



Hibernate Search 6.0.0.Alpha2

Reference Documentation

2019-01-31

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. Indexing	9
1.7. Searching	10
1.8. Analysis	12
1.9. What's next	17
2. Architecture	18
3. Configuration	19
4. Mapping Java entities to the index structure	20
4.1. Direct field mapping	20
4.2. Bridges	23
4.3. Value bridges	24
4.4. Indexed-embedded	24
4.5. Container value extractors	25
5. Indexing	26
5.1. Automatic indexing	26
5.2. Explicit indexing	26
6. Analysis	27
7. Search query	28
7.1. Concept	28
7.2. Sort	28
7.3. Projection	29
8. Search DSLs	32
8.1. Predicate DSL	32
8.2. Sort DSL	32
8.3. Projection DSL	32
9. Lucene backend	33
10. Elasticsearch backend	34
11. Index Optimization	35
12. Monitoring	36
13. Spatial	37
14. Advanced features	38
15. Internals of Hibernate Search	39
15.1. General overview	39

15.2. POJO mapper.....	44
15.3. JSON mapper.....	56
16. Further reading.....	57
17. Credits	58

Preface



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

Chapter 1. Getting started

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



Hibernate Search 6.0.0.Alpha2 is a **technology preview** and is **not ready for production**.

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

Read [the dedicated page on our website](#) for more detailed and up-to-date information.

1.1. Compatibility

Table 1. Compatibility

Java Runtime	Java 8 or greater.
Hibernate ORM (for the ORM mapper)	Hibernate ORM 5.4.1.Final.
JPA (for the ORM mapper)	JPA 2.2.

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:

```
<dependency>
  <groupId>org.hibernate.search</groupId>
  <artifactId>hibernate-search-mapper-orm</artifactId>
  <version>6.0.0.Alpha2</version>
</dependency>
<dependency>
  <groupId>org.hibernate.search</groupId>
  <artifactId>hibernate-search-backend-lucene</artifactId>
  <version>6.0.0.Alpha2</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:

```
<dependency>
  <groupId>org.hibernate.search</groupId>
  <artifactId>hibernate-search-mapper-orm</artifactId>
  <version>6.0.0.Alpha2</version>
</dependency>
<dependency>
  <groupId>org.hibernate.search</groupId>
  <artifactId>hibernate-search-backend-elasticsearch</artifactId>
  <version>6.0.0.Alpha2</version>
</dependency>
```

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

```
<property name="hibernate.search.backends.myBackend.type"
  value="lucene"/> ①
<property name="hibernate.search.backends.myBackend.directory_provider"
  value="local_directory"/> ②
<!--
<property name="hibernate.search.backends.myBackend.root_directory"
  value="some/filesystem/path"/>
--> ③
<property name="hibernate.search.default_backend"
  value="myBackend"/> ④
```

- ① Define a backend named "myBackend" relying on Lucene technology.
- ② Define the storage for that backend as a local filesystem directory.
- ③ The backend will store indexes in the current working directory by default. If you want to store the indexes elsewhere, uncomment this line and set the value of the property.
- ④ Make sure to use the backend we just defined for all indexes.

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

```
<property name="hibernate.search.backends.myBackend.type"
  value="elasticsearch" /> ①
<!--
<property name="hibernate.search.backends.myBackend.host"
  value="https://elasticsearch.mycompany.com"/>
<property name="hibernate.search.backends.myBackend.username"
  value="ironman"/>
<property name="hibernate.search.backends.myBackend.password"
  value="j@rV1s"/>
--> ②
<property name="hibernate.search.default_backend"
  value="myBackend"/> 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 "host" 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.

Example 3. Book and Author entities BEFORE 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;

@Entity
public class Book {

    @Id
    @GeneratedValue
    private Integer id;

    private String title;

    @ManyToMany
    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;

@Entity
public class Author {

    @Id
    @GeneratedValue
    private Integer id;

    private String name;

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

    public Author() {
    }

    // Getters and setters
    // ...

}
```

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 Java entities to the index structure](#).

Below is an example of how the model above can be mapped.

Example 4. 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.GenericField;
import org.hibernate.search.mapper.pojo.mapping.definition.annotation.Indexed;
import org.hibernate.search.mapper.pojo.mapping.definition.annotation.IndexedEmbedded;

@Entity
@Indexed ①
public class Book {

    @Id ②
    @GeneratedValue
    private Integer id;

    @GenericField ③
    private String title;

    @ManyToMany
    @IndexedEmbedded ④
    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 ⑤
public class Author {

    @Id
    @GeneratedValue
    private Integer id;

    @GenericField ③
    private String name;

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

    public Author() {
    }

    // Getters and setters
    // ...
}

```

- ① **@Indexed** marks **Book** as indexed, i.e. an index will be created for that entity, and that index will be kept up to date.
- ② By default, the JPA **@Id** is used to generate a document identifier.
- ③ **@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.
- ④ **@IndexedEmbedded** allows to "embed" the indexed form of associated objects (entities or embeddables) into the indexed form of the embedding entity. Here, the **Author** class defines a single indexed field, **name**. Thus adding **@IndexedEmbedded** to the **authors** property of **Book** will add a single **authors.name** field to the **Book** index. This field will be populated automatically based on the content of the **authors** property, and the books will be reindexed automatically whenever the **name** property of their author changes. See [Indexed-embedded](#) for more information.
- ⑤ Entities that are only **@IndexedEmbedded** in other entities, but do not require to be searchable by themselves, do not need to be annotated with **@Indexed**.

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

- Other **@*Field** annotations exist, some of them allowing full-text search, some of them allowing finer-grained configuration for field of a certain type. You can find out more about **@*Field** annotations in [Direct field mapping](#).

- Properties, or even types, can be mapped with finer-grained control using "bridges". See [Bridges](#) for more information.

1.6. Indexing

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

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

```
// 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 a search query right after that transaction, the search results may not be consistent with the changes you just performed.

See [Synchronization with the index](#) for more information about this behavior and how to tune it.

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

Example 6. Using Hibernate Search MassIndexer API to manually (re)index the already persisted data

```
FullTextEntityManager fullTextEntityManager = Search.getFullTextEntityManager(
    entityManager ); ①

MassIndexer indexer = fullTextEntityManager.createIndexer( Book.class ) ②
    .threadsToLoadObjects( 7 ); ③

indexer.startAndWait(); ④
```

- ① Get the Hibernate Search-specific version of the `EntityManager`, called `FullTextEntityManager`.
- ② Create an "indexer", passing the entity types you want to index. Pass no type to index all of them.
- ③ It is possible to set the number of threads to be used. For the complete option list see [\[manual-index-changes\]](#).
- ④ Invoke the batch indexing process.

1.7. Searching

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

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

Example 7. Using Hibernate Search to query the indexes

```
// Not shown: get the entity manager and open a transaction
FullTextEntityManager fullTextEntityManager = Search.getFullTextEntityManager(
    entityManager ); ①

FullTextSearchTarget<Book> searchTarget = fullTextEntityManager.search( Book.class ); ②

FullTextQuery<Book> query = searchTarget.query() ③
    .asEntity() ④
    .predicate( searchTarget.predicate().match() ⑤
        .onFields( "title", "authors.name" )
        .matching( "Refactoring: Improving the Design of Existing Code" )
        .toPredicate()
    )
    .build(); ⑥

List<Book> result = query.getResultList(); ⑦
// Not shown: commit the transaction and close the entity manager
```

- ① Get the Hibernate Search-specific version of the `EntityManager`, called `FullTextEntityManager`.
- ② Create a "search target", representing the indexed types that will be queried.
- ③ Use the "search target" to start creating the query.
- ④ Define the results expected from the query; here we expect managed Hibernate ORM entities, but other options are available.
- ⑤ Define that only documents matching the given predicate should be returned. The predicate is created using the same search target as the query.
- ⑥ Build the query.
- ⑦ Fetch the results.

If this first example looks too verbose to you, you can use an alternative, lambda-based syntax that spares you the declaration of a variable for the search target:

Example 8. Using Hibernate Search to query the indexes - lambda syntax

```
// Not shown: get the entity manager and open a transaction
FullTextEntityManager fullTextEntityManager = Search.getFullTextEntityManager(
    entityManager );

FullTextQuery<Book> query = fullTextEntityManager.search( Book.class ).query()
    .asEntity()
    .predicate( factory -> factory.match()
        .onFields( "title", "authors.name" )
        .matching( "Refactoring: Improving the Design of Existing Code" )
        .toPredicate()
    )
    .build();

List<Book> result = query.getResultList();
// Not shown: commit the transaction and close the entity manager
```

It is possible to get just the result size, using `getResultSize()` method.

Example 9. Using Hibernate Search to count the indexes

```
// Not shown: get the entity manager and open a transaction
FullTextEntityManager fullTextEntityManager = Search.getFullTextEntityManager(
    entityManager );

FullTextQuery<Book> query = fullTextEntityManager.search( Book.class ).query()
    .asEntity()
    .predicate( factory -> factory.match()
        .onFields( "title", "authors.name" )
        .matching( "Refactoring: Improving the Design of Existing Code" )
        .toPredicate()
    )
    .build();

long resultSize = query.getResultSize(); ①
// Not shown: commit the transaction and close the entity manager
```

① Fetch the result size.

1.8. Analysis

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

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

1.8.1. Concept

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

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

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

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

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

1.8.2. Configuration

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

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

import org.hibernate.search.backend.lucene.analysis.LuceneAnalysisConfigurer;
import org.hibernate.search.backend.lucene.analysis.model.dsl.LuceneAnalysisDefinitionContainerContext;

import org.apache.lucene.analysis.core.LowerCaseFilterFactory;
import org.apache.lucene.analysis.miscellaneous.ASCIIFoldingFilterFactory;
import org.apache.lucene.analysis.snowball.SnowballPorterFilterFactory;
import org.apache.lucene.analysis.standard.StandardTokenizerFactory;

public class MyLuceneAnalysisConfigurer implements LuceneAnalysisConfigurer {
    @Override
    public void configure(LuceneAnalysisDefinitionContainerContext context) {
        context.analyzer( "myAnalyzer" ).custom() ①
            .tokenizer( StandardTokenizerFactory.class ) ②
            .tokenFilter( ASCIIFoldingFilterFactory.class ) ③
            .tokenFilter( LowerCaseFilterFactory.class ) ③
            .tokenFilter( SnowballPorterFilterFactory.class ) ③
            .param( "language", "English" ); ④
    }
}
```

```
<property name="hibernate.search.backends.myBackend.analysis_configurer"
    value=
    "org.hibernate.search.documentation.gettingstarted.withhsearch.withanalysis.MyLuceneAnalysisConfigurer"/> ⑤
```

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

Example 11. Analysis configurer implementation and configuration in `persistence.xml` for a "Hibernate ORM + Elasticsearch" setup

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

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

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

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

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

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

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

Example 12. Book and Author entities after adding Hibernate Search specific annotations

```

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

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

@Entity
@Indexed
public class Book {

    @Id
    @GeneratedValue
    private Integer id;

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

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

    public Book() {
    }

    // Getters and setters
    // ...
}

```

```

import 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 = "myAnalyzer") ①
    private String name;

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

    public Author() {
    }

    // Getters and setters
    // ...
}

```

① Replace the `@GenericField` annotation with `@FullTextField`, and set the `analyzer`

parameter to the name of the custom analyzer configured earlier.

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



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

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

```
// Not shown: get the entity manager and open a transaction
FullTextEntityManager fullTextEntityManager = Search.getFullTextEntityManager(
    entityManager );

FullTextQuery<Book> query = fullTextEntityManager.search( Book.class ).query()
    .asEntity()
    .predicate( factory -> factory.match()
        .onFields( "title", "authors.name" )
        .matching( "refactor" )
        .toPredicate()
    )
    .build();

List<Book> result = query.getResultList();
// Not shown: commit the transaction and close the entity manager
```

1.9. What's next

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

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

Other features that you will probably want to use include [sorts](#) and [projections](#)

Chapter 2. Architecture



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

Chapter 3. Configuration



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

Chapter 4. Mapping Java entities to the index structure



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

4.1. Direct field mapping



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

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

Direct field mapping looks like this:

Example 14. Mapping properties to fields directly

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

@GenericField(projectable = Projectable.YES, sortable = Sortable.YES) ③
private Integer pageCount;
```

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

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

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

③ Map another property to its own field.

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

*The `@*Field` annotation*

In its simplest form, direct field mapping is achieved by applying the `@GenericField` annotation

to a property. This annotation will work for every supported property type, but is rather limited: it does not allow full-text search in particular. To go further, you will need to rely on different, more specific annotations, which offer specific attributes. The available annotations are described in details in [Available field annotations](#).

The type of the property

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

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

`name`

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

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

`sortable`

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

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



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

`projectable`

Whether the field can be [projected on](#), i.e. whether the field value is stored in the index to allow later retrieval when querying.

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

4.1.1. Available field annotations

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

`@GenericField`

A good default choice that will work for every supported property type.

Fields mapped using this annotation do not provide any advanced features such as full-text search: matches on a generic field are exact matches.

@FullTextField

A text field whose value is considered as multiple words. Only works for `String` fields.

Matches on a full-text field can be more subtle than exact matches: match fields which contains a given word, match fields regardless of case, match fields ignoring diacritics, ...

Full-text fields must be assigned an `analyzer`, referenced by its name. See [Analysis](#) for more details about analyzers and full-text analysis.



Full-text fields cannot be sorted on. If you need to sort on the value of a property, it is recommended to use `@KeywordField`, with a normalizer if necessary (see below). Note that multiple fields can be added to the same property, so you can use both `@FullTextField` and `@KeywordField` if you need both full-text search and sorting.

@KeywordField

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

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

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

4.1.2. Mapping custom property types

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

There are two cases to distinguish:

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

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

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

extractors.

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

4.2. Bridges



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

Starting with Hibernate Search 6, there are three main interfaces for bridges:

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

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

`ValueBridge` is a suitable interface for your custom bridge if:

- The property value should be mapped to a single index field.
 - The bridge should be applied to a property whose type is effectively immutable. For example `Integer`, or a custom `enum` type, or a custom bean type whose content never changes would be suitable candidates, but a custom bean type with setters would most definitely not.
- `PropertyBridge` can be used for more complex uses cases when mapping an object's property.

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

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

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

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

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

You can find example of custom bridges in the [Hibernate Search source code](#):

- `org.hibernate.search.integrationtest.showcase.library.bridge.ISBNBridge` implements `ValueBridge`.
- `org.hibernate.search.integrationtest.showcase.library.bridge.MultiKeywordsStringBridge` implements `PropertyBridge`. The corresponding annotation is

`org.hibernate.search.integrationtest.showcase.library.bridge.annotation.MultiKeywordStringBridge`.

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

4.3. Value bridges



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

4.3.1. Built-in value bridges



There are few supported types at the moment. This will be solved before the 6.0.0.Final release, see [HSEARCH-3047](#).

The following types have built-in value bridges, meaning they are supported out-of-the box for [direct field mapping](#) using `@*Field` annotations:

- `java.lang.String`
- `java.lang.Integer`
- `java.lang.Long`
- `java.lang.Boolean`
- `java.time.LocalDate`
- `java.time.Instant`
- `java.util.Date`
- All enum types

4.3.2. Type bridges and property bridges



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

4.4. Indexed-embedded



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

4.5. Container value extractors



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

Chapter 5. Indexing

5.1. Automatic indexing



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

5.1.1. Synchronization with the index



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

5.2. Explicit indexing



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

Chapter 6. Analysis



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

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

- For Lucene, either browse the Lucene JavaDoc or read the corresponding section on the [Solr Wiki](#).
- For Elasticsearch, have a look at the online 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](#).



Why the reference to the Apache Solr wiki for Lucene?

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

Chapter 7. Search query



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

7.1. Concept



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

7.2. Sort

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

Example 15. Using custom sorts

```
FullTextEntityManager fullTextEntityManager = Search.getFullTextEntityManager(
    entityManager );

FullTextSearchTarget<Book> searchTarget = fullTextEntityManager.search( Book.class );

FullTextQuery<Book> query = searchTarget.query() ①
    .asEntity()
    .predicate( searchTarget.predicate().matchAll().toPredicate() )
    .sort(
        searchTarget.sort() ②
        .byField( "pageCount" ).desc()
        .then().byField( "title_sort" )
        .toSort()
    )
    .build();

List<Book> result = query.getResultList(); ③
```

① Start building the query as usual.

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

③ The results are sorted according to instructions.

Or alternatively, using the more concise lambda-based syntax:

Example 16. Using custom sorts - lambda syntax

```
FullTextEntityManager fullTextEntityManager = Search.getFullTextEntityManager(
entityManager );

FullTextSearchTarget<Book> searchTarget = fullTextEntityManager.search( Book
.class );

FullTextQuery<Book> query = searchTarget.query()
    .asEntity()
    .predicate( f -> f.matchAll().toPredicate() )
    .sort( f -> f.byField( "pageCount" ).desc()
        .then().byField( "title_sort" )
        .toSort()
    )
    .build();

List<Book> result = query.getResultList(); ③
```



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

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

7.3. Projection

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

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

Example 17. Using projections to extract data from the index

```
FullTextEntityManager fullTextEntityManager = Search.getFullTextEntityManager(
    entityManager );

FullTextSearchTarget<Book> searchTarget = fullTextEntityManager.search( Book.class );

FullTextQuery<String> query = searchTarget.query() ①
    .asProjection( searchTarget.projection().field( "title", String.class )
    .toProjection() ) ②
    .predicate( searchTarget.predicate().matchAll().toPredicate() )
    .build();

List<String> result = query.getResultList(); ③
```

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

Or alternatively, using the more concise lambda-based syntax:

Example 18. Using projections to extract data from the index - lambda syntax

```
FullTextEntityManager fullTextEntityManager = Search.getFullTextEntityManager(
    entityManager );

FullTextQuery<String> query = fullTextEntityManager.search( Book.class ).query()
    .asProjection( f -> f.field( "title", String.class ).toProjection() )
    .predicate( f -> f.matchAll().toPredicate() )
    .build();

List<String> result = query.getResultList();
```



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

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

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

Example 19. Using advanced projection types

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

```
FullTextEntityManager fullTextEntityManager = Search.getFullTextEntityManager(
    entityManager );

FullTextQuery<MyEntityAndScoreBean<Book>> query = fullTextEntityManager.search( Book.class
).query()
    .asProjection( f ->
        f.composite(
            MyEntityAndScoreBean::new,
            f.object(),
            f.score()
        )
        .toProjection()
    )
    .predicate( f -> f.matchAll().toPredicate() )
    .build();

List<MyEntityAndScoreBean<Book>> result = query.getResultList();
```

The sort DSL offers more projection types, and multiple options for each type of projection. To learn more about the field projection, and all the other types of projection, refer to [Projection DSL](#).

Chapter 8. Search DSLs



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

8.1. Predicate DSL



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

8.2. Sort DSL



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

8.3. Projection DSL



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

Chapter 9. Lucene backend



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

Chapter 10. Elasticsearch backend



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

Chapter 11. Index Optimization



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

Chapter 12. Monitoring



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

Chapter 13. Spatial



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.

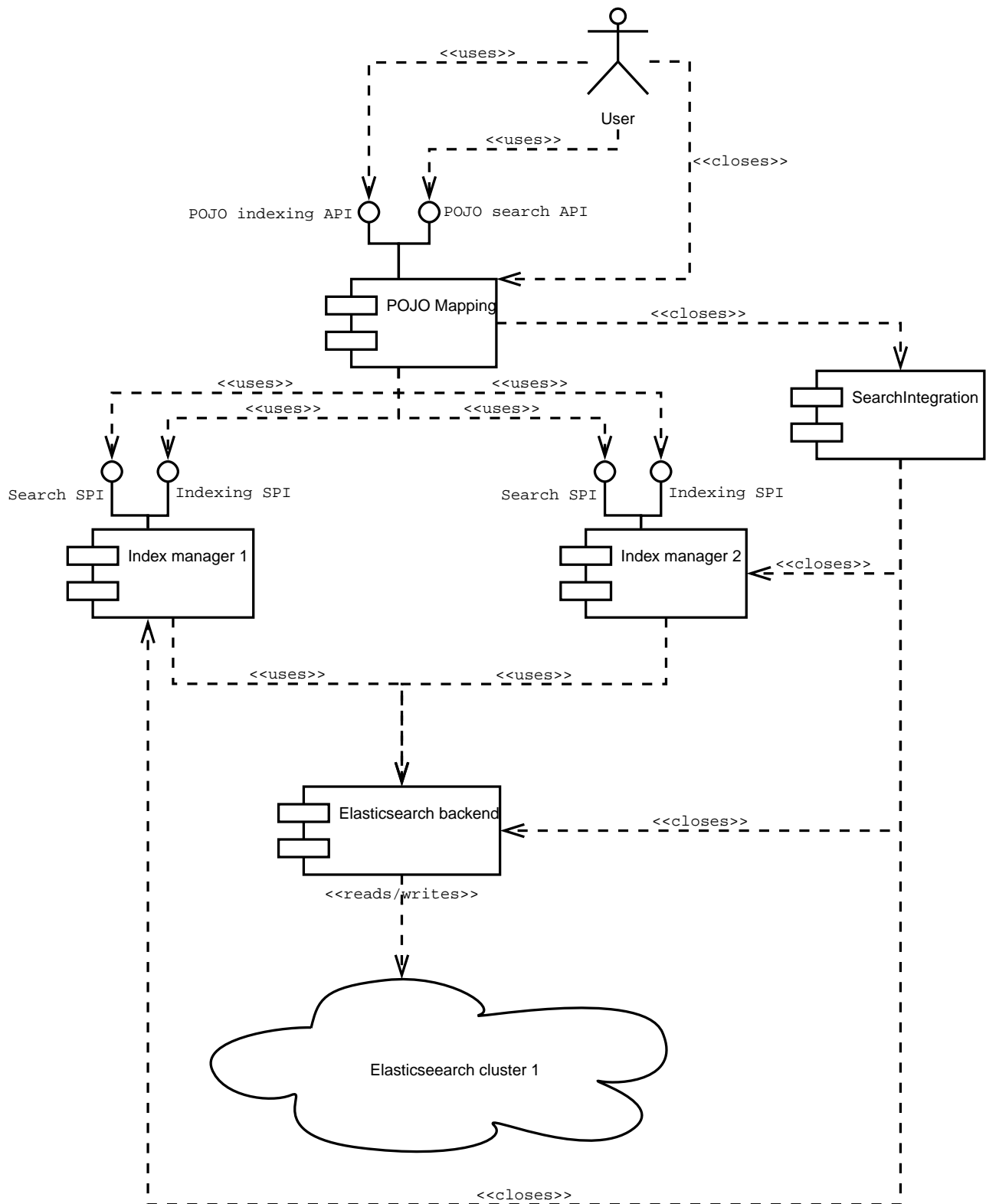
Engine

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.

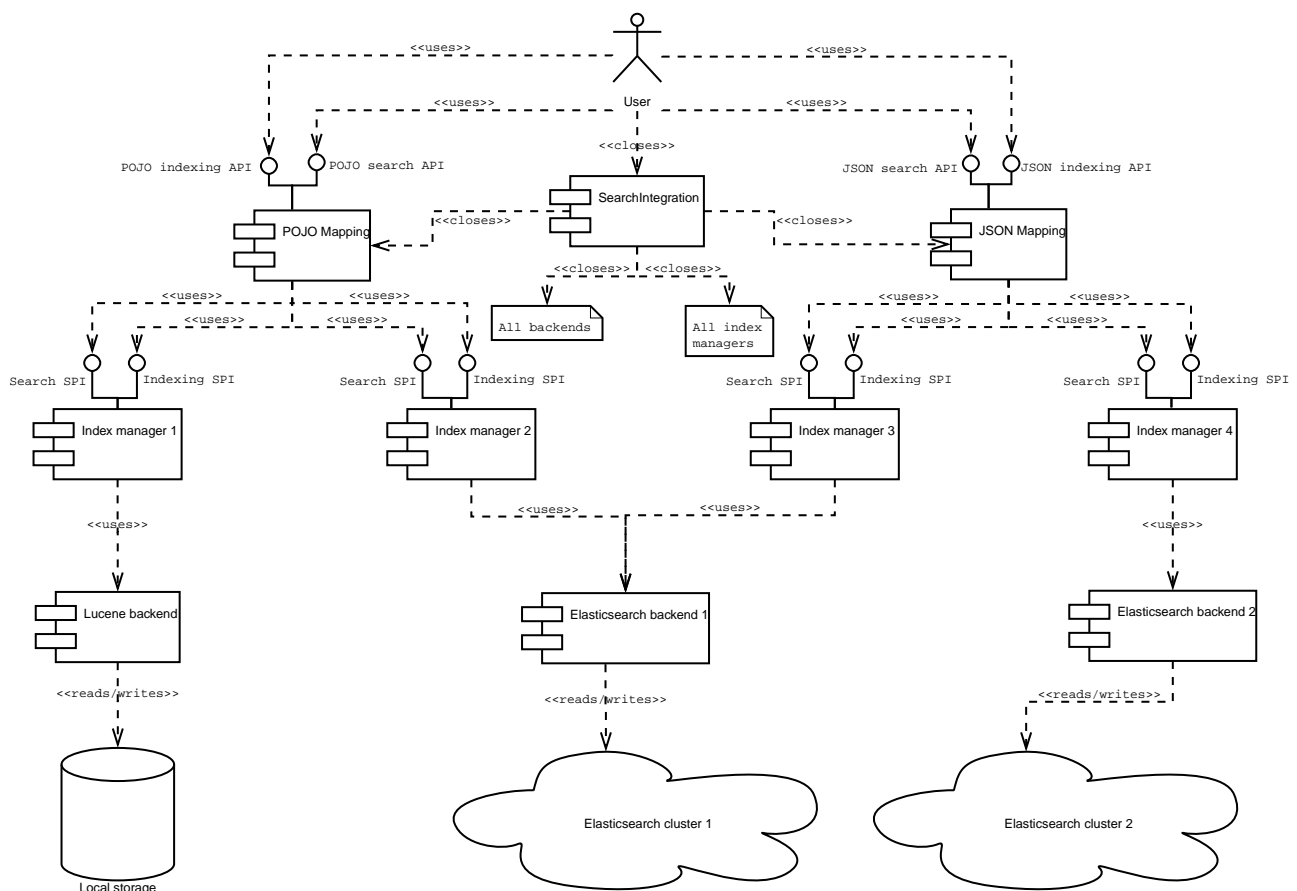
The mapping was provided, during bootstrap, with several "index managers", each exposing SPIs allowing to search and index. The purpose of the mapping is to transform calls to their APIs into call to the index manager SPIs. This requires to perform conversions of:

- 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 index field accessors. During bootstrap, whenever the mapper declared a field, the backend returned an accessor (see `IndexSchemaFieldTerminalContext#createAccessor`). In order to build a document, the mapper extracts data from an object to index, but it then needs to store the extracted data at the appropriate place in the document. The accessor is responsible for exactly this: given a document, it sets the value of the field it represents to some mapper-provided value.

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

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



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

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 target and to provide additional settings. For example `PojoSearchManager#search` allows to define the search target 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`.

Note that the APIs implemented by the engine include ways for the mapper to wrap the resulting search query (`SearchQueryWrappingDefinitionResultContext#asWrappedQuery`). Also, the SPIs implemented by backends allow mappers to inject an "object loader" (see `IndexSearchTarget.query`) that will essentially transform document references into the object 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:

- `PojoIndexingProcessorTypeNode`: A node representing a POJO type (a Java class).
- `PojoIndexingProcessorPropertyNode`: A node representing a POJO property.

- **PojoIndexingProcessorContainerElementNode**: A node representing elements in a container (**List**, **Optional**, ...).

At runtime, the root node will be passed the entity to index and a handle to the document being built. Then each node will "process" its input, i.e. perform one (or more) of the following:

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



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

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

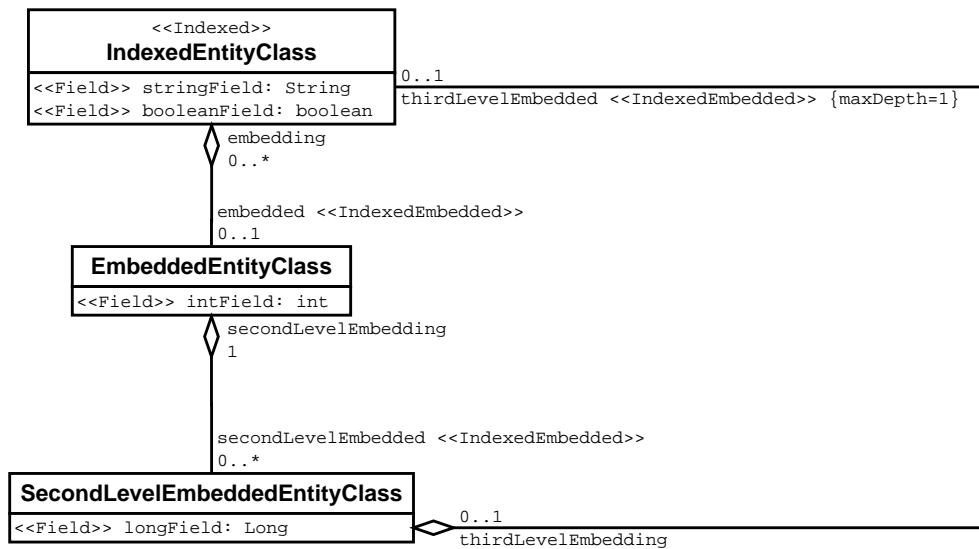


Indexing processors are logged at the debug level during bootstrap. Enable this level of logging for the Hibernate Search classes if you want to understand the indexing processor tree that was generated for a given mapping.

Bootstrap

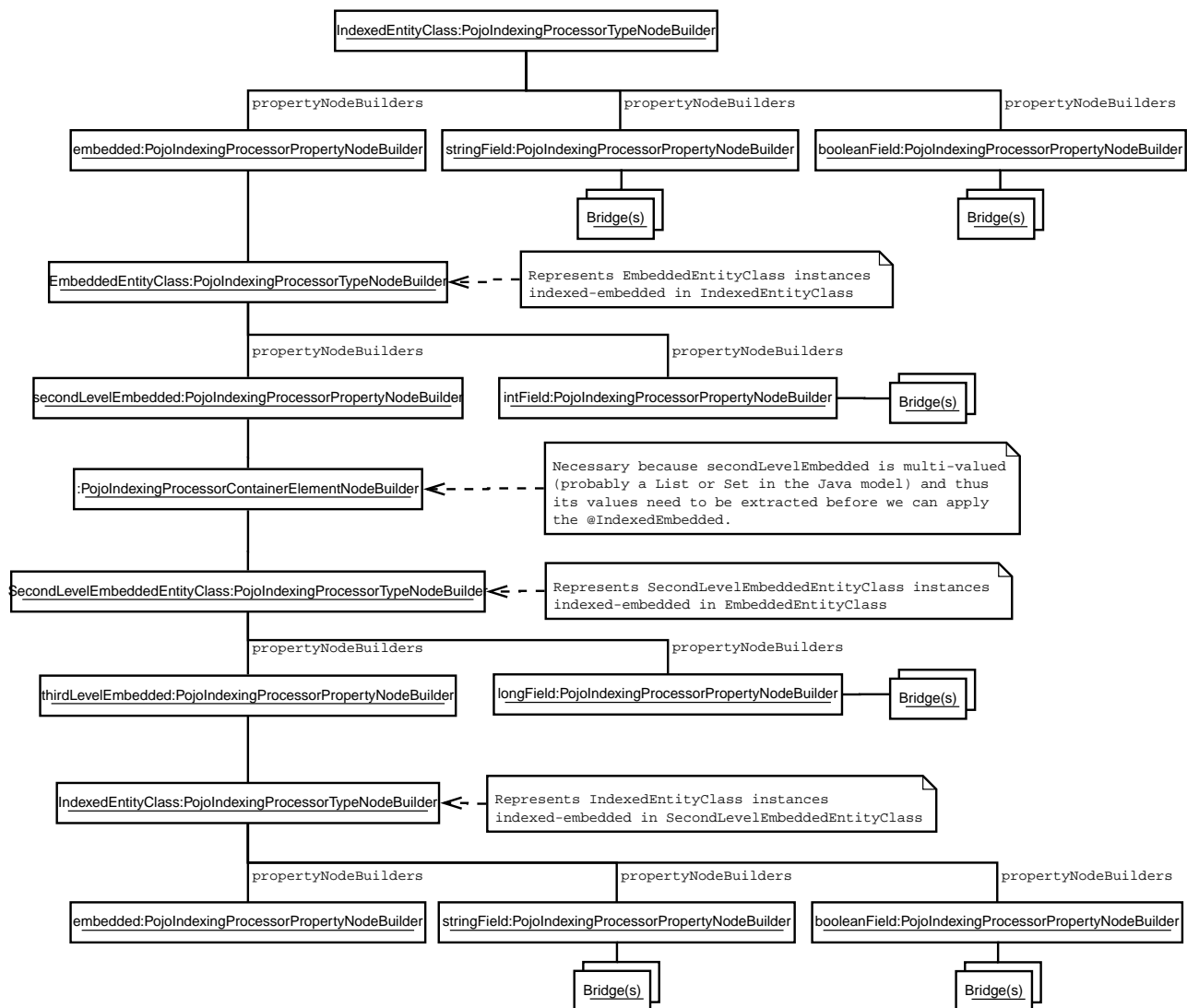
For each indexed type, the building process consists in creating a root **PojoIndexingProcessorTypeNode** builder, and applying metadata contributors to this builder (see [Bootstrap](#)), creating nested builders as the need arises (when a metadata contributor mentions a POJO property, for instance).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

As an example, let's consider the following mapped model:



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

This model is converted using the process described above into this node builder tree:



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

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

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



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

Ultimately, the created indexing process tree will follow approximately the same structure as the builder tree. The indexing processor tree may be a bit different from the builder tree, due to optimizations. In particular, some nodes may be trimmed down if we detect that the node will not

contribute anything to documents at runtime, which may happen for some property nodes when using `@IndexedEmbedded` with path filtering (`includePaths`) or depth filtering (`maxDepth`).

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

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.



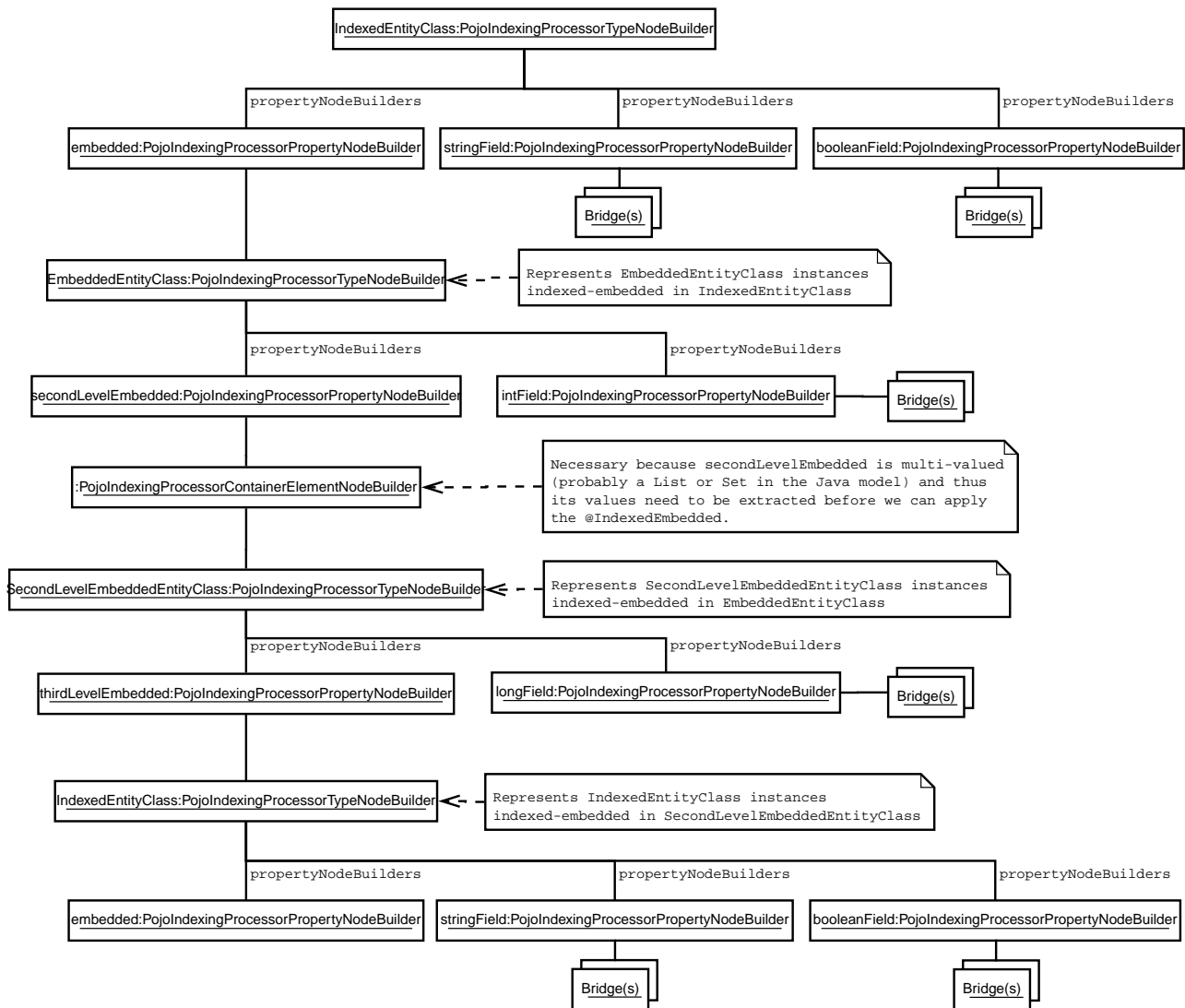
Reindexing resolvers are logged at the debug level during bootstrap. Enable this level of logging for the Hibernate Search classes if you want to understand the reindexing resolver tree that was generated for a given mapping.

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:



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

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

When `collectDependency` is called on this node, the dependency collector will first backtrack to the

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

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

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

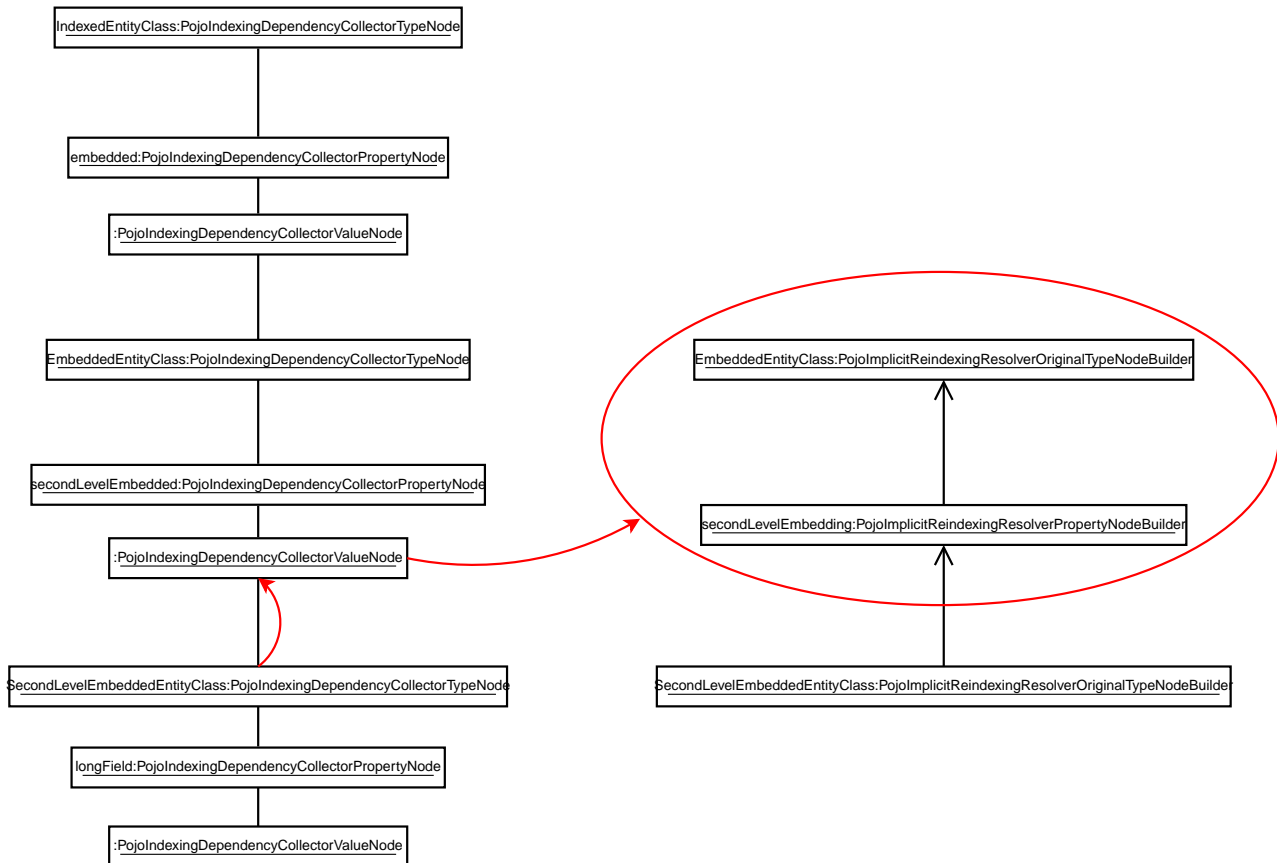


From there, the reindexing resolver builder is passed to the next dependency collector value node using the `PojoIndexingDependencyCollectorValueNode#markForReindexing` method. This method also takes as a parameter the path to the property that is depended on, in this case `longField`.

The value node will then use its knowledge of the dependency tree (using its ancestors in the dependency collector tree) to build a `BoundPojoModelPath` from the previous entity type to that value. In our case, this path is `Type EmbeddedEntityClass ⇒ Property "secondLevelEmbedded" ⇒ No container value extractor`.

This path represents an association between two entity types: `EmbeddedEntityClass` on the containing side, and `SecondLevelEmbeddedEntityClass` on the contained side. In order to complete the reindexing resolver tree, we need to **invert** this association, i.e. find out the inverse path from `SecondLevelEmbeddedEntityClass` to `EmbeddedEntityClass`. This is done in `PojoAssociationPathInverter` using the "additional metadata" mentioned in [Representation of the POJO metamodel](#).

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



The resulting reindexing resolver builder is then passed to the next dependency collector value node, and the process repeats:



Once we reach the dependency collector root, we are almost done. The reindexing resolver builder tree has been populated with every node needed to reindex `IndexedEntityClass` whenever a change occurs in the `longField` property of `SecondLevelEmbeddedEntityClass`.

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

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

Polymorphism

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

There are all sorts of intricate corner cases to take into account, but they are for the main part addressed this way:

- Whenever we create a type node in the reindexing resolver building tree, we take care to determine all the possible concrete entity types for the considered type, and create one reindexing resolver type node builder per possible entity type.
- Whenever we resolve the inverse side of an association, take care to resolve it for every concrete "source" entity type, and to apply all of the resulting inverse paths.

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

Embedded types

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

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

Fine-grained dirty checking

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

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

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

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
- Sanne Grinovero
- Mincong Huang
- Nabeel Ali Memon
- Gunnar Morling
- Yoann Rodière
- Guillaume Smet