Getting started with Hibernate Search's Standalone POJO Mapper

2025-06-06

Table of Contents

1. Assumptions . . . 2
2. Dependencies . . . 3
3. Configuration . . . 5
4. Mapping . . . 7
5. Initialization . . . 11
    5.1. Schema management . . . 11
    5.2. Initial indexing . . . 11
6. Indexing . . . 12
7. Searching . . . 13
8. Analysis . . . 15
9. What's next . . . 20

Features detailed below are *incubating*: they are still under active development.

The usual compatibility policy does not apply: the contract of incubating elements (e.g. types, methods, configuration properties, etc.) may be altered in a backward-incompatible way — or even removed — in subsequent releases.

You are encouraged to use incubating features so the development team can get feedback and improve them, but you should be prepared to update code which relies on them as needed.

This guide will walk you through the initial steps required to index and query your entities using Hibernate Search, when your entities are **not** defined in Hibernate ORM.

If your entities **are** defined in Hibernate ORM, see this other guide instead.

Chapter 1. Assumptions

This getting-started guide aims to be generic and does not tie itself to any framework in particular. If you use Quarkus, WildFly or Spring Boot, make sure to first have a look at framework support for tips and to learn about quirks.

For the sake of simplicity, this guide assumes you are building an application deployed as a single instance on a single node. For more advanced setups, you are encouraged to have a look at the Examples of architectures.

Chapter 2. Dependencies

The Hibernate Search artifacts can be found in Maven's Central Repository. If you get Hibernate Search from Maven, it is recommended to import Hibernate Search BOM as part of your dependency management to keep all its artifact versions aligned:

```
<dependencyManagement>
    <dependencies>
        <!-- Import Hibernate Search BOM to get all of its artifact versions aligned: -->
        <dependency>
            <groupId>org.hibernate.search</groupId>
            <artifactId>hibernate-search-bom</artifactId>
            <version>8.0.0.Final</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
        <!-- Any other dependency management entries -->
    </dependencies>
</dependencyManagement>
```

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 indexing and searching these documents.

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

Standalone POJO Mapper + Lucene

Allows indexing of entities that are not defined in Hibernate ORM, in a single application node, storing the index on the local filesystem.

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

```
<dependencies>
    <!--
        Add dependencies without specifying versions
        as that is already taken care by dependency management:
    -->
    <dependency>
       <groupId>org.hibernate.search</groupId>
       <artifactId>hibernate-search-mapper-pojo-standalone</artifactId>
    </dependency>
    <dependency>
       <groupId>org.hibernate.search</groupId>
       <artifactId>hibernate-search-backend-lucene</artifactId>
    </dependency>
    <!-- Any other dependency entries -->
</dependencies>
```

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

Standalone POJO Mapper + Elasticsearch (or OpenSearch)

Allows indexing of entities that are not defined in Hibernate ORM, on multiple application nodes, storing the index on a remote Elasticsearch or OpenSearch cluster (to be configured separately).

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

```
<dependencies>
    <!--
        Add dependencies without specifying versions
        as that is already taken care by dependency management:
    -->
    <dependency>
       <groupId>org.hibernate.search</groupId>
       <artifactId>hibernate-search-mapper-pojo-standalone</artifactId>
    </dependency>
    <dependency>
       <groupId>org.hibernate.search</groupId>
       <artifactId>hibernate-search-backend-elasticsearch</artifactId>
    </dependency>
    <!-- Any other dependency entries -->
</dependencies>
```

If you get Hibernate Search from the distribution bundle, copy the JARs from `dist/engine`, `dist/mapper/pojo-standalone`, `dist/backend/elasticsearch`, and their respective `lib` subdirectories.

Chapter 3. Configuration

Once you have added all required dependencies to your application, it's time to have a look at the configuration.

The configuration properties of Hibernate Search with the Standalone POJO Mapper are set programmatically when building the mapping.

Example 3.1: Hibernate Search properties for a "Standalone POJO Mapper + Lucene" setup

```
CloseableSearchMapping searchMapping = SearchMapping.builder( AnnotatedTypeSource
.fromClasses( ①
        Book.class, Author.class
) )
        .property( "hibernate.search.backend.directory.root",
                "some/filesystem/path" ) ②
        .build(); ③
```

① Create a builder, passing an `AnnotatedTypeSource` to let Hibernate Search know where to look for annotations.

② Set the location of indexes in the filesystem. By default, the backend will store indexes in the current working directory.

③ Build the `SearchMapping`.

Example 3.2: Hibernate Search properties for a "Standalone POJO Mapper + Elasticsearch/OpenSearch" setup

```
CloseableSearchMapping searchMapping = SearchMapping.builder( AnnotatedTypeSource
.fromClasses( ①
        Book.class, Author.class
) )
        .property( "hibernate.search.backend.hosts",
                "elasticsearch.mycompany.com" ) ②
        .property( "hibernate.search.backend.protocol",
                "https" ) ③
        .property( "hibernate.search.backend.username",
                "ironman" ) ④
        .property( "hibernate.search.backend.password",
                "j@rV1s" )
        .build(); ⑤
```

① Create a builder, passing an `AnnotatedTypeSource` to let Hibernate Search know where to look for annotations.

② Set the Elasticsearch hosts to connect to. By default, the backend will attempt to connect to `localhost:9200`.

③ Set the protocol. The default is `http`, but you may need to use `https`.

④ Set the username and password for basic HTTP authentication. You may also be interested in AWS IAM authentication.

⑤ Build the `SearchMapping`.

Thanks to classpath scanning, your `AnnotatedTypeSource` only needs to include one class from each JAR containing annotated types. Other types should be

automatically discovered.

See also this section to troubleshoot or improve performance of classpath scanning.

A few util methods are available to simplify the dynamic building of property keys.

Chapter 4. Mapping

Let's assume that your application contains the classes `Book` and `Author` and you want to index them in order to search for books.

Example 4.1: Book and Author types BEFORE adding Hibernate Search specific annotations

```
import java.util.HashSet;
import java.util.Set;

public class Book {

    private Integer id;

    private String title;

    private String isbn;

    private int pageCount;

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

    public Book() {
    }

    // Getters and setters
    // ...

}
```

```
import java.util.HashSet;
import java.util.Set;

public class Author {

    private Integer id;

    private String name;

    private Set<Book> books = new HashSet<>();

    public Author() {
    }

    // Getters and setters
    // ...

}
```

To make these types searchable, you will need to mark them as entity types, and map these entity types 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 entities to indexes.

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

Example 4.2: Book and Author entities AFTER adding Hibernate Search specific annotations

```
import java.util.HashSet;
import java.util.Set;

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

@SearchEntity ①
@Indexed ②
public class Book {

    @DocumentId ③
    private Integer id;

    @FullTextField ④
    private String title;

    @KeywordField ⑤
    private String isbn;

    @GenericField ⑥
    private int pageCount;

    @IndexedEmbedded ⑦
    private Set<Author> authors = new HashSet<>();

    public Book() {
    }

    // Getters and setters
    // ...

}
```

```
import java.util.HashSet;
import java.util.Set;

import
org.hibernate.search.mapper.pojo.mapping.definition.annotation.AssociationInverseSide;
import org.hibernate.search.mapper.pojo.mapping.definition.annotation.DocumentId;
import org.hibernate.search.mapper.pojo.mapping.definition.annotation.FullTextField;
import org.hibernate.search.mapper.pojo.mapping.definition.annotation.ObjectPath;
import org.hibernate.search.mapper.pojo.mapping.definition.annotation.PropertyValue;
import org.hibernate.search.mapper.pojo.mapping.definition.annotation.SearchEntity;

@SearchEntity ⑧
⑨
public class Author {

    @DocumentId ⑩
    private Integer id;

    @FullTextField ⑤
    private String name;

    @AssociationInverseSide(inversePath = @ObjectPath(@PropertyValue(propertyName =
"authors"))) ⑪
    private Set<Book> books = new HashSet<>();

    public Author() {
```

```
    }

    // Getters and setters
    // ...

}
```

① `@SearchEntity` marks the `Book` type as an entity type, which is necessary in order for it to be mapped to an index.

② `@Indexed` marks the `Book` type as indexed, i.e. an index will be created for that entity, and that index will be kept up to date. Note `@SearchEntity` could be added here, but is not necessary: with the Standalone POJO Mapper, an `@Indexed` type is implicitly always an entity.

③ `@DocumentId` marks the property used to generate a document identifier.

④ `@FullTextField` maps a property to a full-text index field with the same name and type. Full-text fields are broken down into tokens and normalized (lowercased, …). Here we're relying on default analysis configuration, but most applications need to customize it; this will be addressed further down.

⑤ `@KeywordField` maps a property to a non-analyzed index field. Useful for identifiers, for example.

⑥ Hibernate Search is not just for full-text search: you can index non-`String` types with the `@GenericField` annotation, A broad range of property types are supported out-of-the-box, such as primitive types (`int`, `double`, …) and their boxed counterpart (`Integer`, `Double`, …), enums, date/time types, `BigInteger`/`BigDecimal`, etc.

⑦ `@IndexedEmbedded` "embeds" 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 field named `authors.name` to the `Book` index. This field will be populated automatically based on the content of the `authors` property, and the books will be re-indexed whenever Hibernate Search detects that the `name` property of their author changes. See Mapping associated elements with `@IndexedEmbedded` for more information.

⑧ `@SearchEntity` marks `Author` as an entity type.

⑨ `Author` is `@IndexedEmbedded` in other entities, but does not need to be searchable by itself, so it does not need an index and does not need to be annotated with `@Indexed`.

⑩ `Author` is not indexed, but it is `@IndexedEmbedded`, so Hibernate Search will need to manipulate identifiers of `Author` instances. Putting `@DocumentId` on a uniquely identifying property makes that possible.

⑪ `Book` embeds data from the `Author` entity, so `Book` needs to be reindexed when `Author` changes, and thus Hibernate Search needs to be able to retrieve the books of a particular author when the author changes. For that reason, we need to provide information to Hibernate Search about the inverse side of associations.

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

- Multiple `@*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". This allows the mapping of types that are not supported out-of-the-box. See Binding and bridges for more information.

Chapter 5. 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 another datastore (if any) needs to be indexed.

5.1. 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.

5.2. Initial indexing

As we'll see later, Hibernate Search makes it possible to explicitly index an entity every time it changes in the application.

However, if you use Hibernate Search to index data from another datastore, this datastore may already contain data when you add the Hibernate Search integration. That data is unknown to Hibernate Search, and thus has to be indexed through a batch process.

This is possible thanks to the mass indexer API, but a little more involved than with the Hibernate ORM mapper, because a way to load entities from the other datastore needs to be plugged in.

Due to this (relative) complexity, we won't cover it in this guide, but if you want to know more, have a look at Mass indexing.

Chapter 6. Indexing

Indexing is performed explicitly through an `SearchIndexingPlan`, accessible from a `SearchSession`.

By default, in particular when using the Elasticsearch backend, changes will not be visible right after the session is closed. 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 session, and then execute a search query right after that session is closed, 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.

Use `add` when the entity is first added and you are absolutely certain it does not exist in the index yet:

Example 6.1: Using the Standalone POJO Mapper to index data

```
try ( SearchSession session = searchMapping.createSession() ) { ①
    Author author = new Author(); ②
    author.setId( 1 );
    author.setName( "John Doe" );

    Book book = new Book();
    book.setId( 2 );
    book.setTitle( "Refactoring: Improving the Design of Existing Code" );
    book.setIsbn( "978-0-58-600835-5" );
    book.setPageCount( 200 );
    book.getAuthors().add( author );
    author.getBooks().add( book );

    session.indexingPlan().add( author ); ③
    session.indexingPlan().add( book );
}
```

Use `addOrUpdate` to add/update the entity when the entity may already exist in the index:

Example 6.2: Using the Standalone POJO Mapper to index data

```
try ( SearchSession session = searchMapping.createSession() ) {
    book.setTitle( "On Cleanup of Existing Code" );
    session.indexingPlan().addOrUpdate( book );
}
```

Add use `delete` to delete the entity from the index:

Example 6.3: Using the Standalone POJO Mapper to index data

```
try ( SearchSession session = searchMapping.createSession() ) {
    session.indexingPlan().delete( book );
}
```

Chapter 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` contains the string `refactoring`. The matches are implicitly on words ("tokens") instead of the full string, and are case-insensitive: that's because the targeted fields are **full-text** fields with the default analyzer being applied.

Example 7.1: Using Hibernate Search to query the indexes

```
try ( SearchSession session = searchMapping.createSession() ) { ①
    SearchResult<Integer> result = session.search( Book.class ) ②
            .select( f -> f.id( Integer.class ) ) ③
            .where( f -> f.match() ④
                    .fields( "title", "authors.name" )
                    .matching( "refactoring" ) )
            .fetch( 20 ); ⑤

    long totalHitCount = result.total().hitCount(); ⑥
    List<Integer> hits = result.hits(); ⑦

    List<Integer> hits2 =
            /* ... same DSL calls as above... */
                    .fetchHits( 20 ); ⑧
}
```

① Create a Hibernate Search session, called `SearchSession`.

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

③ Define that we will only retrieve entity identifiers. Retrieving entities directly is possible, but requires more configuration, so we won't discuss it in this guide.

④ 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.

⑤ Build the query and fetch the results, limiting to the top 20 hits.

⑥ Retrieve the total number of matching entities.

⑦ Retrieve the identifiers of matching entities.

⑧ 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 7.2: Using Hibernate Search to query the indexes — object-based syntax

```
try ( SearchSession session = searchMapping.createSession() ) { ①
    SearchScope<Book> scope = session.scope( Book.class ); ②

    SearchResult<Integer> result = session.search( scope ) ③
            .select( f -> f.id( Integer.class ) ) ④
            .where( scope.predicate().match() ⑤
                    .fields( "title", "authors.name" )
                    .matching( "refactoring" )
```

```
                    .toPredicate() )
            .fetch( 20 ); ⑥

    long totalHitCount = result.total().hitCount(); ⑦
    List<Integer> hits = result.hits(); ⑧

    List<Integer> hits2 =
            /* ... same DSL calls as above... */
                    .fetchHits( 20 ); ⑨
}
```

① Create a Hibernate Search session, called `SearchSession`.

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 we will only retrieve entity identifiers. Retrieving entities directly is possible, but requires more configuration, so we won't discuss it in this guide.

⑤ Define that only documents matching the given predicate should be returned. The predicate is created using the same search scope as the query.

⑥ Build the query and fetch the results, limiting to the top 20 hits.

⑦ Retrieve the total number of matching entities.

⑧ Retrieve the identifiers of matching entities.

⑨ 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()`.

Example 7.3: Using Hibernate Search to count the matches

```
try ( SearchSession session = searchMapping.createSession() ) {
    long totalHitCount = session.search( Book.class )
            .where( f -> f.match()
                    .fields( "title", "authors.name" )
                    .matching( "refactoring" ) )
            .fetchTotalHitCount(); ①
}
```

① Fetch the total hit count.

While the examples above retrieved hits as entity identifiers, it is just one of the possible hit types. See Projection DSL for more information.

Chapter 8. Analysis

Full-text search allows fast matches on words in a case-insensitive way, which is one step further than substring search in a relational database. But it can get much better: what if we want a search with the term "refactored" to match our book whose title contains "refactoring"? That's possible with custom analysis.

Analysis defines how both the indexed text and search terms are supposed to be processed. This involves *analyzers*, which are made up of three types of components, applied one after the other:

- zero or (rarely) 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, in particular the default one, which will:

- tokenize (split) the input according to the Word Break rules of the Unicode Text Segmentation algorithm;
- filter (normalize) tokens by turning uppercase letters to lowercase.

The default analyzer is a good fit for most language, but is not very advanced. To get the most out of analysis, you will need to define a custom analyzer by picking the tokenizer and filters 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.

- Finally, an "ascii-folding" filter, which replaces characters with diacritics ("é", "à", ...) with their ASCII equivalent ("e", "a", ...).

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

Example 8.1: Analysis configurer implementation and configuration for a "Standalone POJO Mapper + Lucene" setup

```
package
org.hibernate.search.documentation.mapper.pojo.standalone.gettingstarted.withhsearch.custom
analysis;

import org.hibernate.search.backend.lucene.analysis.LuceneAnalysisConfigurationContext;
import org.hibernate.search.backend.lucene.analysis.LuceneAnalysisConfigurer;

public class MyLuceneAnalysisConfigurer implements LuceneAnalysisConfigurer {
    @Override
    public void configure(LuceneAnalysisConfigurationContext context) {
        context.analyzer( "english" ).custom() ①
                .tokenizer( "standard" ) ②
                .tokenFilter( "lowercase" ) ③
                .tokenFilter( "snowballPorter" ) ③
                        .param( "language", "English" ) ④
                .tokenFilter( "asciiFolding" );

        context.analyzer( "name" ).custom() ⑤
                .tokenizer( "standard" )
                .tokenFilter( "lowercase" )
                .tokenFilter( "asciiFolding" );
    }
}
```

```
CloseableSearchMapping searchMapping = SearchMapping.builder( AnnotatedTypeSource
.fromClasses(
        Book.class, Author.class
) )
        .property( "hibernate.search.backend.directory.root",
                "some/filesystem/path" )
        .property( "hibernate.search.backend.analysis.configurer",
                BeanReference.of( MyLuceneAnalysisConfigurer.class, BeanRetrieval.CLASS ) )
⑥
        .build();
```

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 Lucene-specific names to refer to tokenizers; see Custom analyzers and normalizers for information about available tokenizers, their name and their parameters.

③ Set the token filters. Token filters are applied in the order they are given. Here too, Lucene-specific names are expected; see Custom analyzers and normalizers for information about available token filters, their name and their parameters.

④ Set the value of a parameter for the last added char filter/tokenizer/token filter.

⑤ Define another custom analyzer, called "name", to analyze author names. On contrary to the first one, do not enable stemming (no `snowballPorter` token filter), as it is unlikely to lead to useful results on proper nouns.

⑥ Assign the configurer to the backend in the Hibernate Search configuration properties. For more information about the bean references, see Bean references.

Example 8.2: Analysis configurer implementation and configuration for a "Standalone POJO Mapper + Elasticsearch/OpenSearch" setup

```
package
org.hibernate.search.documentation.mapper.pojo.standalone.gettingstarted.withhsearch.custom
analysis;

import
org.hibernate.search.backend.elasticsearch.analysis.ElasticsearchAnalysisConfigurationConte
xt;
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
                .tokenFilters( "lowercase", "snowball_english", "asciifolding" ); ③

        context.tokenFilter( "snowball_english" ) ④
                .type( "snowball" )
                .param( "language", "English" ); ⑤

        context.analyzer( "name" ).custom() ⑥
                .tokenizer( "standard" )
                .tokenFilters( "lowercase", "asciifolding" );
    }
}
```

```
CloseableSearchMapping searchMapping = SearchMapping.builder(
AnnotatedTypeSource.fromClasses(
        Book.class, Author.class
) )
        .property( "hibernate.search.backend.hosts",
                "elasticsearch.mycompany.com" )
        .property( "hibernate.search.backend.analysis.configurer",
                BeanReference.of( MyElasticsearchAnalysisConfigurer.class,
BeanRetrieval.CLASS ) ) ⑦
        .build();
```

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 Elasticsearch-specific names to refer to tokenizers; see Custom analyzers and normalizers for information about available tokenizers, their name and their parameters.

③ Set the token filters. Token filters are applied in the order they are given. Here too, Elasticsearch-specific names are expected; see Custom analyzers and normalizers for information about available token filters, their name and their parameters.

④ Note that, for Elasticsearch, any parameterized char filter, tokenizer or token filter must be defined separately and assigned a new 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 enable stemming (no `snowball_english` token filter), as it is unlikely to lead

to useful results on proper nouns.

⑦ Assign the configurer to the backend in the Hibernate Search configuration properties. For more information about the bean references, see Bean references.

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

Example 8.3: Book and Author entities after adding Hibernate Search specific annotations

```
import java.util.HashSet;
import java.util.Set;

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

@SearchEntity
@Indexed
public class Book {

    @DocumentId
    private Integer id;

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

    @KeywordField
    private String isbn;

    @GenericField
    private int pageCount;

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

    public Book() {
    }

    // Getters and setters
    // ...

}
```

```
import java.util.HashSet;
import java.util.Set;

import
org.hibernate.search.mapper.pojo.mapping.definition.annotation.AssociationInverseSide;
import org.hibernate.search.mapper.pojo.mapping.definition.annotation.DocumentId;
import org.hibernate.search.mapper.pojo.mapping.definition.annotation.FullTextField;
import org.hibernate.search.mapper.pojo.mapping.definition.annotation.ObjectPath;
import org.hibernate.search.mapper.pojo.mapping.definition.annotation.PropertyValue;
import org.hibernate.search.mapper.pojo.mapping.definition.annotation.SearchEntity;

@SearchEntity
public class Author {

    @DocumentId
    private Integer id;
```

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

    @AssociationInverseSide(inversePath = @ObjectPath(@PropertyValue(propertyName =
"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.

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

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

Example 8.4: Using Hibernate Search to query the indexes after analysis was configured

```
try ( SearchSession session = searchMapping.createSession() ) {
    SearchResult<Integer> result = session.search( Book.class )
            .select( f -> f.id( Integer.class ) )
            .where( f -> f.match()
                    .fields( "title", "authors.name" )
                    .matching( "refactored" ) )
            .fetch( 20 );
}
```

Chapter 9. What's next

The above paragraphs gave you 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 review the Examples of architectures to pick the most appropriate for your use case.

You may also want to explore the basic features in more detail. Two topics which were only briefly touched in this tutorial were analysis configuration (Analysis) and bridges (Binding and bridges). Both are important features required for more fine-grained indexing.

When it comes to initializing your index, you will be interested in schema management and mass indexing.

When querying, you will probably want to know more about predicates, sorts, projections, aggregations, highlights.