePaths**
  The paths of index fields from the indexed-embedded element that should be embedded.
  
  Provided paths must be relative to the indexed-embedded element, i.e. they must not include its name or prefix.
  
  This takes precedence over maxDepth (see below).

- **maxDepth**
  The max recursion depth for indexed-embedded processing.

  maxDepth is the number of `@IndexedEmbedded` that will be traversed and for which all fields of the indexed-embedded element will be included, even if these fields are not included explicitly through includePaths:
  
  - maxDepth=0 means fields of the indexed-embedded element are not included, nor is any field of nested indexed-embedded elements, unless these fields are included explicitly through includePaths.
  
  - maxDepth=1 means fields of the indexed-embedded element are included, but not fields of nested indexed-embedded elements, unless these fields are included explicitly through includePaths.
  
  - And so on.

  The default value depends on the value of the includePaths attribute: if includePaths is empty, the default is `Integer.MAX_VALUE` (no limit) if includePaths is not empty, the default is
Dynamic fields and filtering

Dynamic fields are not directly affected by filtering rules: a dynamic field will be included if and only if its parent is included.

This means in particular that maxDepth and includePaths constraints only need to match the nearest static parent of a dynamic field in order for that field to be included.

Below are two examples: one leveraging includePaths only, and one leveraging includePaths and maxDepth.
Example 26. Filtering indexed-embedded fields with includePaths

This mapping will declare the following fields in the Human index:

- name
- nickname
- parents.name: explicitly included because includePaths on parents includes name.
- parents.nickname: explicitly included because includePaths on parents includes nickname.
- parents.parents.name: explicitly included because includePaths on parents includes parents.name.

The following fields in particular are excluded:

- parents.parents.nickname: not implicitly included because maxDepth is not set and defaults to 0, and not explicitly included either because includePaths on parents does not include parents.nickname.
- parents.parents.parents.name: not implicitly included because maxDepth is not set and defaults to 0, and not explicitly included either because includePaths on parents does not include parents.parents.name.

```java
@Entity
@Indexed
public class Human {
    @Id
    private Integer id;

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

    @FullTextField(analyzer = "name")
    private String nickname;

    @ManyToMany
    @IndexedEmbedded(includePaths = ( "name", "nickname", "parents.name" ))
    private List<Human> parents = new ArrayList<>();

    @ManyToMany(mappedBy = "parents")
    private List<Human> children = new ArrayList<>();

    public Human() {
    }

    // Getters and setters
    // ...
}
```

Example 27. Filtering indexed-embedded fields with includePaths and maxDepth
This mapping will declare the following fields in the *Human* index:

- **name**
- **surname**

- `parents.name`: implicitly at depth 0 because `maxDepth > 0` (so `parents.*` is included implicitly).

- `parents.nickname`: implicitly included at depth 0 because `maxDepth > 0` (so `parents.*` is included implicitly).

- `parents.parents.name`: implicitly included at depth 1 because `maxDepth > 1` (so `parents.parents.*` is included implicitly).

- `parents.parents.nickname`: implicitly included at depth 1 because `maxDepth > 1` (so `parents.parents.*` is included implicitly).

- `parents.parents.parents.name`: not implicitly included at depth 2 because `maxDepth = 2` (so `parents.parents.parents.*` is included implicitly, but sub-fields must be included explicitly) but explicitly included because `includePaths` on `parents` includes `parents.parents.name`.

The following fields in particular are excluded:

- `parents.parents.parents.nickname`: not implicitly included at depth 2 because `maxDepth = 2` (so `parents.parents.parents.*` is included implicitly, but sub-fields must be included explicitly) and not explicitly included either because `includePaths` on `parents` does not include `parents.parents.name`.

- `parents.parents.parents.parents.name`: not implicitly included at depth 3 because `maxDepth = 2` (so `parents.parents.parents.parents.*` is included implicitly, but `parents.parents.parents.parents.*` and sub-fields can only be included explicitly) and not explicitly included either because `includePaths` on `parents` does not include `parents.parents.parents.parents.name`. 
5.5.9. Storing embedded elements in nested documents using storage

Indexed-embedded fields can be stored in one of two ways, configured through the `storage` attribute of the `@IndexedEmbedded` annotation. To illustrate storage options, let’s consider the following object tree, assuming the class `Book` is annotated with `@Indexed` and its `authors` property is annotated with `@IndexedEmbedded`:

- Book instance
  - title = Levianthan Wakes
  - authors =
    - Author instance
      - firstName = Daniel
      - lastName = Abraham
    - Author instance
      - firstName = Ty
      - lastName = Frank

**DEFAULT** or **FLATTENED** storage

By default, indexed-embedded fields are "flattened", meaning that the tree structure is not preserved.
The book instance mentioned above would be indexed with a structure roughly similar to this:

- Book document
  - title = Leviaathan Wakes
  - authors.firstName = [Daniel, Ty]
  - authors.lastName = [Abraham, Frank]

The authors.firstName and authors.lastName fields were "flattened" and now each has two values; the knowledge of which last name corresponds to which first name has been lost.

This is more efficient for storage and querying, but can cause unexpected behavior when querying the index on both the author's first name and the author's last name. The book given in example would show up as a match to a query such as authors.firstname:Ty AND authors.lastname:Abraham, even though "Ty Abraham" is not one of this book's authors.

**NESTED storage**

When indexed-embedded elements are "nested", the tree structure is preserved by transparently creating one separate "nested" document for every indexed-embedded element.

The book instance mentioned above would be indexed with a structure roughly similar to this:

- Book document
  - title = Leviathan Wakes
  - Nested documents
    - Nested documents for "authors"
      - authors.firstName = Daniel
      - authors.lastName = Abraham
    - Nested documents for "authors"
      - authors.firstName = Ty
      - authors.lastName = Frank

The book is effectively indexed as three documents: the root document for the book, and two internal, "nested" documents for the authors, preserving the knowledge of which last name corresponds to which first name at the cost of degraded performance when indexing and querying.

- The nested documents are "hidden" and won't directly show up in search results.
- No need to worry about nested documents being "mixed up" with root documents.

If special care is taken when building predicates on fields within nested documents, using a nested predicate, queries containing predicates on both the author's first name and the author's last name
will behave as one would (intuitively) expect. The book given in example would not show up as a match to a query such as authors.firstname:Ty AND authors.lastname:Abraham, as long as a nested predicate is used.

5.6. Mapping container types with container extractors

5.6.1. Basics

Most built-in annotations applied to properties will work transparently when applied to container types:

- `@GenericField` applied to a property of type `String` will index the property value directly.
- `@GenericField` applied to a property of type `OptionalInt` will index the optional's value (an integer).
- `@GenericField` applied to a property of type `List<String>` will index the list elements (strings).
- `@GenericField` applied to a property of type `Map<Integer, String>` will index the map values (strings).
- `@GenericField` applied to a property of type `Map<Integer, List<String>>` will index the list elements in the map values (strings).
- Etc.

Same goes for other field annotations such as `@FullTextField`, as well as `@IndexedEmbedded` in particular.

What happens behind the scenes is that Hibernate Search will inspect the property type and attempt to apply "container extractors", picking the first that works.

5.6.2. Explicit container extraction

In some cases, you will want to pick the container extractors to use explicitly. This is the case when a map's keys must be indexed, instead of the values. Relevant annotations offer an extraction attribute to configure this, as shown in the example below.

All built-in extractor names are available as constants in org.hibernated.search.mapper.pojo.extractor.builtin.BuiltinContainerExtractors.
Example 28. Mapping map keys to an index field using explicit container extractor definition

```java
@ElementCollection(①)
@JoinTable(name = "book_pricebyformat")
@MapKeyColumn(name = "format")
@Column(name = "price")
@OrderBy("format asc")
@GenericField(②)
    name = "availableFormats",
    extraction = @ContainerExtraction(BuiltinContainerExtractors.MAP_KEY) ③
)
private Map<BookFormat, BigDecimal> priceByFormat = new LinkedHashMap<>();
```

① This annotation, and those below, are just Hibernate ORM configuration.

② Declare an index field based on the `priceByFormat` property.

③ By default, Hibernate Search would index the map values (the book prices). This uses the `extraction` attribute to specify that map keys (the book formats) must be indexed instead.

When multiple levels of extractions are necessary, multiple extractors can be configured:
```
    extraction = @ContainerExtraction(BuiltinContainerExtractors.MAP_KEY,
            BuiltinContainerExtractors.OPTIONAL).
```

However, such complex mappings are unlikely since they are generally not supported by Hibernate ORM.

It is possible to implement and use custom container extractors, but at the moment these extractors will not be handled correctly for automatic reindexing, so the corresponding property must have automatic reindexing disabled.

See HSEARCH-3688 for more information.

5.6.3. Disabling container extraction

In some rare cases, container extraction is not wanted, and the `@GenericField/@IndexedEmbedded` is meant to be applied to the `List/Optional/etc.` directly. To ignore the default container extractors, most annotations offer an `extraction` attribute. Set it as below to disable extraction altogether:
Example 29. Disabling container extraction

```java
@ManyToMany
@GenericField(
    name = "authorCount",
    valueBridge = @ValueBridgeRef(type = MyCollectionSizeBridge.class),
    extraction = @ContainerExtraction(extract = ContainerExtract.NO)
)
private List<Author> authors = new ArrayList<>();
```

1. Declare an index field based on the `authors` property.
2. Instruct Hibernate Search to use the given bridge, which will extract the collection size (the number of authors).
3. Because the bridge is applied to the collection as a whole, and not to each author, the `extraction` attribute is used to disable container extraction.

5.7. Mapping geo-point types

5.7.1. Basics

Hibernate Search provides a variety of spatial features such as a distance predicate and a distance sort. These features require that spatial coordinates are indexed. More precisely, it requires that a geo-point, i.e. a latitude and longitude in the geographic coordinate system, are indexed.

Geo-points are a bit of an exception, because there isn’t any type in the standard Java library to represent them. For that reason, Hibernate Search defines its own interface, `org.hibernate.search.engine.spatial.GeoPoint`. Since your model probably uses a different type to represent geo-points, mapping geo-points requires some extra steps.

Two options are available:

- If your geo-points are represented by a dedicated, immutable type, simply use `@GenericField` and the `GeoPoint` interface, as explained here.
- For every other case, use the more complex (but more powerful) `@GeoPointBinding`, as explained here.

5.7.2. Using `@GenericField` and the `GeoPoint` interface

When geo-points are represented in your entity model by a dedicated, immutable type, you can simply make that type implement the `GeoPoint` interface, and use simple property/field mapping with `@GenericField`:
Example 30. Mapping spatial coordinates by implementing GeoPoint

```java
@Embeddable
public class MyCoordinates implements GeoPoint {
    @Basic
    private Double latitude;
    @Basic
    private Double longitude;

    protected MyCoordinates() {
        // For Hibernate ORM
    }

    public MyCoordinates(double latitude, double longitude) {
        this.latitude = latitude;
        this.longitude = longitude;
    }

    @Override
    public double latitude() {
        return latitude;
    }

    @Override
    public double longitude() {
        return longitude;
    }
}
```

```java
@Entity
@Indexed
public class Author {
    @Id
    @GeneratedValue
    private Integer id;

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

    @Embedded
    @GenericField
    private MyCoordinates placeOfBirth;

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

① Model the geo-point as an embeddable implementing `GeoPoint`. A custom type with a corresponding Hibernate ORM `UserType` would work as well.

② The geo-point type must be immutable: it does not declare any setter.

③ Apply the `@GenericField` annotation to the `placeOfBirth` property holding the coordinates. An index field named `placeOfBirth` will be added to the index. Options generally used on `@GenericField` can be used here as well.
The geo-point type must be immutable, i.e. the latitude and longitude of a given instance may never change.

This is a core assumption of @GenericField and generally all @*Field annotations: changes to the coordinates will be ignored and will not trigger reindexing as one would expect.

If the type holding your coordinates is mutable, do not use @GenericField and refer to Using @GeoPointBinding, @Latitude and @Longitude instead.

If your geo-point type is immutable, but extending the GeoPoint interface is not an option, you can also use a custom value bridge converting between the custom geo-point type and GeoPoint. GeoPoint offers static methods to quickly build a GeoPoint instance.

5.7.3. Using @GeoPointBinding, @Latitude and @Longitude

For cases where coordinates are stored in a mutable object, the solution is the @GeoPointBinding annotation. Combined with the @Latitude and @Longitude annotation, it can map the coordinates of any type that declares a latitude and longitude of type double:
Example 31. Mapping spatial coordinates using @GeoPointBinding

```java
@Entity
@Indexed
@GeoPointBinding(fieldName = "placeOfBirth")
public class Author {

    @Id
    @GeneratedValue
    private Integer id;

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

    @Latitude
    private Double placeOfBirthLatitude;

    @Longitude
    private Double placeOfBirthLongitude;

    public Author() {
    }

    // Getters and setters
    // ...
}
```

① Apply the @GeoPointBinding annotation to the type, setting fieldName to the name of the index field.

② Apply @Latitude to the property holding the latitude. It must be of double or Double type.

③ Apply @Longitude to the property holding the longitude. It must be of double or Double type.

The @GeoPointBinding annotation may also be applied to a property, in which case the @Latitude and @Longitude must be applied to properties of the property's type:

Example 32. Mapping spatial coordinates using @GeoPointBinding on a property
Model the geo-point as an embeddable. An entity would work as well.

In the geo-point type, apply `@Latitude` to the property holding the latitude.
③ In the geo-point type, apply @Longitude to the property holding the longitude.

④ The geo-point type may safely declare setters (it can be mutable).

⑤ Apply the @GeoPointBinding annotation to the property. Setting fieldName to the name of the index field is possible, but optional: the property name will be used by default.

It is possible to handle multiple sets of coordinates by applying the annotations multiple times and setting the markerSet attribute to a unique value:
Example 33. Mapping spatial coordinates using @GeoPointBinding on a property

```java
@Entity
@Indexed
@GeoPointBinding(fieldName = "placeOfBirth", markerSet = "birth") ①
@GeoPointBinding(fieldName = "placeOfDeath", markerSet = "death") ②
public class Author {
    @Id
    @GeneratedValue
    private Integer id;

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

    @Latitude(markerSet = "birth") ③
    private Double placeOfBirthLatitude;

    @Longitude(markerSet = "birth") ④
    private Double placeOfBirthLongitude;

    @Latitude(markerSet = "death") ⑤
    private Double placeOfDeathLatitude;

    @Longitude(markerSet = "death") ⑥
    private Double placeOfDeathLongitude;

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

① Apply the @GeoPointBinding annotation to the type, setting fieldName to the name of the index field, and markerSet to a unique value.

② Apply the @GeoPointBinding annotation to the type a second time, setting fieldName to the name of the index field (different from the first one), and markerSet to a unique value (different from the first one).

③ Apply @Latitude to the property holding the latitude for the first geo-point field. Set the markerSet attribute to the same value as the corresponding @GeoPointBinding annotation.

④ Apply @Longitude to the property holding the longitude for the first geo-point field. Set the markerSet attribute to the same value as the corresponding @GeoPointBinding annotation.

⑤ Apply @Latitude to the property holding the latitude for the second geo-point field. Set the markerSet attribute to the same value as the corresponding @GeoPointBinding annotation.

⑥ Apply @Longitude to the property holding the longitude for the second geo-point field. Set the markerSet attribute to the same value as the corresponding @GeoPointBinding annotation.

5.8. Tuning automatic reindexing
5.8.1. Basics

When an entity property is mapped to the index, be it through @GenericField, @IndexedEmbedded, or a custom bridge, this mapping introduces a dependency: the document will need to be updated when the property changes.

For simpler, single-entity mappings, this only means that Hibernate Search will need to detect when an entity changes and reindex the entity. This will be handled transparently.

If the mapping includes a "derived" property, i.e. a property that is not persisted directly, but instead is dynamically computed in a getter that uses other properties as input, Hibernate Search will be unable to guess which part of the persistent state these properties are based on. In this case, some explicit configuration will be required; see Reindexing when a derived value changes with @IndexingDependency for more information.

When the mapping crosses the entity boundaries, things get more complicated. Let’s consider a mapping where a Book entity is mapped to a document, and that document must include the name property of the Author entity (for example using @IndexedEmbedded). Hibernate Search will need to track changes to the author’s name, and whenever that happens, it will need to retrieve all the books of that author, so as to reindex these books automatically.

In practice, this means that whenever an entity mapping relies on an association to another entity, this association must be bi-directional: if Book.authors is @IndexedEmbedded, Hibernate Search must be aware of an inverse association Author.books. An exception will be thrown on startup if the inverse association cannot be resolved.

Most of the time, Hibernate Search is able to take advantage of Hibernate ORM metadata (the mappedBy attribute of @OneToOne and @OneToMany) to resolve the inverse side of an association, so this is all handled transparently.

In some rare cases, with the more complex mappings, it is possible that even Hibernate ORM is not aware that an association is bi-directional, because mappedBy cannot be used. A few solutions exist:

- The association can simply be ignored. This means the index will be out of date whenever associated entities change, but this can be an acceptable solution if the index is rebuilt periodically. See Disabling reindexing with @IndexingDependency for more information.
- If the association is actually bi-directional, its inverse side can be specified to Hibernate Search explicitly using @AssociationInverseSide. See Enriching the entity model with @AssociationInverseSide for more information.

5.8.2. Enriching the entity model with @AssociationInverseSide

Given an association from an entity type A to entity type B, @AssociationInverseSide defines the inverse side of an association, i.e. the path from B to A.
This is mostly useful when a bi-directional association is not mapped as such in Hibernate ORM (no `mappedBy`).

Example 34. Mapping the inverse side of an association with `@AssociationInverseSide`
① This annotation and the following ones are the Hibernate ORM mapping for a `Map<BookEdition, BigDecimal>` where the keys are `BookEdition` entities and the values are the price of that edition.

② Index-embed the editions that are actually for sale.

③ In Hibernate ORM, it is not possible to use `mappedBy` for an association modeled by a `Map` key. Thus we use `@AssociationInverseSide` to tell Hibernate Search what the inverse side of this association is.

④ We could have applied the `@AssociationInverseSide` annotation here instead: either side will do.

5.8.3. Reindexing when a derived value changes with `@IndexingDependency`

When a property is not persisted directly, but instead is dynamically computed in a getter that uses other properties as input, Hibernate Search will be unable to guess which part of the persistent state these properties are based on, and thus will be unable to trigger automatic reindexing when the relevant persistent state changes. By default, Hibernate Search will detect such cases on bootstrap and throw an exception.

Annotating the property with `@IndexingDependency(derivedFrom = ...)` will give Hibernate Search the information it needs and allow automatic reindexing.
Example 35. Mapping a derived value with @IndexingDependency.derivedFrom

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

@Id
@GeneratedValue
private Integer id;

@FullTextField(analyzer = "name")
private String title;

@ElementCollection
private List<String> authors = new ArrayList<>(); ①

public Book() {
}

@Transient ②
@FullTextField(analyzer = "name") ③
@indexingDependency(derivedFrom = @ObjectPath( ④
    @PropertyValue(propertyName = "authors")
))
public String getMainAuthor() {
    return authors.isEmpty() ? null : authors.get(0);
}
}
```

① Authors are modeled as a list of string containing the author names.

② The transient mainAuthor property dynamically returns the main author (the first one).

③ We use @FullTextField on the getMainAuthor() getter to index the name of the main author.

④ We use @IndexingDependency.derivedFrom to tell Hibernate Search that whenever the list of authors changes, the result of getMainAuthor() may have changed.

5.8.4. Disabling reindexing with @IndexingDependency

In some cases, automatic reindexing is not realistically achievable:

- When a property mapped to the index is updated very frequently, leading to a very frequent reindexing and unacceptable usage of disks or database.

- When an association is massive, for example a single entity instance is indexed-embedded in thousands of other entities.

- Etc.

When that happens, it is possible to tell Hibernate Search to ignore updates to a particular property (and, in the case of @IndexedEmbedded, anything beyond that property). The index will become slightly out-of-sync whenever the property is modified, but this can be solved by reindexing, for example every night.
Example 36. Disabling automatic reindexing with @IndexingDependency.reindexOnUpdate

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

    @Id
    private Integer id;

    @FullTextField(analyzer = "name")
    private String title;

    @ManyToOne ①
    @IndexedEmbedded ②
    @IndexingDependency(reindexOnUpdate = ReindexOnUpdate.NO) ③
    private BookCategory category;

    public Book() {
    }
}
```

① Each book has an association to a BookCategory entity. There are many, potentially thousands of books for each category.

② We want to index-embed the BookCategory into the Book ...

③ ... but we really don’t want to model the (huge) inverse association from BookCategory to Book. Thus we use @IndexingDependency.reindexOnUpdate to tell Hibernate Search that Book should not be reindexed when the content of a BookCategory changes. If we rename a BookCategory, we will need to reindex the corresponding books manually.

5.9. Changing the mapping of an existing application

Over the lifetime of an application, it will happen that the mapping of a particular indexed entity type has to change. When this happens, the mapping changes are likely to require changes to the structure of the index, i.e. its schema. Hibernate Search does not handle this structure change automatically, so manual intervention is required.

The simplest solution when the index structure needs to change is to simply:

1. Drop and re-create the index and its schema, either manually by deleting the filesystem directory for Lucene or using the REST API to delete the index for Elasticsearch, or using Hibernate Search’s schema management features.

2. Re-populate the index, for example using the mass indexer.
Technically, dropping the index and reindexing is not strictly required if the mapping changes include only:

- **adding** new indexed entities that will not have any persisted instance, e.g. adding an `@Indexed` annotation on an entity which has no rows in database.

- **adding** new fields that will be empty for all currently persisted entities, e.g. adding a new property on an entity type and mapping it to a field, but with the guarantee that this property will initially be null for every instance of this entity;

- and/or **removing** data from existing indexes/fields, e.g. removing an index field, or removing the need for a field to be stored.

However, you will still need to:

- **create missing indexes**: this can generally be done automatically by starting up the application with the `create`, `create-or-validate`, or `create-or-update` schema management strategy.

- (Elasticsearch only:) update the schema of existing indexes to declare the new fields. This will be more complex: either do it manually using Elasticsearch’s REST API, or start up the application with the `create-or-update` strategy, but be warned that it may fail.

### 5.10. Programmatic mapping

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.
Using the programmatic mapping API, it is possible to map a "dynamic-map" entity model, i.e. a model based on maps. You just need to refer to the types by their name using `context.programmaticMapping().type("thename"):`

- The entity name for dynamic entity types.
- The "role" for dynamic embedded/component types, i.e. the name of the owning entity, followed by a dot ("."), followed by the dot-separated path to the component in that entity. For example `MyEntity.myEmbedded` or `MyEntity.myEmbedded.myNestedEmbedded`.

However, support for "dynamic-map" entity models is limited. In particular:

- The document ID must be mapped to the entity ID: it can't be mapped properly, even if it is unique.
- Mass indexing dynamic-map entities is not supported.

### 5.11. Custom mapping annotations

By default, Hibernate Search only recognizes built-in mapping annotations such as `@Indexed`, `@GenericField` or `@IndexedEmbedded`.

To use custom annotations in a Hibernate Search mapping, two steps are required:

1. Implementing a processor for that annotation: `TypeMappingAnnotationProcessor` for type annotations or `PropertyMappingAnnotationProcessor` for method/field annotations.
2. Annotating the custom annotation with either `@TypeMapping` or `@PropertyMapping`, passing as an argument the reference to the annotation processor.

Once this is done, Hibernate Search will be able to detect custom annotations in indexed classes. Whenever a custom annotation is encountered, Hibernate Search will instantiate the annotation processor and call its `process` method, passing the following as arguments:

- A `mapping` parameter allowing to define the mapping for the type or property using the programmatic mapping API.
- An `annotation` parameter representing the annotation instance.
- A `context` object with various helpers.

Custom annotations are most frequently used to apply custom, parameterized bridges. You can find examples in these sections in particular:

- Passing parameters to a value bridge
• Passing parameters to a property bridge

• Passing parameters to a type bridge

It is completely possible to use custom annotations for parameter-less bridges, or even for more complex features such as indexed-embedded: every feature available in the programmatic API can be triggered by a custom annotation.

5.12. Inspecting the mapping

After Hibernate Search has successfully booted, the SearchMapping can be used to get a list of indexed entities and get more direct access to the corresponding indexes, as shown in the example below.

Example 37. Accessing indexed entities

```java
SearchMapping mapping = Search.mapping( entityManagerFactory ); ①
SearchIndexedEntity bookEntity = mapping.indexedEntity( Book.class ); ②
String jpaName = bookEntity.jpaName(); ③
IndexManager indexManager = bookEntity.indexManager(); ④
Backend backend = indexManager.backend(); ⑤

SearchIndexedEntity bookEntity2 = mapping.indexedEntity( "Book" ); ⑥
Class<? super Book> javaClass = bookEntity2.javaClass();

for ( SearchIndexedEntity entity : mapping.allIndexedEntities() ) { ⑦
    // ...
}
```

① Retrieve the SearchMapping.

② Retrieve the SearchIndexedEntity by its entity class. SearchIndexedEntity gives access to information pertaining to that entity and its index.

③ Get the JPA name of that entity.

④ Get the index manager for that entity.

⑤ Get the backend for that index manager.

⑥ Retrieve the SearchIndexedEntity by its entity name.

⑦ Retrieve all indexed entities.

From an IndexManager, you can then access the index metamodel, to inspect available fields and their main characteristics, as shown below.
Example 38. Accessing the index metamodel

```java
SearchIndexedEntity bookEntity = mapping.indexedEntity( Book.class );  
IndexManager indexManager = bookEntity.indexManager();  
IndexDescriptor indexDescriptor = indexManager.descriptor();

indexDescriptor.field( "releaseDate" ).ifPresent( field -> {  
  String path = field.absolutePath();  
  String relativeName = field.relativeName();  
  // Etc.

  if ( field.isValueField() ) {  
    IndexValueFieldDescriptor valueField = field.toValueField();  
    IndexValueFieldTypeDescriptor type = valueField.type();  
    boolean projectable = type.projectable();  
    Class<?> dslArgumentClass = type.dslArgumentClass();  
    Class<?> projectedValueClass = type.projectedValueClass();  
    Optional<String> analyzerName = type.analyzerName();  
    Optional<String> searchAnalyzerName = type.searchAnalyzerName();  
    Optional<String> normalizerName = type.normalizerName();  
    // Etc.
  }

  else if ( field.isObjectField() ) {  
    IndexObjectFieldDescriptor objectField = field.toObjectField();  
    IndexObjectFieldTypeDescriptor type = objectField.type();  
    boolean nested = type.nested();  
    // Etc.
  }
});
```

1. Retrieve a **SearchIndexedEntity**.
2. Get the index manager for that entity. **IndexManager** gives access to information pertaining to the index. This includes the metamodel, but not only (see below).
3. Get the descriptor for that index. The descriptor exposes the index metamodel.
4. Retrieve a field by name. The method returns an **Optional**, which is empty if the field does not exist.
5. The field descriptor exposes information about the field structure: path, name, parent, ...
6. Check that the field is a value field, holding a value (integer, text, ...), as opposed to object fields, holding other fields.
7. Narrow down the field descriptor to a value field descriptor.
8. Get the descriptor for the field type. The type descriptor exposes information about the field’s capabilities: is it searchable, sortable, projectable, what is the expected java class for arguments to the **Search DSL**, what are the analyzers/normalizer set on this field, ...
9. Object fields can also be inspected.

The **Backend** and **IndexManager** can also be used to **retrieve the Elasticsearch REST client** or **retrieve Lucene analyzers**.
The SearchMapping also exposes methods to retrieve an IndexManager by name, or even a whole Backend by name.
Chapter 6. Bridges

6.1. Basics

In Hibernate Search, bridges are the components responsible for converting pieces of data from the entity model to the document model.

For example, when @GenericField is applied to a property of a custom enum type, a built-in bridge will be used to convert this enum to a string when indexing, and to convert the string back to an enum when projecting.

Similarly, when a entity identifier of type Long is mapped to a document identifier, a built-in bridge will be used to convert the Long to a String (since all document identifiers are strings) when indexing, and back from a String to a Long when loading search results.

Bridges are not limited to one-to-one mapping: for example, the @GeoPointBinding annotation, which maps two properties annotated with @Latitude and @Longitude to a single field, is backed by another built-in bridge.

While built-in bridges are provided for a wide range of standard types, they may not be enough for complex models. This is why bridges are really interesting: it is possible to implement custom bridges and to refer to them in the Hibernate Search mapping. Using custom bridges, custom types can be mapped, even complex types that require user code to execute at indexing time.

There are multiple types of bridges, detailed in the next sections. If you need to implement a custom bridge, but don’t quite know which type of bridge you need, the following table may help:

<table>
<thead>
<tr>
<th>Bridge type</th>
<th>ValueBridge</th>
<th>PropertyBridge</th>
<th>TypeBridge</th>
<th>IdentifierBridge</th>
<th>RoutingKeyBridge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Built-in annotation(s)</td>
<td>@GenericField, @FulltextField,...</td>
<td>@PropertyBinding</td>
<td>@TypeBinding</td>
<td>@DocumentId</td>
<td>@RoutingKeyBinding</td>
</tr>
<tr>
<td>Maps from...</td>
<td>One property</td>
<td>One property</td>
<td>One property or more</td>
<td>One property (usually the entity identifier)</td>
<td>One property or more</td>
</tr>
<tr>
<td>Maps to...</td>
<td>One index field</td>
<td>One index field or more</td>
<td>One index field or more</td>
<td>Document identifier</td>
<td>Routing key</td>
</tr>
</tbody>
</table>
## 6.2. Value bridges

### 6.2.1. Basics

A value bridge is a pluggable component that implements the mapping of a property to an index field. It is applied to a property with a `@*Field annotation` (@GenericField, @FullTextField, …) or with a custom annotation.

A value bridge is relatively straightforward to implement: in its simplest form, it boils down to converting a value from the property type to the index field type. Thanks to the integration to the `@*Field` annotations, several features come for free:

- The type of the index field can be customized directly in the `@*Field` annotation: it can be defined as sortable, projectable, it can be assigned an analyzer, …

- The bridge can be transparently applied to elements of a container. For example, you can implement a `ValueBridge<ISBN, String>` and transparently use it on a property of type `List<ISBN>`: the bridge will simply be applied once per list element and populate the index field with as many values.

However, due to these features, several limitations are imposed on a value bridge which are not present in a property bridge for example:

- A value bridge only allows one-to-one mapping: one property to one index field. A single value bridge cannot populate more than one index field.

- A value bridge will not work correctly when applied to a mutable type. A value bridge is expected to be applied to "atomic" data, such as a `LocalDate`; if it is applied to an entity, for example, extracting data from its properties, Hibernate Search will not be aware of which properties are used and will not be able to automatically trigger reindexing when these properties change.

Below is an example of a custom value bridge that converts a custom `ISBN` type to its string representation to index it:
Example 39. Implementing and using a `ValueBridge`

```java
public class ISBNValueBridge implements ValueBridge<ISBN, String> { ①
    @Override
    public String toIndexedValue(ISBN value, ValueBridgeToIndexedValueContext context) { ②
        return value == null ? null : value.getStringValue();
    }
}
```

① The bridge must implement the `ValueBridge` interface. Two generic type arguments must be provided: the first one is the type of property values (values in the entity model), and the second one is the type of index fields (values in the document model).

② The `toIndexedValue` method is the only one that must be implemented: all other methods are optional. It takes the property value and a context object as parameters, and is expected to return the corresponding index field value. It is called when indexing, but also when parameters to the search DSL must be transformed.

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

    @Convert(converter = ISBNAttributeConverter.class) ①
    @KeywordField(valueBridge = @ValueBridgeRef(type = ISBNValueBridge.class), ③
                 normalizer = "isbn" ④)
    private ISBN isbn;

    // Getters and setters
    // ...
}
```

① This this is unrelated to the value bridge, but necessary in order for Hibernate ORM to store the data correctly in the database.

② Map the property to an index field.

③ Instruct Hibernate Search to use our custom value bridge. It is also possible to reference the bridge by its name, in the case of a CDI/Spring bean.

④ Customize the field as usual.

The example above is just a minimal implementations. A custom value bridge can do more:

- it can convert the result of projections back to the property type;
- it can parse the value passed to `indexNullAs`;

85
See the next sections for more information.

6.2.2. Type resolution

By default, the value bridge's property type and index field type are determined automatically, using reflection to extract the generic type arguments of the ValueBridge interface: the first argument is the property type while the second argument is the index field type.

For example, in `public class MyBridge implements ValueBridge<ISBN, String>`, the property type is resolved to ISBN and the index field type is resolved to String: the bridge will be applied to properties of type ISBN and will populate an index field of type String.

The fact that types are resolved automatically using reflection brings a few limitations. In particular, it means the generic type arguments cannot be just anything; as a general rule, you should stick to literal types (`MyBridge implements ValueBridge<ISBN, String>`) and avoid generic type parameters and wildcards (`MyBridge<T> implements ValueBridge<List<T>, T>`).

If you need more complex types, you can bypass the automatic resolution and specify types explicitly using a ValueBinder.

6.2.3. Using value bridges in other *@Field annotations*

In order to use a custom value bridge with specialized annotations such as `@FullTextField`, the bridge must declare a compatible index field type.

For example:

- `@FullTextField` and `@KeywordField` require an index field type of type String (`ValueBridge<Whatever, String>`);
- `@ScaledNumberField` requires an index field type of type BigDecimal (`ValueBridge<Whatever, BigDecimal>`) or BigInteger (`ValueBridge<Whatever, BigInteger>`).

Refer to Available field annotations for the specific constraints of each annotation.

Attempts to use a bridge that declares an incompatible type will trigger exceptions at bootstrap.

6.2.4. Supporting projections with fromIndexedValue()

By default, any attempt to project on a field using a custom bridge will result in an exception, because Hibernate Search doesn't know how to convert the projected values obtained from the index back to the property type.
It is possible to disable conversion explicitly to get the raw value from the index, but another way of solving the problem is to simply implement `fromIndexedValue` in the custom bridge. This method will be called whenever a projected value needs to be converted.

*Example 40. Implementing `fromIndexedValue` to convert projected values*

```java
public class ISBNValueBridge implements ValueBridge<ISBN, String> {
    @Override
    public String toIndexedValue(ISBN value, ValueBridgeToIndexedValueContext context) {
        return value == null ? null : value.getStringValue();
    }

    @Override
    public ISBN fromIndexedValue(String value, ValueBridgeFromIndexedValueContext context) {
        return value == null ? null : ISBN.parse(value);
    }
}
```

① Implement `fromIndexedValue` as necessary.

```java
@Entity
@Indexed
public class Book {
    @Id
    @GeneratedValue
    private Integer id;
    @Convert(converter = ISBNAttributeConverter.class) ①
    @KeywordField( ②
        valueBridge = @ValueBridgeRef(type = ISBNValueBridge.class), ③
        normalizer = "isbn",
        projectable = Projectable.YES ④
    )
    private ISBN isbn;
    // Getters and setters
    // ...
}
```

① This is unrelated to the value bridge, but necessary in order for Hibernate ORM to store the data correctly in the database.

② Map the property to an index field.

③ Instruct Hibernate Search to use our custom value bridge.

④ Do not forget to configure the field as projectable.

### 6.2.5. Supporting `indexNullAs` with `parse()`

By default, the `indexNullAs` attribute of `@*Field` annotations cannot be used together with a custom bridge.
In order to make it work, the bridge needs to implement the `parse` method so that Hibernate Search can convert the string assigned to `indexNullAs` to a value of the correct type for the index field.

**Example 41. Implementing `parse` to support `indexNullAs`**

```java
public class ISBNValueBridge implements ValueBridge<ISBN, String> {
    @Override
    public String toIndexedValue(ISBN value, ValueBridgeToIndexedValueContext context) {
        return value == null ? null : value.getStringValue();
    }

    @Override
    public String parse(String value) {
        // Just check the string format and return the string
        return ISBN.parse(value).getStringValue();
    }
}
```

① Implement `parse` as necessary. The bridge may throw exceptions for invalid strings.

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

    @Convert(converter = ISBNAttributeConverter.class) ①
    @KeywordField(valueBridge = @ValueBridgeRef(type = ISBNValueBridge.class), ③
                normalizer = "isbn",
                indexNullAs = "000-0-00-000000-0" ④
    )
    private ISBN isbn;
    // Getters and setters
    // ...
}
```

① This is unrelated to the value bridge, but necessary in order for Hibernate ORM to store the data correctly in the database.

② Map the property to an index field.

③ Instruct Hibernate Search to use our custom value bridge.

④ Set `indexNullAs` to a valid value.

### 6.2.6. Compatibility across indexes with `isCompatibleWith()`

A value bridges is involved in indexing, but also in the various search DSLs, to convert values passed to the DSL to an index field value that the backend will understand.

When creating a predicate targeting a single field across multiple indexes, Hibernate Search will have
multiple bridges to choose from: one per index. Since only one predicate with a single value can be created, Hibernate Search needs to pick a single bridge. By default, when a custom bridge is assigned to the field, Hibernate Search will throw an exception because it cannot decide which bridge to pick.

If the bridges assigned to the field in all indexes produce the same result, it is possible to indicate to Hibernate Search that any bridge will do by implementing `isCompatibleWith`.

This method accepts another bridge in parameter, and returns `true` if that bridge can be expected to always behave the same as this.

**Example 42. Implementing `isCompatibleWith` to support multi-index search**

```java
public class ISBNValueBridge implements ValueBridge<ISBN, String> { ➊
    @Override
    public String toIndexedValue(ISBN value, ValueBridgeToIndexedValueContext context) { ➋
        return value == null ? null : value.getStringValue();
    }

    @Override
    public boolean isCompatibleWith(ValueBridge<?, ?> other) {
        return getClass().equals(other.getClass());
    }
}

➊ Implement `isCompatibleWith` as necessary. Here we just deem any instance of the same class to be compatible.
```

6.2.7. Configuring the bridge more finely with `ValueBinder`

To configure a bridge more finely, it is possible to implement a value binder that will be executed at bootstrap. This binder will be able in particular to define a custom index field type.

**Example 43. Implementing a `ValueBinder`**

```java
```
public class ISBNValueBinder implements ValueBinder {

@Override
public void bind(ValueBindingContext<?> context) {
    context.bridge(ISBN.class, new ISBNValueBridge(), context.typeFactory().asString().normalizer("isbn")
);)
}

private static class ISBNValueBridge implements ValueBridge<ISBN, String> {
    @Override
    public String toIndexedValue(ISBN value, ValueBridgeToIndexedValueContext context) {
        return value == null ? null : value.getStringValue();
    }
}

① The binder must implement the ValueBinder interface.

② Implement the bind method.

③ Call context.bridge(...) to define the value bridge to use.

④ Pass the expected type of property values.

⑤ Pass the value bridge instance.

⑥ Use the context's type factory to create an index field type.

⑦ Pick a base type for the index field using an as*() method.

⑧ Configure the type as necessary. This configuration will set defaults that are applied for any type using this bridge, but they can be overridden. Type configuration is similar to the attributes found in the various @*Field annotations. See Defining index field types for more information.

⑨ The value bridge must still be implemented. Here the bridge class is nested in the binder class, because it is more convenient, but you are obviously free to implement it in a separate java file.

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

    @Convert(converter = ISBNAttributeConverter.class)
    @KeywordField(valueBinder = @ValueBinderRef(type = ISBNValueBinder.class), sortable = Sortable.YES)
    private ISBN isbn;

    // Getters and setters
    // ...
}
This is unrelated to the value bridge, but necessary in order for Hibernate ORM to store the data correctly in the database.

Map the property to an index field.

Instruct Hibernate Search to use our custom value binder. Note the use of `valueBinder` instead of `valueBridge`. It is also possible to reference the binder by its name, in the case of a CDI/Spring bean.

Customize the field as usual. Configuration set using annotation attributes take precedence over the index field type configuration set by the value binder. For example, in this case, the field with be sortable even if the binder didn’t define the field as sortable.

When using a value binder with a specialized `@Field` annotation, the index field type must be compatible with the annotation.

For example, `@FullTextField` will only work if the index field type was created using `asString()`.

These restrictions are similar to those when assigning a value bridge directly; see Using value bridges in other `@Field` annotations.

### 6.2.8. Passing parameters

The value bridges are usually applied with built-in `@Field` annotation, which already accept parameters to configure the field name, whether the field is sortable, etc.

However, these parameters are not passed to the value bridge or value binder. In some cases, it is necessary to pass parameters directly to the value bridge or value binder. This is achieved by defining a custom annotation with attributes:

*Example 44. Passing parameters to a `ValueBridge`*

```java
class BooleanAsStringBridge implements ValueBridge<Boolean, String> {
    private final String trueAsString;
    private final String falseAsString;

    BooleanAsStringBridge(String trueAsString, String falseAsString) {
        this.trueAsString = trueAsString;
        this.falseAsString = falseAsString;
    }

    @Override
    public String toIndexedValue(Boolean value, ValueBridgeToIndexedValueContext context) {
        if (value == null) {
            return null;
        }
        return value ? trueAsString : falseAsString;
    }
}
```
Implement a bridge that does not index booleans directly, but indexes them as strings instead.

The bridge accepts two parameters in its constructors: the string representing `true` and the string representing `false`.

```java
@Retention(RetentionPolicy.RUNTIME) @Target({ElementType.METHOD, ElementType.FIELD}) @PropertyMapping(processor = @PropertyMappingAnnotationProcessorRef(type = BooleanAsStringField.Processor.class))
@Documented @Repeatable(BooleanAsStringField.List.class)
public @interface BooleanAsStringField {
    String trueAsString default "true";
    String falseAsString default "false";
    String name default "";
    ContainerExtraction extraction default @ContainerExtraction();
}

class Processor implements PropertyMappingAnnotationProcessor<BooleanAsStringField> {
    @Override
    public void process(PropertyMappingStep mapping, BooleanAsStringField annotation, PropertyMappingAnnotationProcessorContext context) {
        BooleanAsStringBridge bridge = new BooleanAsStringBridge(annotation.trueAsString(), annotation.falseAsString());
        mapping.genericField( annotation.name().isEmpty() ? null : annotation.name() ).valueBridge( bridge ).extractors( context.toContainerExtractorPath( annotation.extraction() ) );
    }
}
```

Define an annotation with retention **RUNTIME**. Any other retention policy will cause the annotation to be ignored by Hibernate Search.

Since we’re defining a value bridge, allow the annotation to target either methods (getters) or fields.

Mark this annotation as a property mapping, and instruct Hibernate Search to apply the given processor whenever it finds this annotation. It is also possible to reference the processor by its name, in the case of a CDI/Spring bean.

Optionally, mark the annotation as documented, so that it is included in the javadoc of your entities.

Optionally, mark the annotation as repeatable, in order to be able to declare multiple fields on the same property.
Define custom attributes to configure the value bridge. Here we define two strings that the bridge should use to represent the boolean values true and false.

Since we will be using a custom annotation, and not the built-in *@Field annotation, the standard parameters that make sense for this bridge need to be declared here, too.

The processor must implement the PropertyMappingAnnotationProcessor interface, setting its generic type argument to the type of the corresponding annotation. Here the processor class is nested in the annotation class, because it is more convenient, but you are obviously free to implement it in a separate Java file.

In the process method, instantiate the bridge and pass the annotation attributes as constructor arguments.

Declare the field with the configured name (if provided).

Assign our bridge to the field. Alternatively, we could assign a value binder instead, using the valueBinder() method.

Configure the remaining standard parameters. Note that the context object passed to the process method exposes utility methods to convert standard Hibernate Search annotations to something that can be passed to the mapping (here, @ContainerExtraction is converted to a container extractor path).

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

    @Id
    @GeneratedValue
    private Integer id;

    private String title;

    @BooleanAsStringField(trueAsString = "yes", falseAsString = "no")  
    private boolean published;

    @ElementCollection
    @BooleanAsStringField(  
        name = "censorshipAssessments_allYears",  
        trueAsString = "passed", falseAsString = "failed"  
    )
    private Map<Year, Boolean> censorshipAssessments = new HashMap<>();

    // Getters and setters
    // ...
}
```

Apply the bridge using its custom annotation, setting the parameters.

Because we use a value bridge, the annotation can be transparently applied to containers. Here, the bridge will be applied successively to each value in the map.
6.2.9. Accessing the ORM session or session factory from the bridge

Contexts passed to the bridge methods can be used to retrieve the Hibernate ORM session or session factory.

Example 45. Retrieving the ORM session or session factory from a ValueBridge

```java
public class MyDataValueBridge implements ValueBridge<MyData, String> {
    @Override
    public String toIndexedValue(MyData value, ValueBridgeToIndexedValueContext context) {
        SessionFactory sessionFactory = context.extension(HibernateOrmExtension.get()); ①
        sessionFactory(); ②
        // ... do something with the factory ...
    }

    @Override
    public MyData fromIndexedValue(String value, ValueBridgeFromIndexedValueContext context) {
        Session session = context.extension(HibernateOrmExtension.get()); ③
        session(); ④
        // ... do something with the session ...
    }
}
```

① Apply an extension to the context to access content specific to Hibernate ORM.
② Retrieve the SessionFactory from the extended context. The Session is not available here.
③ Apply an extension to the context to access content specific to Hibernate ORM.
④ Retrieve the Session from the extended context.

6.2.10. Injecting beans into the value bridge or value binder

With compatible frameworks, Hibernate Search supports injection of beans into both the ValueBridge and the ValueBinder.

This only applies to beans instantiated by Hibernate Search itself. As a rule of thumb, if you need to call new MyBridge() at some point, the bridge won't get auto-magically injected.

The context passed to the value binder’s bind method also exposes a getBeanResolver method to access the bean resolver and instantiate beans explicitly.

See Bean injection for more details.

6.2.11. Experimental features
These features are experimental. Usual compatibility policies do not apply: incompatible changes may be introduced in any future release.

The context passed to the value binder’s `bind` method exposes a `bridgedElement()` method that gives access to metadata about the value being bound, in particular its type.

See the javadoc for more information.

6.3. Property bridge

6.3.1. Basics

A property bridge, like a value bridge, is a pluggable component that implements the mapping of a property to one or more index fields. It is applied to a property with the `@PropertyBinding` annotation or with a custom annotation.

Compared to the value bridge, the property bridge is more complex to implement, but covers a broader range of use cases:

- A property bridge can map a single property to more than one index field.
- A property bridge can work correctly when applied to a mutable type, provided it is implemented correctly.

However, due to its rather flexible nature, the property bridge does not transparently provide all the features that come for free with a value bridge. They can be supported, but have to be implemented manually. This includes in particular container extractors, which cannot be combined with a property bridge: the property bridge must extract container values explicitly.

Implementing a property bridge requires two components:

1. A custom implementation of `PropertyBinder`, to bind the bridge to a property at bootstrap. This involves declaring the parts of the property that will be used, declaring the index fields that will be populated along with their type, and instantiating the property bridge.

2. A custom implementation of `PropertyBridge`, to perform the conversion at runtime. This involves extracting data from the property, transforming it if necessary, and pushing it to index fields.

Below is an example of a custom property bridge that maps a list of invoice line items to several fields summarizing the invoice.

*Example 46. Implementing and using a `PropertyBridge`*
public class InvoiceLineItemsSummaryBinder implements PropertyBinder {

@Override
public void bind(PropertyBindingContext context) {
    context.dependencies().use("category")
    .use("amount");

    IndexSchemaObjectField summaryField = context.indexSchemaElement("summary");

    IndexFieldType<BigDecimal> amountFieldType = context.typeFactory().asBigDecimal().decimalScale(2).toIndexFieldType();

    context.bridge(new Bridge(
        summaryField.toReference(),
        summaryField.field("total", amountFieldType).toReference(),
        summaryField.field("books", amountFieldType).toReference(),
        summaryField.field("shipping", amountFieldType).toReference()
    ));
}

// ... class continues below

① The binder must implement the PropertyBinder interface.

② Implement the bind method in the binder.

③ Declare the dependencies of the bridge, i.e. the parts of the property value that the bridge will actually use. This is absolutely necessary in order for Hibernate Search to correctly trigger reindexing when these parts are modified. See Declaring dependencies to bridged elements for more information about declaring dependencies.

④ Declare the fields that are populated by this bridge. In this case we're creating a summary object field, which will have multiple sub-fields (see below). See Declaring and writing to index fields for more information about declaring index fields.

⑤ Declare the type of the sub-fields. We're going to index monetary amounts, so we will use a BigDecimal type with two digits after the decimal point. See Defining index field types for more information about declaring index field types.

⑥ Call context.bridge(...) to define the property bridge to use, and pass an instance of the bridge.

⑦ Pass a reference to the summary object field to the bridge.

⑧ Create a sub-field for the total amount of the invoice, a sub-field for the sub-total for books, and a sub-field for the sub-total for shipping. Pass references to these fields to the bridge.
private static class Bridge implements PropertyBridge {

    private final IndexObjectFieldReference summaryField;
    private final IndexFieldReference<BigDecimal> totalField;
    private final IndexFieldReference<BigDecimal> booksField;
    private final IndexFieldReference<BigDecimal> shippingField;

    Bridge(IndexObjectFieldReference summaryField, 
        IndexFieldReference<BigDecimal> totalField,
        IndexFieldReference<BigDecimal> booksField,
        IndexFieldReference<BigDecimal> shippingField) {
        this.summaryField = summaryField;
        this.totalField = totalField;
        this.booksField = booksField;
        this.shippingField = shippingField;
    }

    @Override
    @SuppressWarnings("unchecked")
    public void write(DocumentElement target, Object bridgedElement,
        PropertyBridgeWriteContext context) {
        List<InvoiceLineItem> lineItems = (List<InvoiceLineItem>) bridgedElement;

        BigDecimal total = BigDecimal.ZERO;
        BigDecimal books = BigDecimal.ZERO;
        BigDecimal shipping = BigDecimal.ZERO;
        for (InvoiceLineItem lineItem : lineItems) {
            BigDecimal amount = lineItem.getAmount();
            total = total.add(amount);
            switch (lineItem.getCategory()) {
                case BOOK:
                    books = books.add(amount);
                    break;
                case SHIPPING:
                    shipping = shipping.add(amount);
                    break;
            }
        }

        DocumentElement summary = target.addObject( this.summaryField );
        summary.addValue( this.totalField, total );
        summary.addValue( this.booksField, books );
        summary.addValue( this.shippingField, shipping );
    }
}

1. The bridge must implement the PropertyBridge interface. Here the bridge class is nested in
   the binder class, because it is more convenient, but you are obviously free to implement it in a
   separate java file.

2. The bridge stores references to the fields: it will need them when indexing.

3. Implement the write method in the bridge. This method is called on indexing.

4. The bridged element is passed as an Object, so cast it to the correct type.

5. Extract data from the bridged element, and optionally transform it.

6. Add an object to the summary object field. Note the summary field was declared at the root, so
   we call addObject directly on the target argument.
⑦ Add a value to each of the `summary.total`, `summary.books` and `summary.shipping` fields. Note the fields were declared as sub-fields of `summary`, so we call `addValue` on `summaryValue` instead of `target`.

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

@Id
@GeneratedValue
private Integer id;

@ElementCollection
@OrderColumn
@propertyBinding(binder = @PropertyBinderRef(type = InvoiceLineItemsSummaryBinder.class)) ①
private List<InvoiceLineItem> lineItems = new ArrayList<>();

// Getters and setters
// ...
}
```

① Apply the bridge using the `@PropertyBinding` annotation.

### 6.3.2. Passing parameters

The property bridges are usually applied with the built-in `@PropertyBinding` annotation, which does not accept any parameter other than the property binder.

In some cases, it is necessary to pass parameters directly to the property binder. This is achieved by defining a custom annotation with attributes:

**Example 47. Passing parameters to a PropertyBinder**
Define an annotation with retention \texttt{RUNTIME}. Any other retention policy will cause the annotation to be ignored by Hibernate Search.

Since we’re defining a property bridge, allow the annotation to target either methods (getters) or fields.

Mark this annotation as a property mapping, and instruct Hibernate Search to apply the given processor whenever it finds this annotation. It is also possible to reference the processor by its name, in the case of a CDI/Spring bean.

Optionally, mark the annotation as documented, so that it is included in the javadoc of your entities.

Define an attribute of type \texttt{String} to specify the field name.

The processor must implement the \texttt{PropertyMappingAnnotationProcessor} interface, setting its generic type argument to the type of the corresponding annotation. Here the processor class is nested in the annotation class, because it is more convenient, but you are obviously free to implement it in a separate Java file.

In the annotation processor, instantiate the binder.

Process the annotation attributes and pass the data to the binder. Here we’re using a setter, but passing the data through the constructor would work, too.

Apply the binder to the property.
public class InvoiceLineItemsSummaryBinder implements PropertyBinder {
    private String fieldName = "summary";

    public InvoiceLineItemsSummaryBinder fieldName(String fieldName) {  
        this.fieldName = fieldName;
        return this;
    }

    @Override
    public void bind(PropertyBindingContext context) {
        context.dependencies().
            .use("category")
            .use("amount");

        IndexSchemaObjectField summaryField = context.indexSchemaElement().
            objectField(fieldName);

        IndexFieldType<BigDecimal> amountFieldType = context.typeFactory()
            .asBigDecimal().
            decimalScale(2).toIndexFieldType();

        context.bridge(new Bridge(
            summaryField.toReference(),
            summaryField.field("total", amountFieldType).toReference(),
            summaryField.field("books", amountFieldType).toReference(),
            summaryField.field("shipping", amountFieldType).toReference()
        ));
    }

    private static class Bridge implements PropertyBridge {
        /* ... same implementation as before ... */
    }
}

① Implement setters in the binder. Alternatively, we could expose a parameterized constructor.

② In the bind method, use the value of parameters. Here use the fieldName parameter to set the field name, but we could pass parameters for any purpose: defining the field as sortable, defining a normalizer, ...

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

    @ElementCollection
    @OrderColumn
    @InvoiceLineItemsSummaryBinding(fieldName = "itemSummary")
    private List<InvoiceLineItem> lineItems = new ArrayList<>();

    // Getters and setters
    // ...
}

① Apply the bridge using its custom annotation, setting the fieldName parameter.
6.3.3. Accessing the ORM session from the bridge

Contexts passed to the bridge methods can be used to retrieve the Hibernate ORM session.

**Example 48. Retrieving the ORM session from a `PropertyBridge`**

```java
private static class Bridge implements PropertyBridge {

    private final IndexFieldReference<String> field;

    private Bridge(IndexFieldReference<String> field) {
        this.field = field;
    }

    @Override
    public void write(DocumentElement target, Object bridgedElement, PropertyBridgeWriteContext context) {
        Session session = context.extension(HibernateOrmExtension.get()).session();  // ①
        // ... do something with the session ...
    }
}
```

① Apply an extension to the context to access content specific to Hibernate ORM.
② Retrieve the `Session` from the extended context.

6.3.4. Injecting beans into the binder

With compatible frameworks, Hibernate Search supports injection of beans into the `PropertyMappingAnnotationProcessor`.

The context passed to the property binder’s `bind` method also exposes a `getBeanResolver` method to access the bean resolver and instantiate beans explicitly.

See `Bean injection` for more details.

6.3.5. Experimental features

These features are **experimental**. Usual compatibility policies do not apply: incompatible changes may be introduced in any future release.

The context passed to the property binder’s `bind` method exposes a `bridgedElement()` method that gives access to metadata about the property being bound, in particular its name and type.

The metadata can also be used to inspect the type of the property in details:

- Getting accessors to properties.
- Detecting properties with markers. Markers are applied by specific annotations carrying a
@MarkerBinding meta-annotation.

See the javadoc for more information.

# 6.4. Type bridge

## 6.4.1. Basics

A type bridge is a pluggable component that implements the mapping of a whole type to one or more index fields. It is applied to a type with the `@TypeBinding` annotation or with a custom annotation.

The type bridge is very similar to the property bridge in its core principles and in how it is implemented. The only (obvious) difference is that the property bridge is applied to properties (fields or getters), while the type bridge is applied to the type (class or interface). This entails some slight differences in the APIs exposed to the type bridge.

Implementing a type bridge requires two components:

1. A custom implementation of `TypeBinder`, to bind the bridge to a type at bootstrap. This involves declaring the properties of the type that will be used, declaring the index fields that will be populated along with their type, and instantiating the type bridge.

2. A custom implementation of `TypeBridge`, to perform the conversion at runtime. This involves extracting data from an instance of the type, transforming the data if necessary, and pushing it to index fields.

Below is an example of a custom type bridge that maps two properties of the `Author` class, the `firstName` and `lastName`, to a single `fullName` field.

### Example 49. Implementing and using a `TypeBridge`

```java
public class FullNameBinder implements TypeBinder {  // ①
    @Override
    public void bind(TypeBindingContext context) {  // ②
        context.dependencies()  // ③
            .use("firstName")
            .use("lastName");

        IndexFieldReference<String> fullNameField = context.indexSchemaElement()  // ④
            .field("fullName", f -> f.asString().analyzer("name")  // ⑤
                .toReference();
        context.bridge(new Bridge(  // ⑥
            fullNameField  // ⑦
        ));
    }

    // ... class continues below
}
```

① The binder must implement the `TypeBinder` interface.
② Implement the `bind` method in the binder.

③ Declare the dependencies of the bridge, i.e. the parts of the type instances that the bridge will actually use. This is absolutely necessary in order for Hibernate Search to correctly trigger reindexing when these parts are modified. See Declaring dependencies to bridged elements for more information about declaring dependencies.

④ Declare the field that will be populated by this bridge. In this case we’re creating a single `fullName` String field. Multiple index fields can be declared. See Declaring and writing to index fields for more information about declaring index fields.

⑤ Declare the type of the field. Since we’re indexing a full name, we will use a `String` type with a `name` analyzer (defined separately, see Analysis). See Defining index field types for more information about declaring index field types.

⑥ Call `context.bridge(...)` to define the type bridge to use, and pass an instance of the bridge.

⑦ Pass a reference to the `fullName` field to the bridge.

```java
// ... class FullNameBinder (continued)

private static class Bridge implements TypeBridge {  
    private final IndexFieldReference<String> fullNameField;

    private Bridge(IndexFieldReference<String> fullNameField) {  
        this.fullNameField = fullNameField;
    }

    @Override
    public void write(DocumentElement target, Object bridgedElement,
        TypeBridgeWriteContext context) {  
        Author author = (Author) bridgedElement;  
        String fullName = author.getLastName() + " " + author.getFirstName();  
        target.addValue(this.fullNameField, fullName);
    }
}
```

① The bridge must implement the `TypeBridge` interface. Here the bridge class is nested in the binder class, because it is more convenient, but you are obviously free to implement it in a separate java file.

② The bridge stores references to the fields: it will need them when indexing.

③ Implement the `write` method in the bridge. This method is called on indexing.

④ The bridged element is passed as an `Object`, so cast it to the correct type.

⑤ Extract data from the bridged element, and optionally transform it.

⑥ Set the value of the `fullName` field. Note the `fullName` field was declared at the root, so we call `addValue` directly on the `target` argument.
Apply the bridge using the @TypeBinding annotation.

It is still possible to map properties directly using other annotations, as long as index field names are distinct from the names used in the type binder. But no annotation is necessary on the firstName and lastName properties: these are already handled by the bridge.

6.4.2. Passing parameters

Type bridges are usually applied with the built-in @TypeBinding annotation, which does not accept any parameter other than the type binder.

In some cases, it is necessary to pass parameters directly to the type binder. This is achieved by defining a custom annotation with attributes:

Example 50. Passing parameters to a TypeBinder
1. Define an annotation with retention **RUNTIME**. Any other retention policy will cause the annotation to be ignored by Hibernate Search.

2. Since we're defining a type bridge, allow the annotation to target types.

3. Mark this annotation as a type mapping, and instruct Hibernate Search to apply the given binder whenever it finds this annotation. It is also possible to reference the binder by its name, in the case of a CDI/Spring bean.

4. Optionally, mark the annotation as documented, so that it is included in the javadoc of your entities.

5. Define an attribute of type **boolean** to specify whether a sort field should be added.

6. The processor must implement the `TypeMappingAnnotationProcessor` interface, setting its generic type argument to the type of the corresponding annotation. Here the processor class is nested in the annotation class, because it is more convenient, but you are obviously free to implement it in a separate Java file.

7. In the annotation processor, instantiate the binder.

8. Process the annotation attributes and pass the data to the binder. Here we're using a setter, but passing the data through the constructor would work, too.

9. Apply the binder to the type.
public class FullNameBinder implements TypeBinder {

    private boolean sortField;

    public FullNameBinder sortField(boolean sortField) {
        this.sortField = sortField;
        return this;
    }

    @Override
    public void bind(TypeBindingContext context) {
        context.dependencies().
            use( "firstName" )
            .use( "lastName" );

        IndexFieldReference<String> fullNameField = context.indexSchemaElement()
            .field( "fullName", f -> f.asString().analyzer( "name" ) )
            .toReference();

        IndexFieldReference<String> fullNameSortField = null;
        if ( this.sortField ) {
            fullNameSortField = context.indexSchemaElement()
                .field( "fullName_sort",
                    f -> f.asString().normalizer( "name" ).sortable( Sortable.YES )
                )
                .toReference();
        }

        context.bridge( new Bridge( fullNameField, fullNameSortField ) );
    }

    private static class Bridge implements TypeBridge {

        private final IndexFieldReference<String> fullNameField;
        private final IndexFieldReference<String> fullNameSortField;

        private Bridge(IndexFieldReference<String> fullNameField,
            IndexFieldReference<String> fullNameSortField) {
            this.fullNameField = fullNameField;
            this.fullNameSortField = fullNameSortField;
        }

        @Override
        public void write(DocumentElement target, Object bridgedElement,
            TypeBridgeWriteContext context) {
            Author author = (Author) bridgedElement;
            String fullName = author.getLastName() + " " + author.getFirstName();

            target.addValue( this.fullNameField, fullName );
            if ( this.fullNameSortField != null ) {
                target.addValue( this.fullNameSortField, fullName );
            }
        }
    }
}

① Implement setters in the binder. Alternatively, we could expose a parameterized constructor.

② In the bind method, use the value of parameters. Here use the sortField parameter to decide whether to add a additional, sortable field, but we could pass parameters for any purpose:
defining the field name, defining a normalizer, custom annotation ...

```java
@Entity
@Indexed
@AllArgsConstructor
public class Author {
    @Id
    private Integer id;
    private String firstName;
    private String lastName;
    // Getters and setters
}
```

① Apply the bridge using its custom annotation, setting the `sortField` parameter.

### 6.4.3. Accessing the ORM session from the bridge

Contexts passed to the bridge methods can be used to retrieve the Hibernate ORM session.

**Example 51. Retrieving the ORM session from a `TypeBridge`**

```java
private static class Bridge implements TypeBridge {
    private final IndexFieldReference<String> field;
    private Bridge(IndexFieldReference<String> field) {
        this.field = field;
    }
    @Override
    public void write(DocumentElement target, Object bridgedElement, TypeBridgeWriteContext context) {
        Session session = context.extension(HibernateOrmExtension.get()).session(); ②
        // ... do something with the session ...
    }
}
```

① Apply an extension to the context to access content specific to Hibernate ORM.

② Retrieve the `Session` from the extended context.

### 6.4.4. Injecting beans into the binder

With compatible frameworks, Hibernate Search supports injection of beans into the `TypeMappingAnnotationProcessor`.

The context passed to the type binder’s `bind` method also exposes a `getBeanResolver` method to
access the bean resolver and instantiate beans explicitly.

See Bean injection for more details.

### 6.4.5. Experimental features

These features are experimental. Usual compatibility policies do not apply: incompatible changes may be introduced in any future release.

The context passed to the type binder’s bind method exposes a bridgedElement() method that gives access to metadata about the type being bound.

The metadata can in particular be used to inspect the type in details:

- Getting accessors to properties.
- Detecting properties with markers. Markers are applied by specific annotations carrying a @MarkerBinding meta-annotation.

See the javadoc for more information.

### 6.5. Identifier bridges

#### 6.5.1. Basics

An identifier bridge is a pluggable component that implements the mapping of an entity property to a document identifier. It is applied to a property with the @DocumentId annotation or with a custom annotation.

Implementing an identifier bridge boils down to implementing two methods:

- one method to convert the property value (any type) to the document identifier (a string);
- one method to convert the document identifier back to the original property value.

Below is an example of a custom identifier bridge that converts a custom BookId type to its string representation and back:
Example 52. Implementing and using an IdentifierBridge

```java
public class BookIdBridge implements IdentifierBridge<BookId> {
    @Override
    public String toDocumentIdentifier(BookId value, IdentifierBridgeToDocumentIdentifierContext context) {
        return value.getPublisherId() + "/" + value.getPublisherSpecificBookId();
    }

    @Override
    public BookId fromDocumentIdentifier(String documentIdentifier, IdentifierBridgeFromDocumentIdentifierContext context) {
        String[] split = documentIdentifier.split("/");
        return new BookId(Long.parseLong(split[0]), Long.parseLong(split[1]));
    }
}
```

1. The bridge must implement the IdentifierBridge interface. One generic parameter must be provided: the type of property values (values in the entity model).

2. The toDocumentIdentifier method takes the property value and a context object as parameters, and is expected to return the corresponding document identifier. It is called when indexing, but also when parameters to the search DSL must be transformed, in particular for the ID predicate.

3. The fromDocumentIdentifier method takes the document identifier and a context object as parameters, and is expected to return the original property value. It is called when mapping search hits to the corresponding entity.

```java
@Entity
@Indexed
public class Book {
    @EmbeddedId
    @DocumentId(identifierBridge = @IdentifierBridgeRef(type = BookIdBridge.class))
    private BookId id = new BookId();

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

    // Getters and setters
}
```

1. Map the property to the document identifier.

2. Instruct Hibernate Search to use our custom identifier bridge. It is also possible to reference the bridge by its name, in the case of a CDI/Spring bean.
6.5.2. Type resolution

By default, the identifier bridge’s property type is determined automatically, using reflection to extract the generic type argument of the `IdentifierBridge` interface.

For example, in `public class MyBridge implements IdentifierBridge<BookId>`, the property type is resolved to `BookId`: the bridge will be applied to properties of type `BookId`.

The fact that the type is resolved automatically using reflection brings a few limitations. In particular, it means the generic type argument cannot be just anything; as a general rule, you should stick to literal types (`MyBridge implements IdentifierBridge<BookId>`) and avoid generic type parameters and wildcards (`MyBridge<T extends Number> implements IdentifierBridge<T>, `MyBridge implements IdentifierBridge<List<? extends Number>>`).

If you need more complex types, you can bypass the automatic resolution and specify types explicitly using an `IdentifierBinder`.

6.5.3. Compatibility across indexes with `isCompatibleWith()`

An identifier bridge is involved in indexing, but also in the search DSLs, to convert values passed to the `id` predicate to a document identifier that the backend will understand.

When creating an `id` predicate targeting multiple entity types (and their indexes), Hibernate Search will have multiple bridges to choose from: one per entity type. Since only one predicate with a single value can be created, Hibernate Search needs to pick a single bridge.

By default, when a custom bridge is assigned to the field, Hibernate Search will throw an exception because it cannot decide which bridge to pick.

If the bridges assigned to the field in all indexes produce the same result, it is possible to indicate to Hibernate Search that any bridge will do by implementing `isCompatibleWith`.

This method accepts another bridge in parameter, and returns `true` if that bridge can be expected to always behave the same as `this`. 
**Example 53. Implementing `isCompatibleWith` to support multi-index search**

```
public class BookOrMagazineIdBridge implements IdentifierBridge<BookOrMagazineId> {

    @Override
    public String toDocumentIdentifier(BookOrMagazineId value,
                                        IdentifierBridgeToDocumentIdentifierContext context) {
        return value.getPublisherId() + "/" + value.getPublisherSpecificBookId();
    }

    @Override
    public BookOrMagazineId fromDocumentIdentifier(String documentIdentifier,
                                                   IdentifierBridgeFromDocumentIdentifierContext context) {
        String[] split = documentIdentifier.split("/");
        return new BookOrMagazineId(Long.parseLong(split[0]), Long.parseLong(split[1]));
    }

    @Override
    public boolean isCompatibleWith(IdentifierBridge<? extends Identifier> other) {
        return getClass().equals(other.getClass()); ①
    }
}
```

① Implement `isCompatibleWith` as necessary. Here we just deem any instance of the same class to be compatible.

**6.5.4. Configuring the bridge more finely with `IdentifierBinder`**

To configure a bridge more finely, it is possible to implement a value binder that will be executed at bootstrap. This binder will be able in particular to inspect the type of the property.

**Example 54. Implementing an `IdentifierBinder`**

```
public class BookIdBinder implements IdentifierBinder { ①

    @Override
    public void bind(IdentifierBindingContext<? extends Identifier> context) { ②
        context.bridge( ③
            BookId.class, ④
            new Bridge() ⑤
        );
    }

    private static class Bridge implements IdentifierBridge<BookId> { ⑥
        @Override
        public String toDocumentIdentifier(BookId value,
                                            IdentifierBridgeToDocumentIdentifierContext context) {
            return value.getPublisherId() + "/" + value.getPublisherSpecificBookId();
        }

        @Override
        public BookId fromDocumentIdentifier(String documentIdentifier,
                                               IdentifierBridgeFromDocumentIdentifierContext context) {
            String[] split = documentIdentifier.split("/");
            return new BookId(Long.parseLong(split[0]), Long.parseLong(split[1]));
        }
    }
}
```

The binder must implement the `IdentifierBinder` interface.

Implement the `bind` method.

Call `context.bridge(…)` to define the identifier bridge to use.

Pass the expected type of property values.

Pass the identifier bridge instance.

The identifier bridge must still be implemented. Here the bridge class is nested in the binder class, because it is more convenient, but you are obviously free to implement it in a separate Java file.

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

    @EmbeddedId
    @DocumentId(
        identifierBinder = @IdentifierBinderRef(type = BookIdBinder.class)
    )
    private BookId id = new BookId();

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

    // Getters and setters
}
```

Map the property to the document identifier.

Instruct Hibernate Search to use our custom identifier binder. Note the use of `identifierBinder` instead of `identifierBridge`. It is also possible to reference the binder by its name, in the case of a CDI/Spring bean.

### 6.5.5. Passing parameters

The identifier bridges are usually applied with the built-in `@DocumentId` annotation, which does not accept any parameter other than the identifier bridge/binder.

In some cases, it is necessary to pass parameters directly to the identifier bridge or identifier binder. This is achieved by defining a custom annotation with attributes:

*Example 55. Passing parameters to an IdentifierBridge*
class OffsetIdentifierBridge implements IdentifierBridge<Integer> {
    private final int offset;

    OffsetIdentifierBridge(int offset) {
        this.offset = offset;
    }

    @Override
    public String toDocumentIdentifier(Integer propertyValue, IdentifierBridgeToDocumentIdentifierContext context) {
        return String.valueOf(propertyValue + offset);
    }

    @Override
    public Integer fromDocumentIdentifier(String documentIdentifier, IdentifierBridgeFromDocumentIdentifierContext context) {
        return Integer.parseInt(documentIdentifier) - offset;
    }
}

① Implement a bridge that index the identifier as is, but adds a configurable offset. For example, with an offset of 1 and database identifiers starting at 0, index identifiers will start at 1.

② The bridge accepts one parameter in its constructors: the offset to apply to identifiers.

@Retention(RetentionPolicy.RUNTIME) ①
@Target({ElementType.METHOD, ElementType.FIELD}) ②
@PropertyMapping(processor = @PropertyMappingAnnotationProcessorRef( ③
type = OffsetDocumentId.Processor.class)
)
@Documented ④
public @interface OffsetDocumentId {
    int offset(); ⑤
}

class Processor implements PropertyMappingAnnotationProcessor<OffsetDocumentId> { ⑥
    @Override
    public void process(PropertyMappingStep mapping, OffsetDocumentId annotation, PropertyMappingAnnotationProcessorContext context) {
        OffsetIdentifierBridge bridge = new OffsetIdentifierBridge( ⑦
            annotation.offset()
        );
        mapping.documentId() ⑧
            .identifierBridge( bridge ); ⑨
    }
}

① Define an annotation with retention RUNTIME. Any other retention policy will cause the annotation to be ignored by Hibernate Search.

② Since we’re defining an identifier bridge, allow the annotation to target either methods (getters) or fields.

③ Mark this annotation as a property mapping, and instruct Hibernate Search to apply the given processor whenever it finds this annotation. It is also possible to reference the processor by its name, in the case of a CDI/Spring bean.

④ Optionally, mark the annotation as documented, so that it is included in the javadoc of your
5. Define custom attributes to configure the value bridge. Here we define an offset that the bridge should add to entity identifiers.

6. The processor must implement the PropertyMappingAnnotationProcessor interface, setting its generic type argument to the type of the corresponding annotation. Here the processor class is nested in the annotation class, because it is more convenient, but you are obviously free to implement it in a separate Java file.

7. In the process method, instantiate the bridge and pass the annotation attribute as constructor argument.

8. Declare that this property is to be used to generate the document identifier.

9. Instruct Hibernate Search to use our bridge to convert between the property and the document identifiers. Alternatively, we could pass an identifier binder instead, using the identifierBinder() method.

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

    @Id
    // DB identifiers start at 0, but index identifiers start at 1
    @OffsetDocumentId(offset = 1) ①
    private Integer id;

    private String title;

    // Getters and setters
    // ...
}
```

① Apply the bridge using its custom annotation, setting the parameter.

### 6.5.6. Accessing the ORM session or session factory from the bridge

Contexts passed to the bridge methods can be used to retrieve the Hibernate ORM session or session factory.
Example 56. Retrieving the ORM session or session factory from an **IdentifierBridge**

```java
public class MyDataIdentifierBridge implements IdentifierBridge<MyData> {
    @Override
    public String toDocumentIdentifier(MyData propertyValue,
                                       IdentifierBridgeToDocumentIdentifierContext context) {
        SessionFactory sessionFactory = context.extension(HibernateOrmExtension.get()).sessionFactory();  
        // ... do something with the factory ...  
    }

    @Override
    public MyData fromDocumentIdentifier(String documentIdentifier,
                                           IdentifierBridgeFromDocumentIdentifierContext context) {
        Session session = context.extension(HibernateOrmExtension.get()).session();  
        // ... do something with the session ...  
    }
}
```

---

① Apply an extension to the context to access content specific to Hibernate ORM.

② Retrieve the `SessionFactory` from the extended context. The `Session` is not available here.

③ Apply an extension to the context to access content specific to Hibernate ORM.

④ Retrieve the `Session` from the extended context.

### 6.5.7. Injecting beans into the bridge or binder

With **compatible frameworks**, Hibernate Search supports injection of beans into both the `IdentifierBridge` and the `IdentifierBinder`.

This only applies to beans instantiated by Hibernate Search itself. As a rule of thumb, if you need to call `new MyBridge()` at some point, the bridge won’t get auto-magically injected.

The context passed to the identifier binder’s `bind` method also exposes a `getBeanResolver` method to access the bean resolver and instantiate beans explicitly.

See **Bean injection** for more details.

### 6.5.8. Experimental features

These features are **experimental**. Usual compatibility policies do not apply: incompatible changes may be introduced in any future release.

The context passed to the identifier binder's `bind` method exposes a `bridgedElement()` method that gives access to metadata about the value being bound, in particular its type.
See the javadoc for more information.

6.6. Routing key bridges

6.6.1. Basics

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

6.6.2. Passing parameters

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

6.6.3. Accessing the ORM session from the bridge

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

6.6.4. Injecting beans into the binder

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

6.6.5. Experimental features

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

6.7. Declaring dependencies to bridged elements

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

6.8. Declaring and writing to index fields

6.8.1. Basics

When implementing a PropertyBinder or TypeBinder, it is necessary to declare the index fields
that the bridge will contribute to. This declaration is performed using a dedicated DSL.

The entry point to this DSL is the `IndexSchemaElement`, which represents the part of the document structure that the binder will push data to. From the `IndexSchemaElement`, it is possible to declare fields.

The declaration of each field yields a field `reference`. This reference is to be stored in the bridge, which will use it at runtime to set the value of this field in a given document, represented by a `DocumentElement`.

Below is a simple example using the DSL to declare a single field in a property binder and then write to that field in a property bridge.
Example 57. Declaring a simple index field and writing to that field

```java
public class ISBNBinder implements PropertyBinder {
    @Override
    public void bind(PropertyBindingContext context) {
        context.dependencies();
        /* ... (declaration of dependencies, not relevant) ... */

        IndexSchemaElement schemaElement = context.indexSchemaElement(); ①
        IndexFieldReference<String> field =
            schemaElement.field("isbn", f -> f.asString().normalizer("isbn"))
                .toReference(); ⑤
        context.bridge(new ISBNBridge(field)); ⑥
    }
}
```

① Get the `IndexSchemaElement`, the entry point to the index field declaration DSL.
② Declare a field.
③ Pass the name of the field.
④ Declare the type of the field. This is done through a lambda taking advantage of another DSL. See Defining index field types for more information.
⑤ Get a reference to the declared field.
⑥ Pass the reference to the bridge for later use.

```java
private static class ISBNBridge implements PropertyBridge {
    private final IndexFieldReference<String> fieldReference;
    private ISBNBridge(IndexFieldReference<String> fieldReference) {
        this.fieldReference = fieldReference;
    }

    @Override
    public void write(DocumentElement target, Object bridgedElement, PropertyBridgeWriteContext context) {
        String indexedValue = /* ... (extraction of data, not relevant) ... */
        target.addValue( this.fieldReference, indexedValue ); ①
    }
}
```

① In the bridge, use the reference obtained above to add a value to the field for the current document.

6.8.2. Type objects

The lambda syntax to declare the type of each field is convenient, but sometimes gets in the way, in
particular when multiple fields must be declared with the exact same type.

For that reason, the context object passed to binders exposes a `typeFactory()` method. Using this factory, it is possible to build `IndexFieldType` objects that can be re-used in multiple field declarations.

**Example 58. Re-using an index field type in multiple field declarations**

```java
@override
public void bind(TypeBindingContext context) {
    context.dependencies();
    /* ... (declaration of dependencies, not relevant) ... */
    IndexSchemaElement schemaElement = context.indexSchemaElement();
    IndexFieldType<String> nameType = context.typeFactory().asString().analyzer("name").toIndexFieldType();
    context.bridge(new Bridge( schemaElement.field("firstName", nameType).toReference(),
                                schemaElement.field("lastName", nameType).toReference(),
                                schemaElement.field("fullName", nameType).toReference() ));
}
```

1. Get the type factory.
2. Define the type.
3. Get the resulting type.
4. Pass the type directly instead of using a lambda when defining the field.

### 6.8.3. Multi-valued fields

Fields are considered single-valued by default: if you attempt to add multiple values to a single-valued field during indexing, an exception will be thrown.

In order to add multiple values to a field, this field must be marked as multi-valued during its declaration:
Example 59. Declaring a field as multi-valued

```java
@override
public void bind(TypeBindingContext context) {
    context.dependencies();
    /* ... (declaration of dependencies, not relevant) ... */
    IndexSchemaElement schemaElement = context.indexSchemaElement();
    context.bridge( new Bridge(
        schemaElement.field( "names", f -> f.asString().analyzer( "name" )
            .multiValued() ①
toReference()
    ) );
}
```

① Declare the field as multi-valued.

6.8.4. Object fields

The previous sections only presented flat schemas with atomic fields, but the index schema can actually be organized in a tree structure, with two categories of index fields:

- Value fields, often simply called "fields", which hold an atomic value of a specific type: string, integer, date, ...
- Object fields, which hold a composite value.

Object fields are declared similarly to value fields, with an additional step to declare each sub-field, as shown below.
Example 60. Declaring an object field

```java
@Override
public void bind(PropertyBindingContext context) {
    context.dependencies();
    /* ... (declaration of dependencies, not relevant) ... */

    IndexSchemaElement schemaElement = context.indexSchemaElement();
    IndexSchemaObjectField summaryField =
        schemaElement.objectField( "summary" ); ①
    IndexFieldType<BigDecimal> amountFieldType = context.typeFactory()
        .asBigDecimal().decimalScale( 2 )
        .toIndexFieldType();
    context.bridge( new Bridge(
        summaryField.toReference(), ②
        summaryField.field( "total", amountField ) ③
            .toReference(),
        summaryField.field( "books", amountField ) ③
            .toReference(),
        summaryField.field( "shipping", amountField ) ③
            .toReference()
    ) );
}
```

① Declare an object field with `objectField`, passing its name in parameter.
② Get a reference to the declared object field and pass it to the bridge for later use.
③ Create sub-fields, get references to these fields and pass them to the bridge for later use.

The sub-fields of an object field can include object fields.

Just as value fields, object fields are single-valued by default. Be sure to call `multiValued()` during the object field definition if you want to make it multi-valued.

Object fields as well as their sub-fields are each assigned a reference, which will be used by the bridge to write to documents, as shown in the example below.
Example 61. Writing to an object field

```java
@override
@SuppressWarnings("unchecked")
public void write(DocumentElement target, Object bridgedElement, PropertyBridgeWriteContext context) {
    List<InvoiceLineItem> lineItems = (List<InvoiceLineItem>) bridgedElement;
    BigDecimal total = BigDecimal.ZERO;
    BigDecimal books = BigDecimal.ZERO;
    BigDecimal shipping = BigDecimal.ZERO;
    /* ... (computation of amounts, not relevant) ... */
    DocumentElement summary = target.addObject(this.summaryField); ①
    summary.addValue(this.totalField, total); ②
    summary.addValue(this.booksField, books); ②
    summary.addValue(this.shippingField, shipping); ②
}
```

① Add an object to the summary object field for the current document, and get a reference to that object.

② Add a value to the sub-fields for the object we just added. Note we’re calling `addValue` on the object we just added, not on `target`.

6.8.5. Object field storage

By default, object fields are flattened, meaning that the tree structure is not preserved. See `DEFAULT` or `FLATTENED` storage for more information.

It is possible to switch to nested storage by passing an argument to the `objectField` method, as shown below. Each value of the object field will then transparently be indexed as a separate nested document, without any change to the `write` method of the bridge.
Example 62. Declaring an object field as nested

```java
@Override
public void bind(PropertyBindingContext context) {
    context.dependencies() // ... (declaration of dependencies, not relevant) ...

    IndexSchemaElement schemaElement = context.indexSchemaElement();
    IndexSchemaObjectField lineItemsField =
        schemaElement.objectField("lineItems", ObjectFieldStorage.NESTED)
            .multiValued();

    context.bridge(new Bridge(lineItemsField.toReference(),
        lineItemsField.field("category", f -> f.asString()).toReference(),
        lineItemsField.field("amount", f -> f.asBigDecimal().setScale(2)).toReference()));
}
```

1. Declare an object field with `objectField`.
2. Pass the name of the object field.
3. Pass the storage type of the object field, here `NESTED`.
4. Declare the object field as multi-valued.
5. Get a reference to the declared object field and pass it to the bridge for later use.
6. Create sub-fields, get references to these fields and pass them to the bridge for later use.

6.8.6. Dynamic fields with field templates

Field declared in the sections above are all static: their path and type are known on bootstrap.

In some very specific cases, the path of a field is not known until you actually index it; for example, you may want to index a `Map<String, Integer>` by using the map keys as field names, or index the properties of a JSON object whose schema is not known in advance. The fields, then, are considered dynamic.

Dynamic fields are not declared on bootstrap, but need to match a field template that is declared on bootstrap. The template includes the field types and structural information (multi-valued or not, ...), but omits the field names.

A field template is always declared in a binder: either in a type binder or in a property binder. As for static fields, the entry point to declaring a template is the `IndexSchemaElement` passed to the binder's `bind(...)` method. A call to the `fieldTemplate` method on the schema element will declare a field template.
Assuming a field template was declared during binding, the bridge can then add dynamic fields to the DocumentElement when indexing, by calling addValue and passing the field name (as a string) and the field value.

Below is a simple example using the DSL to declare a field template in a property binder and then write to that field in a property bridge.

**Example 63. Declaring a field template and writing to a dynamic field**

```java
public class UserMetadataBinder implements PropertyBinder {

    @Override
    public void bind(PropertyBindingContext context) {
        /* ... (declaration of dependencies, not relevant) ... */

        IndexSchemaElement schemaElement = context.indexSchemaElement();

        IndexSchemaObjectField userMetadataField =
            schemaElement.objectField( "userMetadata" );

        userMetadataField.fieldTemplate( "userMetadataValueTemplate",
            f -> f.asString().analyzer( "english" ) );

        context.bridge( new UserMetadataBridge( userMetadataField.toReference() ) );
    }
}
```

1. Declare an object field with `objectField`. It’s better to always host your dynamic fields on a dedicated object field, to avoid conflicts with other templates.

2. Declare a field template with `fieldTemplate`.

3. Pass the template name—this is not the field name, and is only used to uniquely identify the template.

4. Define the field type.

5. On contrary to static field declarations, field template declarations do not return a field reference, because you won’t need it when writing to the document.

6. Get a reference to the declared object field and pass it to the bridge for later use.
private static class UserMetadataBridge implements PropertyBridge {

    private final IndexObjectFieldReference userMetadataFieldReference;

    private UserMetadataBridge(IndexObjectFieldReference userMetadataFieldReference) {
        this.userMetadataFieldReference = userMetadataFieldReference;
    }

    @Override
    public void write(DocumentElement target, Object bridgedElement, PropertyBridgeWriteContext context) {
        Map<String, String> userMetadata = (Map<String, String>) bridgedElement;
        DocumentElement indexedUserMetadata = target.addObject(userMetadataFieldReference);

        for (Map.Entry<String, String> entry : userMetadata.entrySet()) {
            String fieldName = entry.getKey();
            String fieldValue = entry.getValue();
            indexedUserMetadata.addValue(fieldName, fieldValue);
        }
    }
}

① Add an object to the userMetadata object field for the current document, and get a reference to that object.

② Add one field per user metadata entry, with the field name and field value defined by the user. Note that field names should usually be validated before that point, in order to avoid exotic characters (whitespaces, dots, ...).

Though rarely necessary, you can also declare templates for object fields using the objectFieldTemplate methods.

It is also possible to add multiple fields with different types to the same object. To that end, make sure that the format of a field can be inferred from the field name. You can then declare multiple templates and assign a path pattern to each template, as shown below.

Example 64. Declaring multiple field templates with different types
public class MultiTypeUserMetadataBinder implements PropertyBinder {

    @Override
    public void bind(PropertyBindingContext context) {
        /* ... (declaration of dependencies, not relevant) ... */

        IndexSchemaElement schemaElement = context.indexSchemaElement();
        IndexSchemaObjectField userMetadataField =
            schemaElement.objectField( "multiTypeUserMetadata" ); ①

        userMetadataField.fieldTemplate( ②
            "userMetadataValueTemplate_int", ③
            f -> f.asInteger().sortable( Sortable.YES ) ④
        )
            .matchingPathGlob( "*_int" ); ⑤

        userMetadataField.fieldTemplate( ⑥
            "userMetadataValueTemplate_default",
            f -> f.asString().analyzer( "english" )
        );

        context.bridge( new Bridge( userMetadataField.toReference() ) );
    }
}

① Declare an object field with objectField.

② Declare a field template for integer with fieldTemplate.

③ Pass the template name.

④ Define the field type as integer, sortable.

⑤ Assign a path pattern to the template, so that only fields ending with _int will be considered as integers.

⑥ Declare another field template, so that fields are considered as english text if they do not match the previous template.
private static class Bridge implements PropertyBridge {

    private final IndexObjectFieldReference userMetadataFieldReference;

    private Bridge(IndexObjectFieldReference userMetadataFieldReference) {
        this.userMetadataFieldReference = userMetadataFieldReference;
    }

    @Override
    public void write(DocumentElement target, Object bridgedElement, PropertyBridgeWriteContext context) {
        Map<String, Object> userMetadata = (Map<String, Object>) bridgedElement;

        DocumentElement indexedUserMetadata = target.addObject(userMetadataFieldReference);

        for (Map.Entry<String, Object> entry : userMetadata.entrySet()) {
            String fieldName = entry.getKey();
            Object fieldValue = entry.getValue();
            indexedUserMetadata.addValue(fieldName, fieldValue);
        }
    }
}

① Add an object to the userMetadata object field for the current document, and get a reference to that object.

② Add one field per user metadata entry, with the field name and field value defined by the user. Note that field values should be validated before that point; in this case, adding a field named foo_int with a value of type String will lead to a SearchException when indexing.

Precedence of field templates

Hibernate Search tries to match templates in the order they are declared, so you should always declare the templates with the most specific path pattern first.

Templates declared on a given schema element can be matched in children of that element. For example, if you declare templates at the root of your entity (through a type bridge), these templates will be implicitly available in every single property bridge of that entity. In such cases, templates declared in property bridges will take precedence over those declared in the type bridge.

6.9. Defining index field types

6.9.1. Basics

A specificity of Lucene-based search engines (including Elasticsearch) is that field types are much more complex than just a data type ("string", "integer", …).

When declaring a field, you must not only declare the data type, but also various characteristics that will define how the data is stored exactly: is the field sortable, is it projectable, is it analyzed and if so with which analyzer, …
Because of this complexity, when field types must be defined in the various binders (ValueBinder, PropertyBinder, TypeBinder), they are defined using a dedicated DSL.

The entry point to this DSL is the IndexFieldTypeFactory. The type factory is generally accessible through the context object passed to the binders (context.typeFactory()). In the case of PropertyBinder and TypeBinder, the type factory can also be passed to the lambda expression passed to the field method to define the field type inline.

The type factory exposes various as*() methods, for example asString or asLocalDate. These are the first steps of the type definition DSL, where the data type is defined. They return other steps, from which options can be set, such as the analyzer. See below for an example.

**Example 65. Defining a field type**

```java
IndexFieldType<String> type = context.typeFactory() ①
    .asString() ②
    .normalizer( "isbn" ) ③
    .sortable( Sortable.YES ) ④
    .toIndexFieldType(); ④
```

① Get the IndexFieldTypeFactory from the binding context.
② Define the data type.
③ Define options. Available options differ based on the field type: for example, normalizer is available for String fields, but not for Double fields.
④ Get the index field type.

In ValueBinder, the call to toIndexFieldType() is omitted: context.bridge(...) expects to be passed the last DSL step, not a fully built type.

toIndexFieldType() is also omitted in the lambda expressions passed to the field method of the field declaration DSL.

### 6.9.2. Available data types

All available data types have a dedicated as*() method in IndexFieldTypeFactory. For details, see the javadoc of IndexFieldTypeFactory, or the backend-specific documentation:

- available data types in the Lucene backend
- available data types in the Elasticsearch backend

### 6.9.3. Available type options

Most of the options available in the index field type DSL are identical to the options exposed by
@*Field* annotations. See [Field annotation attributes](#) for details about them.

Other options are explained in the following sections.

### 6.9.4. DSL converter

This section is not relevant for `ValueBinder`: Hibernate Search sets the DSL converter automatically for value bridges, creating a DSL converter that simply delegates to the value bridge.

The various search DSLs expose some methods that expect a field value: `matching()`, `between()`, `atMost()`, `missingValue().use()`, ... By default, the expected type will be the same as the data type, i.e. `String` if you called `asString()`, `LocalDate` if you called `asLocalDate()`, etc.

This can be annoying when the bridge converts values from a different type when indexing. For example, if the bridge converts an enum to a string when indexing, you probably don’t want to pass a string to search predicates, but rather the enum.

By setting a DSL converter on a field type, it is possible to change the expected type of values passed to the various DSL, See below for an example.

**Example 66. Assigning a DSL converter to a field type**

```java
IndexFieldType<String> type = context.typeFactory()
    .asString() ①
    .normalizer("isbn")
    .sortable(Sortable.YES)
    .dslConverter( ②
        ISBN.class, ③
        (value, convertContext) -> value.getStringValue() ④
    )
    .toIndexFieldType();
```

① Define the data type as `String`.

② Define a DSL converter that converts from `ISBN` to `String`. This converter will be used transparently by the search DSLs.

③ Define the input type as `ISBN` by passing `ISBN.class` as the first parameter.

④ Define how to convert an `ISBN` to a `String` by passing a converter as the second parameter.

```java
ISBN expectedISBN = /* ... */
List<Book> result = /* ... */
    .where( f -> f.match().field("isbn")
        .matching(expectedISBN) ) ①
    .fetchHits(20);
```

① Thanks to the DSL converter, predicates targeting fields using our type accept `ISBN` values by default.
DSL converters can be disabled in the various DSLs where necessary. See Type of arguments passed to the DSL.

### 6.9.5. Projection converter

This section is not relevant for ValueBinder: Hibernate Search sets the projection converter automatically for value bridges, creating a projection converter that simply delegates to the value bridge.

By default, the type of values returned by field projections or aggregations will be the same as the data type of the corresponding field, i.e. String if you called asString(), LocalDate if you called asLocalDate(), etc.

This can be annoying when the bridge converts values from a different type when indexing. For example, if the bridge converts an enum to a string when indexing, you probably don’t want projections to return a string, but rather the enum.

By setting a projection converter on a field type, it is possible to change the type of values returned by field projections or aggregations. See below for an example.

**Example 67. Assigning a projection converter to a field type**

```java
IndexFieldType<String> type = context.typeFactory()
   .asString(①)
   .projectable(Projectable.YES)
   .projectionConverter(②)
   ISBN.class, ③
   (value, convertContext) -> ISBN.parse(value) ④
   .toIndexFieldType();
```

**①** Define the data type as String.

**②** Define a projection converter that converts from String to ISBN. This converter will be used transparently by the search DSLs.

**③** Define the converted type as ISBN by passing ISBN.class as the first parameter.

**④** Define how to convert a String to an ISBN by passing a converter as the second parameter.

```java
List<ISBN> result = searchSession.search( Book.class )
   .select( f -> f.field("isbn", ISBN.class) ) ①
   .where( f -> f.matchAll() )
   .fetchHits(20);
```

**①** Thanks to the projection converter, fields using our type are projected to an ISBN by default.
Projection converters can be disabled in the projection DSL where necessary. See Type of projected values.

6.9.6. Backend-specific types

Backends define extensions to this DSL to define backend-specific types.

See:

- Lucene index field type DSL extension
- Elasticsearch index field type DSL extension

6.10. Assigning default bridges with the bridge resolver

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.
Chapter 7. Managing the index schema

7.1. Basics

Before indexes can be used for indexing or searching, they must be created on disk (Lucene) or in the remote cluster (Elasticsearch). With Elasticsearch in particular, this creation may not be obvious since it requires to describe the schema for each index, which includes in particular:

- the definition of every analyzer or normalizer used in this index;
- the definition of every single field used in this index, including in particular its type, the analyzer assigned to it, whether it requires doc values, etc.

Hibernate Search has all the necessary information to generate this schema automatically, so it is possible to delegate the task of managing the schema to Hibernate Search.

7.2. Automatic schema management on startup/shutdown

The property `hibernate.search.schema_management.strategy` can be set to one of the following values in order to define what to do with the indexes and their schema on startup and shutdown.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Definition</th>
<th>Warnings</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>A strategy that does not do anything on startup or shutdown. Indexes and their schema will not be created nor deleted on startup or shutdown. Hibernate Search will <strong>not even check</strong> that the index actually exists.</td>
<td>With Elasticsearch, indexes and their schema will have to be created explicitly before startup.</td>
</tr>
<tr>
<td>Strategy</td>
<td>Definition</td>
<td>Warnings</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>validate</td>
<td>A strategy that does not change indexes nor their schema, but checks that indexes exist and validates their schema on startup.</td>
<td>Indexes and their schema will have to be created explicitly before startup.                                                                                           With the Lucene backend, validation is limited to checking that the indexes exist, because local Lucene indexes don't have a schema.</td>
</tr>
<tr>
<td></td>
<td>An exception will be thrown on startup if:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Indexes are missing</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• OR, with the Elasticsearch backend only, indexes exist but their schema does not match the requirements of the Hibernate Search mapping: missing fields, fields with incorrect type, missing analyzer definitions or normalizer definitions, ...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;Compatible&quot; differences such as extra fields are ignored.</td>
<td></td>
</tr>
<tr>
<td>create</td>
<td>A strategy that creates missing indexes and their schema on startup, but does not touch existing indexes and assumes their schema is correct without validating it.</td>
<td>Creating a schema does not populate indexed data.</td>
</tr>
<tr>
<td>create-or-validate (default)</td>
<td>A strategy that creates missing indexes and their schema on startup, and validates the schema of existing indexes.</td>
<td>Creating a schema does not populate indexed data.</td>
</tr>
<tr>
<td></td>
<td>With the Elasticsearch backend only, an exception will be thrown on startup if some indexes already exist but their schema does not match the requirements of the Hibernate Search mapping: missing fields, fields with incorrect type, missing analyzer definitions or normalizer definitions, ...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&quot;Compatible&quot; differences such as extra fields are ignored.</td>
<td></td>
</tr>
<tr>
<td>Strategy</td>
<td>Definition</td>
<td>Warnings</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>create-or-update</td>
<td>A strategy that creates missing indexes and their schema on startup, and updates the schema of existing indexes if possible.</td>
<td>Updating a schema does not update indexed data.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This strategy is unfit for production environments, due to several limitations including the impossibility to change the type of an existing field or the requirement to close indexes while updating analyzer definitions (which is not possible at all on AWS).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>With the Lucene backend, schema update is a no-op, because local Lucene indexes don’t have a schema.</td>
</tr>
<tr>
<td>drop-and-create</td>
<td>A strategy that drops existing indexes and recreates them and their schema on startup.</td>
<td>All indexed data will be lost on startup.</td>
</tr>
<tr>
<td>drop-and-create-and-drop</td>
<td>A strategy that drops existing indexes and recreates them and their schema on startup, then drops the indexes on shutdown.</td>
<td>All indexed data will be lost on startup and shutdown.</td>
</tr>
</tbody>
</table>

7.3. Manual schema management

Schema management does not have to happen automatically on startup and shutdown.

Using the SearchSchemaManager interface, it is possible to trigger schema management operations explicitly after Hibernate Search has started.
The most common use case is to set the `automatic schema management strategy` to `none` and handle the creation/deletion of indexes manually when some other conditions are met, for example the Elasticsearch cluster has finished booting.

After schema management operations are complete, you will often want to populate indexes. To that end, use the `mass indexer`.

The `SearchSchemaManager` interface exposes the following methods.

<table>
<thead>
<tr>
<th>Method</th>
<th>Definition</th>
<th>Warnings</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>validate()</code></td>
<td>Does not change indexes nor their schema, but checks that indexes exist and validates their schema.</td>
<td>With the Lucene backend, validation is limited to checking that the indexes exist, because local Lucene indexes don't have a schema.</td>
</tr>
<tr>
<td><code>createIfMissing()</code></td>
<td>Creates missing indexes and their schema, but does not touch existing indexes and assumes their schema is correct without validating it.</td>
<td>Creating a schema does not populate indexed data.</td>
</tr>
<tr>
<td><code>createOrValidate()</code></td>
<td>Creates missing indexes and their schema, and validates the schema of existing indexes.</td>
<td>Creating a schema does not populate indexed data.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>With the Lucene backend, validation is limited to checking that the indexes exist, because local Lucene indexes don't have a schema.</td>
</tr>
<tr>
<td>Method</td>
<td>Definition</td>
<td>Warnings</td>
</tr>
<tr>
<td>--------------------</td>
<td>---------------------------------------------------------------------------</td>
<td>------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>createOrUpdate()</code></td>
<td>Creates missing indexes and their schema, and updates the schema of existing indexes if possible.</td>
<td>Updating a schema does not update indexed data. With the Elasticsearch backend, updating a schema may fail. With the Elasticsearch backend, updating a schema may close indexes while updating analyzer definitions (which is not possible at all on AWS). With the Lucene backend, schema update is a no-op, because local Lucene indexes don’t have a schema. (it just creates missing indexes).</td>
</tr>
<tr>
<td><code>dropIfExisting()</code></td>
<td>Drops existing indexes.</td>
<td>All indexed data will be lost.</td>
</tr>
<tr>
<td><code>dropAndCreate()</code></td>
<td>Drops existing indexes and re-creates them and their schema.</td>
<td>All indexed data will be lost.</td>
</tr>
</tbody>
</table>

Below is an example using a `SearchSchemaManager` to drop and create indexes, then using a mass indexer to re-populate the indexes. The `dropAndCreateSchemaOnStart` setting of the mass indexer would be an alternative solution to achieve the same results.
**Example 68. Reinitializing indexes using a `SearchSchemaManager`**

```java
SearchSession searchSession = Search.session( entityManager ); ①
SearchSchemaManager schemaManager = searchSession.schemaManager(); ②
schemaManager.dropAndCreate(); ③
searchSession.massIndexer().startAndWait(); ④
```

① Get a `SearchSession`.
② Get a schema manager.
③ Drop and create the indexes. This method is synchronous and will only return after the operation is complete.
④ Optionally, trigger mass indexing.

You can also select entity types when creating a schema manager, so as to manage the indexes of these types only (and their indexed subtypes, if any):

**Example 69. Reinitializing only some indexes using a `SearchSchemaManager`**

```java
SearchSchemaManager schemaManager = searchSession.schemaManager( Book.class ); ①
schemaManager.dropAndCreate(); ②
```

① Get a schema manager targeting the index mapped to the `Book` entity type.
② Drop and create the index for the `Book` entity only. Other indexes are unaffected.

### 7.4. How schema management works

**Creating/updating a schema does not create/update indexed data**

Creating or updating indexes and their schema through schema management will not populate the indexes:

- newly created indexes will always be empty.
- indexes with a recently updated schema will still contain the same indexed data, i.e. new fields won’t be added to documents just because they were added to the schema.

This is by design: reindexing is a potentially long-running task that should be triggered explicitly. To populate indexes with pre-existing data from the database, use mass indexing.

**Dropping the schema means losing indexed data**

Dropping a schema will drop the whole index, including all indexed data.

A dropped index will need to be re-created through schema management, then populated with pre-existing data from the database through mass indexing.
Schema validation and update are not effective with Lucene

The Lucene backend will only validate that the index actually exists and create missing indexes, because there is no concept of schema in Lucene beyond the existence of index segments.

Schema validation is permissive

With Elasticsearch, schema validation is as permissive as possible:

- Fields that are unknown to Hibernate Search will be ignored.
- 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.
- Analyzer/normalizer definitions that are unknown to Hibernate Search will be ignored.

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

Schema updates may fail

A schema update, triggered by the create-or-update strategy, may very well fail. This is because schemas may changed in an incompatible way, such as a field having its type changed, updating the schema may be impossible without manual intervention, and then the schema update

Worse, since updates are handled on a per-index basis, a schema update may succeed for one index but fail on another, leaving your schema as a whole half-updated.

For these reasons, using schema updates in a production environment is not recommended. Whenever the schema changes, you should either:

- drop and create indexes, then reindex.
- OR update the schema manually through custom scripts.

In this case, the create-or-update strategy will prevent Hibernate Search from starting, but it may already have successfully updated the schema for another index, making a rollback difficult.

Schema updates on Elasticsearch may close indexes

Elasticsearch does not allow updating analyzer/normalizer definitions on an open index. Thus, when analyzer or normalizer definitions have to be updated during a schema update, Hibernate Search will temporarily stop the affected indexes.

For this reason, the create-or-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.

Also, since Elasticsearch on Amazon Web Services (AWS) does not support the _close/_open
operations, the schema update will fail when trying to update analyzer definitions on an AWS Elasticsearch cluster. The only workaround is to avoid the schema update on AWS. It should be avoided in production environments regardless: see [mapper-orm-schema-management-concepts-update-failure].
Chapter 8. Indexing Hibernate ORM entities

8.1. 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.

To be precise, index updates happen on transaction commit or, if working outside of a transaction, on session flush.

8.1.1. Configuration

Automatic indexing may be unnecessary if your index is read-only or if you update it regularly by reindexing, either using the MassIndexer or manually. You can enable or disable automatic indexing by setting the configuration property hibernate.search.automatic_indexing.strategy:

- when set to session (the default), 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.

8.1.2. 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, either using the MassIndexer or manually.

Entity data is retrieved from entities upon session flushes

When a Hibernate ORM session is flushed, Hibernate Search will extract data from the entities to build documents to index, and will put these documents in an internal buffer for later indexing (see the next paragraphs).

This means in particular that you can safely clear() the session after a flush(): entity changes performed up to the flush will be indexed correctly.
If you come from Hibernate Search 5 or earlier, you may see this as a significant improvement: there is no need to call `flushToIndexes()` and update indexes in the middle of a transaction anymore, except for larger volumes of data (see Controlling entity reads and index writes with `SearchIndexingPlan`).

**Inside transactions, indexing happens after transactions are committed**

When entity changes happen inside a transaction, indexes are not updated immediately, but only after the transaction is successfully committed. That way, if a transaction is rolled back, the indexes will be left in a state consistent with the database, discarding all the index changes that were planned during the transaction.

However, if you perform a batch process inside a transaction, and perform flush/clear, regularly to save memory, be aware that Hibernate Search's internal buffer holding documents to index will grow on each flush, and will not be cleared until the transaction is committed or rolled back. If you encounter memory issues because of that, see Controlling entity reads and index writes with `SearchIndexingPlan` for a few solutions.

**Outside of transactions, indexing happens on session flush**

When entity changes happen outside of any transaction (not recommended), indexes are updated immediately upon session `flush()`. Without that flush, indexes will not be updated automatically.

**Index changes may not be visible immediately**

By default, indexing will resume the application thread 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 indexing 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.

## 8.1.3. Synchronization with the indexes

### Basics

For a preliminary introduction to writing to and reading from indexes in Hibernate Search, including in particular the concepts of **commit** and **refresh**, see Commit and refresh.

When a transaction is committed, automatic indexing can (and, by default, will) block the application thread until indexing reaches a certain level of completion.

There are two main reasons for blocking the thread:

1. **Indexed data safety**: if, once the database transaction completes, index data must be safely stored to disk, an **index commit** is necessary. Without it, index changes may only be safe after a few seconds, when a periodic index commit happens in the background.

2. **Real-time search queries**: if, once the database transaction completes, any search query must immediately take the index changes into account, an **index refresh** is necessary. Without it, index changes may only be visible after a few seconds, when a periodic index refresh happens in the background.

These two requirements are controlled by the **synchronization strategy**. The default strategy is defined by the configuration property `hibernate.search.automatic_indexing.synchronization.strategy`. Below is a reference of all available strategies and their guarantees.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Guarantees when the application thread resumes</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Changes applied (with or without <strong>commit</strong>)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Changes safe from crash/power loss (<strong>commit</strong>)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Changes visible on search (<strong>refresh</strong>)</td>
<td></td>
</tr>
<tr>
<td><strong>async</strong></td>
<td>No guarantee</td>
<td>Best</td>
</tr>
<tr>
<td>Strategy</td>
<td>Indexing</td>
<td>Extraction</td>
</tr>
<tr>
<td>------------</td>
<td>----------</td>
<td>------------</td>
</tr>
<tr>
<td><code>write-sync</code> (default)</td>
<td>Guaranteed</td>
<td>Guaranteed</td>
</tr>
<tr>
<td><code>read-sync</code></td>
<td>Guaranteed</td>
<td>No guarantee</td>
</tr>
<tr>
<td><code>sync</code></td>
<td>Guaranteed</td>
<td>Guaranteed</td>
</tr>
</tbody>
</table>

Depending on the backend and its configuration, the `sync` and `read-sync` strategies may lead to poor indexing throughput, because the backend may not be designed for frequent, on-demand index refreshes.

This is why this strategy is only recommended if you know your backend is designed for it, or for integration tests. In particular, the `sync` strategy will work fine with the default configuration of the Lucene backend, but will perform poorly with the Elasticsearch backend.

Indexing failures may be reported differently depending on the chosen strategy:

- Failure to extract data from entities:
  - Regardless of the strategy, throws an exception in the application thread.

- Failure to apply index changes (i.e. I/O operations on the index):
  - For strategies that apply changes immediately: throws an exception in the application thread.
  - For strategies that do **not** apply changes immediately: forwards the failure to the failure handler, which by default will simply log the failure.

- Failure to commit index changes:
  - For strategies that guarantee an index commit: throws an exception in the application thread.
  - For strategies that do **not** guarantee an index commit: forwards the failure to the failure handler, which by default will simply log the failure.

**Per-session override**

While the configuration property mentioned above defines a default, it is possible to override this default on a particular session by calling `SearchSession#automaticIndexingSynchronizationStrategy(…)` and passing a different strategy.

The built-in strategies can be retrieved by calling:

- `AutomaticIndexingSynchronizationStrategy.async()`
• `AutomaticIndexingSynchronizationStrategy.writeSync()`
• `AutomaticIndexingSynchronizationStrategy.readSync()`
• or `AutomaticIndexingSynchronizationStrategy.sync()`

Example 70. Overriding the automatic indexing synchronization strategy

```java
SearchSession searchSession = Search.session(entityManager); ①
searchSession-automaticIndexingSynchronizationStrategy(
    AutomaticIndexingSynchronizationStrategy.sync() 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)
    .where(f -> f.match().field("title" ).matching("2nd edition" ))
    .fetchHits(20); 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.

Custom strategy

You can also implement custom strategy. The custom strategy can then be set just like the built-in strategies:

• as the default by setting the configuration property `hibernate.search.automatic_indexing.synchronization.strategy` to a bean reference pointing to the custom implementation.

• at the session level by passing an instance of the custom implementation to `SearchSession#automaticIndexingSynchronizationStrategy(...)`. 
8.2. Reindexing large volumes of data with the **MassIndexer**

8.2.1. Basics

There are cases where automatic indexing is not enough, because a pre-existing database has to be indexed:

- when restoring a database backup;
- when indexes had to be wiped, for example because the Hibernate Search mapping or some core settings changed;
- when automatic indexing had to be disabled for performance reasons, and periodic reindexing (every night, ...) is preferred.

To address these situations, Hibernate Search provides the **MassIndexer**: a tool to rebuild indexes completely based on the content of the database. It can be told to reindex a few selected indexed types, or all of them.

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

- Indexes are purged completely when mass indexing starts.
- Mass indexing is performed by several parallel threads, each loading data from the database and sending indexing requests to the indexes.

Because of the initial index purge, and because mass indexing is a very resource-intensive operation, it is recommended to take your application offline while the **MassIndexer** works.

Querying the index while a **MassIndexer** is busy may be slower than usual and will likely return incomplete results.

The following snippet of code will rebuild the index of all indexed entities, deleting the index and then reloading all entities from the database.

*Example 71. Reindexing everything using a **MassIndexer***

```java
SearchSession searchSession = Search.session( entityManager );
searchSession.massIndexer().startAndWait();
```

1. Get the `SearchSession`.
2. Create a `MassIndexer` targeting every indexed entity type.
3. Start the mass indexing process and return when it is over.
The MassIndexer creates its own, separate sessions and (read-only) transactions, so there is no need to begin a database transaction before the MassIndexer is started or to commit a transaction after it is done.

A note to MySQL users: the MassIndexer uses forward only scrollable results to iterate on the primary keys to be loaded, but MySQL’s JDBC driver will pre-load all values in memory.

To avoid this "optimization" set the idFetchSize parameter to Integer.MIN_VALUE.

You can also select entity types when creating a mass indexer, so as to reindex only these types (and their indexed subtypes, if any):

Example 72. Reindexing selected types using a MassIndexer

```java
searchSession.massIndexer( Book.class ) ➊ .startAndWait(); ➋
```

➊ Create a MassIndexer targeting the Book type and its indexed subtypes (if any).

➋ Start the mass indexing process for the selected types and return when it is over.

It is possible to run the mass indexer asynchronously, because, the mass indexer does not rely on the original Hibernate ORM session. When used asynchronously, the mass indexer will return a future to track the completion of mass indexing:

Example 73. Reindexing asynchronously using a MassIndexer

```java
CompletableFuture<? extends ?> future = searchSession.massIndexer() ➊ .start(); ➋
```

➊ Create a MassIndexer.

➋ Start the mass indexing process, but do not wait for the process to finish. A CompletableFuture is returned.

Although the MassIndexer is simple to use, some tweaking is recommended to speed up the process. Several optional parameters are available, and can be set as shown below, before the mass indexer starts. See MassIndexer parameters for a reference of all available parameters, and Tuning the MassIndexer for best performance for details about key topics.
Example 74. Using a tuned MassIndexer

searchSession.massIndexer() ①
  .idFetchSize( 150 ) ②
  .batchSizeToLoadObjects( 25 ) ③
  .threadsToLoadObjects( 12 ) ④
  .startAndWait(); ⑤

① Create a MassIndexer.
② Load Book identifiers by batches of 150 elements.
③ Load Book entities to reindex by batches of 25 elements.
④ Create 12 parallel threads to load the Book entities.
⑤ Start the mass indexing process and return when it is over.

Running the MassIndexer with many threads will require many connections to the database. If you don’t have a sufficiently large connection pool, the MassIndexer itself and/or your other applications could starve and be unable to serve other requests: make sure you size your connection pool according to the mass indexing parameters, as explained in Threads and JDBC connections.

8.2.2. MassIndexer parameters

Table 5. MassIndexer parameters

<table>
<thead>
<tr>
<th>Setter</th>
<th>Default value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>typesToIndexInParallel(int)</td>
<td>1</td>
<td>The number of types to index in parallel.</td>
</tr>
<tr>
<td>threadsToLoadObjects(int)</td>
<td>6</td>
<td>The number of threads for entity loading, for each type indexed in parallel. That is to say, the number of threads spawned for entity loading will be typesToIndexInParallel * threadsToLoadObjects (+ 1 thread per type to retrieve the IDs of entities to load).</td>
</tr>
<tr>
<td>Setter</td>
<td>Default value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>idFetchSize(int)</td>
<td>100</td>
<td>The fetch size to be used when loading primary keys. Some databases accept special values, for example MySQL might benefit from using $\text{Integer}$.MIN_VALUE, otherwise it will attempt to preload everything in memory.</td>
</tr>
<tr>
<td>batchSizeToLoadObjects(int)</td>
<td>10</td>
<td>The fetch size to be used when loading entities from database. Some databases accept special values, for example MySQL might benefit from using $\text{Integer}$.MIN_VALUE, otherwise it will attempt to preload everything in memory.</td>
</tr>
<tr>
<td>Setter</td>
<td>Default value</td>
<td>Description</td>
</tr>
<tr>
<td>------------------------------------------</td>
<td>---------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>dropAndCreateSchemaOnStart(boolean)</td>
<td>false</td>
<td>Drops the indexes and their schema (if they exist) and re-creates them before indexing. Indexes will be unavailable for a short time during the dropping and re-creation, so this should only be used when failures of concurrent operations on the indexes (automatic indexing, ...) are acceptable. This should be used when the existing schema is known to be obsolete, for example when the Hibernate Search mapping changed and some fields now have a different type, a different analyzer, new capabilities (projectable, ...), etc. This may also be used when the schema is up-to-date, since it can be faster than a purge (purgeAllOnStart) on large indexes, especially with the Elasticsearch backend. As an alternative to this parameter, you can also use a schema manager to manage schemas manually at the time of your choosing: Manual schema management.</td>
</tr>
<tr>
<td>purgeAllOnStart(boolean)</td>
<td>true</td>
<td>Removes all entities from the indexes before indexing. Only set this to false if you know the index is already empty; otherwise, you will end up with duplicates in the index.</td>
</tr>
<tr>
<td>Setter</td>
<td>Default value</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>mergeSegmentsAfterPurge(boolean)</code></td>
<td>true</td>
<td>Force merging of each index into a single segment after the initial index purge, just before indexing. This setting has no effect if purgeAllOnStart is set to false.</td>
</tr>
<tr>
<td><code>mergeSegmentsOnFinish(boolean)</code></td>
<td>false</td>
<td>Force merging of each index into a single segment after indexing. This operation does not always improve performance: see Merging segments and performance.</td>
</tr>
<tr>
<td><code>cacheMode(CacheMode)</code></td>
<td><code>CacheMode.IGNORE</code></td>
<td>The Hibernate CacheMode when loading entities. The default is <code>CacheMode.IGNORE</code>, and it will be the most efficient choice in most cases, but using another mode such as <code>CacheMode.GET</code> may be more efficient if many of the entities being indexed refer to a small set of other entities.</td>
</tr>
<tr>
<td><code>transactionTimeout</code></td>
<td>-</td>
<td>Only supported in JTA-enabled environments. Timeout of transactions for loading ids and entities to be re-indexed. The timeout should be long enough to load and index all entities of one type. Note that these transactions are read-only, so choosing a large value (e.g. 1800, meaning 30 minutes) should not cause any problem.</td>
</tr>
<tr>
<td>Setter</td>
<td>Default value</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>---------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>limitIndexedObjectsTo(long)</code></td>
<td>-</td>
<td>The maximum number of results to load per entity type. This parameter let you define a threshold value to avoid loading too many entities accidentally. The value defined must be greater than 0. The parameter is not used by default. It is equivalent to keyword <code>LIMIT</code> in SQL.</td>
</tr>
<tr>
<td><code>monitor(MassIndexingMonitor)</code></td>
<td>A logging monitor.</td>
<td>The component responsible for monitoring progress of mass indexing. As a <code>MassIndexer</code> can take some time to finish its job, it is often necessary to monitor its progress. The default, built-in monitor logs progress periodically at the <code>INFO</code> level, but a custom monitor can be set by implementing the <code>MassIndexingMonitor</code> interface and passing an instance using the <code>monitor</code> method. Implementations of <code>MassIndexingMonitor</code> must be thread-safe.</td>
</tr>
</tbody>
</table>
### 8.2.3. Tuning the MassIndexer for best performance

**Basics**

The MassIndexer was designed to finish the re-indexing task as quickly as possible, but there is no
one-size-fits-all solution, so some configuration is required to get the best of it.

Performance optimization can get quite complex, so keep the following in mind while you attempt to configure the MassIndexer:

- Always test your changes to assess their actual effect: advice provided in this section is true in general, but each application and environment is different, and some options, when combined, may produce unexpected results.
- Take baby steps: before tuning mass indexing with 40 indexed entity types with two million instances each, try a more reasonable scenario with only one entity type, optionally limiting the number of entities to index to assess performance more quickly.
- Tune your entity types individually before you try to tune a mass indexing operation that indexes multiple entity types in parallel.

Threads and JDBC connections

Increasing parallelism usually helps as the bottleneck usually is the latency to the database connection: it’s probably worth it to experiment with a number of threads significantly higher than the number of actual cores available.

However, each thread requires one JDBC connection, and JDBC connections are usually in limited supply. In order to increase the number of threads safely:

1. You should make sure your database can actually handle the resulting number of connections.
2. Your JDBC connection pool should be configured to provide a sufficient number of connections.
3. The above should take into account the rest of your application (request threads in a web application): ignoring this may bring other processes to a halt while the MassIndexer is working.

There is a simple formula to understand how the different options applied to the MassIndexer affect the number of used worker threads and connections:

\[
\text{threads} = \text{typesToIndexInParallel} \times (\text{threadsToLoadObjects} + 1);
\]

\[
\text{required JDBC connections} = \text{threads};
\]

Here are a few suggestions for a roughly sane tuning starting point for the parameters that affect parallelism:

**typesToIndexInParallel**

Should probably be a low value, like 1 or 2, depending on how much of your CPUs have spare cycles and how slow a database round trip will be.

**threadsToLoadObjects**

Higher increases the pre-loading rate for the picked entities from the database, but also increases
memory usage and the pressure on the threads working on subsequent indexing. Note that each thread will extract data from the entity to reindex, which depending on your mapping might require to access lazy associations and load associated entities, thus making blocking calls to the database, so you will probably need a high number of threads working in parallel.

All internal thread groups have meaningful names prefixed with "Hibernate Search", so they should be easily identified with most diagnostic tools, including simply thread dumps.

8.3. Reindexing large volumes of data with the JSR-352 integration

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

8.4. Manual indexing

8.4.1. Basics

While automatic indexing and the MassIndexer or the mass indexing job should take care of most needs, it is sometimes necessary to control indexing manually, for example to reindex just a few entity instances that were affected by changes to the database that automatic indexing cannot detect, such as JPQL/SQL insert, update or delete queries.

To address these use cases, Hibernate Search exposes several APIs explained in the following sections.

As with everything in Hibernate Search, these APIs only affect the Hibernate Search indexes: they do not write anything to the database.

8.4.2. Controlling entity reads and index writes with SearchIndexingPlan

A fairly common use case when manipulating large datasets with JPA is the periodic "flush-clear" pattern, where a loop reads or writes entities for every iteration and flushes then clears the session every n iterations. This pattern allows processing a large number of entities while keeping the memory footprint reasonably low.

Below is an example of this pattern to persist a large number of entities when not using Hibernate Search.
Example 75. A batch process with JPA

```java
entityManager.getTransaction().begin();
try {
    for ( int i = 0 ; i < NUMBER_OF_BOOKS ; ++i ) { ①
        Book book = newBook( i );
        entityManager.persist( book ); ②

        if ( ( i + 1 ) % BATCH_SIZE == 0 ) { ③
            entityManager.flush();
            entityManager.clear(); ④
        }
    }
    entityManager.getTransaction().commit();
} catch (RuntimeException e) {
    entityManager.getTransaction().rollback();
    throw e;
}
```

① Execute a loop for a large number of elements, inside a transaction.

② For every iteration of the loop, instantiate a new entity and persist it.

③ Every `BATCH_SIZE` iterations of the loop, flush the entity manager to send the changes to the database-side buffer.

④ After a flush, clear the ORM session to release some memory.

With Hibernate Search 6 (on contrary to Hibernate Search 5 and earlier), this pattern will work as expected: documents will be built on flushes, and sent to the index upon transaction commit.

However, each flush call will potentially add data to an internal document buffer, which for large volumes of data may lead to an `OutOfMemoryException`, depending on the JVM heap size and on the complexity and number of documents.

If you run into memory issues, the first solution is to break down the batch process into multiple transactions, each handling a smaller number of elements: the internal document buffer will be cleared after each transaction.

See below for an example.

With this pattern, if one transaction fails, part of the data will already be in the database and in indexes, with no way to roll back the changes.

However, the indexes will be consistent with the database, and it will be possible to (manually) restart the process from the last transaction that failed.
Example 76. A batch process with Hibernate Search using multiple transactions

```java
try {
    int i = 0;
    while (i < NUMBER_OF_BOOKS) {  
        entityManager.getTransaction().begin();  
        int end = Math.min(i + BATCH_SIZE, NUMBER_OF_BOOKS);  
        for (; i < end; ++i) {  
            Book book = newBook(i);
            entityManager.persist(book);  
        }  
        entityManager.getTransaction().commit();  
    }
} catch (RuntimeException e) {
    entityManager.getTransaction().rollback();  
    throw e;
}
```

1. Add an outer loop that creates one transaction per iteration.
2. Begin the transaction at the beginning of each iteration of the outer loop.
3. Only handle a limited number of elements per transaction.
4. For every iteration of the loop, instantiate a new entity and persist it. Note we're relying on automatic indexing to index the entity, but this would work just as well if automatic indexing was disabled, only requiring an extra call to index the entity. See Explicitly indexing and deleting specific documents.
5. Commit the transaction at the end of each iteration of the outer loop. The entities will be flushed and indexed automatically.

The multi-transaction solution and the original `flush()`/`clear()` loop pattern can be combined, breaking down the process in multiple medium-sized transactions, and periodically calling `flush/clear` inside each transaction.

This combined solution is the most flexible, hence the most suitable if you want to fine-tune your batch process.

If breaking down the batch process into multiple transactions is not an option, a second solution is to just write to indexes after the call to `session.flush()`/`session.clear()`, without waiting for the database transaction to be committed: the internal document buffer will be cleared after each write to indexes.

This is done by calling the `execute()` method on the indexing plan, as shown in the example below.
With this pattern, if an exception is thrown, part of the data will already be in the
index, with no way to roll back the changes, while the database changes will have
been rolled back. The index will thus be inconsistent with the database.

To recover from that situation, you will have to either execute the exact same
database changes that failed manually (to get the database back in sync with the
index), or reindex the entities affected by the transaction manually (to get the
index back in sync with the database).

Of course, if you can afford to take the indexes offline for a longer period of time, a
simpler solution would be to wipe the indexes clean and reindex everything.

Example 77. A batch process with Hibernate Search using `execute()`

```java
SearchSession searchSession = Search.session(entityManager); ①
SearchIndexingPlan indexingPlan = searchSession.indexingPlan(); ②
entityManager.getTransaction().begin();
try {
    for ( int i = 0 ; i < NUMBER_OF_BOOKS ; ++i ) {
        Book book = newBook( i );
        entityManager.persist( book ); ③
        if ( ( i + 1 ) % BATCH_SIZE == 0 ) {
            entityManager.flush();
            entityManager.clear();
            indexingPlan.execute(); ④
        }
    }
    entityManager.getTransaction().commit(); ⑤
} catch (RuntimeException e) {
    entityManager.getTransaction().rollback();
    throw e;
}
```

① Get the `SearchSession`.
② Get the search session's indexing plan.
③ For every iteration of the loop, instantiate a new entity and persist it. Note we're relying on
automatic indexing to index the entity, but this would work just as well if automatic indexing was
disabled, only requiring an extra call to index the entity. See Explicitly indexing and deleting
specific documents.
④ After after a `flush()|clear()`, call `indexingPlan.execute()`. The entities will be
processed and the changes will be sent to the indexes immediately. Hibernate Search will wait
for index changes to be "completed" as required by the configured synchronization strategy.
⑤ After the loop, commit the transaction. The remaining entities that were not flushed/cleared will
be flushed and indexed automatically.
8.4.3. Explicitly indexing and deleting specific documents

When automatic indexing is disabled, the indexes will start empty and stay that way until explicit indexing commands are sent to Hibernate Search.

Indexing is done in the context of an ORM session using the SearchIndexingPlan interface. This interface represents the (mutable) set of changes that are planned in the context of a session, and will be applied to indexes upon transaction commit.

This interface offers the following methods:

**addOrUpdate(Object entity)**
Add or update a document in the index if the entity type is mapped to an index (@Indexed), and re-index documents that embed this entity (through @IndexedEmbedded for example).

**delete(Object entity)**
Delete a document from the index if the entity type is mapped to an index (@Indexed), and re-index documents that embed this entity (through @IndexedEmbedded for example).

**purge(Class<?> entityType, Object id)**
Delete the entity from the index, but do not try to re-index documents that embed this entity.

Compared to delete, this is mainly useful if the entity has already been deleted from the database and is not available, even in a detached state, in the session. In that case, reindexing associated entities will be the user’s responsibility, since Hibernate Search cannot know which entities are associated to an entity that no longer exists.

**purge(String entityName, Object id)**
Same as purge(Class<?> entityType, Object id), but the entity type is referenced by its name (see @javax.persistence.Entity#name).

**process() and execute()**
Respectively, process the changes and apply them to indexes.

These methods will be executed automatically on commit, so they are only useful when processing large number of items, as explained in Controlling entity reads and index writes with SearchIndexingPlan.

Below are examples of using addOrUpdate and delete.
Example 78. Explicitly adding or updating an entity in the index using `SearchIndexingPlan`

```java
SearchSession searchSession = Search.session( entityManager ); ①  
SearchIndexingPlan indexingPlan = searchSession.indexingPlan(); ②  
entityManager.getTransaction().begin();  
try {  
    Book book = entityManager.getReference( Book.class, 5 ); ③  
    indexingPlan.addOrUpdate( book ); ④  
    entityManager.getTransaction().commit(); ⑤  }  
catch (RuntimeException e) {  
    entityManager.getTransaction().rollback();  
    throw e;  
}
```

① Get the `SearchSession`.
② Get the search session’s indexing plan.
③ Fetch from the database the `Book` we want to index.
④ Submit the `Book` to the indexing plan for an add-or-update operation. The operation won’t be executed immediately, but will be delayed until the transaction is committed.
⑤ Commit the transaction, allowing Hibernate Search to actually write the document to the index.

Example 79. Explicitly deleting an entity from the index using `SearchIndexingPlan`

```java
SearchSession searchSession = Search.session( entityManager ); ①  
SearchIndexingPlan indexingPlan = searchSession.indexingPlan(); ②  
entityManager.getTransaction().begin();  
try {  
    Book book = entityManager.getReference( Book.class, 5 ); ③  
    indexingPlan.delete( book ); ④  
    entityManager.getTransaction().commit(); ⑤  }  
catch (RuntimeException e) {  
    entityManager.getTransaction().rollback();  
    throw e;  
}
```

① Get the `SearchSession`.
② Get the search session’s indexing plan.
③ Fetch from the database the `Book` we want to un-index.
④ Submit the `Book` to the indexing plan for a delete operation. The operation won’t be executed immediately, but will be delayed until the transaction is committed.
⑤ Commit the transaction, allowing Hibernate Search to actually delete the document from the index.
Multiple operations can be performed in a single indexing plan. The same entity can even be changed multiple times, for example added and then removed: Hibernate Search will simplify the operation as expected.

This will work fine for any reasonable number of entities, but changing or simply loading large numbers of entities in a single session requires special care with Hibernate ORM, and then some extra care with Hibernate Search. See Controlling entity reads and index writes with SearchIndexingPlan for more information.

8.4.4. Explicitly altering a whole index

Some index operations are not about a specific entity/document, but rather about a large number of documents, possibly all of them. This includes, for example, purging the index to remove all of its content.

The operations are performed outside of the context of an ORM session, using the SearchWorkspace interface. This interface exposes various large-scale operations that can be applied to an index or a set of indexes. These operations are triggered as soon as they are requested, without waiting for the transaction commit.

This interface offers the following methods:

**purge()**

Delete all documents from indexes targeted by this workspace.

With multi-tenancy enabled, only documents of the current tenant will be removed: the tenant of the session from which this workspace originated.

**purgeAsync()**

Asynchronous version of purge() returning a CompletableFuture.

**purge(Set<String> routingKeys)**

Delete documents from indexes targeted by this workspace that were indexed with any of the given routing keys.

With multi-tenancy enabled, only documents of the current tenant will be removed: the tenant of the session from which this workspace originated.

**purgeAsync(Set<String> routingKeys)**

Asynchronous version of purge(Set<String>) returning a CompletableFuture.

**flush()**

Flush to disk the changes to indexes that have not been committed yet. In the case of backends with a transaction log (Elasticsearch), also apply operations from the transaction log that were not
applied yet.

This is generally not useful as Hibernate Search commits changes automatically. See Commit and refresh for more information.

flushAsync()
Asynchronous version of flush() returning a CompletableFuture.

refresh()
Refresh the indexes so that all changes executed so far will be visible in search queries.

This is generally not useful as indexes are refreshed automatically. See Commit and refresh for more information.

refreshAsync()
Asynchronous version of refresh() returning a CompletableFuture.

mergeSegments()
Merge each index targeted by this workspace into a single segment. This operation does not always improve performance: see Merging segments and performance.

mergeSegmentsAsync()
Asynchronous version of mergeSegments() returning a CompletableFuture. This operation does not always improve performance: see Merging segments and performance.
Merging segments and performance

The merge-segments operation may affect performance positively as well as negatively.

This operation will regroup all index data into a single, huge segment (a file). This may speed up search at first, but as documents are deleted, this huge segment will begin to fill with "holes" which have to be handled as special cases during search, degrading performance.

Elasticsearch/Lucene do address this by rebuilding the segment at some point, but only once a certain ratio of deleted documents is reached. If all documents are in a single, huge segment, this ratio is less likely to be reached, and the index performance will continue to degrade for a long time.

There are, however, two situations in which merging segments may help:

1. No deletions or document updates are expected for an extended period of time.
2. Most, or all documents have just been removed from the index, leading to segments consisting mostly of deleted documents. In that case, it makes sense to regroup the few remaining documents into a single segment, though Elasticsearch/Lucene will probably do it automatically.

Below is an example using a SearchWorkspace to purge several indexes.

Example 80. Purging indexes using a SearchWorkspace

```java
SearchSession searchSession = Search.session(entityManager); ①
SearchWorkspace workspace = searchSession.workspace(Book.class, Author.class); ②
workspace.purge(); ③
```

① Get a SearchSession.
② Get a workspace targeting the indexes mapped to the Book and Author entity types.
③ Trigger a purge. This method is synchronous and will only return after the purge is complete, but an asynchronous method, purgeAsync, is also available.

There are multiple ways to retrieve a SearchWorkspace to target one, several or all indexes:
Example 81. Retrieving a `SearchWorkspace`

```java
SearchSession searchSession = Search.session( entityManager ); ①
SearchWorkspace workspace1 = searchSession.workspace(); ②
SearchWorkspace workspace2 = searchSession.workspace( Book.class ); ③
SearchWorkspace workspace3 = searchSession.workspace( Book.class, Author.class ); ④
```

① Get a `SearchSession`.
  
② Get a workspace targeting all indexes.
  
③ Get a workspace targeting the index mapped to the `Book` entity type.
  
④ Get a workspace targeting the indexes mapped to the `Book` and `Author` entity types.
Chapter 9. Searching

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

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

9.1. Query DSL

9.1.1. Basics

Preparing and executing a query requires just a few lines:

Example 82. Executing a search query

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

SearchResult<Book> result = searchSession.search(Book.class) ②
    .where(f -> f.match() ③
        .field("title")
        .matching("robot")
    )
    .fetch(20); ④

long totalHitCount = result.totalHitCount(); ⑤
List<Book> hits = result.hits(); ⑥
// 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. See `Predicate DSL` for more information about predicates.
4. Build the query and fetch the results, limiting to the top 20 hits.
5. Retrieve the total number of matching entities.
6. Retrieve matching entities.

By default, the hits of a search query will be entities managed by Hibernate ORM, bound to the entity manager used to create the search session. This provides all the benefits of Hibernate ORM, in particular the ability to navigate the entity graph to retrieve associated entities if necessary.

The query DSL offers many features, detailed in the following sections. Some commonly used features include:
• **predicates**, the main component of a search query, i.e. the condition that every document must satisfy in order to be included in search results.

• **fetching the results differently**: getting the hits directly as a list, using pagination, scrolling, etc.

• **sorts**, to order the hits in various ways: by score, by the value of a field, by distance to a point, etc.

• **projections**, to retrieve hits that are not just managed entities: data can be extracted from the index (field values), or even from both the index and the database.

• **aggregations**, to group hits and compute aggregated metrics for each group—hit count by category, for example.

### 9.1.2. Advanced entity types targeting

#### Targeting multiple entity types

When multiple entity types have similar indexed fields, it is possible to search across these multiple types in a single search query: the search result will contain hits from any of the targeted types.

*Example 83. Targeting multiple entity types in a single search query*

```java
SearchResult<Person> result = searchSession.search( Arrays.asList( Manager.class, Associate.class )
  .where( f -> f.match() //
    .field( "name" )
    .matching( "james" )
  ).fetch( 20 ); //
```

1. Initiate a search query targeting the indexes mapped to the `Manager` and `Associate` entity types. Since both entity types implement the `Person` interface, search hits will be instances of `Person`.

2. Continue building the query as usual. There are restrictions regarding the fields that can be used: see the note below.

3. Fetch the search result. Hits will all be instances of `Person`.

Multi-entity (multi-index) searches will only work well as long as the fields referenced in predicates/sorts/etc. are identical in all targeted indexes (same type, same analyzer, ...). Fields that are defined in only one of the targeted indexes will also work correctly.

If you want to reference index fields that are even slightly different in one of the targeted indexes (different type, different analyzer, ...), see Targeting multiple fields.
Targeting entity types by name

Though rarely necessary, it is also possible to use entity names instead of classes to designate the entity types targeted by the search:

Example 84. Targeting entity types by name

```java
SearchResult<Person> result = searchSession.search(  
    searchSession.scope(  
        Person.class,  
        Arrays.asList( "Manager", "Associate" )  
    )  
)  
    .where( f -> f.match()  
        .field( "name" )  
        .matching( "james" )  
    )  
    .fetch( 20 );
```

1. Initiate a search query.
2. Pass a custom scope encompassing the indexes mapped to the Manager and Associate entity types, expecting those entity types to implement the Person interface (Hibernate Search will check that).
3. Continue building the query as usual.
4. Fetch the search result. Hits will all be instances of Person.

9.1.3. Fetching results

Basics

In Hibernate Search, the default search result is a little bit more complicated than just "a list of hits". This is why the default methods return a composite SearchResult object offering getters to retrieve the part of the result you want, as shown in the example below.
Example 85. Getting information from a `SearchResult`

```java
SearchResult<Book> result = searchSession.search(Book.class)  // ①
    .where(f -> f.matchAll())
    .fetch(20);  // ②

long totalHitCount = result.totalHitCount();  // ③
List<Book> hits = result.hits();  // ④
// ...  // ⑤
```

1. Start building the query as usual.
2. Fetch the results, limiting to the top 20 hits.
3. Retrieve the total hit count, i.e. the total number of matching entities/documents, which could be 10,000 even if you only retrieved the top 20 hits. This is useful to give end users and idea of how many more hits they query produced.
4. Retrieve the top hits, in this case the top 20 matching entities/documents.
5. Other kinds of results and information can be retrieved from `SearchResult`. They are explained in dedicated sections, such as Aggregation DSL.

It is possible to retrieve the total hit count alone, for cases where only the number of hits is of interest, not the hits themselves:

Example 86. Getting the total hit count directly

```java
long totalHitCount = searchSession.search(Book.class)
    .where(f -> f.matchAll())
    .fetchTotalHitCount();
```

The top hits can also be obtained directly, without going through a `SearchResult`, which can be handy if only the top hits are useful, and not the total hit count:

Example 87. Getting the top hits directly

```java
List<Book> hits = searchSession.search(Book.class)
    .where(f -> f.matchAll())
    .fetchHits(20);
```

If only zero to one hit is expected, it is possible to retrieve it as an `Optional`. An exception will be thrown if more than one hits are returned.
Example 88. Getting the only hit directly

```java
Optional<Book> hit = searchSession.search(Book.class)
    .where(f -> f.id().matching(1))
    .fetchSingleHit();
```

Fetching all hits

Fetching all hits is rarely a good idea: if the query matches many entities/documents, this may lead to loading millions of entities in memory, which will likely crash the JVM, or at the very least slow it down to a crawl.

If you know your query will always have less than N hits, consider setting the limit to N to avoid memory issues.

If there is no bound to the number of hits you expect, you should consider Pagination or Scrolling to retrieve data in batches.

If you still want to fetch all hits in one call, be aware that the Elasticsearch backend will only ever return 10,000 hits at a time, due to internal safety mechanisms in the Elasticsearch cluster.

Example 89. Getting all hits in a SearchResult

```java
SearchResult<Book> result = searchSession.search(Book.class)
    .where(f -> f.id().matchingAny(Arrays.asList(1, 2)))
    .fetchAll();
long totalHitCount = result.totalHitCount();
List<Book> hits = result.hits();
```

Example 90. Getting all hits directly

```java
List<Book> hits = searchSession.search(Book.class)
    .where(f -> f.id().matchingAny(Arrays.asList(1, 2)))
    .fetchAllHits();
```

Pagination

Pagination is the concept of splitting hits in successive "pages", all pages containing a fixed number of elements (except potentially the last one). When displaying results on a web page, the user will be able to go to an arbitrary page and see the corresponding results, for example "results 151 to 170 of 14,265".
Pagination is achieved in Hibernate Search by passing an offset and a limit to the `fetch` or `fetchHits` method:

- The offset defines the number of documents that should be skipped because they were displayed in previous pages. It is a number of documents, not a number of pages, so you will usually want to compute it from the page number and page size this way: \( \text{offset} = \text{zero-based-page-number} \times \text{page-size} \).

- The limit defines the maximum number of hits to return, i.e. the page size.

**Example 91. Pagination retrieving a `SearchResult`**

```java
SearchResult<Book> result = searchSession.search( Book.class )
    .where( f -> f.matchAll() )
    .fetch( 40, 20 ); ①
```

① Set the offset to 40 and the limit to 20.

**Example 92. Pagination retrieving hits directly**

```java
List<Book> hits = searchSession.search( Book.class )
    .where( f -> f.matchAll() )
    .fetchHits( 40, 20 ); ①
```

① Set the offset to 40 and the limit to 20.

**Scrolling**

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

### 9.1.4. Routing

For a preliminary introduction to sharding, including how it works in Hibernate Search and what its limitations are, see [Sharding and routing](#).

If, for a given index, there is one immutable value that documents are often filtered on, for example a "category" or a "user id", it is possible to match documents with this value using a routing key instead of a predicate.

The main advantage of a routing key over a predicate is that, on top of filtering documents, the routing key will also filter shards. If sharding is enabled, this means only part of the index will be scanned during query execution, potentially increasing search performance.
A pre-requisite to using routing in search queries is to map your entity in such a way that it is assigned a routing key at indexing time.

Specifying routing keys is done by calling the `.routing(String)` or `.routing(Collection<String>)` methods when building the query:

Example 93. Routing a query to a subset of all shards

```java
SearchResult<Book> result = searchSession.search(Book.class) ①
  .where(f -> f.match() ②
    .field("genre")
    .matching(Genre.SCIENCE_FICTION)
  .routing(Genre.SCIENCE_FICTION.name()) ③
  .fetch(20); ④
```

1. Start building the query.
2. Define that only documents matching the given genre should be returned.
3. In this case, the entity is mapped in such a way that the genre is also used as a routing key. We know all documents will have the given genre value, so we can specify the routing key to limit the query to relevant shards.
4. Build the query and fetch the results.

9.1.5. Entity loading options

Hibernate Search executes database queries to load entities that are returned as part of the hits of a search query.

This section presents all available options related to entity loading in search queries.

Cache lookup strategy

By default, Hibernate Search will load entities from the database directly, without looking at any cache. This is a good strategy when the size of caches (Hibernate ORM session or second level cache) is much lower than the total number of indexed entities.

If a significant portion of your entities are present in the second level cache, you can force Hibernate Search to retrieve entities from the persistence context (the session) and/or the second level cache if possible. Hibernate Search will still need to execute a database query to retrieve entities missing from the cache, but the query will likely have to fetch fewer entities, leading to better performance and lower stress on your database.

This is done through the cache lookup strategy, which can be configured by setting the configuration property `hibernate.search.query.loading.cache_lookup.strategy:`
• **skip** (the default) will not perform any cache lookup.

• **persistence-context** will only look into the persistence context, i.e. will check if the entities are already loaded in the session. Useful if most search hits are expected to already be loaded in session, which is generally unlikely.

• **persistence-context-then-second-level-cache** will first look into the persistence context, then into the second level cache, if enabled in Hibernate ORM for the searched entity. Useful if most search hits are expected to be cached, which may be likely if you have a small number of entities and a large cache.

Before a second-level cache can be used for a given entity type, some configuration is required in Hibernate ORM.

See the caching section of the Hibernate ORM documentation for more information.

It is also possible to override the configured strategy on a per-query basis, as shown below.

**Example 94. Overriding the cache lookup strategy in a single search query**

```java
SearchResult<Book> result = searchSession.search(Book.class) ①
    .where(f -> f.match()  
        .field("title")  
        .matching("robot") 
    ).loading(o -> o.cacheLookupStrategy(EntityLoadingCacheLookupStrategy.PERSISTENCE_CONTEXT_THEN_SECOND_LEVEL_CACHE) ②)
    .fetch(20); ③
```

① Start building the query.

② Access the loading options of the query, then mention that the persistence context and second level cache should be checked before entities are loaded from the database.

③ Fetch the results. The more entities found in the persistence context or second level cache, the less entities will be loaded from the database.

**Fetch size**

By default, Hibernate Search will use a fetch size of **100**, meaning that for a single `fetch*()` call on a single query, it will run a first query to load the first 100 entities, then if there are more hits it will run a second query to load the next 100, etc.

The fetch size can be configured by setting the configuration property `hibernate.search.query.loading.fetch_size`. This property expects a strictly positive **integer value**.
It is also possible to override the configured fetch size on a per-query basis, as shown below.

Example 95. Overriding the fetch size in a single search query

```java
SearchResult<Book> result = searchSession.search( Book.class )
    .where( f -> f.match() )
    .field( "title" )
    .matching( "robot" )
    .loading( o -> o.fetchSize( 50 ) )
    .fetch( 200 );
```

1. Start building the query.
2. Access the loading options of the query, then set the fetch size to an arbitrary value (must be 1 or more).
3. Fetch the results, limiting to the top 200 hits. One query will be executed to load the hits if there are less hits than the given fetch size; two queries if there are more hits than the fetch size but less than twice the fetch size, etc.

Entity graph

By default, Hibernate Search will load associations according to the defaults of your mappings: associations marked as lazy won’t be loaded, while associations marked as eager will be loaded before returning the entities.

It is possible to force the loading of a lazy association, or to prevent the loading of an eager association, by referencing an entity graph in the query. See below for an example, and this section of the Hibernate ORM documentation for more information about entity graphs.
Example 96. Applying an entity graph to a search query

```java
EntityManager entityManager = /* ... */

EntityGraph<Manager> graph = entityManager.createEntityGraph(Manager.class); ①
graph.addAttributeNodes("associates");

SearchResult<Manager> result = Search.session(entityManager).search(Manager.class) ②
  .where(f -> f.match())
  .field("name")
  .matching("james")
  .loading(o -> o.graph(graph, GraphSemantic.FETCH)) ③
  .fetch(20); ④
```

① Build an entity graph.
② Start building the query.
③ Access the loading options of the query, then set the entity graph to the graph built above. You must also pass a semantic: GraphSemantic.FETCH means only associations referenced in the graph will be loaded; GraphSemantic.LOAD means associations referenced in the graph and associations marked as EAGER in the mapping will be loaded.
④ Fetch the results. All managers loaded by this search query will have their associates association already populated.

Instead of building the entity graph on the spot, you can also define the entity graph statically using the @NamedEntityGraph annotation, and pass the name of your graph to Hibernate Search, as shown below. See this section of the Hibernate ORM documentation for more information about @NamedEntityGraph.

Example 97. Applying a named entity graph to a search query

```java
SearchResult<Manager> result = Search.session(entityManager).search(Manager.class) ①
  .where(f -> f.match())
  .field("name")
  .matching("james")
  .loading(o -> o.graph("preload-associates", GraphSemantic.FETCH)) ②
  .fetch(20); ③
```

① Start building the query.
② Access the loading options of the query, then set the entity graph to "preload-associates", which was defined elsewhere using the @NamedEntityGraph annotation.
③ Fetch the results. All managers loaded by this search query will have their associates association already populated.

9.1.6. Timeout

You can limit the time it takes for a search query to execute in two ways:
• Aborting (throwing an exception) when the time limit is reached with `failAfter()`.

• Truncating the results when the time limit is reached with `truncateAfter()`.

Currently, the two approaches are incompatible: trying to set both `failAfter` and `truncateAfter` will result in unspecified behavior.

`failAfter()`: Aborting the query after a given amount of time

By calling `failAfter(...)` when building the query, it is possible to set a time limit for the query execution. Once the time limit is reached, Hibernate Search will stop the query execution and throw a `SearchTimeoutException`.

Timeouts are handled on a best-effort basis.

Depending on the resolution of the internal clock and on how often Hibernate Search is able to check that clock, it is possible that a query execution exceeds the timeout. Hibernate Search will try to minimize this excess execution time.

**Example 98. Triggering a failure on timeout**

```java
try {
    SearchResult<Book> result = searchSession.search(Book.class)  // 1
        .where(f -> f.match())
        .field("title")
        .matching("robot")
        .failAfter(500, TimeUnit.MILLISECONDS)  // 2
        .fetch(20);  // 3
}
catch (SearchTimeoutException e) {  // 4
    // ...
}
```

1. Build the query as usual.
2. Call `failAfter` to set the timeout.
3. Fetch the results.
4. Catch the exception if necessary.

`explain()` does not honor this timeout: this method is used for debugging purposes and in particular to find out why a query is slow.

`truncateAfter()`: Truncating the results after a given amount of time

By calling `truncateAfter(...)` when building the query, it is possible to set a time limit for the collection of search results. Once the time limit is reached, Hibernate Search will stop collecting hits.
Timeouts are handled on a best-effort basis.

Depending on the resolution of the internal clock and on how often Hibernate Search is able to check that clock, it is possible that a query execution exceeds the timeout. Hibernate Search will try to minimize this excess execution time.

**Example 99. Truncating the results on timeout**

```java
SearchResult<Book> result = searchSession.search(Book.class) ①
  .where(f -> f.match()
    .field("title")
    .matching("robot")
  ).truncateAfter(500, TimeUnit.MILLISECONDS) ②
  .fetch(20); ③
Duration took = result.took(); ④
Boolean timedOut = result.timedOut(); ⑤
```

① Build the query as usual.

② Call `truncateAfter` to set the timeout.

③ Fetch the results.

④ Optionally extract `took`: how much time the query took to execute.

⑤ Optionally extract `timedOut`: whether the query timed out.

explain() and `fetchTotalHitCount()` do not honor this timeout. The former is used for debugging purposes and in particular to find out why a query is slow. For the latter it does not make sense to return a partial result.

### 9.1.7. Obtaining a query object

The example presented in most of this documentation fetch the query results directly at the end of the query definition DSL, not showing any "query" object that can be manipulated. This is because the query object generally only makes code more verbose without bringing anything worthwhile.

However, in some cases a query object can be useful. To get a query object, just call `toQuery()` at the end of the query definition:
Example 100. Getting a SearchQuery object

```java
SearchQuery<Book> query = searchSession.search(Book.class) ①
  .where(f -> f.matchAll())
  .toQuery(); ②
List<Book> hits = query.fetchHits(20); ③
```

① Build the query as usual.
② Retrieve a SearchQuery object.
③ Fetch the results.

This query object supports all fetch* methods supported by the query DSL. The main advantage over calling these methods directly at the end of a query definition is mostly related to debugging (see Debugging a query), but the query object can also be useful if you need an adapter to another API.

Hibernate Search provides an adapter to JPA and Hibernate ORM’s native APIs, i.e. a way to turn a SearchQuery into a javax.persistence.TypedQuery (JPA) or a org.hibernate.query.Query (native ORM API):

Example 101. Getting a SearchQuery object

```java
SearchQuery<Book> query = searchSession.search(Book.class) ①
  .where(f -> f.matchAll())
  .toQuery(); ②
javax.persistence.TypedQuery<Book> jpaQuery = Search.toJpaQuery(query); ③
org.hibernate.query.Query<Book> ormQuery = Search.toOrmQuery(query); ④
```

① Build the query as usual.
② Retrieve a SearchQuery object.
③ Turn the SearchQuery object into a JPA query.
④ Turn the SearchQuery object into a Hibernate ORM query.
The resulting query does not support all operations, so it is recommended to only convert search queries when absolutely required, for example when integrating with code that only works with Hibernate ORM queries.

The following operations are expected to work correctly in most cases, even though they may behave slightly differently from what is expected from a JPA TypedQuery or Hibernate ORM Query in some cases (including, but not limited to, the type of thrown exceptions):

- Hit retrieval methods (list, getResultList, uniqueResult, ...).
- setFirstResult/setMaxResults and getters.
- setFetchSize
- unwrap

The following operations are known not to work correctly, with no plan to fix them at the moment:

- Hints (setHint, ...).
- Parameter-related methods (setParameter, ...).
- Result transformer (setResultTransformer, ...); use composite projections instead.
- Lock-related methods (setLockOptions, ...).
- And more (this list is not exhaustive).

9.1.8. Debugging a query

Explaining matches

When some documents unexpectedly match or don’t match, you will need information about the exact query being executed, and about the index content.

To gain insight about what ends up being executed exactly, one option is to create a SearchQuery object using toQuery() at the end of the query definition, then call toString() to get a String representation of that query.

Another option is to take advantage of logs: all executed search queries are logged to the log category org.hibernate.search.query at the DEBUG level.

You may also need to inspect the content of the index. This is rather obvious with Elasticsearch: run simpler queries using either Hibernate Search or the REST APIs directly. For the Lucene backend, use the Luke tool distributed as part of the Lucene binary packages.
Explaining scores

When the score of some documents is higher or lower than expected, the best way to gain insight is to create a SearchQuery object using toQuery() at the end of the query definition, and then use the backend-specific explain methods; the result of these methods will explain how the score of a specific document was computed. See below for examples.

To retrieve an explanation for all matches in one call, explanation projections are available: see here for Lucene and here for Elasticsearch.

Regardless of the API used, explanations are rather costly performance-wise: only use them for debugging purposes.

Example 102. Retrieving score explanation — Lucene

```java
LuceneSearchQuery<Book> query = searchSession.search(Book.class)
  .extension(LuceneExtension.get())
  .where(f -> f.match().field("title"))
  .matching("robot")
  .toQuery(); ①

Explanation explanation1 = query.explain("1"); ②
Explanation explanation2 = query.explain("Book", "1"); ③

LuceneSearchQuery<Book> luceneQuery = query.extension(LuceneExtension.get()); ④
```

① Build the query as usual, but using the Lucene extension so that the retrieved query exposes Lucene-specific operations.

② Retrieve a SearchQuery object.

③ Retrieve the explanation of the score of the document with ID 1. The explanation is of type Explanation, but you can convert it to a readable string using toString().

④ For multi-index queries, it is necessary to refer to the document not only by its ID, but also by the name of the index it’s located in.

⑤ If you cannot change the code building the query to use the Lucene extension, you can instead use the Lucene extension on the SearchQuery to convert it after its creation.
Example 103. Retrieving score explanation – Elasticsearch

ElasticsearchSearchQuery<Book> query = searchSession.search(Book.class)  
  .extension(ElasticsearchExtension.get()) ①  
  .where(f -> f.match()  
      .field("title")  
      .matching("robot") )  
  .toQuery(); ②

JsonObject explanation1 = query.explain("1"); ③  
JsonObject explanation2 = query.explain("Book", "1"); ④

ElasticsearchSearchQuery<Book> elasticsearchQuery = query.extension(ElasticsearchExtension.get()); ⑤

① Build the query as usual, but using the Elasticsearch extension so that the retrieved query exposes Elasticsearch-specific operations.
② Retrieve a SearchQuery object.
③ Retrieve the explanation of the score of the document with ID 1.
④ For multi-index queries, it is necessary to refer to the document not only by its ID, but also by the name of the index it’s located in.
⑤ If you cannot change the code building the query to use the Elasticsearch extension, you can instead use the Elasticsearch extension on the SearchQuery to convert it after its creation.

Query metadata in SearchResult: took and timed_out

Example 104. Returning query execution time and whether a timeout occurred

SearchQuery<Book> query = searchSession.search(Book.class)  
  .where(f -> f.match()  
      .field("title")  
      .matching("robot") )  
  .toQuery();

SearchResult<Book> result = query.fetch(20); ①  

Duration took = result.took(); ②  
Boolean timedOut = result.timedOut(); ③

① Fetch the results.
② Extract took: how much time the query took (in case of Elasticsearch, ignoring network latency between the application and the Elasticsearch cluster).
③ Extract timedOut: whether the query timed out (in case of Elasticsearch, ignoring network latency between the application and the Elasticsearch cluster).
9.1.9. Elasticsearch: leveraging advanced features with JSON manipulation

This feature is under active development. Usual compatibility policies do not apply: incompatible changes may be introduced in any future release.

Elasticsearch ships with many features. It is possible that at some point, one feature you need will not be exposed by the Search DSL.

To work around such limitations, Hibernate Search provides ways to:

- Transform the HTTP request sent to Elasticsearch for search queries.
- Read the raw JSON of the HTTP response received from Elasticsearch for search queries.

Direct changes to the HTTP request may conflict with Hibernate Search features and be supported differently by different versions of Elasticsearch.

Similarly, the content of the HTTP response may change depending on the version of Elasticsearch, depending on which Hibernate Search features are used, and even depending on how Hibernate Search features are implemented.

Thus, features relying on direct access to HTTP requests or responses cannot be guaranteed to continue to work when upgrading Hibernate Search, even for micro upgrades (x.y.z to x.y.(z+1)).

Use this at your own risk.

Most simple use cases will only need to change the HTTP request slightly, as shown below.
Example 105. Transforming the Elasticsearch request manually in a search query

```java
List<Book> hits = searchSession.search( Book.class )
  .extension( ElasticsearchExtension.get() )
  .where( f -> f.match()
             .field( "title" )
             .matching( "robot" ) )
  .requestTransformer( context -> {
    Map<String, String> parameters = context.parametersMap();
    parameters.put( "search_type", "dfs_query_then_fetch" );
    JsonObject body = context.body();
    body.addProperty( "min_score", 0.5f );
  } )
  .fetchHits( 20 );
```

1. Build the query as usual, but using the Elasticsearch extension so that Elasticsearch-specific options are available.

2. Add a request transformer to the query. Its `transform` method will be called whenever a request is about to be sent to Elasticsearch.

3. Inside the `transform` method, alter the HTTP query parameters.

4. It is also possible to alter the request’s JSON body as shown here, or even the request’s path (not shown in this example).

5. Retrieve the result as usual.

For more complicated use cases, it is possible to access the raw JSON of the HTTP response, as shown below.
Example 106. Accessing the Elasticsearch response body manually in a search query

```java
ElasticsearchSearchResult<Book> result = searchSession.search( Book.class )
    .extension( ElasticsearchExtension.get() ) ①
    .where( f -> f.match()
        .field( "title" )
        .matching( "robt" )
    )
    .requestTransformer( context -> {
        JsonObject body = context.body();
        body.add( "suggest", jsonObject( suggest -> {
            suggest.add( "my-suggest", jsonObject( mySuggest -> {
                mySuggest.addProperty( "text", "robt" );
                mySuggest.add( "term", jsonObject( term -> {
                    term.addProperty( "field", "title" );
                } ) );
            } ) );
        } ) );
    } ); ②
    .fetch( 20 ); ④

JsonObject responseBody = result.responseBody(); ⑤
JsonArray mySuggestResults = responseBody.getAsJsonObject( "suggest" ) ⑥
    .getAsJsonArray( "my-suggest" );
```

① Build the query as usual, but using the Elasticsearch extension so that Elasticsearch-specific options are available.

② Add a request transformer to the query.

③ Add content to the request body, so that Elasticsearch will return more data in the response. Here we’re asking Elasticsearch to apply a suggester.

④ Retrieve the result as usual. Since we used the Elasticsearch extension when building the query, the result is an ElasticsearchSearchResult instead of the usual SearchResult.

⑤ Get the response body as a JsonObject.

⑥ Extract useful information from the response body. Here we’re extracting the result of the suggester we configured above.

---

Gson’s API for building JSON objects is quite verbose, so the example above relies on a small, custom helper method to make the code more readable:

```java
private static JsonObject jsonObject(Consumer<JsonObject> instructions) {
    JsonObject object = new JsonObject();
    instructions.accept( object );
    return object;
}
```

When data needs to be extracted from each hit, it is often more convenient to use the jsonHit projection than parsing the whole response.
9.1.10. Lucene: retrieving low-level components

Lucene queries allow to retrieve some low-level components. This should only be useful to integrators, but is documented here for the sake of completeness.

Example 107. Accessing low-level components in a Lucene search query

```java
LuceneSearchQuery<Book> query = searchSession.search( Book.class )
    .extension( LuceneExtension.get() ) ①
    .where( f -> f.match() )
    .field( "title" )
    .matching( "robot" )
    .sort( f -> f.field( "title_sort" ) )
    .toQuery(); ②

Sort sort = query.luceneSort(); ③
LuceneSearchResult<Book> result = query.fetch( 20 ); ④
TopDocs topDocs = result.topDocs(); ⑤
```

① Build the query as usual, but using the Lucene extension so that Lucene-specific options are available.

② Since we used the Lucene extension when building the query, the query is a 
   LuceneSearchQuery instead of the usual SearchQuery.

③ Retrieve the org.apache.lucene.search.Sort this query relies on.

④ Retrieve the result as usual. LuceneSearchQuery returns a LuceneSearchResult instead of the usual SearchResult.

⑤ Retrieve the org.apache.lucene.search.TopDocs for this result. Note that the TopDocs are offset according to the arguments to the fetch method, if any.

9.2. Predicate DSL

9.2.1. Basics

The main component of a search query is the predicate, i.e. the condition that every document must satisfy in order to be included in search results.

The predicate is configured when building the search query:
Example 108. Defining the predicate of a search query

```java
SearchSession searchSession = Search.session( entityManager );
List<Book> result = searchSession.search( Book.class )
    .where( f -> f.match().field( "title" )
    .matching( "robot" )
    )
    .fetchHits( 20 );
```

1. Start building the query.
2. Mention that the results of the query are expected to have a title field matching the value robot. If the field does not exist or cannot be searched on, an exception will be thrown.
3. Fetch the results, which will match the given predicate.

Or alternatively, if you don’t want to use lambdas:

Example 109. Defining the predicate of a search query – object-based syntax

```java
SearchSession searchSession = Search.session( entityManager );
SearchScope<Book> scope = searchSession.scope( Book.class );
List<Book> result = searchSession.search( scope )
    .where( scope.predicate().match().field( "title" )
    .matching( "robot" )
    .toPredicate() )
    .fetchHits( 20 );
```

The predicate DSL offers more predicate types, and multiple options for each type of predicate. To learn more about the match predicate, and all the other types of predicate, refer to the following sections.

9.2.2. Options common to multiple predicate types

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

9.2.3. matchAll: match all documents

Example 110. Matching all documents

```java
List<Book> hits = searchSession.search( Book.class )
    .where( f -> f.matchAll() )
    .fetchHits( 20 );
```
Example 111. Matching all documents except those matching a given predicate

```java
List<Book> hits = searchSession.search( Book.class )
    .where( f -> f.matchAll()
        .except( f.match().field( "title" )
                    .matching( "robot" )
        )
    )
    .fetchHits( 20 );
```

9.2.4. id: match a document identifier

Example 112. Matching a document with a given identifier

```java
List<Book> hits = searchSession.search( Book.class )
    .where( f -> f.id().matching( 1 )
    )
    .fetchHits( 20 );
```

Example 113. Matching all documents with an identifier among a given collection

```java
List<Integer> ids = new ArrayList<>();
ids.add( 1 );
ids.add( 2 );
List<Book> hits = searchSession.search( Book.class )
    .where( f -> f.id().matchingAny( ids )
    )
    .fetchHits( 20 );
```

9.2.5. match: match a value

Example 114. Matching a value

```java
List<Book> hits = searchSession.search( Book.class )
    .where( f -> f.match().field( "title" )
        .matching( "robot" )
    )
    .fetchHits( 20 );
```
Example 115. Matching multiple terms

```java
List<Book> hits = searchSession.search(Book.class)
    .where( f -> f.match().field("title")
            .matching("robot dawn") )
    .fetchHits(20); ①
```

① For full-text fields, the value passed to `matching` may be a string containing multiple terms. The string will be analyzed and each term identified.

② All returned hits will match at least one term of the given string. Hits matching multiple terms will have a higher score.

Example 116. Matching a value in any of multiple fields

```java
List<Book> hits = searchSession.search(Book.class)
    .where( f -> f.match().field("title").field("description")
            .matching("robot") )
    .fetchHits(20);
```

```java
List<Book> hits = searchSession.search(Book.class)
    .where( f -> f.match().fields("title", "description")
            .matching("robot") )
    .fetchHits(20);
```

Example 117. Matching a text value approximately

```java
List<Book> hits = searchSession.search(Book.class)
    .where( f -> f.match().field("title")
            .matching("robto")
            .fuzzy() )
    .fetchHits(20);
```

Example 118. Matching a value, analyzing it with a different analyzer

```java
List<Book> hits = searchSession.search(Book.class)
    .where( f -> f.match().field("title_autocomplete")
            .matching("robo")
            .analyzer("autocomplete_query") )
    .fetchHits(20);
```
Example 119. Matching a value without analyzing it

```java
List<Book> hits = searchSession.search( Book.class )
    .where( f -> f.match() )
    .field( "title" )
    .matching( "robot" )
    .skipAnalysis() )
    .fetchHits( 20 );
```

9.2.6. range: match a range of values

Example 120. Matching a range of values

```java
List<Book> hits = searchSession.search( Book.class )
    .where( f -> f.range().field( "pageCount" )
    .between( 210, 250 ) )
    .fetchHits( 20 );
```

Example 121. Matching values equal to or greater than a given value

```java
List<Book> hits = searchSession.search( Book.class )
    .where( f -> f.range().field( "pageCount" )
    .atLeast( 400 ) )
    .fetchHits( 20 );
```

Example 122. Matching values strictly greater than a given value

```java
List<Book> hits = searchSession.search( Book.class )
    .where( f -> f.range().field( "pageCount" )
    .greaterThan( 400 ) )
    .fetchHits( 20 );
```

Example 123. Matching values equal to or less than a given value

```java
List<Book> hits = searchSession.search( Book.class )
    .where( f -> f.range().field( "pageCount" )
    .atMost( 400 ) )
    .fetchHits( 20 );
```
Example 124. Matching values strictly less than a given value

```
List<Book> hits = searchSession.search( Book.class )
    .where( f -> f.range().field( "pageCount" )
          .lessThan( 400 ) )
    .fetchHits( 20 );
```

Example 125. Matching a range of values with explicit bound inclusion/exclusion

```
List<Book> hits = searchSession.search( Book.class )
    .where( f -> f.range().field( "pageCount" )
            .between( 200, RangeBoundInclusion.EXCLUDED,
                      250, RangeBoundInclusion.EXCLUDED )
      )
    .fetchHits( 20 );
```

9.2.7. phrase: match a sequence of words

Example 126. Matching a sequence of words

```
List<Book> hits = searchSession.search( Book.class )
    .where( f -> f.phrase().field( "title" )
            .matching( "robots of dawn" )
      )
    .fetchHits( 20 );
```

Example 127. Matching a sequence of words approximately

```
List<Book> hits = searchSession.search( Book.class )
    .where( f -> f.phrase().field( "title" )
            .matching( "dawn robot" )
            .slop( 3 )
      )
    .fetchHits( 20 );
```

9.2.8. exists: match fields with content

The exists predicate, applied to a field, will match all documents for which this field has a non-null value.

Example 128. Matching fields with content

```
List<Book> hits = searchSession.search( Book.class )
    .where( f -> f.exists().field( "comment" )
      )
    .fetchHits( 20 );
```
There isn’t any built-in predicate to match fields with exclusively null values, but you can easily create one yourself using an `exists` predicate in a `mustNot` clause in a boolean predicate.

The `exists` predicate can also be applied to an object field. In that case, it will match all documents for which at least one sub-field of the object field has a non-null value.

**Example 129. Matching object fields with content**

```java
List<Author> hits = searchSession.search( Author.class )
    .where( f -> f.exists().field( "placeOfBirth" ) )
    .fetchHits( 20 );
```

Object fields need to have at least one sub-field with content in order to be considered as "existing".

Let’s consider the example above, and let’s assume the `placeOfBirth` object field only has one sub-field: `placeOfBirth.country`:

- an author whose `placeOfBirth` is null will not match.
- an author whose `placeOfBirth` is not null and has the `country` filled in will match.
- an author whose `placeOfBirth` is not null but does not have the `country` filled in will not match.

Because of this, it is preferable to use the `exists` predicate on object fields that are known to have at least one sub-field that is never null: an identifier, a name, ...

**9.2.9. wildcard: match a simple pattern**

**Example 130. Matching a simple pattern**

```java
List<Book> hits = searchSession.search( Book.class )
    .where( f -> f.wildcard().field( "description" )
        .matching( "rob*t" ) )
    .fetchHits( 20 );
```
If a normalizer has been defined on the field, the patterns used in wildcard predicates will be normalized.

If an analyzer has been defined on the field:

- when using the Elasticsearch backend, the patterns won’t be analyzed nor normalized, and will be expected to match a single indexed token, not a sequence of tokens.
- when using the Lucene backend the patterns will be normalized, but not tokenized: the pattern will still be expected to match a single indexed token, not a sequence of tokens.

For example, a pattern such as `Cat*` could match `cat` when targeting a field having a normalizer that applies a lowercase filter when indexing.

A pattern such as `john gr*` will not match anything when targeting a field that tokenizes on spaces. `gr*` may match, since it doesn’t include any space.

When the goal is to match user-provided query strings, the simple query string predicate should be preferred.

### 9.2.10. bool: combine predicates (or/and/…)

Example 131. Matching a document that matches any of multiple given predicates (~OR operator)

```java
List<Book> hits = searchSession.search( Book.class )
   .where( f -> f.bool()
         .should( f.match().field( "title" )
             .matching( "robot" ) ) ①
         .should( f.match().field( "description" )
             .matching( "investigation" ) ) ② )
   .fetchHits( 20 ); ③
```

① The hits should have a `title` field matching the text `robot`, or they should match any other clause in the same boolean predicate.

② The hits should have a `description` field matching the text `investigation`, or they should match any other clause in the same boolean predicate.

③ All returned hits will match at least one of the clauses above: they will have a `title` field matching the text `robot` or they will have a `description` field matching the text `investigation`. 

190
**Example 132. Matching a document that matches all of multiple given predicates (~AND operator)**

```
List<Book> hits = searchSession.search( Book.class )
   .where( f -> f.bool()
       .must( f.match().field( "title" )
           .matching( "robot" ) ) ①
       .must( f.match().field( "description" )
           .matching( "crime" ) ) ②
     )
   .fetchHits( 20 ); ③
```

① The hits must have a `title` field matching the text `robot`, independently from other clauses in the same boolean predicate.

② The hits must have a `description` field matching the text `crime`, independently from other clauses in the same boolean predicate.

③ All returned hits will match all of the clauses above: they will have a `title` field matching the text `robot` and they will have a description field matching the text `crime`.

**Example 133. Matching a document that does not match a given predicate**

```
List<Book> hits = searchSession.search( Book.class )
   .where( f -> f.bool()
       .must( f.match().field( "title" )
           .matching( "robot" ) ) ①
       .mustNot( f.match().field( "description" )
             .matching( "investigation" ) ) ②
     )
   .fetchHits( 20 ); ③
```

① The hits must have a `title` field matching the text `robot`, independently from other clauses in the same boolean predicate.

② The hits must not have a `description` field matching the text `investigation`, independently from other clauses in the same boolean predicate.

③ All returned hits will match all of the clauses above: they will have a `title` field matching the text `robot` and they will not have a `description` field matching the text `investigation`.

While it is possible to execute a boolean predicate with only "negative" clauses (mustNot), performance may be disappointing because the full power of indexes cannot be leveraged in that case.
Example 134. Matching a document that matches a given predicate without affecting the score

```java
List<Book> hits = searchSession.search(Book.class)
    .where( f -> f.bool() ①
        .should( f.bool() ②
            .filter( f.match().field("genre")
                .matching( Genre.SCIENCE_FICTION ) ) ③
            .must( f.match().fields("description")
                .matching( "crime" ) ) ④
        )
    .should( f.bool() ⑤
        .filter( f.match().field("genre")
            .matching( Genre.CRIME_FICTIO) ) ⑥
        .must( f.match().fields("description")
            .matching( "robot" ) ) ⑦
    )
    .fetchHits(20); ⑧
```

① Create a top-level boolean predicate, with two should clauses.

② In the first should clause, create a nested boolean predicate.

③ Use a filter clause to require documents to have the science-fiction genre, without taking this predicate into account when scoring.

④ Use a must clause to require documents with the science-fiction genre to have a title field matching crime, and take this predicate into account when scoring.

⑤ In the second should clause, create a nested boolean predicate.

⑥ Use a filter clause to require documents to have the crime fiction genre, without taking this predicate into account when scoring.

⑦ Use a must clause to require documents with the crime fiction genre to have a description field matching robot, and take this predicate into account when scoring.

⑧ The score of hits will ignore the filter clauses, leading to fairer sorts if there are much more "crime fiction" documents than "science-fiction" documents.
Example 135. Using optional **should** clauses to boost the score of some documents

```java
List<Book> hits = searchSession.search( Book.class )
    .where( f -> f.bool()
        .must( f.match().field( "title" )
            .matching( "robot" ) ) ①
        .should( f.match().field( "description" )
            .matching( "crime" ) ) ②
        .should( f.match().field( "description" )
            .matching( "investigation" ) ) ③
    )
    .fetchHits( 20 ); ④
```

① The hits **must** have a **title** field matching the text **robot**, independently from other clauses in the same boolean predicate.

② The hits **should** have a **description** field matching the text **crime**, but they may not, because matching the **must** clause above is enough. However, matching this **should** clause will improve the score of the document.

③ The hits **should** have a **description** field matching the text **investigation**, but they may not, because matching the **must** clause above is enough. However, matching this **should** clause will improve the score of the document.

④ All returned hits will match the **must** clause, and optionally the **should** clauses: they will have a **title** field matching the text **robot**, and the ones whose description matches either **crime** or **investigation** will have a better score.

Example 136. Fine-tuning **should** clauses matching requirements with **minimumShouldMatch**

```java
List<Book> hits = searchSession.search( Book.class )
    .where( f -> f.bool()
        .minimumShouldMatchNumber( 2 ) ①
        .should( f.match().field( "description" )
            .matching( "robot" ) ) ②
        .should( f.match().field( "description" )
            .matching( "investigation" ) ) ③
        .should( f.match().field( "description" )
            .matching( "disappearance" ) ) ④
    )
    .fetchHits( 20 ); ⑤
```

① At least two "should" clauses must match for this boolean predicate to match.

② The hits **should** have a **description** field matching the text **robot**.

③ The hits **should** have a **description** field matching the text **investigate**.

④ The hits **should** have a **description** field matching the text **crime**.

⑤ All returned hits will match at least two of the **should** clauses: their description will match either **robot** and **investigate**, **robot** and **crime**, **investigate** and **crime**, or all three of these terms.
Example 137. Easily adding clauses dynamically with the lambda syntax

```java
MySearchParameters searchParameters = getSearchParameters(); ①
List<Book> hits = searchSession.search( Book.class )
    .where( f -> f.bool(b -> ( ②
        b.must( f.matchAll() )); ③
        if ( searchParameters.getGenreFilter() != null ) ( ④
            b.must( f.match() .field( "genre" )
                .matching( searchParameters.getGenreFilter() ));
        )
        if ( searchParameters.getFullTextFilter() != null ) ( ⑤
            b.must( f.match() .fields( "title", "description" )
                .matching( searchParameters.getFullTextFilter() ));
        )
        if ( searchParameters.getPageCountMaxFilter() != null ) {
            b.must( f.range() .field( "pageCount" )
                .atMost( searchParameters.getPageCountMaxFilter() ));
        }
    ) .fetchHits( 20 ); ⑤
```

① Get a custom object holding the search parameters provided by the user through a web form, for example.

② Call `.bool(Consumer)`. The consumer, implemented by a lambda expression, will receive a builder as an argument and will add clauses to that builder as necessary.

③ By default, a boolean predicate will match nothing if there is no clause. To match every document when there is no clause, add a `must` clause that matches everything.

④ Inside the lambda, the code is free to check conditions before adding clauses. In this case, we only add clauses if the relevant parameter was filled in by the user.

⑤ The hits will match the clauses added by the lambda expression.

9.2.11. simpleQueryString: match a user-provided query

Example 138. Matching a simple query string: AND/OR operators

```java
List<Book> hits = searchSession.search( Book.class )
    .where( f -> f.simpleQueryString().field( "description" )
        .matching( "robots + (crime | investigation | disappearance)" ));
    .fetchHits( 20 );
```

Example 139. Matching a simple query string: NOT operator

```java
List<Book> hits = searchSession.search( Book.class )
    .where( f -> f.simpleQueryString().field( "description" )
        .matching( "robots + -investigation" ));
    .fetchHits( 20 );
```
Example 140. Matching a simple query string: AND as default operator

```java
List<Book> hits = searchSession.search( Book.class )
    .where( f -> f.simpleQueryString().field( "description" )
        .matching( "robots investigation" )
        .defaultOperator( BooleanOperator.AND ) )
    .fetchHits( 20 );
```

Example 141. Matching a simple query string: prefix

```java
List<Book> hits = searchSession.search( Book.class )
    .where( f -> f.simpleQueryString().field( "description" )
        .matching( "rob*" )
    ).fetchHits( 20 );
```

Example 142. Matching a simple query string: fuzzy

```java
List<Book> hits = searchSession.search( Book.class )
    .where( f -> f.simpleQueryString().field( "description" )
        .matching( "robo~2" )
    ).fetchHits( 20 );
```

Example 143. Matching a simple query string: phrase

```java
List<Book> hits = searchSession.search( Book.class )
    .where( f -> f.simpleQueryString().field( "title" )
        .matching( "\"robots of dawn\"" )
    ).fetchHits( 20 );
```

Example 144. Matching a simple query string: phrase with slop

```java
List<Book> hits = searchSession.search( Book.class )
    .where( f -> f.simpleQueryString().field( "title" )
        .matching( "\"dawn robot\"-3" )
    ).fetchHits( 20 );
```

Example 145. Matching a simple query string: enabling only specific syntax constructs

```java
List<Book> hits = searchSession.search( Book.class )
    .where( f -> f.simpleQueryString().field( "title" )
        .flags( SimpleQueryFlag.AND, SimpleQueryFlag.OR, SimpleQueryFlag.NOT )
    ).fetchHits( 20 );
```
9.2.12. **nested**: match nested documents

Example 146. Matching a simple pattern

```java
List<Book> hits = searchSession.search(Book.class)
    .where(f -> f.nested().objectField("authors")
        .nest(f.bool())
        .must(f.match().field("authors.firstName")
            .matching("isaac")
        )
        .must(f.match().field("authors.lastName")
            .matching("asimov")
        )
    )
    .fetchHits(20); ④
```

① Create a nested predicate on the authors object field.
② The author must have a first name matching isaac.
③ The author must have a last name matching asimov.
④ All returned hits will be books for which at least one author has a first name matching isaac and a last name matching asimov. Books that happen to have multiple authors, one of which has a first name matching isaac and another of which has a last name matching asimov, will not match.

Example 147. Use the implicit nested form

```java
List<Book> hits = searchSession.search(Book.class)
    .where(f -> f.bool())
        .must(f.match().field("authors.firstName")
            .matching("isaac")
        )
        .must(f.match().field("authors.lastName")
            .matching("asimov")
        )
    )
    .fetchHits(20); ④
```

① The nested predicate is created implicitly, since target fields here belong to a nested object.
② The author must have a first name matching isaac.
③ The author must have a last name matching asimov.
④ All returned hits will be books for which at least one author has a first name matching isaac and a last name matching asimov. Books that happen to have multiple authors, one of which has a first name matching isaac and another of which has a last name matching asimov, will match, because we apply the nested predicate separately to each match predicate.

9.2.13. **within**: match points within a circle, box, polygon
Example 148. Matching points within a circle

```java
GeoPoint center = GeoPoint.of(53.970000, 32.150000);
List<Author> hits = searchSession.search(Author.class)
    .where(f -> f.spatial().within().field("placeOfBirth.coordinates")
        .circle(center, 50, DistanceUnit.KILOMETERS))
    .fetchHits(20);
```

Example 149. Matching points within a box

```java
GeoBoundingBox box = GeoBoundingBox.of(53.99, 32.13, 53.95, 32.17);
List<Author> hits = searchSession.search(Author.class)
    .where(f -> f.spatial().within().field("placeOfBirth.coordinates")
        .boundingBox(box))
    .fetchHits(20);
```

Example 150. Matching points within a polygon

```java
GeoPolygon polygon = GeoPolygon.of(
    GeoPoint.of(53.976177, 32.138627),
    GeoPoint.of(53.986177, 32.148627),
    GeoPoint.of(53.979177, 32.168627),
    GeoPoint.of(53.876177, 32.159627),
    GeoPoint.of(53.956177, 32.155627),
    GeoPoint.of(53.976177, 32.138627)
);
List<Author> hits = searchSession.search(Author.class)
    .where(f -> f.spatial().within().field("placeOfBirth.coordinates")
        .polygon(polygon))
    .fetchHits(20);
```

9.2.14. More like this

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

9.2.15. Backend-specific extensions

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

Lucene: fromLuceneQuery
Example 151. Matching a native org.apache.lucene.search.Query

```java
List<Book> hits = searchSession.search( Book.class )
    .extension( LuceneExtension.get() )
    .where( f -> f.fromLuceneQuery( new RegexpQuery( new Term( "description", "neighbor|neighbour" ) ) )
    .fetchHits( 20 );
```

Elasticsearch: `fromJson`

Example 152. Matching a native Elasticsearch JSON query provided as a JsonObject

```java
JsonObject jsonObject = /* ... */;
List<Book> hits = searchSession.search( Book.class )
    .extension( ElasticsearchExtension.get() )
    .where( f -> f.fromJson( jsonObject ) )
    .fetchHits( 20 );
```

Example 153. Matching a native Elasticsearch JSON query provided as a JSON-formatted string

```java
List<Book> hits = searchSession.search( Book.class )
    .extension( ElasticsearchExtension.get() )
    .where( f -> f.fromJson( "{" + "\"regexp\": {" + "\"description\": \"neighbor|neighbour\"" + "\"}" + "}" ) )
    .fetchHits( 20 );
```

9.3. Sort DSL

9.3.1. Basics

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 154. Using custom sorts

```java
SearchSession searchSession = Search.session( entityManager );
List<Book> result = searchSession.search( Book.class )
    .where( f -> f.matchAll() )
    .sort( f -> f.field( "pageCount" ).desc() )
    .then().field( "title_sort" )
    .fetchHits( 20 );
```

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. Fetch the results, which will be sorted according to instructions.

Or alternatively, if you don't want to use lambdas:

Example 155. Using custom sorts – object-based syntax

```java
SearchSession searchSession = Search.session( entityManager );
SearchScope<Book> scope = searchSession.scope( Book.class );
List<Book> result = searchSession.search( scope )
    .where( scope.predicate().matchAll().toPredicate() )
    .sort( scope.sort() )
    .field( "pageCount" ).desc()
    .then().field( "title_sort" )
    .toSort()
    .fetchHits( 20 );
```

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 the following sections.

9.3.2. score: sort by matching score (relevance)
Example 156. Sorting by relevance

```
List<Book> hits = searchSession.search( Book.class )
  .where( f -> f.match().field( "title" )
          .matching( "robot dawn" )
          )
  .sort( f -> f.score() )
  .fetchHits( 20 );
```

9.3.3. **indexOrder**: sort according to the order of documents on storage

Example 157. Sorting according to the order of documents on storage

```
List<Book> hits = searchSession.search( Book.class )
  .where( f -> f.matchAll() )
  .sort( f -> f.indexOrder() )
  .fetchHits( 20 );
```

9.3.4. **field**: sort by field values

*field* will sort documents according to the value of a given field.

Example 158. Sorting by field values

```
List<Book> hits = searchSession.search( Book.class )
  .where( f -> f.matchAll() )
  .sort( f -> f.field( "title_sort" ) )
  .fetchHits( 20 );
```

Several options are available:

- The sort order is ascending by default, but can be **controlled explicitly with** `.asc()`/*.desc()`.

- The behavior on missing values can be **controlled explicitly with** `.missing()`.

- The behavior on multi-valued fields can be **controlled explicitly with** `.mode(...)`.

- For fields in nested objects, all nested objects are considered by default, but that can be **controlled explicitly with** `.filter(...)`.

9.3.5. **distance**: sort by distance to a point

*distance* will sort documents according to the distance between a given center and the point specified by a given field document field.
Example 159. Sorting by distance to a point

```java
GeoPoint center = GeoPoint.of(47.506060, 2.473916);
List<Author> hits = searchSession.search(Author.class)
    .where(f -> f.matchAll())
    .sort(f -> f.distance("placeOfBirth", center))
    .fetchHits(20);
```

Several options are available:

- The sort order is ascending by default, but can be controlled explicitly with `.asc()/.desc()`.
- The behavior on multi-valued fields can be controlled explicitly with `.mode(...)`.
- For fields in nested objects, all nested objects are considered by default, but that can be controlled explicitly with `.filter(...)`.

9.3.6. composite: combine sorts

Example 160. Sorting by multiple composed sorts using `composite()`

```java
List<Book> hits = searchSession.search(Book.class)
    .where(f -> f.matchAll())
    .sort(f -> f.composite() ①
        .add(f.field("genre_sort") ②
        .add(f.field("title_sort") ) ③)
    .fetchHits(20); ④
```

Example 161. Sorting by multiple composed sorts using `then()`

```java
List<Book> hits = searchSession.search(Book.class)
    .where(f -> f.matchAll())
    .sort(f -> f.field("genre_sort") ②
    .then().field("title_sort") ③
    .fetchHits(20); ④
```
Example 162. Easily composing sorts dynamically with the lambda syntax

```java
MySearchParameters searchParameters = getSearchParameters(); ➊
List<Book> hits = searchSession.search( Book.class )
    .where( f -> f.matchAll() )
    .sort( f -> f.compose( b -> { ➋
        for ( MySort mySort : searchParameters.getSorts() ) { ➌
            switch ( mySort.getType() ) {
                case GENRE:
                    b.add( f.field( "genre_sort" ).order( mySort.getOrder() ) );
                    break;
                case TITLE:
                    b.add( f.field( "title_sort" ).order( mySort.getOrder() ) );
                    break;
                case PAGE_COUNT:
                    b.add( f.field( "pageCount" ).order( mySort.getOrder() ) );
                    break;
            }
        }
    } )
    .fetchHits( 20 ); ➍
```

➊ Get a custom object holding the search parameters provided by the user through a web form, for example.

➋ Call `.composite(Consumer)`. The consumer, implemented by a lambda expression, will receive a builder as an argument and will add sorts to that builder as necessary.

➌ Inside the lambda, the code is free to do whatever is necessary before adding sorts. In this case, we iterate over user-selected sorts and add sorts accordingly.

➍ The hits will be sorted according to sorts added by the lambda expression.

9.3.7. Backend-specific extensions

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

**Lucene:** `fromLuceneSort`  

Example 163. Sorting by a native `org.apache.lucene.search.Sort`

```java
List<Book> hits = searchSession.search( Book.class )
    .extension( LuceneExtension.get() )
    .where( f -> f.matchAll() )
    .sort( f -> f.fromLuceneSort(  
        new Sort(  
            new SortedSetSortField( "genre_sort", false ),
            new SortedSetSortField( "title_sort", false )  
        )
    )
    .fetchHits( 20 );
```
### Lucene: `fromLuceneSortField`

#### Example 164. Sorting by a native `org.apache.lucene.search.SortField`

```java
List<Book> hits = searchSession.search(Book.class)
    .extension(LuceneExtension.get())
    .where(f -> f.matchAll())
    .sort(f -> f.fromLuceneSortField(new SortedSetSortField("title_sort", false)))
    .fetchHits(20);
```

### Elasticsearch: `fromJson`

#### Example 165. Sorting by a native Elasticsearch JSON sort provided as a `JsonObject`

```java
JsonObject jsonObject = /* ... */;
List<Book> hits = searchSession.search(Book.class)
    .extension(ElasticsearchExtension.get())
    .where(f -> f.matchAll())
    .sort(f -> f.fromJson(jsonObject))
    .fetchHits(20);
```

#### Example 166. Sorting by a native Elasticsearch JSON sort provided as a JSON-formatted string

```java
List<Book> hits = searchSession.search(Book.class)
    .extension(ElasticsearchExtension.get())
    .where(f -> f.matchAll())
    .sort(f -> f.fromJson("
        "{"title_sort": "asc"}
        
    ")
    .fetchHits(20);
```

### 9.3.8. Options common to multiple sort types

#### Sort order

Most sorts use the ascending order by default, with the notable exception of the `score sort`.

The order controlled explicitly through the following options:

- `.asc()` for an ascending order.
- `.desc()` for a descending order.
- `.order(...)` for an order defined by the given argument: `SortOrder.ASC/SortOrder.DESC`.

Below are a few examples with the `field sort`. 
Example 167. Sorting by field values in explicitly ascending order with `asc()`

```java
List<Book> hits = searchSession.search( Book.class )
  .where( f -> f.matchAll() )
  .sort( f -> f.field( "title_sort" ).asc() )
  .fetchHits( 20 );
```

Example 168. Sorting by field values in explicitly descending order with `desc()`

```java
List<Book> hits = searchSession.search( Book.class )
  .where( f -> f.matchAll() )
  .sort( f -> f.field( "title_sort" ).desc() )
  .fetchHits( 20 );
```

Example 169. Sorting by field values in explicitly descending order with `order(…)`

```java
List<Book> hits = searchSession.search( Book.class )
  .where( f -> f.matchAll() )
  .sort( f -> f.field( "title_sort" ).order( SortOrder.DESC ) )
  .fetchHits( 20 );
```

Missing values

Documents that do not have any value for a sort field will appear in the last position by default.

The behavior for missing values can be controlled explicitly through the `.missing()` option:

- `.missing().first()` puts documents with no value in first position (regardless of the sort order).
- `.missing().last()` puts documents with no value in last position (regardless of the sort order).
- `.missing().use(…)` uses the given value as a default for documents with no value.

Below are a few examples with the field sort.

Example 170. Sorting by field values, documents with no value in first position

```java
List<Book> hits = searchSession.search( Book.class )
  .where( f -> f.matchAll() )
  .sort( f -> f.field( "pageCount" ).missing().first() )
  .fetchHits( 20 );
```
Example 171. Sorting by field values, documents with no value in last position

```java
List<Book> hits = searchSession.search(Book.class)
    .where(f -> f.matchAll())
    .sort(f -> f.field("pageCount").missing().last())
    .fetchHits(20);
```

Example 172. Sorting by field values, documents with no value using a given default value

```java
List<Book> hits = searchSession.search(Book.class)
    .where(f -> f.matchAll())
    .sort(f -> f.field("pageCount").missing().use(300))
    .fetchHits(20);
```

Sort mode for multi-valued fields

Documents that have multiple values for a sort field can be sorted too. A single value is picked for each document in order to compare it with order documents. How the value is picked is called the sort mode, specified using the .mode(...) option. The following sort modes are available:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description</th>
<th>Supported value types</th>
<th>Unsupported value types</th>
</tr>
</thead>
<tbody>
<tr>
<td>SortMode.MIN</td>
<td>Picks the lowest value for field sorts, the lowest distance for distance sorts. This is default for ascending sorts.</td>
<td>All.</td>
<td>-</td>
</tr>
<tr>
<td>SortMode.MAX</td>
<td>Picks the highest value for field sorts, the highest distance for distance sorts. This is default for descending sorts.</td>
<td>All.</td>
<td>-</td>
</tr>
<tr>
<td>SortMode.SUM</td>
<td>Computes the sum of all values for each document, and picks that sum for comparison with other documents.</td>
<td>Numeric fields (long, ...)</td>
<td>Text and temporal fields (String, LocalDate, ...), distance.</td>
</tr>
<tr>
<td><strong>SortMode.AVG</strong></td>
<td>Computes the arithmetic mean of all values for each document and picks that average for comparison with other documents.</td>
<td>Numeric and temporal fields (long, LocalDate,...), distance.</td>
<td>Text fields (String,...).</td>
</tr>
<tr>
<td><strong>SortMode.MEDIAN</strong></td>
<td>Computes the median of all values for each document, and picks that median for comparison with other documents.</td>
<td>Numeric and temporal fields (long, LocalDate,...), distance.</td>
<td>Text fields (String,...).</td>
</tr>
</tbody>
</table>

Below is an example with the field sort.

**Example 173. Sorting by field values using the average value for each document**

```java
List<Author> hits = searchSession.search( Author.class )
   .where( f -> f.matchAll() )
   .sort( f -> f.field( "books.pageCount" )).mode( SortMode.AVG )
   .fetchHits( 20 );
```

**Filter for fields in nested objects**

When the sort field is located in a nested object, by default all nested objects will be considered for the sort and their values will be combined using the configured sort mode.

It is possible to filter the nested documents whose values will be considered for the sort using one of the `filter(...)` methods.

Below is an example with the field sort: authors are sorted by the average page count of their books, but only books of the "crime fiction" genre are considered:

**Example 174. Sorting by field values using a filter for nested objects**

```java
List<Author> hits = searchSession.search( Author.class )
   .where( f -> f.matchAll() )
   .sort( f -> f.field( "books.pageCount" )
      .mode( SortMode.AVG )
      .filter( pf -> pf.match().field( "books.genre" )
         .matching( Genre.CRIME_FICTION ) )
   ).fetchHits( 20 );
```
9.4. Projection DSL

9.4.1. Basics

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 175. Using projections to extract data from the index

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

List<String> result = searchSession.search( Book.class )
    .select( f -> f.field( "title", String.class ) )
    .where( f -> f.matchAll() )
    .fetchHits( 20 );
```

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. Fetch the results, which will have the expected type.

Or alternatively, if you don’t want to use lambdas:

Example 176. Using projections to extract data from the index — object-based syntax

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

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

List<String> result = searchSession.search( scope )
    .select( scope.projection().field( "title", String.class )
        .toProjection() )
    .where( scope.predicate().matchAll().toPredicate() )
    .fetchHits( 20 );
```

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 related to the search query itself (score,
To learn more about the field projection, and all the other types of projection, refer to the following sections.

9.4.2. Options common to multiple projection types

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

9.4.3. `documentReference`: return references to matched documents

**Example 177. Returning references to matched documents**

```java
List<DocumentReference> hits = searchSession.search(Book.class)
    .select(f -> f.documentReference())
    .where(f -> f.matchAll())
    .fetchHits(20);
```

9.4.4. `entityReference`: return references to matched entities

**Example 178. Returning references to matched entities**

```java
List<EntityReference> hits = searchSession.search(Book.class)
    .select(f -> f.entityReference())
    .where(f -> f.matchAll())
    .fetchHits(20);
```

9.4.5. `entity`: return matched entities loaded from the database

**Example 179. Returning matched entities loaded from the database**

```java
List<Book> hits = searchSession.search(Book.class)
    .select(f -> f.entity())
    .where(f -> f.matchAll())
    .fetchHits(20);
```

9.4.6. `field`: return field values from matched documents
Example 180. Returning field values from matched documents

```java
List<Genre> hits = searchSession.search(Book.class)
    .select(f -> f.field("genre", Genre.class))
    .where(f -> f.matchAll())
    .fetchHits(20);
```

Example 181. Returning field values from matched documents, without specifying the field type

```java
List<Object> hits = searchSession.search(Book.class)
    .select(f -> f.field("genre"))
    .where(f -> f.matchAll())
    .fetchHits(20);
```

Example 182. Returning field values from matched documents, without converting the field value

```java
List<String> hits = searchSession.search(Book.class)
    .select(f -> f.field("genre", String.class, ValueConvert.NO))
    .where(f -> f.matchAll())
    .fetchHits(20);
```

9.4.7. **score**: return the score of matched documents

Example 183. Returning the score of matched documents

```java
List<Float> hits = searchSession.search(Book.class)
    .select(f -> f.score())
    .where(f -> f.match().field("title")
    .matching("robot dawn")
    .fetchHits(20);
```

9.4.8. **distance**: return the distance to a point

Example 184. Returning the distance to a point

```java
GeoPoint center = GeoPoint.of(47.506060, 2.473916);
SearchResult<Double> result = searchSession.search(Author.class)
    .select(f -> f.distance("placeOfBirth", center))
    .where(f -> f.matchAll())
    .fetch(20); ③
```
Example 185. Returning the distance to a point with a given distance unit

```java
GeoPoint center = GeoPoint.of( 47.506060, 2.473916 );
SearchResult<Double> result = searchSession.search( Author.class )
    .select( f -> f.distance( "placeOfBirth", center )
               .unit( DistanceUnit.KILOMETERS ) )
    .where( f -> f.matchAll() )
    .fetch( 20 );
```

9.4.9. composite: combine projections

Example 186. Returning custom objects created from multiple projected values

```java
List<MyPair<String, Genre>> hits = searchSession.search( Book.class )
    .select( f -> f.composite( MyPair::new, f.field( "title", String.class ), f.field( "genre", Genre.class )
    ) )
    .where( f -> f.matchAll() )
    .fetchHits( 20 );
```

① Call `composite(...)`.  
② Use the constructor of a custom object, MyPair, as the combining function. The combining function can be a Function, a BiFunction, or a `org.hibernate.search.util.common.function.TriFunction`. It will combine values returned by other projections and create an object returned by the composite projection. Depending on the type of function, either one, two, or three additional arguments are expected.  
③ Define the first projection to combine as a projection on the title field, meaning the constructor of MyPair will be called for each matched document with the value of the title field as its first argument.  
④ Define the second projection to combine as a projection on the genre field, meaning the constructor of MyPair will be called for each matched document with the value of the genre field as its second argument.  
⑤ The hits will be the result of calling the combining function for each matched document, in this case MyPair instances.
Example 187. Returning a List of projected values

```java
List<List<?>> hits = searchSession.search( Book.class )
        .select( f -> f.composite( ①
                f.field( "title", String.class ), ②
                f.field( "genre", Genre.class ) ③
        )
        .where( f -> f.matchAll() )
        .fetchHits( 20 ); ④
```

① Call .composite(…).
② Define the first projection to combine as a projection on the title field, meaning the hits will be List instances with the value of the title field of the matched document at index 0.
③ Define the second projection to combine as a projection on the genre field, meaning the hits will be List instances with the value of the genre field of the matched document at index 1.
④ The hits will be List instances holding the result of the given projections, in order for each matched document.

9.4.10. Backend-specific extensions

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

Lucene: document

Example 188. Returning the matched document as a native org.apache.lucene.document.Document

```java
List<Document> hits = searchSession.search( Book.class )
        .extension( LuceneExtension.get() )
        .select( f -> f.document() )
        .where( f -> f.matchAll() )
        .fetchHits( 20 );
```

Lucene: explanation

Explanations are rather costly performance-wise: only use them for debugging purposes.
Example 189. Returning the score explanation as a native `org.apache.lucene.search.Explanation`

```
List<Explanation> hits = searchSession.search(Book.class)
    .extension(LuceneExtension.get())
    .select(f -> f.explanation())
    .where(f -> f.matchAll())
    .fetchHits(20);
```

**Elasticsearch:** `source`

Example 190. Returning the matched document source as a `JsonObject`

```
List<JsonObject> hits = searchSession.search(Book.class)
    .extension(ElasticsearchExtension.get())
    .select(f -> f.source())
    .where(f -> f.matchAll())
    .fetchHits(20);
```

**Elasticsearch:** `explanation`

Explanations are rather costly performance-wise: only use them for debugging purposes.

Example 191. Returning the score explanation as a `JsonObject`

```
List<JsonObject> hits = searchSession.search(Book.class)
    .extension(ElasticsearchExtension.get())
    .select(f -> f.explanation())
    .where(f -> f.matchAll())
    .fetchHits(20);
```

**Elasticsearch:** `jsonHit`

This is particularly useful when customizing the request’s JSON to ask for additional data within each hit.
Direct changes to the HTTP request may conflict with Hibernate Search features and be supported differently by different versions of Elasticsearch.

Similarly, the content of the HTTP response may change depending on the version of Elasticsearch, depending on which Hibernate Search features are used, and even depending on how Hibernate Search features are implemented.

Thus, features relying on direct access to HTTP requests or responses cannot be guaranteed to continue to work when upgrading Hibernate Search, even for micro upgrades ($x.y.z$ to $x.y.(z+1)$).

Use this at your own risk.

Example 192. Returning the Elasticsearch hit as a JsonObject

```
List<JsonObject> hits = searchSession.search( Book.class )
    .extension( ElasticsearchExtension.get() )
    .select( f -> f.jsonHit() )
    .where( f -> f.matchAll() )
    .fetchHits( 20 );
```

9.5. Aggregation DSL

9.5.1. Basics

Sometimes, you don’t just need to list query hits directly: you also need to group and aggregate the hits.

For example, almost any e-commerce website you can visit will have some sort of "faceting", which is a simple form of aggregation. In the "book search" webpage of an online bookshop, beside the list of matching books, you will find "facets", i.e. a count of matching documents in various categories. These categories can be taken directly from the indexed data, e.g. the genre of the book (science-fiction, crime fiction, ...), but also derived from the indexed data slightly, e.g. a price range ("less than $5", "less than $10", ...).

Aggregations allow just that (and, depending on the backend, much more): they allow the query to return "aggregated" hits.

Aggregations can be configured when building the search query:
Example 193. Defining an aggregation in a search query

```java
SearchSession searchSession = Search.session( entityManager );
AggregationKey<Map<Genre, Long>> countsByGenreKey = AggregationKey.of( "countsByGenre" );

SearchResult<Book> result = searchSession.search( Book.class )
    .where( f -> f.match().field( "title" )
        .matching( "robot" )
        .aggregation( countsByGenreKey, f -> f.terms() )
        .field( "genre", Genre.class )
    .fetch( 20 );

Map<Genre, Long> countsByGenre = result.aggregation( countsByGenreKey );
```

1. Define a key that will uniquely identify the aggregation. Make sure to give it the correct type (see <6>).
2. Start building the query as usual.
3. Define a predicate: the aggregation will only take into account documents matching this predicate.
4. Request an aggregation on the genre field, with a separate count for each genre: science-fiction, crime fiction, … If the field does not exist or cannot be aggregated, an exception will be thrown.
5. Fetch the results.
6. Retrieve the aggregation from the results as a Map, with the genre as key and the hit count as value of type Long.

Or alternatively, if you don’t want to use lambdas:

Example 194. Defining an aggregation in a search query – object-based syntax

```java
SearchSession searchSession = Search.session( entityManager );
SearchScope<Book> scope = searchSession.scope( Book.class );
AggregationKey<Map<Genre, Long>> countsByGenreKey = AggregationKey.of( "countsByGenre" );

SearchResult<Book> result = searchSession.search( scope )
    .where( scope.predicate().match().field( "title" )
        .matching( "robot" )
        .toPredicate() )
    .aggregation( countsByGenreKey, scope.aggregation().terms() )
    .field( "genre", Genre.class )
    .toAggregation() )
    .fetch( 20 );

Map<Genre, Long> countsByGenre = result.aggregation( countsByGenreKey );
```
There are a few constraints regarding aggregations. In particular, in order for a field to be "aggregable", it must be marked as such in the mapping, so that it is correctly stored in the index.

Faceting generally involves a concept of "drill-down", i.e. the ability to select a facet and restrict the hits to only those that match that facet.

Hibernate Search 5 used to offer a dedicated API to enable this "drill-down", but in Hibernate Search 6 you should simply create a new query with the appropriate predicate.

The aggregation DSL offers more aggregation types, and multiple options for each type of aggregation. To learn more about the terms aggregation, and all the other types of aggregations, refer to the following sections.

9.5.2. terms: group by the value of a field

Example 195. Counting hits grouped by the value of a field

```java
AggregationKey<Map<Genre, Long>> countsByGenreKey = AggregationKey.of( "countsByGenre" );
SearchResult<Book> result = searchSession.search( Book.class )
    .where( f -> f.matchAll() )
    .aggregation( countsByGenreKey, f -> f.terms()
        .field( "genre", Genre.class ) ) ①
    .fetch( 20 );
Map<Genre, Long> countsByGenre = result.aggregation( countsByGenreKey );
```

① Define the path and type of the field whose values should be considered.

Example 196. Counting hits grouped by the value of a field, without converting field values

```java
AggregationKey<Map<String, Long>> countsByGenreKey = AggregationKey.of( "countsByGenre" );
SearchResult<Book> result = searchSession.search( Book.class )
    .where( f -> f.matchAll() )
    .aggregation( countsByGenreKey, f -> f.terms()
        .field( "genre", String.class, ValueConvert.NO ) )
    .fetch( 20 );
Map<String, Long> countsByGenre = result.aggregation( countsByGenreKey );
```
Example 197. Setting the maximum number of returned entries in a `terms` aggregation

```java
AggregationKey<Map<Genre, Long>> countsByGenreKey = AggregationKey.of( "countsByGenre" );
SearchResult<Book> result = searchSession.search( Book.class )
  .where( f -> f.matchAll() )
  .aggregation( countsByGenreKey, f -> f.terms() )
    .field( "genre", Genre.class )
    .maxTermCount( 1 )
  .fetch( 20 );
Map<Genre, Long> countsByGenre = result.aggregation( countsByGenreKey );
```

Example 198. Including values from unmatched documents in a `terms` aggregation

```java
AggregationKey<Map<Genre, Long>> countsByGenreKey = AggregationKey.of( "countsByGenre" );
SearchResult<Book> result = searchSession.search( Book.class )
  .where( f -> f.matchAll() )
  .aggregation( countsByGenreKey, f -> f.terms() )
    .field( "genre", Genre.class )
    .minDocumentCount( 0 )
  .fetch( 20 );
Map<Genre, Long> countsByGenre = result.aggregation( countsByGenreKey );
```

Example 199. Excluding the rarest terms from a `terms` aggregation

```java
AggregationKey<Map<Genre, Long>> countsByGenreKey = AggregationKey.of( "countsByGenre" );
SearchResult<Book> result = searchSession.search( Book.class )
  .where( f -> f.matchAll() )
  .aggregation( countsByGenreKey, f -> f.terms() )
    .field( "genre", Genre.class )
    .minDocumentCount( 2 )
  .fetch( 20 );
Map<Genre, Long> countsByGenre = result.aggregation( countsByGenreKey );
```

With the Lucene backend, due to limitations of the current implementation, using any order other than the default one (by descending count) may lead to incorrect results. See HSEARCH-3666 for more information.

Example 200. Ordering entries by ascending value in a `terms` aggregation

```java
AggregationKey<Map<Genre, Long>> countsByGenreKey = AggregationKey.of( "countsByGenre" );
SearchResult<Book> result = searchSession.search( Book.class )
  .where( f -> f.matchAll() )
  .aggregation( countsByGenreKey, f -> f.terms() )
    .field( "genre", Genre.class )
    .orderByTermAscending()
  .fetch( 20 );
Map<Genre, Long> countsByGenre = result.aggregation( countsByGenreKey );
```
Example 201. Ordering entries by descending value in a \textit{terms} aggregation

```java
AggregationKey<
    Map<Genre, Long>
  > countsByGenreKey = AggregationKey.of("countsByGenre");
SearchResult<Book> result = searchSession.search(Book.class)
  .where( f -> f.matchAll() )
  .aggregation( countsByGenreKey, f -> f.terms() )
  .field( "genre", Genre.class )
  .orderByTermDescending() ;
Map<Genre, Long> countsByGenre = result.aggregation( countsByGenreKey );
```

Example 202. Ordering entries by ascending count in a \textit{terms} aggregation

```java
AggregationKey<
    Map<Genre, Long>
  > countsByGenreKey = AggregationKey.of("countsByGenre");
SearchResult<Book> result = searchSession.search(Book.class)
  .where( f -> f.matchAll() )
  .aggregation( countsByGenreKey, f -> f.terms() )
  .field( "genre", Genre.class )
  .orderByCountAscending() ;
Map<Genre, Long> countsByGenre = result.aggregation( countsByGenreKey );
```

When ordering entries by ascending count in a \textit{terms} aggregation, hit counts are approximate.

For fields in nested objects, all nested objects are considered by default, but that can be controlled explicitly with \texttt{.filter(...)}.

\subsection{9.5.3. \texttt{range}: grouped by ranges of values for a field}

Range aggregations are not available on String-based fields.
Example 203. Counting hits grouped by range of values for a field

```java
AggregationKey<
    Map<
        Range<Double>,
        Long
    >
>
countsByPriceKey = AggregationKey.of("countsByPrice" );

SearchResult<Book>
result = searchSession.search( Book.class )
    .where( f -> f.matchAll() )
    .aggregation( countsByPriceKey,
        f -> f.range() )
    .field( "price", Double.class )
    .range( 0.0, 10.0 )
    .range( 10.0, 20.0 )
    .range( 20.0, null )
)
    .fetch( 20 );

Map<
    Range<Double>,
    Long
>
countsByPrice = result.aggregation( countsByPriceKey );
```

① Define the path and type of the field whose values should be considered.

② Define the ranges to group hits into. The range can be passed directly as the lower bound (included) and upper bound (excluded). Other syntaxes exist to define different bound inclusion (see other examples below).

③ null means "to infinity".

Example 204. Counting hits grouped by range of values for a field – passing Range objects

```java
AggregationKey<
    Map<
        Range<Double>,
        Long
    >
>
countsByPriceKey = AggregationKey.of("countsByPrice" );

SearchResult<Book>
result = searchSession.search( Book.class )
    .where( f -> f.matchAll() )
    .aggregation( countsByPriceKey,
        f -> f.range() )
    .field( "price", Double.class )
    .range( Range.canonical( 0.0, 10.0 ) )
    .range( Range.between( 10.0, RangeBoundInclusion.INCLUDED, 20.0, RangeBoundInclusion.EXCLUDED ) )
    .range( Range.atLeast( 20.0 ) )
)
    .fetch( 20 );

Map<
    Range<Double>,
    Long
>
countsByPrice = result.aggregation( countsByPriceKey );
```

① With Range.of(Object, Object), the lower bound is included and the upper bound is excluded.

② Range.of(Object, RangeBoundInclusion, Object, RangeBoundInclusion) is more verbose, but allows setting the bound inclusion explicitly.

③ Range also offers multiple static methods to create ranges for a variety of use cases ("at least", "greater than", "at most", ...).
With the Elasticsearch backend, due to a limitation of Elasticsearch itself, all ranges must have their lower bound included (or \texttt{null}) and their upper bound excluded (or \texttt{null}). Otherwise, an exception will be thrown.

If you need to exclude the lower bound, or to include the upper bound, replace that bound with the immediate next value instead. For example with integers, \texttt{.range(0, 100)} means "0 (included) to 100 (excluded)". Call \texttt{.range(0, 101)} to mean "0 (included) to 100 (included)", or \texttt{.range(1, 100)} to mean "0 (excluded) to 100 (excluded)".

\textbf{Example 205. Counting hits grouped by range of values for a field, without converting field values}

```java
AggregationKey<Map<Range<Instant>, Long>> countsByPriceKey = AggregationKey.of("countsByPrice");
SearchResult<Book> result = searchSession.search(Book.class)
  .where(f -> f.matchAll())
  .aggregation(countsByPriceKey, f -> f.range() // Assuming "releaseDate" is of type "java.util.Date" or "java.sql.Date"
    .field("releaseDate", Instant.class, ValueConvert.NO)
    .range(null, LocalDateTime.of(1970, 1, 1))
    .atStartOfDay().toInstant(ZoneOffset.UTC)
    .range(LocalDateTime.of(1970, 1, 1), LocalDateTime.of(2000, 1, 1))
    .atStartOfDay().toInstant(ZoneOffset.UTC)
    .range(LocalDateTime.of(2000, 1, 1), null)
  )
  .fetch(20);
Map<Range<Instant>, Long> countsByPrice = result.aggregation(countsByPriceKey);
```

For fields in nested objects, all nested objects are considered by default, but that can be \texttt{controlled explicitly with .filter(...)}.

\textbf{9.5.4. Backend-specific extensions}

\textbf{Elasticsearch: fromJson}

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.
Example 206. Defining a native Elasticsearch JSON aggregation as a `JsonObject`

```
JsonObject jsonObject = /* ... */;
AggregationKey<JsonObject> countsByPriceHistogramKey = AggregationKey.of("countsByPriceHistogram");
SearchResult<Book> result = searchSession.search(Book.class)
  .extension(ElasticsearchExtension.get())
  .where(f -> f.matchAll())
  .aggregation(countsByPriceHistogramKey, f -> f.fromJson(jsonObject))
  .fetch(20);
JsonObject countsByPriceHistogram = result.aggregation(countsByPriceHistogramKey);
```

① The aggregation result is a `JsonObject`.

Example 207. Defining a native Elasticsearch JSON aggregation as a JSON-formatted string

```
AggregationKey<JsonObject> countsByPriceHistogramKey = AggregationKey.of("countsByPriceHistogram");
SearchResult<Book> result = searchSession.search(Book.class)
  .extension(ElasticsearchExtension.get())
  .where(f -> f.matchAll())
  .aggregation(countsByPriceHistogramKey, f -> f.fromJson("{""histogram": {
    "field": "price",
    "interval": 10
  }"})
  .fetch(20);
JsonObject countsByPriceHistogram = result.aggregation(countsByPriceHistogramKey);
```

① The aggregation result is a `JsonObject`.

9.5.5. Options common to multiple aggregation types

Filter for fields in nested objects

When the aggregation field is located in a nested object, by default all nested objects will be considered for the aggregation, and the document will be counted once for each value found in any nested object.

It is possible to filter the nested documents whose values will be considered for the aggregation using one of the `filter(...)` methods.

Below is an example with the range aggregation: the result of the aggregation is a count of books for each price range, with only the price of "paperback" editions being taken into account; the price of e-book editions, for example, is ignored.
9.6. Field types and compatibility

9.6.1. 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), atLeast(Object), ...`). Similarly, it is possible to pass an argument of type `Object` in the sort DSL when defining the behavior for missing values (`missing().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:

Example 209. Transparent conversion of DSL parameters

```java
List<AuthenticationEvent> result = searchSession.search(AuthenticationEvent.class)
  .where(f -> f.match().field("outcome")
   .matching(AuthenticationOutcome.INVALID_PASSWORD))
  .fetchHits(20);
```

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 `ValueConvert` parameter, and pass `ValueConvert.NO`:

Example 210. Disabling the DSL converter

```java
List<AuthenticationEvent> result = searchSession.search(AuthenticationEvent.class)
  .where(f -> f.match().field("outcome")
   .matching("Invalid password", ValueConvert.NO))
  .fetchHits(20);
```

All methods that apply DSL converters offer a variant that accepts a `ValueConvert` parameter: `matching, between, atLeast, atMost, greaterThan, lessThan, range, ...`

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

See Type bridge or Property bridge for more information.

9.6.2. 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 211. Transparent conversion of projections**

```java
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:```

```java
List<OrderStatus> result = searchSession.search(Order.class)
    .select(f -> f.field( "status", OrderStatus.class ) )
    .where( f -> f.matchAll() )
    .fetchHits(20);
```

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 `ValueConvert` parameter, and pass `ValueConvert.NO`:

**Example 212. Disabling the projection converter**

```java
List<String> result = searchSession.search(Order.class)
    .select(f -> f.field( "status", String.class, ValueConvert.NO ) )
    .where( f -> f.matchAll() )
    .fetchHits(20);
```
Projection converters must be configured explicitly in custom bridges. See Value bridges, Property bridge or Type bridge for more information.

### 9.6.3. Targeting multiple fields

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

- when multiple field names are passed to the `fields` 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.

### 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.

**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)/between(Object)/atLeast(Object)/greaterThan(Object)/etc. in the predicate DSL, `missing().use(Object)` in the sort DSL, `range(Object, Object)` in the aggregation 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`).

**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`).

**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 10. Lucene backend

10.1. Basic 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.

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`

10.2. Index storage (Directory)

The component responsible for index storage in Lucene is the `org.apache.lucene.store.Directory`. The implementation of the directory determines where the index will be stored: on the filesystem, in the JVM’s heap, ...

By default, the Lucene backend stores the indexes on the filesystem, in the JVM’s working directory.

The type of directory is set at the backend level:

```
hibernate.search.backends.<backend-name>.directory.type = local-filesystem
```

The following directory types are available:

- `local-filesystem`: Store the index on the local filesystem. See Local filesystem storage for details and configuration options.
- `local-heap`: Store the index in the local JVM heap. Local heap directories and all contained indexes are lost when the JVM shuts down. See Local heap storage for details and configuration options.

10.2.1. Local filesystem storage

The `local-filesystem` directory type will store each index in a subdirectory of a configured filesystem directory.
Local filesystem directories really are designed to be local to one server and one application.

In particular, they should not be shared between multiple Hibernate Search instances. Even if network shares allow to share the raw content of indexes, using the same index files from multiple Hibernate Search would require more than that: non-exclusive locking, routing of write requests from one node to another, ... These additional features are simply not available on local-filesystem directories.

If you need to share indexes between multiple Hibernate Search instances, the Elasticsearch backend will be a better choice. Refer to Architecture for more information.

Index location

Each index is assigned a subdirectory under a root directory.

By default, the root directory is the JVM’s working directory. It can be configured at the backend level:

```
hibernate.search.backends.<backend-name>.directory.root = /path/to/my/root
```

For example, with the configuration above, an entity of type `com.mycompany.Order` will be indexed in directory `/path/to/my/root/com.mycompany.Order/`. If that entity is explicitly assigned the index name "orders" (see `@Indexed(index = …)` in Entity/index mapping), it will instead be indexed in directory `/path/to/my/root/orders/`.

Filesystem access strategy

The default strategy for accessing the filesystem is determined automatically based on the operating system and architecture. It should work well in most situations.

For situations where a different filesystem access strategy is needed, Hibernate Search exposes a configuration property at the backend level:

```
hibernate.search.backends.<backend-name>.directory.filesystem_access.strategy = auto (default)
```

Allowed values are:

- **auto** (default): lets Lucene select the most appropriate implementation based on the operating system and architecture.

- **simple**: a straightforward strategy based on `Files.newByteChannel`. See `org.apache.lucene.store.SimpleFSDirectory`. 

228
• **mmap**: uses `mmap` for reading, and `FSDirectory.FSIndexOutput` for writing. See `org.apache.lucene.store.MMapDirectory`.

• **nio**: uses `java.nio.channels.FileChannel`'s positional read for concurrent reading, and `FSDirectory.FSIndexOutput` for writing. See `org.apache.lucene.store.NIOFSDirectory`.

Make sure to refer to Javadocs of these Directory implementations before changing this setting. Implementations offering better performance also bring issues of their own.

### Other configuration options

The **local-filesystem** directory also allows configuring a locking strategy.

#### 10.2.2. Local heap storage

The **local-heap** directory type will store indexes in the local JVM's heap.

As a result, indexes contained in a local-heap directory are **lost when the JVM shuts down**.

This directory type is only provided for use in testing configurations with small indexes and low concurrency, where it could slightly improve performance. In setups requiring larger indexes and/or high concurrency, a filesystem-based directory will achieve better performance.

The **local-heap** directory does not offer any specific option beyond the locking strategy.

#### 10.2.3. Locking strategy

In order to write to an index, Lucene needs to acquire a lock to ensure no other application instance writes to the same index concurrently. Each directory type comes with a default locking strategy that should work well enough in most situations.

For those (very) rare situations where a different locking strategy is needed, Hibernate Search exposes a configuration property at the backend level:

```
hibernate.search.backends.<backend-name>.directory.locking.strategy = native-filesystem
```

The following strategies are available:

• **simple-filesystem**: Locks the index by creating a marker file and checking it before write operations. This implementation is very simple and based Java's File API. If for some reason an application ends abruptly, the marker file will stay on the filesystem and will need to be removed manually.
This strategy is only available for filesystem-based directories.

See org.apache.lucene.store.SimpleFSLockFactory.

- **native-filesystem**: Similarly to `simple-filesystem`, locks the index by creating a marker file, but using native OS file locks instead of Java’s File API, so that locks will be cleaned up if the application ends abruptly.

  This is the default strategy for the `local-filesystem` directory type.

  This implementation has known problems with NFS: it should be avoided on network shares.

  This strategy is only available for filesystem-based directories.

  See org.apache.lucene.store.NativeFSLockFactory.

- **single-instance**: Locks using a Java object held in the JVM’s heap. Since the lock is only accessible by the same JVM, this strategy will only work properly when it is known that only a single application will ever try to accesses the indexes.

  This is the default strategy for the `local-heap` directory type.


- **none**: Does not use any lock. Concurrent writes from another application will result in index corruption. Test your application carefully and make sure you know what it means.

  See org.apache.lucene.store.NoLockFactory.

10.3. Sharding

For a preliminary introduction to sharding, including how it works in Hibernate Search and what its limitations are, see Sharding and routing.

In the Lucene backend, sharding is disabled by default, but can be enabled by selecting a sharding strategy at the index level. Multiple strategies are available:

**hash**

```java
hibernate.search.backends.<backend name>.indexes.<index name>.sharding.strategy = hash
hibernate.search.backends.<backend name>.indexes.<index name>.sharding.number_of_shards = 2 (no default)
# OR
hibernate.search.backends.<backend name>.index_defaults.sharding.strategy = hash
hibernate.search.backends.<backend name>.index_defaults.sharding.number_of_shards = 2 (no default)
```
The hash strategy requires to set a number of shards through the `number_of_shards` property.

This strategy will set up an explicitly configured number of shards, numbered from 0 to the chosen number minus one (e.g. for 2 shards, there will be shard "0" and shard "1").

When routing, the routing key will be hashed to assign it to a shard. If the routing key is null, the document ID will be used instead.

This strategy is suitable when there is no explicit routing key configured in the mapping, or when the routing key has a large number of possible values that need to be brought down to a smaller number (e.g. "all integers").

```
explicit
```

```
hibernate.search.backends.<backend name>.indexes.<index name>.sharding.strategy = explicit
hibernate.search.backends.<backend name>.indexes.<index name>.sharding.shard_identifiers = fr,en,de (no default)
# OR
hibernate.search.backends.<backend name>.index_defaults.sharding.strategy = explicit
hibernate.search.backends.<backend name>.index_defaults.sharding.shard_identifiers = fr,en,de (no default)
```

The explicit strategy requires to set a list of shard identifiers through the `shard_identifiers` property. The identifiers must be provided as a String containing multiple shard identifiers separated by commas, or a `Collection<String>` containing shard identifiers. A shard identifier can be any string.

This strategy will set up one shard per configured shard identifier.

When routing, the routing key will be validated to make sure it matches a shard identifier exactly. If it does, the document will be routed to that shard. If it does not, an exception will be thrown. The routing key cannot be null, and the document ID will be ignored.

This strategy is suitable when there is an explicit routing key configured in the mapping, and that routing key has a limited number of possible values that are known before starting the application.

### 10.4. 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_8_1_1
```

Depending on the specific version of Lucene you're using, you might have different options available:
see `org.apache.lucene.util.Version` contained in `lucene-core.jar` for a list of allowed
values.

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.

10.5. Schema

Lucene does not really have a concept of centralized schema to specify the data type and capabilities
of each field, but Hibernate Search maintains such a schema in memory, in order to remember which
predicates/projections/sorts can be applied to each field.

For the most part, the schema is inferred from the mapping configured through Hibernate Search’s
mapping APIs, which are generic and independent from Elasticsearch.

Aspects that are specific to the Lucene backend are explained in this section.

10.5.1. Field types

Available 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 Supported property types for more information.

Field types that are not in this list can still be used with a little bit more work:

- If a property in the entity model has an unsupported type, but can be converted to a supported type, you will need a bridge. See Bridges.
- If you need an index field with a specific type that is not supported by Hibernate Search, you will need a bridge that defines a native field type. See Index field type DSL extensions.

<table>
<thead>
<tr>
<th>Field type</th>
<th>Limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.String</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.Byte</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.Short</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.Integer</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.Long</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.Double</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.Float</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.Boolean</td>
<td>-</td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td>-</td>
</tr>
<tr>
<td>java.math.BigInteger</td>
<td>-</td>
</tr>
<tr>
<td>java.time.Instant</td>
<td>Lower range/resolution</td>
</tr>
<tr>
<td>java.time.LocalDate</td>
<td>Lower range/resolution</td>
</tr>
<tr>
<td>java.time.LocalTime</td>
<td>Lower range/resolution</td>
</tr>
<tr>
<td>java.time.LocalDateTime</td>
<td>Lower range/resolution</td>
</tr>
<tr>
<td>java.time.ZonedDateTime</td>
<td>Lower range/resolution</td>
</tr>
<tr>
<td>java.time.OffsetDateTime</td>
<td>Lower range/resolution</td>
</tr>
<tr>
<td>java.time.OffsetTime</td>
<td>Lower range/resolution</td>
</tr>
<tr>
<td>Field type</td>
<td>Limitations</td>
</tr>
<tr>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>java.time.Year</td>
<td>Lower range/resolution</td>
</tr>
<tr>
<td>java.time.YearMonth</td>
<td>Lower range/resolution</td>
</tr>
<tr>
<td>java.time.MonthDay</td>
<td>-</td>
</tr>
<tr>
<td>org.hibernate.search.engine.spatial.GeoPoint</td>
<td>Lower resolution</td>
</tr>
</tbody>
</table>

**Range and resolution of date/time fields**

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 \(292.278.993\).

Values that are out of range will trigger indexing failures.

Resolution for time types is also lower:

- `java.time` supports nanosecond-resolution.
- The Lucene backend supports millisecond-resolution.

Precision beyond the millisecond will be lost when indexing.

**Range and resolution of GeoPoint fields**

GeoPoints are indexed as LatLonPoints in the Lucene backend. According to `LatLonPoint`'s javadoc, there is a loss of precision when the values are encoded:

Values are indexed with some loss of precision from the original `double` values \(4.190951585769653\times10^{-8}\) for the latitude component and \(8.381903171539307\times10^{-8}\) for longitude).

This effectively means indexed points can be off by about 13 centimeters (5.2 inches) in the worst case.

**Index field type DSL extensions**

Not all Lucene field types have built-in support in Hibernate Search. Unsupported field types can still be used, however, by taking advantage of the "native" field type. Using this field type, Lucene `IndexableField` instances can be created directly, giving access to everything Lucene can offer.
Below is an example of how to use the Lucene "native" type.

**Example 213. Using the Lucene "native" type**

```java
public class PageRankValueBinder implements ValueBinder {  
    @Override
    public void bind(ValueBindingContext<?> context) {
        context.bridge(
            Float.class,
            new PageRankValueBridge(),
            context.typeFactory().extension(LuceneExtension.get()).asNative(Float.class),
            "pageRank", value -> {
                collector.accept(new FeatureField(absoluteFieldPath, "pageRank", value));
                collector.accept(new StoredField(absoluteFieldPath, value));
                field -> (Float) field.numericValue()
            },
            field -> Float.class)
        private static class PageRankValueBridge implements ValueBridge<Float, Float> {
            @Override
            public Float toIndexedValue(Float value, ValueBridgeToIndexedValueContext context) {
                return value;
            }
            @Override
            public Float fromIndexedValue(Float value, ValueBridgeFromIndexedValueContext context) {
                return value;
            }
        }
    }
}
```

1. Define a custom binder and its bridge. The "native" type can only be used from a binder, it cannot be used directly with annotation mapping. Here we're defining a value binder, but a type binder, or a property binder would work as well.

2. Get the context’s type factory.

3. Apply the Lucene extension to the type factory.

4. Call `asNative` to start defining a native type.

5. Define the field value type.

6. Define the LuceneFieldContributor. The contributor will be called upon indexing to add as many fields as necessary to the document. All fields must be named after the `absoluteFieldPath` passed to the contributor.

7. Optionally, if projections are necessary, define the LuceneFieldValueExtractor. The extractor will be called upon projecting to extract the projected value from a single stored field.

8. The value bridge is free to apply a preliminary conversion before passing the value to Hibernate.
Search, which will pass it along to the `LuceneFieldContributor`.

```java
@Entity
@Indexed
public class WebPage {
    @Id
    private Integer id;

    @NonStandardField(①)
    valueBinder = @ValueBinderRef(type = PageRankValueBinder.class) ②
    private Float pageRank;
    // Getters and setters
    // ...
}
```

① Map the property to an index field. Note that value bridges using a non-standard field type (such as Lucene's "native" type) must be mapped using the `@NonStandardField` annotation: other annotations such as `@GenericField` will fail.

② Instruct Hibernate Search to use our custom value binder.

### 10.5.2. 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, a strategy must be selected in order to enable multi-tenancy.

The multi-tenancy strategy is set a the backend level:

```java
hibernate.search.backends.<backend name>.multi_tenancy.strategy = none (default)
```

See the following subsections for details about available strategies.

**none**: single-tenancy

The `none` strategy (the default) disables multi-tenancy completely.

Attempting to set a tenant ID will lead to a failure when indexing.
discriminator: type name mapping using the index name

With the discriminator strategy, all documents from all tenants are stored in the same index.

When indexing, a discriminator field holding the tenant ID is populated transparently for each document.

When searching, a filter targeting the tenant ID field is added transparently to the search query to only return search hits for the current tenant.

10.6. Analysis

10.6.1. Basics

Analysis is the text processing performed by analyzers, both when indexing (document processing) and when searching (query processing).

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

- Define a class that implements the org.hibernate.search.backend.lucene.analysis.LuceneAnalysisConfigurer interface.

  - Configure the backend to use that implementation by setting the configuration property hibernate.search.backends.<backend name>.analysis.configurer to a bean reference pointing to the implementation.

Hibernate Search will call the configure method of this implementation on startup, and the configurer will be able to take advantage of a DSL to define analyzers and normalizers or even (for more advanced use) the similarity. See below for examples.

10.6.2. Analyzers and normalizers

The context passed to the configurer exposes a DSL to define analyzers and normalizers:

Example 214. Implementing and using an analysis configurer to define analyzers and normalizers with the Lucene backend
package org.hibernate.search.documentation.analysis;

import org.hibernate.search.backend.lucene.analysis.LuceneAnalysisConfigurationContext;
import org.hibernate.search.backend.lucene.analysis.LuceneAnalysisConfigurer;
import org.apache.lucene.analysis.charfilter.HTMLStripCharFilterFactory;
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(LuceneAnalysisConfigurationContext context) {
        context.analyzer("english").custom()
            .tokenizer(StandardTokenizerFactory.class)
            .charFilter(HTMLStripCharFilterFactory.class)
            .tokenFilter(LowerCaseFilterFactory.class)
            .tokenFilter(SnowballPorterFilterFactory.class)
            .param("language", "English")
            .tokenFilter(ASCIIFoldingFilterFactory.class);

        context.normalizer("lowercase").custom()
            .tokenizer(LowerCaseFilterFactory.class)
            .tokenFilter(ASCIIFoldingFilterFactory.class);

        context.analyzer("french").custom()
            .tokenizer(StandardTokenizerFactory.class)
            .charFilter(HTMLStripCharFilterFactory.class)
            .tokenFilter(LowerCaseFilterFactory.class)
            .tokenFilter(SnowballPorterFilterFactory.class)
            .param("language", "French")
            .tokenFilter(ASCIIFoldingFilterFactory.class);
    }
}

① Define a custom analyzer named "english", because it will be used to analyze English text such as book titles.
② Set the tokenizer to a standard tokenizer: components are referenced by their factory class.
③ Set the char filters. Char filters are applied in the order they are given, before the tokenizer.
④ Set the token filters. Token filters are applied in the order they are given, after the tokenizer.
⑤ Set the value of a parameter for the last added char filter/tokenizer/token filter.
⑥ Normalizers are defined in a similar way, the only difference being that they cannot use a tokenizer.
⑦ Multiple analyzers/normalizers can be defined in the same configurer.

① hibernate.search.backends.myBackend.analysis.configurer
    org.hibernate.search.documentation.analysis.MyLuceneAnalysisConfigurer

① Assign the configurer to the backend myBackend using a Hibernate Search configuration property.
It is also possible to assign a name to a built-in analyzer, or a custom analyzer implementation:

**Example 215. Naming an analyzer instance in the Lucene backend**

```java
context.analyzer( "standard" ).instance( new StandardAnalyzer() );
```

To know which analyzers, character filters, tokenizers and token filters are available, either browse the Lucene Javadoc or read the corresponding section on the Solr Wiki (you don’t need Solr to use these analyzers, it’s just that there is no documentation page for Lucene proper).

### 10.6.3. Similarity

When searching, scores are assigned to documents based on statistics recorded at index time, using a specific formula. Those statistics and the formula are defined by a single component called the similarity, implementing Lucene’s `org.apache.lucene.search.similarities.Similarity` interface.

By default, Hibernate Search uses `BM25Similarity` with its defaults parameters ($k_1 = 1.2, b = 0.75$). This should provide satisfying scoring in most situations.

If you have advanced needs, you can set a custom `Similarity` in your analysis configurer, as shown below.

**Example 216. Implementing an analysis configurer to change the Similarity with the Lucene backend**

```java
public class CustomSimilarityLuceneAnalysisConfigurer implements LuceneAnalysisConfigurer {
    @Override
    public void configure(LuceneAnalysisConfigurationContext context) {
        context.similarity( new ClassicSimilarity() ); ①
        context.analyzer( "english" ).custom() ②
c .tokenizer( StandardTokenizerFactory.class )
c .tokenFilter( LowerCaseFilterFactory.class )
c .tokenFilter( ASCIIFoldingFilterFactory.class );
    }
}
```

① Set the similarity to `ClassicSimilarity`.

② Define analyzers and normalizers as usual.
For more information about Similarity, its various implementations, and the pros and cons of each implementation, see the javadoc of Similarity and Lucene’s source code.

You can also find useful resources on the web, for example in Elasticsearch’s documentation.

10.7. Threads

The Lucene backend relies on an internal thread pool to execute write operations on the index.

By default, the pool contains exactly as many threads as the number of processors available to the JVM on bootstrap. That can be changed using a configuration property:

```
hibernate.search.backends.<backend-name>.thread_pool.size = 4
```

This number is *per backend*, not per index. Adding more indexes will not add more threads.

Operations happening in this thread-pool include blocking I/O, so raising its size above the number of processor cores available to the JVM can make sense and may improve performance.

10.8. Indexing queues

Among all the write operations performed by Hibernate Search on the indexes, it is expected that there will be a lot of "indexing" operations to create/update/delete a specific document. We generally want to preserve the relative order of these requests when they are about the same documents.

For this reasons, Hibernate Search pushes these operations to internal queues and applies them in batches. Each index maintains 10 queues holding at most 1000 elements each. Queues operate independently (in parallel), but each queue applies one operation after the other, so at any given time there can be at most 10 batches of indexing requests being applied for each index.

Indexing operations relative to the same document ID are always pushed to the same queue.

It is possible to customize the queues in order to reduce resource consumption, or on the contrary to improve throughput. This is done through the following configuration properties, at the index level:
• `indexing.queue_count` defines the number of queues. Expects a strictly positive integer value.

Higher values will lead to more indexing operations being performed in parallel, which may lead to higher indexing throughput if CPU power is the bottleneck when indexing.

Note that raising this number above the number of threads is never useful, as the number of threads limits how many queues can be processed in parallel.

• `indexing.queue_size` defines the maximum number of elements each queue can hold. Expects a strictly positive integer value.

Lower values may lead to lower memory usage, especially if there are many queues, but values that are too low will increase the likeliness of application threads blocking because the queue is full, which may lead to lower indexing throughput.

When a queue is full, any attempt to request indexing will block until the request can be put into the queue.

In order to achieve a reasonable level of performance, be sure to set the size of queues to a high enough number that this kind of blocking only happens when the application is under very high load.

When sharding is enabled, each shard will be assigned its own set of queues.

If you use the hash sharding strategy based on the document ID (and not based on a provided routing key), make sure to set the number of queues to a number with no common denominator with the number of shards; otherwise, some queues may be used much less than others.

For example, if you set the number of shards to 8 and the number of queues to 4, documents ending up in the shard #0 will always end up in queue #0 of that shard. That’s because both the routing to a shard and the routing to a queue take the hash of the document ID then apply a modulo operation to that hash, and `<some hash> % 8 == 0` (routed to shard #0) implies `<some hash> % 4 == 0` (routed to queue #0 of shard #0). Again, this is only true if you rely on the document ID and not on a provided routing key for sharding.
10.9. Writing and reading

10.9.1. Commit

For a preliminary introduction to writing to and reading from indexes in Hibernate Search, including in particular the concepts of commit and refresh, see Commit and refresh.

In Lucene terminology, a commit is when changes buffered in an index writer are pushed to the index itself, so that a crash or power loss will no longer result in data loss.

Some operations are critical and are always committed before they are considered complete. This is the case for changes triggered by automatic indexing (unless configured otherwise), and also for large-scale operations such as a purge. When such an operation is encountered, a commit will be performed immediately, guaranteeing that the operation is only considered complete after all changes are safely stored on disk.

Other operations, however, are not expected to be committed immediately. This is the case for changes contributed by the mass indexer, or by automatic indexing when using the async synchronization strategy.

Performance-wise, committing may be an expensive operation, which is why Hibernate Search tries not to commit too often. By default, when changes that do not require an immediate commit are applied to the index, Hibernate Search will delay the commit by one second. If other changes are applied during that second, they will be included in the same commit. This dramatically reduces the amount of commits in write-intensive scenarios (e.g. mass indexing), leading to much better performance.

It is possible to control exactly how often Hibernate Search will commit by setting the commit interval (in milliseconds) at the index level:

```properties
hibernate.search.backends.<backend name>.indexes.<index name>.io.commit_interval = 1000 (default)
# OR
hibernate.search.backends.<backend name>.index_defaults.io.commit_interval = 1000 (default)
```

Setting the commit interval to 0 will force Hibernate Search to commit after every batch of changes, which may result in a much lower throughput, both for automatic indexing and mass indexing.
Remember that individual write operations may force a commit, which may cancel out the potential performance gains from setting a higher commit interval.

By default, the commit interval may only improve throughput of the mass indexer. If you want changes triggered by automatic indexing to benefit from it too, you will need to select a non-default synchronization strategy, so as not to require a commit after each change.

### 10.9.2. Refresh

For a preliminary introduction to writing to and reading from indexes in Hibernate Search, including in particular the concepts of commit and refresh, see Commit and refresh.

In Lucene terminology, a refresh is when a new index reader is opened, so that the next search queries will take into account the latest changes to the index.

Performance-wise, refreshing may be an expensive operation, which is why Hibernate Search tries not to refresh too often. The index reader is refreshed upon every search query, but only if writes have occurred since the last refresh.

In write-intensive scenarios where refreshing after each write is still too frequent, it is possible to refresh less frequently and thus improve read throughput by setting a refresh interval in milliseconds. When set to a value higher than 0, the index reader will no longer be refreshed upon every search query: if, when a search query starts, the refresh occurred less than X milliseconds ago, then the index reader will not be refreshed, even though it may be out-of-date.

The refresh interval is set at the index level:

```
hibernate.search.backends.<backend name>.indexes.<index name>.io.refresh_interval = 0 (default)
# OR
hibernate.search.backends.<backend name>.index_defaults.io.refresh_interval = 0 (default)
```

### 10.9.3. IndexWriter settings

Lucene’s IndexWriter, used by Hibernate Search to write to indexes, exposes several settings that can be tweaked to better fit your application, and ultimately get better performance.

Hibernate Search exposes these settings through configuration properties prefixed with `io.writer`, at the index level.

Below is a list of all index writer settings. They can be set through configuration properties, at the index level. For example, `io.writer.ram_buffer_size` can be set like this:
<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[...].io.writer.max_buffered_docs</code></td>
<td>The maximum number of documents that can be buffered in-memory before they are flushed to the Directory. Large values mean faster indexing, but more RAM usage. When used together with <code>ram_buffer_size</code> a flush occurs for whichever event happens first.</td>
</tr>
<tr>
<td><code>[...].io.writer.ram_buffer_size</code></td>
<td>The maximum amount of RAM that may be used for buffering added documents and deletions before they are flushed to the Directory. Large values mean faster indexing, but more RAM usage. Generally for faster indexing performance it’s best to use this setting rather than <code>max_buffered_docs</code>. When used together with <code>max_buffered_docs</code> a flush occurs for whichever event happens first.</td>
</tr>
<tr>
<td><code>[...].io.writer.infostream</code></td>
<td>Enables low level trace information about Lucene’s internal components; <code>true</code> or <code>false</code>. Logs will be appended to the logger <code>org.hibernate.search.backend.lucene.infostream</code> at the TRACE level. This may cause significant performance degradation, even if the logger ignores the TRACE level, so this should only be used for troubleshooting purposes. Disabled by default.</td>
</tr>
</tbody>
</table>

Refer to Lucene’s documentation, in particular the javadoc and source code of `IndexWriterConfig`, for more information about the settings and their defaults.
10.9.4. Merge settings

A Lucene index is not stored in a single, continuous file. Instead, each flush to the index will generate a small file containing all the documents added to the index. This file is called a "segment". Search can be slower on an index with too many segments, so Lucene regularly merges small segments to create fewer, larger segments.

Lucene’s merge behavior is controlled through a MergePolicy. Hibernate Search uses the LogByteSizeMergePolicy, which exposes several settings that can be tweaked to better fit your application, and ultimately get better performance.

Below is a list of all merge settings. They can be set through configuration properties, at the index level. For example, `io.merge.factor` can be set like this:

```
hibernate.search.backends.<backend name>.indexes.<index name>.io.merge.factor = 10
# OR
hibernate.search.backends.<backend name>.index_defaults.io.merge.factor = 10
```

Table 8. Configuration properties related to merges

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[...].io.merge.max_docs</td>
<td>The maximum number of documents that a segment can have before merging. Segments with more than this number of documents will not be merged.</td>
</tr>
<tr>
<td></td>
<td>Smaller values perform better on frequently changing indexes, larger values provide better search performance if the index does not change often.</td>
</tr>
<tr>
<td>[...].io.merge.factor</td>
<td>The number of segments that are merged at once.</td>
</tr>
<tr>
<td></td>
<td>With smaller values, merging happens more often and thus uses more resources, but the total number of segments will be lower on average, increasing read performance. Thus, larger values (&gt; 10) are best for mass indexing, and smaller values (≤ 10) are best for automatic indexing.</td>
</tr>
<tr>
<td></td>
<td>The value must not be lower than 2.</td>
</tr>
<tr>
<td>Property</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>[...].io.merge.min_size</td>
<td>The minimum target size of segments, in MB, for background merges. Segments smaller than this size are merged more aggressively. Setting this too large might result in expensive merge operations, even tough they are less frequent.</td>
</tr>
<tr>
<td>[...].io.merge.max_size</td>
<td>The maximum size of segments, in MB, for background merges. Segments larger than this size are never merged in the background. Settings this to a lower value helps reduce memory requirements and avoids some merging operations at the cost of optimal search speed. When forcefully merging an index, this value is ignored and max_forced_size is used instead (see below).</td>
</tr>
<tr>
<td>[...].io.merge.max_forced_size</td>
<td>The maximum size of segments, in MB, for forced merges. This is the equivalent of io.merge.max_size for forceful merges. You will generally want to set this to the same value as max_size or lower, but setting it too low will degrade search performance as documents are deleted.</td>
</tr>
<tr>
<td>[...].io.merge.calibrate_by_deletes</td>
<td>Whether the number of deleted documents in an index should be taken into account; true or false. When enabled, Lucene will consider that a segment with 100 documents, 50 of which are deleted, actually contains 50 documents. When disabled, Lucene will consider that such a segment contains 100 documents. Setting calibrate_by_deletes to false will lead to more frequent merges caused by io.merge.max_docs, but will more aggressively merge segments with many deleted documents, improving search performance.</td>
</tr>
</tbody>
</table>
Refer to Lucene’s documentation, in particular the javadoc and source code of `LogByteSizeMergePolicy`, for more information about the settings and their defaults.

The options `io.merge.max_size` and `io.merge.max_forced_size` do not directly define the maximum size of all segment files.

First, consider that merging a segment is about adding it together with another existing segment to form a larger one. `io.merge.max_size` is the maximum size of segments before merging, so newly merged segments can be up to twice that size.

Second, merge options do not affect the size of segments initially created by the index writer, before they are merged. This size can be limited with the setting `io.writer.ram_buffer_size`, but Lucene relies on estimates to implement this limit; when these estimates are off, it is possible for newly created segments to be slightly larger than `io.writer.ram_buffer_size`.

So, for example, to be fairly confident no file grows larger than 15MB, use something like this:

```
hibernate.search.backends.<backend name>.index_defaults.io.writer.ram_buffer_size = 10
hibernate.search.backends.<backend name>.index_defaults.io.merge.max_size = 7
hibernate.search.backends.<backend name>.index_defaults.io.merge.max_forced_size = 7
```

### 10.10. Retrieving analyzers and normalizers

Lucene analyzers and normalizers defined in Hibernate Search can be retrieved from the Lucene backend.
Example 217. Retrieving the Lucene analyzers by name from the backend

```
SearchMapping mapping = Search.mapping( entityManagerFactory ); ①
Backend backend = mapping.backend( "myBackend" ); ②
LuceneBackend luceneBackend = backend.unwrap( LuceneBackend.class ); ③
Optional<? extends Analyzer> analyzer = luceneBackend.analyzer( "english" ); ④
Optional<? extends Analyzer> normalizer = luceneBackend.normalizer( "isbn" ); ⑤
```

1. Retrieve the `SearchMapping`.
2. Retrieve the `Backend`.
3. Narrow down the backend to the `LuceneBackend` type.
4. Get an analyzer by name. The method returns an `Optional`, which is empty if the analyzer does not exist. The analyzer must have been defined in Hibernate Search, otherwise it won't exist.
5. Get a normalizer by name. The method returns an `Optional`, which is empty if the normalizer does not exist. The normalizer must have been defined in Hibernate Search, otherwise it won't exist.

Alternatively, you can also retrieve the (composite) analyzers for a whole index. These analyzers behaves differently for each field, delegating to the analyzer configured in the mapping for each field.

Example 218. Retrieving the Lucene analyzers for a whole index

```
SearchMapping mapping = Search.mapping( entityManagerFactory ); ①
IndexManager indexManager = mapping.indexManager( "Book" ); ②
LuceneIndexManager luceneIndexManager = indexManager.unwrap( LuceneIndexManager.class ); ③
Analyzer indexingAnalyzer = luceneIndexManager.indexingAnalyzer(); ④
Analyzer searchAnalyzer = luceneIndexManager.searchAnalyzer(); ⑤
```

1. Retrieve the `SearchMapping`.
2. Retrieve the `IndexManager`.
3. Narrow down the index manager to the `LuceneIndexManager` type.
4. Get the indexing analyzer. This is the analyzer used when indexing documents. It ignores the `search analyzer` in particular.
5. Get the search analyzer. This is the analyzer used when building search queries through the `Search DSL`. On contrary to the indexing analyzer, it takes into account the `search analyzer`.

248
Chapter 11. Elasticsearch backend

11.1. 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></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>

11.1.1. 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.

11.2. Basic 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, so you will need to set the address of your cluster by configuring the client.

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

### 11.3. Configuration of the Elasticsearch cluster

Most of the time, Hibernate Search does not require any specific configuration to be applied by hand to the Elasticsearch cluster, beyond the index mapping (schema) which can be automatically generated.

The only exception is Sharding, which needs to be enabled explicitly.

### 11.4. Client configuration

An Elasticsearch backend communicates with an Elasticsearch cluster through a REST client. Below are the options that affect this client.

#### 11.4.1. Target hosts

The following property configures the Elasticsearch host (or hosts) to send indexing requests and search queries to:

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

This property may be set to a String representing a host and port such as localhost or es.mycompany.com:4400, or a String containing multiple such host-and-port strings separated by commas, or a `Collection<String>` containing such host-and-port strings.

You may change the protocol used to communicate with the hosts using this configuration property:

```java
hibernate.search.backends.<backend name>.protocol = http (default)
```

This property may be set to either http or https.
11.4.2. Node 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 configuration).

Automatic discovery is controlled by the following properties:

```
hibernate.search.backends.<backend name>.discovery.enabled = false (default)
hibernate.search.backends.<backend name>.discovery.refresh_interval = 10 (default)
```

- `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.

11.4.3. HTTP authentication

HTTP authentication is disabled by default, but may be enabled by setting the following configuration 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.

![Warning](https://example.com/warning.png)

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

11.4.4. 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:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-search-backend-elasticsearch-aws</artifactId>
    <version>6.0.0.Beta8</version>
</dependency>
```

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

```java
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.

**11.4.5. Connection tuning**

*Timeouts*

```java
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.

*Connection pool*

```java
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.

11.4.6. 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.

If necessary, you can disable the call to the Elasticsearch cluster for version checks on startup. To do that, set hibernate.search.backends.<backend name>.version_check.enabled to false, and set the property hibernate.search.backends.<backend name>.version 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 (alphanumeric or _).

The major and minor version numbers (x and y in the format above) are mandatory, but other numbers are optional.

The property hibernate.search.backends.<backend name>.version can also be set when hibernate.search.backends.<backend name>.version_check.enabled is true (the default).

In that case, Hibernate Search will still query the Elasticsearch cluster to know the actual version of the cluster, and will check that the configured version matches the actual version. This can be helpful while developing, in particular.

11.4.7. 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.

11.5. Sharding

For a preliminary introduction to sharding, including how it works in Hibernate Search and what its limitations are, see Sharding and routing.

Elasticsearch disables sharding by default. To enable it, set the property index.number_of_shards in your cluster.
11.6. Index lifecycle

Elasticsearch indexes need to be created before they can be used for indexing and searching; see Managing the index schema for more information about how to create indexes and their schema in Hibernate Search.

For Elasticsearch specifically, some fine-tuning is available through the following options:

```java
hibernate.search.backends.<backend name>.indexes.<index name>.schema_management.minimal_required_status green (default)
hibernate.search.backends.<backend name>.indexes.<index name>.schema_management.minimal_required_status_wait_timeout 10000 (default)

# OR
hibernate.search.backends.<backend name>.index_defaults.schema_management.minimal_required_status green (default)
hibernate.search.backends.<backend name>.index_defaults.schema_management.minimal_required_status_wait_timeout 10000 (default)
```

- **minimal_required_status** defines the minimal required status of an index before creation is considered complete.
- **minimal_required_status_wait_timeout** defines the maximum time to wait for this status, as an integer value in milliseconds.

These properties are only effective when creating or validating an index as part of schema management.

11.7. Index layout

Hibernate Search works with aliased indexes. This means an index with a given name in Hibernate Search will not directly be mapped to an index with the same name in Elasticsearch.

By default, for an index whose name in Hibernate Search is `myIndex`:

- Write operations (indexing, purge, ...) will target the alias `myindex-write`.
- Read operations (searching, explaining, ...) will target the alias `myindex-read`.
- If Hibernate Search creates the index automatically, it will name the index `myindex-000001` and will automatically create the write and read aliases.
This layout is a bit more complex than it could be, but it follows the best practices.

Using aliases has a significant advantage over directly targeting the index: it makes full reindexing on a live application possible. With aliases, you just need to direct the read alias (used by search queries) to an old copy of the index, while the write alias (used by document writes) is redirected to a new copy of the index.

This "zero-downtime" reindexing, which shares some characteristics with "blue/green" deployment, is not currently provided by Hibernate Search itself. However, you can implement it in your application by directly issuing commands to Elasticsearch’s REST APIs. The basic sequence of actions is the following:

1. Create a new index, `myindex-000002`.
2. Switch the write alias, `myindex-write`, from `myindex-000001` to `myindex-000002`.
3. Reindex, for example using the mass indexer.
4. Switch the read alias, `myindex-read`, from `myindex-000001` to `myindex-000002`.
5. Delete `myindex-000001`.

If the default names and aliases used by Hibernate Search do not match your needs, you can define a custom layout in two simple steps:

1. Define a class that implements the interface `org.hibernate.search.backend.elasticsearch.index.layout.IndexLayoutStrategy`.
2. Configure the backend to use that implementation by setting the configuration property `hibernate.search.backends.<backend name>.layout.strategy` to a bean reference pointing to the implementation.

For example, the implementation below will lead to the following layout for an index named `myIndex`:

- Write operations (indexing, purge, ...) will target the alias `myindex-write`.
- Read operations (searching, explaining, ...) will target the alias `myindex` (no suffix).
- If Hibernate Search creates the index automatically at exactly 19:19:00 on November 6th, 2017, it will name the index `myindex-20171106-191900-0000000000`.
import java.time.Clock;
import java.time.Instant;
import java.time.ZoneOffset;
import java.time.format.DateTimeFormatter;
import java.util.Locale;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import org.hibernate.search.backend.elasticsearch.index.layout.IndexLayoutStrategy;

public class CustomLayoutStrategy implements IndexLayoutStrategy {

    private static final DateTimeFormatter INDEX_SUFFIX_FORMATTER =
            DateTimeFormatter.ofPattern("uuuuMMdd-HHmmss-SSSSSSSSS", Locale.ROOT).
            withZone(ZoneOffset.UTC);

    private static final Pattern UNIQUE_KEY_PATTERN =
            Pattern.compile("(.*)-\d+-\d+-\d+\d+");

    @Override
    public String createInitialElasticsearchIndexName(String hibernateSearchIndexName) {
        // Clock is Clock.systemUTC() in production, may be overridden in tests
        Clock clock = MyApplicationClock.get();
        return hibernateSearchIndexName + INDEX_SUFFIX_FORMATTER.format(Instant.now(clock));
    }

    @Override
    public String createWriteAlias(String hibernateSearchIndexName) {
        return hibernateSearchIndexName + "-write";
    }

    @Override
    public String createReadAlias(String hibernateSearchIndexName) {
        return hibernateSearchIndexName;
    }

    @Override
    public String extractUniqueKeyFromHibernateSearchIndexName(String hibernateSearchIndexName) {
        return hibernateSearchIndexName;
    }

    @Override
    public String extractUniqueKeyFromElasticsearchIndexName(String elasticsearchIndexName) {
        Matcher matcher = UNIQUE_KEY_PATTERN.matcher(elasticsearchIndexName);
        if (!matcher.matches()) {
            throw new IllegalArgumentException("Unrecognized index name: " + elasticsearchIndexName);
        }
        return matcher.group(1);
    }
}

11.8. Schema ("mapping")

What Elasticsearch calls the "mapping" is the schema assigned to each index, specifying the data type and capabilities of each "property" (called an "index field" in Hibernate Search).
For the most part, the Elasticsearch mapping is inferred from the mapping configured through Hibernate Search’s mapping APIs, which are generic and independent from Elasticsearch.

Aspects that are specific to the Elasticsearch backend are explained in this section.

Hibernate Search can be configured to push the mapping to Elasticsearch when creating the indexes through the index lifecycle strategy.

### 11.8.1. Field types

Available 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 Supported property types for more information.

Field types that are not in this list can still be used with a little bit more work:

- If a property in the entity model has an unsupported type, but can be converted to a supported type, you will need a bridge. See Bridges.
- If you need an index field with a specific type that is not supported by Hibernate Search, you will need a bridge that defines a native field type. See Index field type DSL extension.

**Table 9. Field types supported by the Elasticsearch backend**

<table>
<thead>
<tr>
<th>Field type</th>
<th>Data type in Elasticsearch</th>
<th>Limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.String</td>
<td>text if an analyzer is defined, keyword otherwise</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.Byte</td>
<td>byte</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.Short</td>
<td>short</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.Integer</td>
<td>integer</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.Long</td>
<td>long</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.Double</td>
<td>double</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.Float</td>
<td>float</td>
<td>-</td>
</tr>
<tr>
<td>java.lang.Boolean</td>
<td>boolean</td>
<td>-</td>
</tr>
<tr>
<td>Field type</td>
<td>Data type in Elasticsearch</td>
<td>Limitations</td>
</tr>
<tr>
<td>------------------------</td>
<td>--------------------------------------------------------------------------------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>java.math.BigDecimal</td>
<td>scaled_float with a scaling_factor equal to $10^{\text{decimalScale}}$</td>
<td>-</td>
</tr>
<tr>
<td>java.math.BigInteger</td>
<td>scaled_float with a scaling_factor equal to $10^{\text{decimalScale}}$</td>
<td>-</td>
</tr>
<tr>
<td>java.time.Instant</td>
<td>date with format `uuuu-MM-dd'T'HH:mm:ss.SSSSSSSSSZZ (ES7 and above) or yyyy-MM-dd'T'HH:mm:ss.SSS'SZ'</td>
<td></td>
</tr>
<tr>
<td>java.time.LocalDate</td>
<td>date with format <code>uuuu-MM-dd</code> (ES7 and above) or <code>yyyy-MM-dd</code> (ES6 and below)</td>
<td>Lower range/resolution</td>
</tr>
<tr>
<td>java.time.LocalTime</td>
<td>date with format <code>HH:mm:ss.SSSSSSSSS</code> (ES7 and above) or <code>HH:mm:ss.SSS</code></td>
<td></td>
</tr>
<tr>
<td>java.time.LocalDateTime</td>
<td>date with format <code>uuuu-MM-dd'T'HH:mm:ss.SSSSSSSSS</code> (ES7 and above) or <code>yyyy-MM-dd'T'HH:mm:ss.SSS</code></td>
<td></td>
</tr>
<tr>
<td>Field type</td>
<td>Data type in Elasticsearch</td>
<td>Limitations</td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>java.time.ZonedDateTime</td>
<td>date with format uuuu-MM-dd'T'HH:mm:ss.SSSSSSSSSZZZ['VV'] (ES7 and above) or yyyy-MM-dd'T'HH:mm:ss.SSSSSSSSZZZ['ZZ']</td>
<td>Lower range/resolution</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.time.OffsetDateTime</td>
<td>date with format uuuu-MM-dd'T'HH:mm:ss.SSSSSSSSSZZZ (ES7 and above) or yyyy-MM-dd'T'HH:mm:ss.SSSSSSZZZ</td>
<td>Lower range/resolution</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.time.OffsetTime</td>
<td>date with format HH:mm:ss.SSSSSSSZZZZ (ES7 and above) or HH:mm:ss.SSSZZZ:HH:mm:ss.SSSSSSSZZ (ES6 and below)</td>
<td>Lower range/resolution</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.time.Year</td>
<td>date with format uuuu (ES7 and above) or yyyy</td>
<td>Lower range/resolution</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>java.time.YearMonth</td>
<td>date with format uuuu-MM (ES7 and above) or yyyy-MM</td>
<td>Lower range/resolution</td>
</tr>
<tr>
<td></td>
<td></td>
<td></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>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>org.hibernate.search.engine.spatial.GeoPoint</td>
<td>geo_point</td>
<td>-</td>
</tr>
</tbody>
</table>
Range and resolution of date/time fields

The Elasticsearch date type does 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 \(999.999.999\).
- Elasticsearch’s date type supports dates ranging from year \(-292.275.054\) to year \(292.278.993\).

Values that are out of range will trigger indexing failures.

Resolution is also lower:

- java.time supports nanosecond-resolution.
- Elasticsearch’s date type supports millisecond-resolution.

Precision beyond the millisecond will be lost when indexing.

Index field type DSL extension

Not all Elasticsearch field types have built-in support in Hibernate Search. Unsupported field types can still be used, however, by taking advantage of the "native" field type. Using this field type, the Elasticsearch "mapping" can be defined as JSON directly, giving access to everything Elasticsearch can offer.

Below is an example of how to use the Elasticsearch "native" type.

Example 220. Using the Elasticsearch "native" type
Define a custom binder and its bridge. The "native" type can only be used from a binder, it cannot be used directly with annotation mapping. Here we’re defining a value binder, but a type binder, or a property binder would work as well.

Get the context’s type factory.

Apply the Elasticsearch extension to the type factory.

Call asNative to start defining a native type.

Pass the Elasticsearch mapping as JSON.

Values of native fields are represented as a JsonElement in Hibernate Search. JsonElement is a type from the Gson library. Do not forget to format them correctly before you pass them to the backend. Here we are creating a JsonPrimitive (a subtype of JsonElement) from a String because we just need a JSON string, but it’s completely possible to handle more complex objects, or even to convert directly from POJOs to JSON using Gson.

For nicer projections, you can also implement this method to convert from JsonElement to the mapped type (here, String).
public class CompanyServer {
  private Integer id;
  private String ipAddress;
  // Getters and setters
}

① Map the property to an index field. Note that value bridges using a non-standard type (such as
Elasticsearch’s "native" type) must be mapped using the @NonStandardField annotation:
other annotations such as @GenericField will fail.

② Instruct Hibernate Search to use our custom value binder.

11.8.2. Type name mapping

When Hibernate Search performs a search query targeting multiple entity types, and thus multiple
indexes, it needs to determine the type of each search hit in order to map it back to an entity.

There are multiple strategies to handle this "type name resolution", and each has pros and cons.

The strategy is set a the backend level:

hibernate.search.backends.<backend-name>.mapping.type_name.strategy = discriminator (default)

See the following subsections for details about available strategies.

discriminator: type name mapping using a discriminator field

With the discriminator strategy, a discriminator field is used to retrieve the type name directly
from each document.

When indexing, the _entity_type field is populated transparently with type name for each
document.

When searching, the docvalues for the type field is transparently requested to Elasticsearch and
extracted from the response.

Pros:
Works correctly when targeting index aliases.

Cons:

- Small storage overhead: a few bytes of storage per document.
- Requires reindexing if an entity name changes, even if the index name doesn't change.

**index-name**: type name mapping using the index name

With the index-name strategy, the _index meta-field returned for each search hit is used to resolve the index name, and from that the type name.

Pros:

- No storage overhead.

Cons:

- Relies on the actual index name, not aliases, because the _index meta-field returned by Elasticsearch contains the actual index name (e.g. myindex-000001), not the alias (e.g. myindex-read). If indexes do not follow the default naming scheme <hibernateSearchIndexName>-<6 digits>, a custom index layout must be configured.

### 11.8.3. 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, a strategy must be selected in order to enable multi-tenancy.

The multi-tenancy strategy is set at the backend level:

```
hibernate.search.backends.<backend name>.multi_tenancy.strategy = none (default)
```

See the following subsections for details about available strategies.

**none**: single-tenancy

The none strategy (the default) disables multi-tenancy completely.

Attempting to set a tenant ID will lead to a failure when indexing.
**discriminator**: type name mapping using the index name

With the discriminator strategy, all documents from all tenants are stored in the same index. The Elasticsearch ID of each document is set to the concatenation of the tenant ID and original ID.

When indexing, two fields are populated transparently for each document:

- `_tenant_id`: the "discriminator" field holding the tenant ID.
- `_tenant_doc_id`: a field holding the the original (tenant-scoped) document ID.

When searching, a filter targeting the tenant ID field is added transparently to the search query to only return search hits for the current tenant. The ID field is used to retrieve the original document IDs.

### 11.9. Analysis

Analysis is the text processing performed by analyzers, both when indexing (document processing) and when searching (query processing).

All built-in Elasticsearch analyzers can be used transparently, without any configuration in Hibernate Search: just use their name wherever Hibernate Search expects an analyzer name. However, in order to define custom analyzers, analysis must be configured explicitly.

Elasticsearch analysis configuration is not applied immediately on startup: it needs to be pushed to the Elasticsearch cluster. Hibernate Search will only push the configuration to the cluster if specific conditions are met, and only if instructed to do so through the lifecycle configuration.

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

- Define a class that implements the `org.hibernate.search.backend.elasticsearch.analysis.ElasticsearchAnalysisConfigurer` interface.
- Configure the backend to use that implementation by setting the configuration property `hibernate.search.backends.<backend name>.analysis.configurer` to a bean reference pointing to the implementation.

Hibernate Search will call the `configure` method of this implementation on startup, and the configurer will be able to take advantage of a DSL to define analyzers:
package org.hibernate.search.documentation.analysis;

import org.hibernate.search.backend.elasticsearch.analysis.ElasticsearchAnalysisConfigurationContext;
import org.hibernate.search.backend.elasticsearch.analysis.ElasticsearchAnalysisConfigurer;

public class MyElasticsearchAnalysisConfigurer implements ElasticsearchAnalysisConfigurer {
  @Override
  public void configure(ElasticsearchAnalysisConfigurationContext context) {
    context.analyzer("english").custom() ①
      .tokenizer("standard") ②
      .charFilters("html_strip") ③
      .tokenFilters("lowercase", "snowball_english", "asciifolding"); ④

    context.tokenFilter("snowball_english") ⑤
      .type("snowball")
      .param("language", "English"); ⑥

    context.normalizer("lowercase").custom() ⑦
      .tokenFilters("lowercase", "asciifolding");

    context.analyzer("french").custom() ⑧
      .tokenizer("standard")
      .tokenFilters("lowercase", "snowball_french", "asciifolding");

    context.tokenFilter("snowball_french")
      .type("snowball")
      .param("language", "French");
  }
}

① Define a custom analyzer named "english", because it will be used to analyze English text such as book titles.

② Set the tokenizer to a standard tokenizer.

③ Set the char filters. Char filters are applied in the order they are given, before the tokenizer.

④ Set the token filters. Token filters are applied in the order they are given, after the tokenizer.

⑤ Note that, for Elasticsearch, any parameterized char filter, tokenizer or token filter must be defined separately and assigned a name.

⑥ Set the value of a parameter for the char filter/tokenizer/token filter being defined.

⑦ Normalizers are defined in a similar way, the only difference being that they cannot use a tokenizer.

⑧ Multiple analyzers/normalizers can be defined in the same configurer.

hibernate.search.backends.myBackend.analysis.configurer
org.hibernate.search.documentation.analysis.MyElasticsearchAnalysisConfigurer

① Assign the configurer to the backend myBackend using a Hibernate Search configuration property.
It is also possible to assign a name to a parameterized built-in analyzer:

Example 222. Naming a parameterized built-in analyzer in the Elasticsearch backend

```java
context.analyzer( "english_stopwords" ).type( "standard" ) ①
  .param( "stopwords", "_english_" ); ②
```

① Define an analyzer with the given name and type.

② Set the value of a parameter for the analyzer being defined.

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.

11.10. Threads

The Elasticsearch backend relies on an internal thread pool to orchestrate indexing requests (add/update/delete) and to schedule request timeouts.

By default, the pool contains exactly as many threads as the number of processors available to the JVM on bootstrap. That can be changed using a configuration property:

```java
hibernate.search.backends.<backend-name>.thread_pool.size = 4
```

This number is per backend, not per index. Adding more indexes will not add more threads.

As all operations happening in this thread-pool are non-blocking, raising its size above the number of processor cores available to the JVM will not bring noticeable performance benefits.

The only reason to alter this setting would be to reduce the number of threads; for example, in an application with a single index with a single indexing queue, running on a machine with 64 processor cores, you might want to bring down the number of threads.
11.11. Indexing queues

Among all the requests sent by Hibernate Search to Elasticsearch, it is expected that there will be a lot of “indexing” requests to create/update/delete a specific document. Sending these requests one by one would be inefficient (mainly because of network latency). Also, we generally want to preserve the relative order of these requests when they are about the same documents.

For these reasons, Hibernate Search pushes these requests to ordered queues and relies on the Bulk API to send them in batches. Each index maintains 10 queues holding at most 1000 elements each, and each queue will send bulk requests of at most 100 indexing requests. Queues operate independently (in parallel), but each queue sends one bulk request after the other, so at any given time there can be at most 10 bulk requests being sent for each index.

Indexing operations relative to the same document ID are always pushed to the same queue.

It is possible to customize the queues in order to reduce the load on the Elasticsearch server, or on the contrary to improve throughput. This is done through the following configuration properties, at the index level:

```
hibernate.search.backends.<backend name>.indexes.<index name>.indexing.queue_count 10 (default)
hibernate.search.backends.<backend name>.indexes.<index name>.indexing.queue_size 1000 (default)
hibernate.search.backends.<backend name>.indexes.<index name>.indexing.max_bulk_size 100 (default)
# OR
hibernate.search.backends.<backend name>.index_defaults.indexing.queue_count 10 (default)
hibernate.search.backends.<backend name>.index_defaults.indexing.queue_size 1000 (default)
hibernate.search.backends.<backend name>.index_defaults.indexing.max_bulk_size 100 (default)
```

- **indexing.queue_count** defines the number of queues. Expects a strictly positive integer value.
  
  Higher values will lead to more connections being used in parallel, which may lead to higher indexing throughput, but incurs a risk of overloading Elasticsearch, leading to Elasticsearch giving up on some requests and resulting in indexing failures.

- **indexing.queue_size** defines the maximum number of elements each queue can hold. Expects a strictly positive integer value.
  
  Lower values may lead to lower memory usage, especially if there are many queues, but values that are too low will reduce the likeliness of reaching the max bulk size and increase the likeliness of application threads blocking because the queue is full, which may lead to lower indexing throughput.

- **indexing.max_bulk_size** defines the maximum number of indexing requests in each bulk request. Expects a strictly positive integer value.
Higher values will lead to more documents being sent in each HTTP request sent to Elasticsearch, which may lead to higher indexing throughput, but incurs a risk of overloading Elasticsearch, leading to Elasticsearch giving up on some requests and resulting in indexing failures.

Note that raising this number above the queue size has no effect, as bulks cannot include more requests than are contained in the queue.

When a queue is full, any attempt to request indexing will block until the request can be put into the queue.

In order to achieve a reasonable level of performance, be sure to set the size of queues to a high enough number that this kind of blocking only happens when the application is under very high load.

Elasticsearch nodes can only handle so many parallel requests, and in particular they limit the amount of memory available to store all pending requests at any given time.

In order to avoid indexing failures, avoid using overly large numbers for the number of queues and the maximum bulk size, especially if you expect your index to hold large documents.

11.12. Writing and reading

For a preliminary introduction to writing to and reading from indexes in Hibernate Search, including in particular the concepts of commit and refresh, see Commit and refresh.

11.12.1. Commit

When writing to indexes, Elasticsearch relies on a transaction log to make sure that changes, even uncommitted, are always safe as soon as the REST API call returns.

For that reason, the concept of "commit" is not as important to the Elasticsearch backend, and commit requirements are largely irrelevant.

11.12.2. Refresh

When reading from indexes, Elasticsearch relies on a periodically refreshed index reader, meaning that search queries will return slightly out-of-date results, unless a refresh was forced: this is called near-real-time behavior.
By default, the index reader is refreshed every second, but this can be customized on the Elasticsearch side through index settings: see the `refresh_interval` setting on this page.

### 11.13. Retrieving the REST client

When writing complex applications with advanced requirements, it may be necessary from time to time to send requests to the Elasticsearch cluster directly, in particular if Hibernate Search does not support this kind of requests out of the box.

To that end, you can retrieve the Elasticsearch backend, then get access the Elasticsearch client used by Hibernate Search internally. See below for an example.

**Example 223. Accessing the low-level REST client**

```java
SearchMapping mapping = Search.mapping( entityManagerFactory ); ①
Backend backend = mapping.backend( "myBackend" ); ②
ElasticsearchBackend elasticsearchBackend = backend.unwrap( ElasticsearchBackend.class ); ③
RestClient client = elasticsearchBackend.client( RestClient.class ); ④
```

1. Retrieve the `SearchMapping`.
2. Retrieve the `Backend`.
3. Narrow down the backend to the `ElasticsearchBackend` type.
4. Get the client, passing the expected type of the client as an argument.

The client itself is not part of the Hibernate Search API, but of the official Elasticsearch REST client API.

Hibernate Search may one day switch to another client with a different Java type, without prior notice. If that happens, the snippet of code above will throw an exception.
Chapter 12. Index Optimization

⚠️ This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.
Chapter 13. Monitoring

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

⚠️ This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.
Chapter 15. 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.

15.1. 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.

15.1.1. 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.

15.1.2. 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 `IndexSchemaFieldFinalStep#toReference`). 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 `IndexIndexingPlan` class. The mapper should create an indexing plan whenever it needs to add, update or delete a document.

`IndexIndexingPlan` 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 indexing plan whenever this context changes.

Index-scale operations such as flush, merge-segments, etc. are unavailable from indexing plans. They are accessed through a separate class, `IndexWorkspace`.

15.1.3. 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.selectEntity`) that will essentially transform document references into the entity that was initially indexed.

## 15.2. 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.

### 15.2.1. 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**).

### 15.2.2. 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:

- `PojoIndexingProcessorOriginalTypeNode`: 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 `PojoIndexingProcessorOriginalTypeNode` 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.

15.2.3. 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:

```
IndexedEntityClass:PojoIndexingDependencyCollectorTypeNode

embedded:PojoIndexingDependencyCollectorPropertyNode

<table>
<thead>
<tr>
<th>PojoIndexingDependencyCollectorValueNode</th>
<th>PojoIndexingDependencyCollectorValueNode</th>
<th>PojoIndexingDependencyCollectorValueNode</th>
</tr>
</thead>
</table>

EmbeddedEntityClass:PojoIndexingDependencyCollectorTypeNode

secondLevelEmbedded:PojoIndexingDependencyCollectorPropertyNode

<table>
<thead>
<tr>
<th>PojoIndexingDependencyCollectorValueNode</th>
<th>PojoIndexingDependencyCollectorValueNode</th>
</tr>
</thead>
</table>

SecondLevelEmbeddedEntityClass:PojoIndexingDependencyCollectorTypeNode

firstLevelEmbedded:PojoIndexingDependencyCollectorPropertyNode

<table>
<thead>
<tr>
<th>PojoIndexingDependencyCollectorValueNode</th>
<th>PojoIndexingDependencyCollectorValueNode</th>
</tr>
</thead>
</table>

IndexedEntityClass:PojoIndexingProcessorTypeNodeBuilder

embedded:PojoIndexingDependencyCollectorPropertyNode

<table>
<thead>
<tr>
<th>PojoIndexingDependencyCollectorValueNode</th>
<th>PojoIndexingDependencyCollectorValueNode</th>
</tr>
</thead>
</table>
```

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:

![Dependency Collector Tree Diagram]

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).

15.3. JSON mapper

The JSON mapper does not currently exist, but there are plans to work on it.
Chapter 16. Further reading

This section is incomplete. It will be completed during the Alpha/Beta phases of Hibernate Search 6.0.0.
Chapter 17. 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
- Fabio Massimo Ercoli
- Sanne Grinovero
- Mincong Huang
- Nabeel Ali Memon
- Gunnar Morling
- Yoann Rodière
- Guillaume Smet